Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Developing a CUDA solver for large sparse matrices for MARIN

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

MARTIJN DE JONG

Delft, the Netherlands
February 2012

MSc THESIS APPLIED MATHEMATICS

"Developing a CUDA solver for large sparse matrices for MARIN"

MARTIJN DE JONG

Delft University of Technology

Daily supervisor

Prof. dr. ir. C. Vuik

Responsible professor

Prof. dr. ir. C. Vuik

Other thesis committee members

Dr. ir. A. Ditzel

Dr. ir. A. van der Ploeg

Dr. ir. H.X. Lin

March 12, 2012

Delft, the Netherlands

Preface

This masters thesis has been written for the degree of Master of Science in Applied Mathematics at the faculty of Electrical Engineering, Mathematics and Computer Sciences of Delft University of Technology. The report ends a nine month internship carried out at Maritime Research Institute Netherlands (MARIN).

MARIN supplies innovative products for the offshore industry and shipping companies. One of their products is a full-scale bridge simulator which can be used to train captains, steersmen and other ship workers, but also for research and consultancy purposes. Also much smaller real-time simulators are available in the form of software that can be installed and run on a single or multiple desktop PCs.

To bring the simulator to a next level a new wave model is under development. The project that deals with this new wave model is called the "Interactive Waves" project. As the name explains, the project focusses on developing a simulator in which ships and waves interact. The new wave model is the Variational Boussinesq model (VBM) as suggested by Gert Klopman. However, this new realistic model brings much more computation effort with it. The VBM namely requires a solver that solves each frame ($20\,fps$) a system $S\psi = b$.

Using VBM most of the computational time is absorbed by the solver, which makes it impossible to simulate domains larger than 200×400 nodes in real-time. The focus of the Master's project was thus on developing a fast `CUDA` solver that would deliver a really good speed-up, so that the solving the system $S\psi = b$ is no longer the time bottleneck in the simulator.

Outline

The layout of the report as follows. The report consists of six parts (I, II, III, IV, V and VI), each consisting of multiple chapters.

Part I provides background information and explains our problem in detail. Also, the Variational Boussinesq model (VBM), test problems and test equipment are discussed. Part I ends with a plan of approach.

Part II is on linear solvers. After some preliminaries, mostly linear algebra results, we discuss consecutively direct methods, basic iterative methods (BIMs), the Conjugate Gradient (CG) algorithm, preconditioners, deflation, and Multigrid (MG).

In Part III we discuss the architecture of a GPU and strategies for fast `CUDA` software. Also notions on performance, timing and speed-up are discussed. At the end of Part II we find two `CUDA` applications which illustrate nicely the most important concepts in designing fast `CUDA` code.

In Part IV the new `CUDA` solvers are discussed in great detail. The discussion on the RRB-solver may serve as a manual. The part contains all the source code of the kernels for

our two new CUDA solvers: the RRB-solver and IPDIAG-solver. The full source code can be obtained via MARIN.

Part V deals with the experiments and corresponding results. First our testing strategy is explained so that there can be no confusion on how we have timed the code and how speed-up factors are computed.

Finally, in Part VI we find conclusions and recommendations for future research.

Reading advice

Not all chapters and sections are equally important. As the report is quite thick, we recommend reading at least the chapters and sections listed in the table below completely if one has little time; the rest of the report can be "scan read".

Part:	Chapter	Sections to read:
I	1-3	all
	5	all
II	7	7.1, 7.2
	10	10.1.1, 10.2.2, 10.2.3
III	13-15	all
	16	16.1, 16.3, 16.4
	17	all
IV	18	18.1
	19	19.1, 19.2, 19.5
	20	20.1
V	21-23, 25	all
VI	26, 27	all

Acknowledgements

I would like to thank MARIN and in particular Auke Ditzel for giving me the opportunity to do research at MARIN on this challenging project, Prof. Vuik for providing me feedback on my work and support, and Anneke Sicherer-Roetman and John Jaspersen for their help and company. Also, I would like to thank Elwin van 't Wout, his work helped me alot.

Martijn de Jong,

Delft, February 2012.

Contents

Part I

PROBLEM FORMULATION AND DESIGN PLAN

Chapter 1

Backgrounds: the Interactive Waves project

The Interactive Waves ("Interactieve Golven") project was initiated a few years ago to bring MARIN's real-time simulator, and in particular the real-time wave model, to a next level.

In the next sections we discuss the backgrounds of the Interactive Waves project and the work that has been performed on this project so far. The information in these sections comes for a good part from the presentation "Interactieve Golven: ups and downs" held by Anneke Sicherer-Roetman on 16 februari 2010 at MARIN. Also several memos are used that were sent regularly to all stakeholders with updates on the progress and latest developments in the Interactive Waves project.

1.1 Backgrounds and history

In the current version of the Mermaid simulator the wave model is deterministic. The wave-induced motions come from a deterministic force model in combination with a database that links waves to ship movements. Because of deterministic waves the waves are not influenced at all by ships, moles, breakwaters, piers, or any other arbitrary object.

However, the underlying model in Mermaid simulator is already much more realistic and complete than the visualization models from the film industry (e.g. Waterworld, Titanic, Perfect Storm). In contrast to these models in the Mermaid simulator the ship movements are realistic.

A better model was suggested by Gert Klopman: the Variational Boussinesq model (VBM). The model is discussed in Chapter 2. The big advantage of this Variational Boussinesq model over the current deterministic model is its completeness: in the "Klopman model" the waves and ships really interact, i.e., the movements of the ship are influenced by the waves and the waves in their turn are influenced by the ship, hence the project name "Interactive Waves". Moreover, the model can deal with deep waters with varying depth, which is quite unique. However, logically, one pays for this additional future: the model is much more computation-intensive and therefore a really fast solver is needed.

Gert Klopman delivered in the 3th quarter of 2007 `Fortran` source code in which the VBM was embedded: the `lin_wacu` program. The underlying test problem in the software was the so-called "Klopman's harbour", see Section 3.2, that includes a varying bathymetry. The original program already had wave makers, comparable to the wave makers in the MARIN

bassins, to generate all kinds of waves. In the program the ship was modelled by an elliptical "pressure puls", and the visualization is made with `PGPLOT` ("moving heat map"). Experiments showed that the sequential solver in Klopman's code took about 75% of the computation time, which was way too much.

The next steps were to incorporate the VBM in the Mermaid simulator, which is written in `C++`, and to design a faster parallel solver. Klopman's code had to be translated into `C++`, and simultaneously be restructered so that it became more modular. This work was carried out by Anneke Sicherer-Roetman in the 4th quarter of 2007, and the first two quarters of 2008. Her efforts have resulted in highly efficient and transparant `C++` code with a class structure. For the visualization now use was made of `OpenSceneGraph` (`OSG`) instead of `PGPLOT`, which allows the user to watch the simulations in 3D.

For designing a new fast parallel solver MARIN contacted the department of Numerical Analysis from Delft University of Technology led by Prof. Kees Vuik. While MARIN was waiting on a math student, Anneke worked during the 3th quarter of 2008 on parallelizing the software, except from the solver part, further optimizing the code, and running more tests with different pressure pulses ("bath-tub" shapes) and multiple pressure pulses (two ships).

In the 4th quarter of 2008, Elwin van 't Wout from Delft Univiversity of Technology arrived at MARIN to do research on a fast parallel solver for his Master's project. His efforts have resulted in a fast preconditioned Conjugate Gradient solver. The work of Elwin was carried out under supervision of Prof. Vuik, Auke Ditzel and Auke van der Ploeg and it was completed in August 2009. Because programming on the GPU is difficult and modern CPUs contain more and more computing units, in first instance the focus was on a fast solver for the CPU. The fastest solver turned out to be a CG solver preconditioned by the RRB-method, or briefly "the RRB-solver" which has been used since then.

1.2 Earlier work

1.2.1 Gert Klopman: model maker

Dr.ir. Gert Klopman came up with the Variational Boussineq model (VBM) that is used in the Interactive Waves software. Gert Klopman provided `Fortran` source code, called "`lin_wacu`" in which the VBM was embedded. Besides his contributions to MARIN's Interactive Waves project, he has published many papers on the Variational Boussinesq model, modelling of linear water waves, waves in varying bathymetries, etc.

1.2.2 Elwin van 't Wout: faster solvers and model explanation

For his Master's project Ir. Elwin van 't Wout has written an excellent thesis [28]. The project was carried out upon instructions from MARIN. The project was about improving the linear solver that is used in a real-time ship simulator. The underlying wave model is the variational Boussinesq model as suggested by Gert Klopman. In the thesis we find a very(!) complete and accurate description of the model, and lots of details on its discretization. Below follows a brief summary of his work.

Elwin starts with the derivation of the Variational Boussinesq model (VBM). The corresponding sections are based on several papers by Gert and some emails and conversations with Gert. We read that essential in the VBM is the use of so-called vertical shape functions

to reduce the 3D model to a 2D model. Moreover, the resulting model is linearized to reduce the computational effort even further.

After deriving the VBM the model is discretized. For time discretization the Leapfrog method is used, and for spatial discretization the finite volume method is used to discretize the equations, yielding a linear system $S\psi = b$, with S a large SPD matrix, resulting from a 5-point stencil. This system must be solved in real-time (frame rate: $20\,fps \rightarrow$ frame time $= 0.05\,s$).

Elwin points out that at this moment typical problems that can be solved real-time involve a matrix S of size 20000×20000 (resulting from a $5\,m \times 5\,m$ mesh, and 200 grid points in the x-direction and 100 grid points in the y-direction). Much bigger problems cannot be solved real-time, and this is the main problem: as captains can see miles away, for the simulator to become realistic it is necessary that the waves can be computed for much larger domains. Therefore, the solver used in the Interactive Waves model is reviewed.

The solver that was used was the RRB-solver, with as many levels as the problem allows, implemented in `Fortran` by Auke van der Ploeg [19]. This solver was slightly modified such that the number of grid levels was no longer fixed. The modified version is called the RRB-k method (k is the number of grid levels). Besides this solver Elwin investigated several other solvers all based on CG, but with different preconditioners (diagonal scaling, RIC), and with or without deflation. Therefore, at first the properties of the matrix and its spectrum are analyzed in order to gain insight into topics as convergence and condition number. Next, we find a good discussion on Krylov subspace methods, CG, preconditioners, and deflation. The most important results for MARIN are accompanied with mathematical reasonings, formulas and proofs.

Elwin also gives some suggestions on how to implement the methods efficiently (Eisenstat's implementation, termination criterium, using `LAPACK` routines, etc.). The solvers have been tested with a set of test problems: (i) open sea, (ii) Klopman's harbour, (iii) IJssel. Lots of settings have been investigated, e.g., for RRB-k the number of levels is varied, and for RICCG with deflation (RICDEF) the amount of deflation vectors is varied. After lots of experiments the RRB-k method and the RICDEF method turn out to be the most promising methods. Elwin succeeded to reduce the computation time by 25% by optimally choosing the number of levels in RRB-k. Furthermore, we find some first attempts for a parallel implementation and some hints, e.g., as incomplete Cholesky is inherently sequential, a block-version of the algorithm is proposed. Besides finding out to what extent the methods can be parallelized on multi-CPU or GPU-machines, investigating the Multigrid method is suggested as future research by Elwin.

1.2.3 Anneke Sicherer-Roetman: code optimization

Dr. Anneke Sicherer-Roetman is the main programmer of the Interactive Waves project. In first instance Anneke ported the `Fortran` code by Gert Klopman to `C++` code with a very transparent class structure. Moreover, the code was restructured so that hardcoded parts of the code that should have been external inputs from the beginning, e.g., the shape of the pressure puls, now can be freely chosen by the user.

Also, Anneke has improved, optimized and parallelized parts of the code. For example, the amount of 2D-arrays that have the size of the system matrix has been strongly reduced, so that less memory is required; also, depending on the situation the user can now choose to use either floats or doubles. Furthermore, where possible, except from the solvers, the code

has been parallelized with OpenMP yielding a nice speed up of about 1.3-1.8× on a quad-core system.

Anneke also investigated to what extent the RRB-solver could be parallelized; unfortunately, however, it turned out to be very difficult to parallelize the RRB-solver with OpenMP. At the time, Anneke had to conclude that parallelizing this part of the code was not viable.

In the beginning of 2010 Anneke investigated if ready CUDA solvers could be used for the MARIN problem. Five CUDA solvers were found: Multigrid with CUDA by Chris Bunch, OpenCurrent 1.0.0 by Jonathan Cohen, Iterative CUDA by Andreas Klöckner, and OpenNL 3.0.2. Gert Klopman suggested ParaSails (which uses MPI not CUDA). Three out of five (the last three) turned out to be good candidates and could readily be applied without too many difficulties or modifications to the code.

The results were not very promising: for Klopman's harbour the CUDA solvers were much slower than the C++ RRB-solver. However, for larger problems, such as the open sea, Iterative CUDA did show a significant speed up. One reason for the disappointing results is the fact that the ready CUDA solvers do not exploit the structure of our problem, e.g., the fact that the matrix is a pentadiagonal matrix is not exploited, moreover, the Iterative CUDA solver, which is a CG-type solver, uses each time a zero initial guess in the CG algorithm, while the solution from the previous time step would be a much better initial guess.

1.3 What we are going to do

Our main target is to design and implement fast CUDA solvers. The solvers should be able to solve a set of test problems accurately and fast enough for real-time simulators. Also, the solvers should come well documented, so that in the future other programmers can work with, understand and hopefully even modify the CUDA solvers, so that they can be kept up-to-date.

Chapter 2

The model

The model that is used in the Interactive Waves software is the Variational Boussinesq model (VBM) as proposed by Gert Klopman. In contrast to the standard Boussinesq model (1871), the VBM can be used to model water waves in deep water with varying current and depth.

The VBM and related topics can be found in Gert Klopman's PhD thesis [13]. However, a first and very complete and detailed description of the model is given in the MSc thesis of Elwin van 't Wout [28]. For this reason, we shall only give a global description about the model without deriving it or going into details; full details can be found in [13, 28] and the references therein.

2.1 The Variational Boussinesq model (VBM)

We jump in at the point where things get interesting for us: the point where the VBM equations have been linearized for the sake of simplicity and reduction of the required computational effort. The governing linearized VBM equations are given by (cf. [28]; Chapter 5):

$$\frac{\partial \zeta}{\partial t} + \nabla \cdot (\zeta \mathbf{U} + h\nabla\varphi - h\mathcal{D}\nabla\psi) = 0, \tag{2.1.1a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U} \cdot \nabla\varphi + g\zeta = P_s, \tag{2.1.1b}$$

$$\mathcal{M}\psi + \nabla \cdot (h\mathcal{D}\nabla\varphi - \mathcal{N}\nabla\psi) = 0. \tag{2.1.1c}$$

Herein is:

ζ	water level[1,2]	h	water depth[4]
φ	surface velocity potential[2]	$\mathbf{U}$	current[5]
ψ	vertical structure[2,3]	P_s	pressure puls ship[6]
g	gravity	$\mathcal{D}, \mathcal{M}, \mathcal{N}$	model parameters[7]

Notes:

1. The water level $\zeta = \zeta(x, y, t)$ is relative to a reference level $z = 0$. The reference level can be thought of as the level if the water surface were completely flat (no waves at all), like the MARIN water bassins before a simulation.

2. The VBM equations are derived from the Euler equations for irrotatianal flow, leading to the instationary Bernoulli equation. Because the vertical structure of water flows is often known, the velocity potential ϕ occuring in the instationary Bernoulli equation is expanded in predefined shape functions f_m, i.e.,

$$\phi(x,y,z,t) = \varphi(x,y,t) + \sum_{m=1}^{M} f_m(z,\zeta)\psi_m(x,y,t).$$

This reduces the 3D-model to a 2D-model and the computational effort. Two realistic choices for f_m are parabolic and cosine-hyperbolic shapes for which the expansion consists of only one shape function. With $M = 1$ we can simply write

$$\phi(x,y,z,t) = \varphi(x,y,t) + f(z)\psi(x,y,t). \tag{2.1.2}$$

The basic variables of the VBM thus become ζ, φ, and ψ. Note that f does no longer depend on ζ. This is the consequence of an approximating argument during the linearization process.

3. Together with the shape function f the vertical structure ψ allows variations in the vertical water velocity. ψ occurs in the expansion in shape functions, see expression (2.1.2). ψ is not known a priori and thus is one of the three basis variables to solve for in the linearized VBM.

4. The water depth $h = h(x,y,t)$ is also relative to the reference level $z = 0$. The bottom of a bassin, river or sea is thus at level $z = -h$.

5. $\mathbf{U} = \mathbf{U}(x,y,t)$ is the average horizontal velocity of the current over time and is delivered to the model in the form of external data.

6. Although physically not correct at MARIN we like to speak about "pressure puls" (actually we only say it in Dutch: "drukpuls"); in fact, we have $P_s := -\frac{p_s}{\rho}$, where p_s is a pressure (predefined) modelling the ship, and ρ is the density of water. So P_s is in $[m^2 s^{-2}]$ instead of [Pascal] $= [kg \cdot m s^{-2}]$. The pressure puls is computed via $P_s = gd$, where g is the gravity and d is the draft (Dutch: "diepgang") of the ship.

7. The model parameters (functionals) are implied by the shape function f through

$$\mathcal{D} = -\frac{1}{h}\int_{-h}^{0} f \, \mathrm{d}z, \quad \mathcal{M} = \int_{-h}^{0} (f')^2 \, \mathrm{d}z, \quad \text{and} \quad \mathcal{N} = \int_{-h}^{0} f^2 \, \mathrm{d}z. \tag{2.1.3}$$

Physically realistic choices for the vertical shape functions f are parabolic and cosine-hyperbolic shape functions [13]. For the linearized parabolic model we have

$$\mathcal{D}^{(p)} = \frac{1}{3}h, \quad \mathcal{M}^{(p)} = \frac{1}{3}h, \quad \text{and} \quad \mathcal{N}^{(p)} = \frac{2}{15}h^3. \tag{2.1.4}$$

For the linearized cosine-hyperbolic model we have

$$\begin{aligned} \mathcal{D}^{(c)} &= \mathcal{C} - \frac{\mathcal{S}}{\kappa h}, \\ \mathcal{M}^{(c)} &= \frac{1}{2}\kappa\mathcal{S}\mathcal{C} - \frac{1}{2}\kappa^2 h, \\ \mathcal{N}^{(c)} &= -\frac{3}{2}\frac{1}{\kappa}\mathcal{S}\mathcal{C} + \frac{1}{2}h + h\mathcal{C}^2, \end{aligned} \tag{2.1.5}$$

where

$$\mathcal{S} = \sinh(\kappa h) \quad \text{and} \quad \mathcal{C} = \cosh(\kappa h). \tag{2.1.6}$$

2.2 The computational domain

The domain is chosen to be rectangular. Its dimensions are $L_x \times L_y$. For discretization of the VBM equations this domain is covered with an equidistant grid with $N_x \times N_y$ grid points or nodes. The outermost nodes are placed on the physical boundary. The mesh spacing in the x- and y-direction are $\Delta x = \frac{L_x}{N_x - 1}$ and $\Delta y = \frac{L_y}{N_y - 1}$, respectively. The nodes are numbered as (i, j) with $i = 1, 2, \ldots, N_x$ and $j = 1, 2, \ldots, N_y$. An example is given in Figure 2.1.

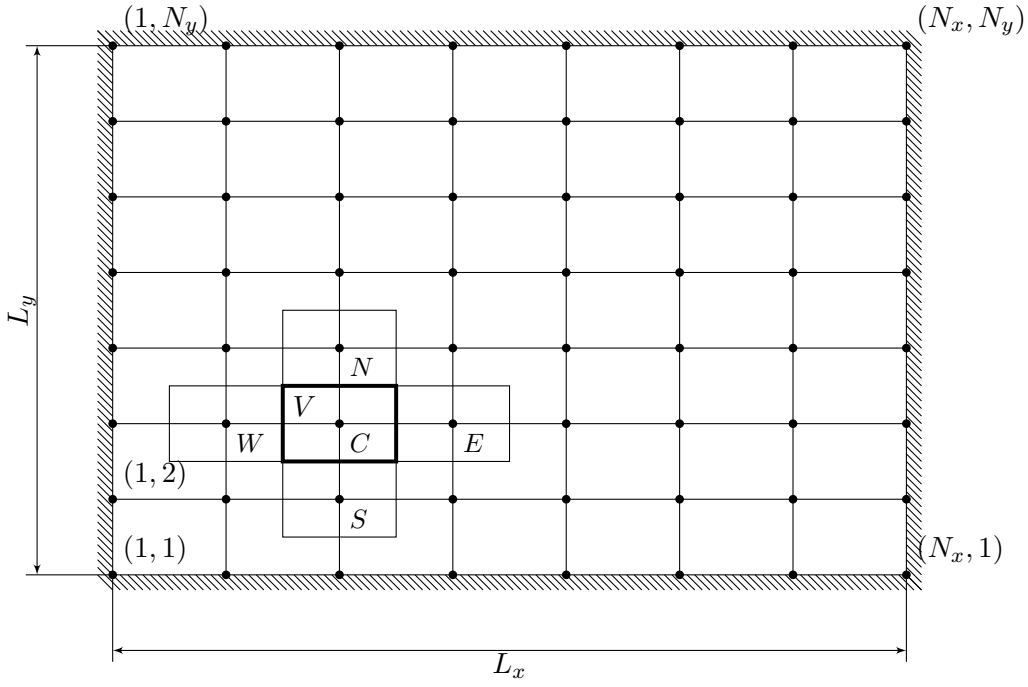


Figure 2.1: The physical domain and its corresponding grid necessary for discretization of the VBM equations. Here: $N_x = 8$ and $N_y = 8$.

2.3 Discretization of the VBM equations

The VBM equations are discretized in space with the finite volume method (FVM). For the grid point located at C (= center) the surrounding control volume V and its four nearest neighbours (N = north, E = east, S = south, W = west) are indicated by rectangles of size $\Delta x \times \Delta y$. For time integration the Leapfrog method is used. Discretization of the equations

(2.1.1) with the FVM yields:

$$\begin{aligned}
&\Delta x\Delta y\frac{\mathrm{d}\zeta_C}{\mathrm{d}t} + \tfrac{1}{2}\big(\overline{V_N}\Delta x\zeta_N + \overline{U_E}\Delta y\zeta_E - \overline{V_S}\Delta x\zeta_S - \overline{U_W}\Delta y\zeta_W\big)\\
&\qquad + \tfrac{1}{2}\big(\overline{V_N}\Delta x + \overline{U_E}\Delta y - \overline{V_S}\Delta x - \overline{U_W}\Delta y\big)\zeta_C\\
&\qquad + \tfrac{\Delta x}{\Delta y}\overline{h_N}\varphi_N + \tfrac{\Delta y}{\Delta x}\overline{h_E}\varphi_E + \tfrac{\Delta x}{\Delta y}\overline{h_S}\varphi_S + \tfrac{\Delta y}{\Delta x}\overline{h_W}\varphi_W\\
&\qquad - \big(\tfrac{\Delta x}{\Delta y}\overline{h_N} + \tfrac{\Delta y}{\Delta x}\overline{h_E} + \tfrac{\Delta x}{\Delta y}\overline{h_S} + \tfrac{\Delta y}{\Delta x}\overline{h_W}\big)\varphi_C\\
&- \tfrac{\Delta x}{\Delta y}\overline{h_N}\,\overline{\mathcal{D}_N}\psi_N - \tfrac{\Delta y}{\Delta x}\overline{h_E}\,\overline{\mathcal{D}_E}\psi_E - \tfrac{\Delta x}{\Delta y}\overline{h_S}\,\overline{\mathcal{D}_S}\psi_S - \tfrac{\Delta y}{\Delta x}\overline{h_W}\,\overline{\mathcal{D}_W}\psi_W\\
&\qquad + \big(\tfrac{\Delta x}{\Delta y}\overline{h_N}\,\overline{\mathcal{D}_N} + \tfrac{\Delta y}{\Delta x}\overline{h_E}\,\overline{\mathcal{D}_E} + \tfrac{\Delta x}{\Delta y}\overline{h_S}\,\overline{\mathcal{D}_S} + \tfrac{\Delta y}{\Delta x}\overline{h_W}\,\overline{\mathcal{D}_W}\big)\psi_C = 0, \qquad (2.3.1a)\\
&\Delta x\Delta y\frac{\mathrm{d}\varphi_C}{\mathrm{d}t} + \tfrac{1}{2}\big(\overline{V_N}\Delta x\varphi_N + \overline{U_E}\Delta y\varphi_E - \overline{V_S}\Delta x\varphi_S - \overline{U_W}\Delta y\varphi_W\big)\\
&\qquad - \tfrac{1}{2}\big(\overline{V_N}\Delta x + \overline{U_E}\Delta y + \overline{V_S}\Delta x - \overline{U_W}\Delta y\big)\varphi_C + \Delta x\Delta y g\zeta_C = P_{sC}\Delta x\Delta y, \qquad (2.3.1b)\\
&\tfrac{\Delta x}{\Delta y}\overline{h_N}\,\overline{\mathcal{D}_N}\varphi_N + \tfrac{\Delta y}{\Delta x}\overline{h_E}\,\overline{\mathcal{D}_E}\varphi_E + \tfrac{\Delta x}{\Delta y}\overline{h_S}\,\overline{\mathcal{D}_S}\varphi_S + \tfrac{\Delta y}{\Delta x}\overline{h_W}\,\overline{\mathcal{D}_W}\varphi_W\\
&\qquad - \big(\tfrac{\Delta x}{\Delta y}\overline{h_N}\,\overline{\mathcal{D}_N} + \tfrac{\Delta y}{\Delta x}\overline{h_E}\,\overline{\mathcal{D}_E} + \tfrac{\Delta x}{\Delta y}\overline{h_S}\,\overline{\mathcal{D}_S} + \tfrac{\Delta y}{\Delta x}\overline{h_W}\,\overline{\mathcal{D}_W}\big)\varphi_C\\
&\qquad - \tfrac{\Delta x}{\Delta y}\overline{\mathcal{N}_N}\psi_N - \tfrac{\Delta y}{\Delta x}\overline{\mathcal{N}_E}\psi_E - \tfrac{\Delta x}{\Delta y}\overline{\mathcal{N}_S}\psi_S - \tfrac{\Delta y}{\Delta x}\overline{\mathcal{N}_W}\psi_W\\
&\qquad + \big(\tfrac{\Delta x}{\Delta y}\overline{\mathcal{N}_N} + \tfrac{\Delta y}{\Delta x}\overline{\mathcal{N}_E} + \tfrac{\Delta x}{\Delta y}\overline{\mathcal{N}_S} + \tfrac{\Delta y}{\Delta x}\overline{\mathcal{N}_W}\big)\psi_C + \Delta x\Delta y\mathcal{M}_C\psi_C = 0, \qquad (2.3.1c)
\end{aligned}$$

The overbar notation is used to indicate the average of two nodes, e.g., $\overline{\zeta_N} = (\zeta_N + \zeta_C)/2$. The system can be put in the form (cf. [28]; page 34):

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\vec{\zeta}\\ \vec{\varphi}\\ \mathbf{0}\end{bmatrix} + \begin{bmatrix} S_{\zeta\zeta} & S_{\zeta\varphi} & S_{\zeta\psi}\\ S_{\varphi\zeta} & S_{\varphi\varphi} & S_{\varphi\psi}\\ S_{\psi\zeta} & S_{\psi\varphi} & S_{\psi\psi}\end{bmatrix}\begin{bmatrix}\vec{\zeta}\\ \vec{\varphi}\\ \vec{\psi}\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \vec{P}_s\\ \mathbf{0}\end{bmatrix}, \qquad (2.3.2)$$

or, equivalently,

$$\frac{\mathrm{d}q}{\mathrm{d}t} = Lq + f, \qquad (2.3.3a)$$

$$S\vec{\psi} = b. \qquad (2.3.3b)$$

The first equation is solved using the Leapfrog integration scheme. The second equation, i.e., the system (2.3.3b), is the system that has to be solved by the solver.

2.4 The system

The problem that has to be solved, is

$$S\psi = b, \qquad (2.4.1)$$

wherein the matrix S is given by the 5-point stencil

$$\begin{bmatrix} 0 & -\frac{\Delta x}{\Delta y}\overline{\mathcal{N}_N} & 0\\ -\frac{\Delta y}{\Delta x}\overline{\mathcal{N}_W} & \frac{\Delta x}{\Delta y}\overline{\mathcal{N}_N} + \frac{\Delta y}{\Delta x}\overline{\mathcal{N}_E} + \Delta x\Delta y\mathcal{M}_C + \frac{\Delta x}{\Delta y}\overline{\mathcal{N}_S} + \frac{\Delta y}{\Delta x}\overline{\mathcal{N}_W} & -\frac{\Delta y}{\Delta x}\overline{\mathcal{N}_E}\\ 0 & -\frac{\Delta x}{\Delta y}\overline{\mathcal{N}_E} & 0\end{bmatrix}. \qquad (2.4.2)$$

The problem has to be solved multiple times per second needed for real-time simulation.

2.5 Properties of the matrix S

In [28]; Appendix F it has been shown that $\mathcal{D}, \mathcal{M}, \mathcal{N} \geq 0$. Also recall that the overbar notation just means taking the average of two neighbouring nodes. Therefore, the center coefficient of the stencil is positive and the other coefficients are negative, and hence the matrix S is diagonally dominant. In fact it can be shown that $\mathcal{M}_C > 0$ and therefore the matrix is strictly diagonally dominant, and thereby an M-matrix.

Next, by taking $\Delta x = k\Delta y$, we see that the center coefficient of the matrix is $\mathcal{O}(1 + h^2)$, whereas the other coefficients are $\mathcal{O}(1)$. Hence for small h the diagonal dominance is not very strong.

Because of the diagonal dominance of the matrix Gershgorin's circle Theorem can be applied to show that the real part of the eigenvalues are strictly positive. Furthermore, by writing out the outer coefficients it is easily found that the matrix is symmetric, so, for all $\lambda \in \sigma(S)$, we have $\lambda \in \mathbb{R}_{>0}$. Thus, summarizing, the matrix S is an SPD matrix.

2.6 Problem size and the real-time issue

Typical mesh sizes for a 2D rectangular grid are $5\,m \times 5\,m$. In a simulation environment, a domain of $20\,km \times 20\,km$ is common. Hence in that case the grid consists of about 16 million nodes. The system $S\psi = b$ that must be solved thus involves a matrix S of 16 million by 16 million! Moreover, the simulator has a frame rate of say $20\,fps$, hence the system $S\psi = b$ must be solved within 0.05 seconds. This example already points out the difficulty of our problem; the main problem is the size in combination with the little time available. A huge amount of computational effort must be performed in almost zero time, and so a massive parallel solver and a tremendous amount of crunch power is required.

To put things in perspective: nowadays with MARIN's current C++ RRB-solver problems on a domain of no more than 200 by 400 nodes can be computed in real time.

Chapter 3

Test problems

3.1 Mathematical problem: Poisson's equation

As a first test problem we shall consider Poisson's equation on a 2D square grid with Dirichlet boundary conditions, i.e.,

$$\begin{aligned} -\Delta u &= f(x,y) \quad \text{on} \quad \Omega = (0,1)\times(0,1), \\ u(x,y) &= 0 \qquad\quad \text{on} \quad \partial\Omega. \end{aligned} \tag{3.1.1}$$

This is a very suitable test problem since the 2D Poisson's problem is very well studied, and our problem is a Poisson-like problem. Hence results and insight obtained using this test system can be applied to our system. However, the rate of convergence for this problem is not representive for the realistic problems.

For the discretization we use a square grid with a total of $(N+1)^2$ equidistant grid cells of size h^2, $h = 1/(N+1)$, see Figure 3.1. The unknows $u_{i,j}$ are defined at the cell vertices of the grid. Since all vertices on the boundary are known the number of unknows equals N^2. Let us introduce the number $n = N^2$ for convenience. By applying finite differences in the x-direction, we obtain a second order accurate discretization,

$$-\left.\frac{\partial^2 u}{\partial x^2}\right|_{x,y} = \frac{-u(x-h,y)+2u(x,y)-u(x+h,y)}{h^2} + \mathcal{O}(h^2). \tag{3.1.2}$$

A similar discretization can be derived for the y-direction. If we let $u_{i,j}$ denote the approximation of $u(x,y)$ in the point (x_i, y_j) with $x_i = ih$, $y_j = jh$ discretization of system (3.1.1) yields, using approximations like (3.1.2), the following system of linear equations:

$$\begin{aligned} 4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} &= h^2 f(x_i, y_j), \\ u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} &= 0 \quad \text{for} \quad i,j \in \{1,2,\ldots,N\}. \end{aligned}$$

Equivalently, in stencil notation for internal points this system reads,

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} u_{i,j} = h^2 f(x_i, y_j).$$

Using a natural ordering (lexicographic) in matrix-vector notation this translates to

$$Au = f$$

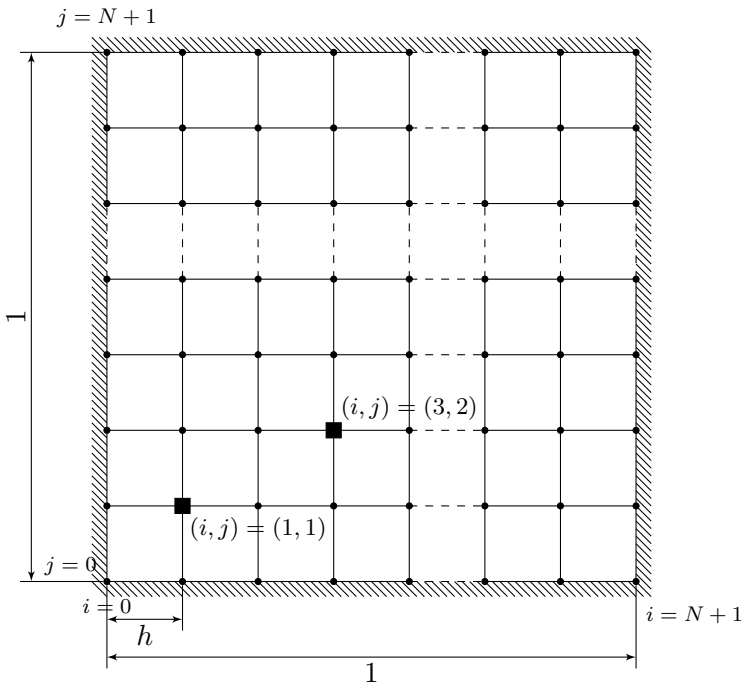


Figure 3.1: The square grid for discretization of Poisson's equation (3.1.1).

with $A \in \mathbb{R}^{n \times n}$ being the block matrix

$$A = \begin{bmatrix} T & -I & 0 & 0 & \cdots & \emptyset \\ -I & T & -I & 0 & \cdots & \\ 0 & -I & T & -I & & \\ 0 & 0 & -I & T & \ddots & \\ & & & \ddots & \ddots & -I \\ \emptyset & & & & -I & T \end{bmatrix},$$

where the I's are $N \times N$ identity matrices and where the T's are tridiagonal $N \times N$ matrices given by

$$T = \begin{bmatrix} 4 & -1 & 0 & 0 & \cdots & \emptyset \\ -1 & 4 & -1 & 0 & \cdots & \\ 0 & -1 & 4 & -1 & & \\ 0 & 0 & -1 & 4 & \ddots & \\ & & & \ddots & \ddots & -1 \\ \emptyset & & & & -1 & 4 \end{bmatrix},$$

where $u = (u_{i,j}) \in \mathbb{R}^n$ is the vector of unknowns, and $f = (h^2 f(x_i, y_j)) \in \mathbb{R}^n$ the right-hand side.

3.2 Small harbour

A representative model with all important phenomena was created to study all kind of wave phenomena, see Figure 3.2. There is a beach, a harbour and a shoal. The overall depth is $30\,m$ except from the cone shaped shoal, radius $125\,m$, with its top lying 2 meter under the water surface. The beach has a width of $300\,m$ and length $200\,m$. The total rectangular domain has size $600\,m \times 1200\,m$ with grid cells of $5\,m \times 5\,m$.

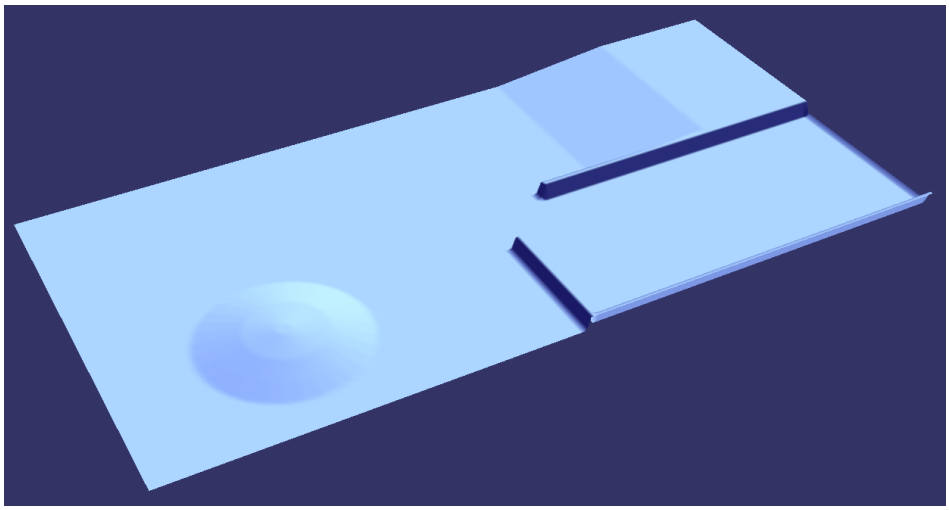


Figure 3.2: Klopman's harbour.

3.3 Realistic problems: IJssel, Plymouth, Port Presto

Three realistic problems are obtained from MARIN's database. We have selected parts from the following regions:

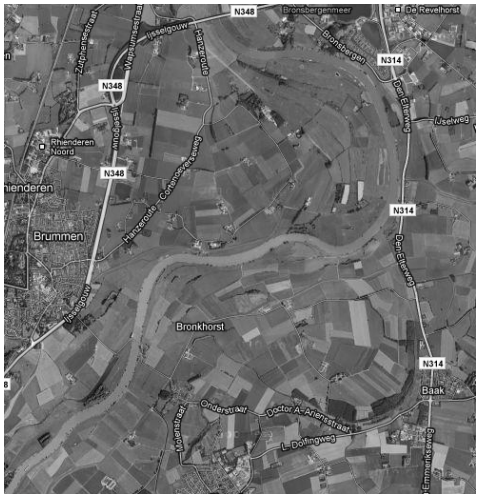
1. The Gelderse IJssel (Netherlands);
2. Plymouth Sound (United Kingdom);
3. Port Presto (fictional; inspired by Barcelona, Spain);

3.3.1 The Gelderse IJssel

The Gelderse IJssel, a small river, is a branch from the Rhine in the Dutch provinces Gelderland and Overijssel. The river flows from Westervoort and discharges in the IJsselmeer. In Figure 3.3 a (small) part of the river is shown. From this part several test problems are extracted. This is done by cropping the displayed region to smaller rectangular shapes, see the table attached to the figure. For the discretization an equidistant $2\,m$ by $2\,m$ grid is used.

3.3.2 Plymouth Sound

Plymouth Sound, locally just The Sound, is a bay located at Plymouth, a town in the South shore region of England, United Kingdom. In the centre of The Sound is Plymouth Breakwater, a dam which protects anchored ships in the Northern part of The Sound against



L_x [m]	L_y [m]	N_x	N_y	n (#nodes)
1,000	400	500	200	100,000
1,600	500	800	250	200,000
2,000	1,000	1,000	500	500,000
3,200	1,250	1,600	625	1,000,000
3,000	2,000	1,500	1,000	1,500,000

Figure 3.3: The Gelderse IJssel and extracted test regions (Google Maps).

south-western storms. From this region also rectangular shaped test problems are extracted, see Figure 3.4. For the discretization an equidistant $5\,m$ by $5\,m$ grid is used.

L_x [m]	L_y [m]	N_x	N_y	n (#nodes)
2,000	1,250	400	250	100,000
2,500	2,000	500	400	200,000
4,000	3,125	800	625	500,000
4,000	6,250	800	1,250	1,000,000
6,000	6,250	1,200	1,250	1,500,000

Figure 3.4: Plymouth Sound and extracted test regions (Google Maps).

3.3.3 Port Presto

Port Presto is a fictional region that shows great similarities with Barcelona, Spain. Is is used frequently as a reference harbour in real-time simulator studies and assesments of mariners. In Figure 3.5 a `OpenSceneGraph` (`OSG`) screenshot is shown as well as a rectangular test problem that is extracted from it. The rectangular box corresponds with the 1,500,000 nodes test problem. For the discretization an equidistant $5\,m$ by $5\,m$ grid is used.

L_x [m]	L_y [m]	$Nx1$	$Nx2$	n (#nodes)
2,000	1,250	400	250	100,000
2,500	2,000	500	400	200,000
4,000	3,125	800	625	500,000
5,000	5,000	1,000	1,000	1,000,000
6,000	6,250	1,200	1,250	1,500,000

Figure 3.5: Port Presto and extracted test regions (`OpenSceneGraph`).

Chapter 4

Test systems

Below is an overview of the machines that will be used to do our experiments. The two machines are equipped with different CPUs and GPUs. System I is a system with hardware a couple of years old. System II is a system equipped with the latest and pretty much the best hardware available today. The different machines are used to make sure that our `CUDA` solvers run on GPUs with different architectures.

4.1 System I: GTX 285

A somewhat older machine equipped with the — back in the days — very popular GeForce GTX 285.

Brand / Type	Dell Precision Workstation T3400
Owner / System no.	MARIN LIN0143
CPU	Intel Core 2 Duo E6850 @ 3.00 GHz
No. of cores	2
Cache	64 kB L1 / 4 MB L2
Memory	4 GB (4 × 1 GB) RAM DDR2 @ 667 MHz
Motherboard	Dell Custom
Operating System	Ubuntu 10.04.3 LTS
System kernel	2.6.32-27-generic (x86_64)
CUDA release	3.2
Driver version	260.19.26
GCC version	4.4.3
GPU (CUDA + screen)	Asus NVIDIA GeForce GTX 285
Memory	1024 MB
No. of cores	30 SM × 8 (cores/SM) = 240 cores
Compute capability	1.3

4.2 System II: GTX 580

A state-of-the-art machine equipped with the ultimate GeForce GTX 580 graphics processing unit, and about the fastest CPU: the Xeon W3520. The Xeon W3520 is comparable to the i7 920 processor, or even slightly better. We belief that using this machine we get honest speed up evaluations for CPU versus GPU code. If we were to compare the CPU from System I, the C2D E6850, with the GPU in this system, we would argue that the comparisons are dishonest as the CPU is quite old and the GPU is very modern.

Brand / Type	Dell Precision Workstation T3500
Owner / System no.	MARIN LIN0169
CPU	Intel Xeon W3520 @ 2.67 GHz
No. of cores	4
Cache	256 kB L1 / 1 MB L2 / 8 MB L3
Memory	6 GB (3 × 2 GB) RAM DDR2 @ 1066 MHz
Motherboard	Dell Custom
Operating System	Ubuntu 10.04.3 LTS
System kernel	2.6.32-34-generic (x86_64)
CUDA release	4.0
Driver version	270.41.19
GCC version	4.4.3
GPU 0 (CUDA)	Asus NVIDIA GeForce GTX 580
Memory	1536 MB
No. of cores	16 SM × 32 (cores/SM) = 512 cores
Compute capability	2.0
GPU 1 (screen)	Asus NVIDIA Quadro NVS 295
Memory	256 MB
No. of cores	1 SM × 8 (cores/SM) = 8 cores
Compute capability	1.1

Chapter 5

Design plan

In this chapter we explain how we came up with our new solvers. Among the first steps is to decide in what language we were going to program the solvers, and also what specific solver(s) we were going to implement. Conjugate Gradients (CG), Multigrid (MG), or a maybe a combination? And in case of CG: which preconditioner are we going to use? And in case of MG: which smoother, which coarse grid solver, etcetera?

5.1 `CUDA` rather than `OpenCL`

We have chosen to implement our new solvers in `CUDA`. There are several reasons for this decision. To start with, on the one hand, `OpenCL` (Open Computing Language) is an open standard to write parallel software supporting many platforms including CPUs and GPUs. The framework hs been developed by Apple and includes a programming language based on the `C99` standard. Because of its generality a performance hit is likely expected. It is almost impossible to achieve maximal performance on every different platform. On the other hand, `CUDA` is a parallel computing architecture developed by NVIDIA themselves for the NVIDIA GPUs only. Its very limited range of support is quite a disadvantage of `CUDA`; however, in this way NVIDIA can easily safeguard maximal performance.

In [11] we find a performance comparison of `CUDA` and `OpenCL`. Although the experiments are not very interesting for us, the results and conclusions are. The authors state that both data transfers and kernel execution in `CUDA` is 15% - 60% faster than in `OpenCL`, depending on the specific size and application. For MARIN at this point maximal performance is more important than generalism, and so `CUDA` is preferred over `OpenCL`.

Also, it is said that the `CUDA` programming lanuage is more compact (fewer lines of code) and somewhat easier to comprehend. Especially if you are a `C` programmer (like us) the CUDA runtime API is easier to use than `OpenCL`. If at one point one desires `OpenCL` it is not too difficult to port the software from `CUDA` to `OpenCL`. Last but not least, we have already quite some experience with `CUDA` from which we can greatly benefit.

5.2 Our choice: PCG with the RRB-method, shortly: the RRB-solver

As the matrix S of the system $S\psi = b$ is SPD, and given by a 5-point stencil, thus yielding a very sparse pentadiagonal matrix, the most proficient method to solve this system is the

preconditioned Conjugate Gradient (PCG) method. No question about that. The real question is what preconditioner we should choose. Given the desire of MARIN to be able to solve really large systems in the nearby future (open sea), taking the Multigrid (MG) method as preconditioner seems to be a good option because of its linear complexity and the fact that the MG method offers good parallelization opportunities. In our literature study we chose for the MGCG-solver, i.e., CG with MG as preconditioner.

However, across time we changed our mind and rather chose to port the existing `C++` RRB-solver, i.e., CG preconditioned with the RRB-method, to `CUDA`. There were several reasons to make this switch. Firstly, we found that the existing `C++` RRB-solver did a great job when we ranthe `lin_wacu` software with the new set of larger test problems (IJssel, Plymouth, Port Presto). We found that the RRB-solver was able to solve all the test problems (up to 1.5 million nodes) within 6 or 7 CG-iterations! This number of CG-iterations is considered small in numerical science. So, if we were to design a new MGCG-solver we would have to beat this number, which seems already to be a difficult task.

Secondly, the RRB-solver has great similarities with the MGCG-solver: both contain the "finer-to-coarser" grid level hierarchy. This implies that for both methods we would encounter the same implementation issues such as overhead due to communication and idle threads on coarse grids. So, regarding implementation difficulties, both methods are "equally" difficult to parallelize efficiently.

Thirdly, if we were to design a MGCG-solver we would encounter additional problems. The MGCG-solver would be a difficult solver to tune to our problem because of its great complexity: the number of levels, the smoother, the form of the cycles, etcetera, are all variables that need to be set and chosen properly, fully depending on our specific problem. It may turn out, after already putting much effort in it, that we do not succeed in designing a robust MGCG-solver that can solve the full set of test problems. And then what? We would stand empty-handed.

Finally, the problems that we can solve real-time later on with the new `CUDA` solver are possibly not big enough for the linear complexity of the Multigrid method to really kick in; possibly the RRB-solver outperforms the MGCG-solver for medium sized grids like our test problems (1.5 million nodes).

Therefore, as porting the `C++` to `CUDA` appears to be already difficult enough, we have chosen to port at least a solver that already works properly. Moreover, the parallelization techniques can possibly be used in the future to make an efficient `CUDA` MGCG-solver.

5.3 A second `CUDA` solver: the IPDIAG-solver

A few months earlier Rohit Gupta has been investigating for his Master's project a specific `CUDA` solver for bubbly flow problems [7]: the IP solver with deflation, i.e., CG with the Incomplete Poisson preconditioner and deflation on top of it, shortly the IPDEF-solver. The speed up results were quite promising. Our very first idea was thus to apply Gupta's IPDEF-solver to our test problems and see what it does. However, unfortunately, Gupta's code could not be used for grids with arbitrary sizes (which is mandatory for us), and especially his deflation implementation lacks generalism. Moreover, the code was written according to a low standard and overall structure was missing.

Therefore, we could not run his code and we decided to implement an own version of the IPDEF-solver. Unfortunately, we did not have enough time to complete our flexible

implementation of deflation (we were almost there), and so deflation has been left out of the code. What remains is a CG solver with the IP preconditioner, and diagonal scaling (if enabled), shortly the IPDIAG-solver. Funny enough the IPDIAG-solver is not a very good solver for the basic 2D Poisson problem; however, for our test problems the IPDIAG-solver does a good job, because of the bigger diagonal dominance of these problems.

Another motivation for this solver was that the CUDA IPDIAG-solver is an embarrassingly parallel type solver that can be far more easily divided over a cluster of computers and GPUs than the CUDA RRB-solver. Thus, although the IPDIAG-solver is inferior to the RRB-solver on its own, on multiple GPUs the IPDIAG-solver may be a good alternative. The IPDIAG-solver is thus much more "future-proof".

5.4 Get CUDA and OpenMP to work simultaneously

Before we started with porting the RRB-solver to CUDA we had to fix an issue. For some reason CUDA and OpenMP could not work together. OpenMP is used to parallelize the code around the solver, and CUDA is used to parallelize the solver. With a first basic CUDA solver, plain CG, we tested why OpenMP and CUDA could not be used simultaneously. The cause turned out to be easy but tricky. The constructor phase (allocating memory on the GPU) was done by host thread A while the solve phase (solving $S\psi = b$) was called by say thread B. Thread B could simply not address the allocated memory on the GPU (the pointers to the memory were gone). The solution is to make sure that the same thread, say thread A, calls both the constructor and the solve phase. One option is to use p-threads, or, even easier, to use `#pragma omp critical` to tell the host that only the main thread (which is never cleared) is allowed to call CUDA routines.

Part II

THEORY: LINEAR SOLVERS

Chapter 6

Preliminaries and notation

In this chapter we introduce some basic terminology and results from linear algebra that are needed for the rest of Part II.

6.1 Linear algebra

Definition 6.1.1. *$A \in \mathbb{R}^{n\times n}$ is called* symmetric *if $A = A^T$, where T stands for transposing, thus, if the entries of A are written as $A = (a_{ij})$, then it holds that $a_{ji} = a_{ij}$.*

Definition 6.1.2. *$A \in \mathbb{R}^{n\times n}$ is called* diagonally dominant *if*

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$$

for all i.

Definition 6.1.3. *$A \in \mathbb{R}^{n\times n}$, write $A = (a_{ij})$ is called an M-*matrix *if and only if*

1. *$a_{ii} > 0$ for all $i = 1, 2, \ldots, n$;*
2. *$a_{ij} \leq 0$ for all $i \neq j$, $i, j = 1, 2, \ldots, n$;*
3. *A^{-1} exists, and $A^{-1} \geq 0$.*

Theorem 6.1.4. *If $A \in \mathbb{R}^{n\times n}$, write $A = (a_{ij})$, satisfies*

1. *$a_{ii} > 0$ for all $i = 1, 2, \ldots, n$;*
2. *$a_{ij} \leq 0$ for all $i \neq j$, $i, j = 1, 2, \ldots, n$;*
3. *A is irreducibly diagonal dominant,*

then A is an M-matrix.

Definition 6.1.5. *If there exists a $v \in \mathbb{C}^n, v \neq 0$ such that*

$$Av = \lambda v,$$

where $A \in \mathbb{R}^{n\times n}$, and λ a scalar, then v is called an eigenvector *of A, and λ is the corresponding* eigenvalue.

Lemma 6.1.6. *If v is an eigenvector of A with corresponding eigenvalue λ, then v is also an eigenvector of $P(A)$ with corresponding eigenvalue $P(\lambda)$ for any polynomial P.*

Definition 6.1.7. *The set of all eigenvalues is called the* spectrum *of A, denoted $\sigma(A)$; thus, $\sigma(A) := \{\lambda \mid \lambda \text{ is an eigenvalue of } A\}$.*

Definition 6.1.8. *The* spectral radius*, denoted $\rho(A)$, of a matrix A is given by*

$$\rho(A) = \max_{\lambda \in \sigma(A)} |\lambda|. \tag{6.1.1}$$

Lemma 6.1.9. *The eigenvalues of a symmetric matrix A are real.*

The previous Lemma 6.1.9 implies that the eigenvalues of a symmetric $n \times n$ matrix A can be ordered as

$$\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n, \tag{6.1.2}$$

where eigenvalues with algebraic multiplicity $k > 1$ are listed k times in a row. Note that for a symmetric $n \times n$ matrix A we just have $\rho(A) = \lambda_n$.

Definition 6.1.10. *$A \in \mathbb{R}^{n \times n}$ is called* symmetric positive definite*, abbreviated* SPD*, if A is symmetric, and $x^T Ax > 0$ for all $x \neq 0$.*

Lemma 6.1.11. *$A \in \mathbb{R}^{n \times n}$ is SPD if and only if A is symmetric and all its eigenvalues are positive.*

The previous Lemma 6.1.11 allows the introduction of the so-called spectral condition number:

Definition 6.1.12. *Let $A \in \mathbb{R}^{n \times n}$ be SPD, and order the eigenvalues as done in (6.1.2). Then, the number*

$$\kappa(A) := \frac{\lambda_n}{\lambda_1} \in [1, \infty)$$

is called the spectral condition number*.*

The following result underlies the so-called Cholesky factorization, see Section 7.3.3.

Theorem 6.1.13. *If $A \in \mathbb{R}^{n \times n}$ is an SPD matrix, then there exists a unique $L \in \mathbb{R}^{n \times n}$ with positive diagonal entries such that $A = LL^T$.*

Theorem 6.1.14 (Gershgorin)**.** *Let A be an $n \times n$ matrix, possibly complex-valued, with entries a_{ij}. For $i = 1, 2, \ldots, n$ let*

$$R_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|,$$

and let $D_i = D(a_{ii}, R_i)$ be the closed disc centered at a_{ii} with radius R_i, called a Gershgorin circle*. Then, for all $\lambda \in \sigma(A)$, there exists an i such that $\lambda \in D_i$.*

Definition 6.1.15. *The A-*inner product*, denoted $\langle x, y \rangle_A$ is defined to be $\langle x, y \rangle_A = \langle Ax, y \rangle_2$.*

Chapter 7

Solvers for $Ax = b$: a brief overview

7.1 The system

The goal throughout the upcoming sections is to solve the linear system

$$Ax = b, \tag{7.1.1}$$

where $A \in \mathbb{R}^{n \times n}$ is an SPD matrix (see 6.1.10), $x \in \mathbb{R}^n$ the *vector of unknowns*, and $b \in \mathbb{R}^n$ a given vector, the *right-hand side* (RHS). Although not necessary, we confine ourselves to SPD matrices A which are *sparse*, that is, matrices that are mainly filled with zeros, and having just a few nonzero elements, say $\mathcal{O}(n)$ nonzeros compared to a total of $\mathcal{O}(n^2)$ elements.

We shall very often refer to this system; by writing "linear system (7.1.1)" we mean the system with A specified as above: size $n \times n$, SPD and sparse.

7.2 An overview

Below a figure is given that summarizes what methods there are to solve system (7.1.1).

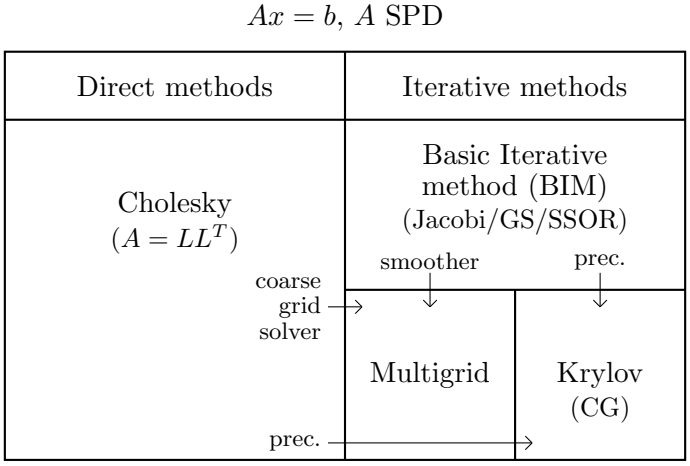


Figure 7.1: Overview of solvers for $Ax = b$, with A an SPD matrix.

7.3 Direct methods

In the next sections we are interested in solving the linear system (7.1.1) by means of a direct method. In contrast to iterative methods, direct methods attempt to solve a problem in a finite number of operations, without occurence of roundoff errors resulting in an exact solution. Although a direct method in itself is not very useful for large systems (too expensive in terms of memory and work) they can be used for, e.g., preconditioning in the Conjugate Gradient method, see Chapter 9, or as exact solver on the coarsest grid in Multigrid, see Chapter 12.

7.3.1 Introduction

For general sparse A (thus not SPD) one typically applies Gaussian elimination to obtain a factorization $A = LU$, whereafter one successively solves the systems $Ly = b$ and $Ux = y$, where L and U respectively are lower and upper triangular matrices, hence the name LU *decomposition*. The systems are easily solved by forward- and backward substitution. For systems in which A is SPD one can apply the *Cholesky factorization* to obtain a factorization,

$$A = GG^T, \tag{7.3.1}$$

where G is a lower triangular matrix known as the *Cholesky factor*. The system $Ax = b$ is then readily solved by first solving the system $Gy = b$ for y, where $y = G^T x$, using forward substition, and then solving the system $G^T x = y$ for x by using backward substition. Cholesky factorization is advantageous for a couple of reasons. Firstly, pivoting (row and/or column interchanges), which ensures numerical stability, is proven to be no longer necessary. Secondly, both memory and work are halved compared to Gaussian elimination. Furthermore, the underlying problem of reordering, which we will describe in a moment, can better be described. For example, for SPD matrices one can predict the locations of nonzero entries during the process, and hence the data structure can be chosen and memory can be reserved before the actual computations are executed.

7.3.2 Occurence of fill-in and reordering

Usually, when applying Cholesky factorization to a sparse matrix A, the matrix suffers so-called *fill-in*, that is, G has nonzeros in positions which are zero in the lower triangular part of A. This effect is best illustrated by an example.

Consider the matrix for the 2D Poisson equation with Dirichlet boundary conditions obtained via finite differences on a square grid with 25 interior grid points and a lexicographic numbering. The structure of this matrix is given below in Figure 7.2. Using the Cholesky algorithm we obtain a factor which structure is shown in Figure 7.2 on the right. As one can see, the Cholesky factor suffers lots of fill-in (gray).

By reordering of the system we hope for a factorization with less fill-in. By setting $\tilde{x} = P^T x$ where P denotes a permutation matrix, solving the system $Ax = b$ for x becomes equivalent with solving the reordered system,

$$\tilde{A}\tilde{x} = \tilde{b} \tag{7.3.2}$$

for $\tilde{x}$, where $\tilde{A} = PAP^T$ and $\tilde{b} = Pb$. In this way the matrix $\tilde{A}$ is again SPD, and hence we can apply the Cholesky decomposition to obtain a factorization, $\tilde{A} = \tilde{G}\tilde{G}^T$. In Figure 7.3 the system is reordered by applying the Reverse Cuthill-McKee method while in Figure 7.4 the

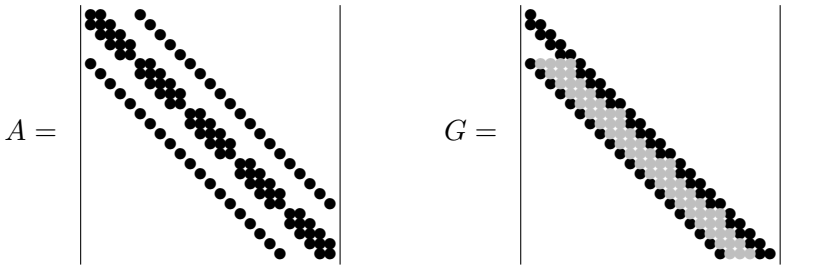


Figure 7.2: Matrix A and its Cholesky factor G for the Poisson equation.

system is reordered by applying the Minimum Degree method. As one can see, in both cases the amount of fill-in is significantly reduced. Note also that Reverse Cuthill-McKee provides for this example the standard diagonal numbering of the grid nodes.

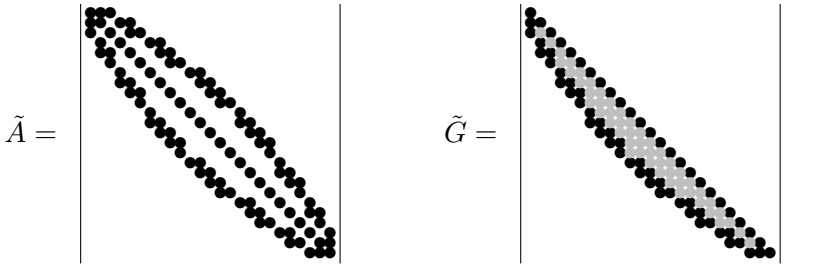


Figure 7.3: Reordered matrix $\tilde{A}$ and its Cholesky factor $\tilde{G}$ for the Poisson equation. Reordering by Reverse Cuthill-Mckee.

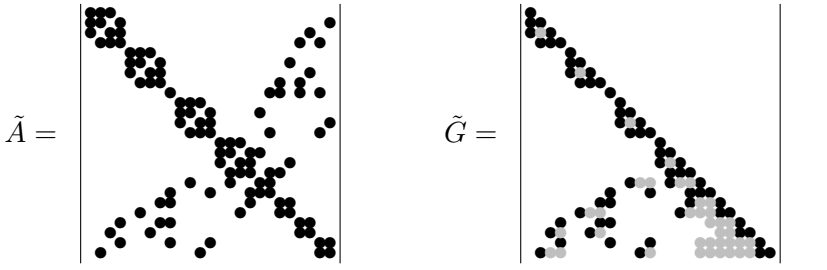


Figure 7.4: Reordered matrix $\tilde{A}$ and its Cholesky factor $\tilde{G}$ for the Poisson equation. Reordering by the Minimum Degree Method.

7.3.3 Cholesky factorization algorithm

Let us derive the *Cholesky-Crout* version of the Cholesky factorization algorithm. This version computes the elements G column by column starting from the upper left corner of G. Consider a 4×4 SPD matrix $A = (a_{ij})$ and write $G = (g_{ij})$, then

$$\begin{bmatrix} a_{11} & \cdot & \cdot & \cdot \\ a_{21} & a_{22} & \cdot & \cdot \\ a_{31} & a_{32} & a_{33} & \cdot \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = A = GG^T = \begin{bmatrix} g_{11} & 0 & 0 & 0 \\ g_{21} & g_{22} & 0 & 0 \\ g_{31} & g_{32} & g_{33} & 0 \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ 0 & g_{22} & g_{32} & g_{42} \\ 0 & 0 & g_{33} & g_{43} \\ 0 & 0 & 0 & g_{44} \end{bmatrix}$$
$$= \begin{bmatrix} g_{11}^2 & \cdot & \cdot & \cdot \\ g_{11}g_{21} & g_{21}^2 + g_{22}^2 & \cdot & \cdot \\ g_{11}g_{31} & g_{21}g_{31} + g_{22}g_{32} & g_{31}^2 + g_{32}^2 + g_{33}^2 & \cdot \\ g_{11}g_{41} & g_{21}g_{41} + g_{22}g_{42} & g_{31}g_{41} + g_{32}g_{42} + g_{33}g_{43} & g_{41}^2 + g_{42}^2 + g_{43}^2 + g_{44}^2 \end{bmatrix},$$

where the dots $(\cdot)$ indicate the symmetric counterparts. We now have to solve a system of equations in the 10 unknows g_{ij}. By working columnwise through G starting from g_{11} we find:

1. $g_{11}^2 = a_{11} \Longrightarrow g_{11} = \sqrt{a_{11}}$,
2. $g_{11}g_{21} = a_{21} \Longrightarrow g_{21} = \dfrac{a_{21}}{g_{11}}$,
   $g_{11}g_{31} = a_{31} \Longrightarrow g_{31} = \dfrac{a_{31}}{g_{11}}$,
   $g_{11}g_{41} = a_{41} \Longrightarrow g_{41} = \dfrac{a_{41}}{g_{11}}$,
3. $g_{21}^2 + g_{22}^2 = a_{22} \Longrightarrow g_{22} = \sqrt{a_{22} - g_{21}^2}$,
4. $g_{21}g_{31} + g_{22}g_{32} = a_{32} \Longrightarrow g_{32} = \dfrac{1}{g_{22}}(a_{32} - g_{21}g_{32})$,
   $g_{21}g_{41} + g_{22}g_{42} = a_{42} \Longrightarrow g_{42} = \dfrac{1}{g_{22}}(a_{42} - g_{21}g_{42})$,
5. $g_{31}^2 + g_{32}^2 + g_{33}^2 = a_{33} \Longrightarrow g_{33} = \sqrt{a_{33} - (g_{31}^2 + g_{32}^2)}$,
6. $g_{31}g_{41} + g_{32}g_{42} + g_{33}g_{43} = a_{43} \Longrightarrow g_{43} = \dfrac{1}{g_{33}}(a_{43} - (g_{31}g_{41} + g_{32}g_{42}))$,
7. $g_{41}^2 + g_{42}^2 + g_{43}^2 + g_{44}^2 = a_{44} \Longrightarrow g_{44} = \sqrt{a_{44} - (g_{41}^2 + g_{42}^2 + g_{43}^2)}$.

From this we see that for general SPD matrices $A \in \mathbb{R}^{n \times n}$ we find G via

$$\begin{aligned} g_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} g_{jk}^2}, \\ g_{ij} &= \frac{1}{g_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} g_{ik}g_{jk} \right), \qquad \text{for } i > j. \end{aligned} \tag{7.3.3}$$

Also note that for $A \in \mathbb{R}^{n\times n}$ one computes the elements in the following order:

$$\begin{bmatrix} 1 & & & & & \\ 2 & 3 & & & & \\ 2 & 4 & 5 & & & \\ \vdots & 4 & 6 & 7 & & \\ \vdots & \vdots & \vdots & & \ddots & \\ 2 & 4 & 6 & 8 & \cdots & 2n-1 \end{bmatrix}$$

Since A is an SPD matrix, the expression under the root is always positive. However, to avoid square roots we may consider a factorization $A = GDG^T$ instead. Further note that matrix A can be overwritten by its Cholesky factor G if the original matrix A is no longer needed. Typical code for the Cholesky Factorization is given in Algorithm 1.

Input: A
Output: G
for $k = 1, 2, \ldots, n$ **do**
  $g_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} a_{kj}}$;
  for $i = k+1, \ldots, n$ **do**
    $g_{ik} = (a_{ik} - \sum_{j=1}^{k-1} a_{ij}a_{kj})/a_{kk}$;
  end
end

Algorithm 1: Cholesky factorization (column version).

The number of flops for this algorithm is $n^3/3$.

7.4 Iterative methods

The iterative methods are discussed in the following chapters.

Chapter 8

Basic Iterative Methods

In this chapter we shall briefly discuss some classical methods, usually called the Basic Iterative Methods (BIMs). Although these methods themselves are of no practical use to us as they will usually take a too large number of iterations to convergen, the BIMs can be used as building blocks for more advanced solvers, e.g., they may serve as preconditioners in the CG method or as smoothers in the MG method.

8.1 Introduction

Suppose we want to solve the linear system (7.1.1) for vector x, the vector of unknowns, where A is a *nonsingular*[1] matrix, and b a given vector, the right-hand side (RHS). By introducing a *splitting* $A = M - N$, where M is a nonsingular matrix, we can formulate the *fixed point iteration*

$$Mx^{k+1} = Nx^k + b, \tag{8.1.1}$$

with x^0 a given start vector. By defining $Q = M^{-1}N$ and $s = M^{-1}b$ we can identify scheme (8.1.1) with the following type of BIMs:

$$x^{k+1} = Qx^k + s, \tag{8.1.2}$$

in which Q is called the *iteration matrix*. The iteration matrix is usually not computed explicitly as it is too expensive or just impossible to do so.

The BIM (8.1.2) can be *damped* by incorporating a *damping factor* ω as follows:

$$x^* = Qx^k + s, \tag{8.1.3a}$$

$$x^{k+1} = \omega x^* + (1 - \omega)x^k. \tag{8.1.3b}$$

Thus actually we take some kind of weighted average between x^k and the update x^*, therefore we frequently see terminology as *weighted scheme* and *weight factor* ω. Also, the words *relaxation parameter* or *smoothing factor*[2] are frequently used for ω, especially in the context of SOR (see below). By eliminating x^* from the equations (8.1.3) we get

$$x^{k+1} = (\omega Q + (1 - \omega)I)x^k + \omega s. \tag{8.1.4}$$

[1] A nonsingular matrix A is a matrix that is not *singular*, that is, for matrix A the inverse, A^{-1}, exists.
[2] This word may lead to confusion, see smoothing in Multigrid.

Damping can be needed to ensure the BIM has the smoothing property, which we shall explain in a later chapter. By optimally choosing ω we can optimize the speed of convergence.

To be able to specify some popular BIMs, suppose that A is decomposed as $A = D+L+U$, where D is a diagonal matrix containing the main diagonal entries of A, L the strictly lower triangular part of A, and U the strictly upper triangular part of A. Then, by choosing M as indicated we get the following methods:

Method	Choice of M
Jacobi	D
Weighted Jacobi	$\frac{1}{\omega}D$
Gauss-Seidel (GS)	$D + L$
Gauss-Seidel backward	$D + U$
SOR (= weighted GS)	$\frac{1}{\omega}(D + \omega L)$
SSOR	$\frac{1}{\omega(2-\omega)}(D + \omega L)D^{-1}(D + \omega U)$

Table 8.1: Popular Basic Iterative Methods.

SOR stands for Successive Overrelaxation, and SSOR stands for Symmetric SOR, which is a version that yields a symmetric iteration matrix Q. SOR follows from Gauss-Seidel (GS) by incorporating damping as discussed above. SSOR follows from a forward SOR sweep followed by a backward SOR sweep. Note that for $\omega = 1$ we get back the GS method. Thanks to this relaxation parameter the SOR method is a major improvement over Jacobi and GS in terms of convergence speed. Next we discuss some of the listed methods in more detail.

8.2 Some popular methods

8.2.1 Jacobi

For the Jacobi method we take

$$M = D, \tag{8.2.1a}$$

$$N = -(L + U). \tag{8.2.1b}$$

This gives us

$$Dx^{k+1} = b - (L + U)x^k, \tag{8.2.2}$$

or,

$$x^{k+1} = Q_{\mathrm{JAC}}x^k + D^{-1}b, \tag{8.2.3}$$

where the iteration matrix Q_{JAC} is given by

$$Q_{\mathrm{JAC}} = -D^{-1}(L + U). \tag{8.2.4}$$

In componentwise form we have

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^k\right). \tag{8.2.5}$$

8.2.2 Gauss-Seidel (GS)

For the GS method we take

$$M = D + L, \tag{8.2.6a}$$
$$N = -U. \tag{8.2.6b}$$

This gives us

$$(D + L)x^{k+1} = b - Ux^k, \tag{8.2.7}$$

or,

$$x^{k+1} = Q_{\mathrm{GS}}x^k + (D + L)^{-1}b, \tag{8.2.8}$$

where the iteration matrix Q_{GS} is given by

$$Q_{\mathrm{GS}} = -(D + L)^{-1}U. \tag{8.2.9}$$

In componentwise form we have

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right). \tag{8.2.10}$$

8.2.3 SOR

For the SOR method we take

$$M = \frac{1}{\omega}(D + \omega L), \tag{8.2.11a}$$
$$N = \frac{1 - \omega}{\omega}D - U. \tag{8.2.11b}$$

This gives us

$$(D + \omega L)x^{k+1} = \omega b + ((1 - \omega)D - \omega U)x^k, \tag{8.2.12}$$

or,

$$x^{k+1} = Q_{\mathrm{SOR}}x^k + (D + \omega L)^{-1}\omega b, \tag{8.2.13}$$

where the iteration matrix Q_{SOR} is given by

$$Q_{\mathrm{SOR}} = (D + \omega L)^{-1}((1 - \omega)D - \omega U). \tag{8.2.14}$$

In componentwise form we have

$$x_i^{k+1} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right) + (1 - \omega)x_i^k. \tag{8.2.15}$$

Note that this last equation can readily be derived from the equation (8.2.10) for the GS method and the damped scheme (8.1.3). Furthermore, for SOR we have $\omega > 1$, hence the name "overrelaxation". "Successive" refers to the fact that an overrelaxation is carried out regularly at every iteration step.

8.3 Some basic results

The *error* e^k is defined to be the difference between the exact solution x^* and the approximation x^k, i.e., $e^k := x^k - x^*$. The *residual* r^k is defined to be the difference between the RHS b and Ax^k, i.e., $r^k := b - Ax^k$. The relationship between the error and residual is as follows:

$$Ae^k = A(x^k - x) = Ax^k - b = -r^k. \tag{8.3.1}$$

By replacing b by $Ax^* = (M - N)x^*$ in the fixed point iteration (8.1.1) we find

$$Mx^{k+1} = Nx^k + (M - N)x^*,$$

or,

$$M(x^{k+1} - x^*) = N(x^k - x^*),$$

hence

$$e^{k+1} = Qe^k, \tag{8.3.2}$$

where $Q = M^{-1}N$ is the iteration matrix.

Lemma 8.3.1. *It holds that* $e^k = Q^k e^0$.

Proof. We use induction to k. The case $k = 1$ is clear, indeed $e^1 = Qe^0$ by (8.3.2). Assume that it holds for $k = i$ (induction hypothesis), i.e., $e^i = Q^i e^0$. Then for $k = i + 1$ we find

$$e^{i+1} = Qe^i \overset{\text{(I.H.)}}{=} QQ^i e^0 = Q^{i+1} e^0,$$

and we are done. □

8.4 Convergence results of BIMs

In this section we discuss some standard convergence results, and explain why the BIMs on themselves are not very good solvers. The section also introduces some terminology which will be used in other sections as well. To start with, let us clarify what we mean with convergence in the context of iterative methods.

Definition 8.4.1. *An iterative method* $\left\{x^k\right\}, k = 0, 1, 2, \ldots$ *is said to be* convergent *if*

$$\lim_{k\to\infty} \|x - x^k\| = 0.$$

If the BIM given by (8.1.2) converges it must be towards the unique solution of (7.1.1). This follows from its construction. For $k \to \infty$ (8.1.2) becomes $x = Qx + s = M^{-1}(Nx + b)$, or, equivalently, $Mx = Nx + b$. With $A = M - N$ this comes down to solving $Ax = b$, the original system (7.1.1).

So, now the interesting question is under what conditions (8.1.2) converges. Let us introduce the following terminology.

Definition 8.4.2. *The* spectral radius*, denoted* $\rho(A)$*, of a matrix* A *is given by*

$$\rho(A) = \max\left\{|\lambda| \ : \lambda \text{ is an eigenvalue of } A\right\}. \tag{8.4.1}$$

Without a proof we state the following important result (for a proof, see [14]).

Theorem 8.4.3. *The BIM (8.1.2) converges for every x^0 if and only if $\rho(Q) < 1$.*

The BIM thus converges if all the eigenvalues of the iteration matrix $Q = M^{-1}N$ in absolute value are smaller than 1. The spectral radius can be thought of as the asymptotic rate of convergence, in other words, at each iterate the error is reduced by a factor $\rho(Q)$, i.e., asymptotically

$$\|x - x^{k+1}\| \leq \rho(Q)\|x - x^k\|.$$

For BIMs the M-matrix property guarantees convergence.

Definition 8.4.4. *Matrix A is called an* M-matrix *if A is nonsingular, $A^{-1} \geq 0$ (elementwise) and $a_{ij} \leq 0$ for all i, j with $i \neq j$.*

Accordingly, we have the following result.

Theorem 8.4.5. *If the system matrix A in (7.1.1) is an M-matrix then Jacobi and GS converge and $\rho(Q_{\mathrm{GS}}) < \rho(Q_{\mathrm{JAC}})$ for $\rho(Q_{\mathrm{JAC}}) \neq 0$.*

GS thus converges faster than Jacobi. This is not surprising as GS always uses the "newest information that is available", and Jacobi does not.

As stated earlier, the BIMs are of little practical use as the rate of convergence is usually very poor. This is motivated by the unfortunate fact that for many problems the spectral radius nears 1 as the grid size becomes smaller, and the closer the spectral radius is to 1, the poorer the speed with which the BIM converges. A standard example is the Poisson equation on a Cartesian grid (unit square) with mesh size h. In that case $\rho(Q_{\mathrm{JAC}}) = \cos \pi h = 1 - (\pi h)^2/2 + \mathcal{O}(h^4)$, and $\rho(Q_{\mathrm{GS}}) = \cos^2 \pi h = 1 - (\pi h)^2 + \mathcal{O}(h^4)$, see [26]; e.g., for $h = 1/64$ we find $\rho(Q_{\mathrm{JAC}}) = 0.998$, and $\rho(Q_{\mathrm{JAC}}) = 0.9976$. With such numbers Jacobi and GS take literally tens of thousands of iterates to approximate the solution with good enough accuracy.

However, by incorporating damping, see equations (8.1.3), the rate of convergence can greatly be improved. For Jacobi we obtain Weighted Jacobi by doing so, and for GS we obtain the SOR method. For the Poisson example above we find with optimal ω, call it ω_{opt}, this time $\rho(Q_{\mathrm{SOR}}) \approx 0.90$ for $h = 1/64$, and hence SOR converges must faster than GS. For the Poisson problem an analytic solution for ω_{opt} can be found; unfortunately, however, for general problems this is not possible and ω_{opt} must be found by experiments.

Although the relaxed methods generally are significantly faster than standard Jacobi and GS their rates of convergence are still not very impressive. The BIMs can be speed up much further by using a Krylov subspace method as accelerator, such as CG. The opposite viewpoint is saying that we use a BIM as preconditioner for the CG method. Then, the BIMs become valuable building blocks.

Chapter 9

The Conjugate Gradient (CG) method

9.1 Derivation of the CG method

For the derivation of the CG method we follow the approach that links solving system (7.1.1) to minimizing a function in n variables [22]. In this respect CG is just a line-search method with a special choice for the search directions (to be explained below). Some results and their proofs come from lecture notes[1] by Wen Shen[2] which are on their turn based on work by Douglas Arnold[2] [2] A same kind of derivation, results and proofs can be found in [6].

Other approaches are to consider CG as a variant of the Arnoldi/Lanczos iteration [20]. The relationship between Lanczos and CG is extensively discussed in [15]. A very first description of CG is published by Hestenes and Stiefel [9].

On our way we shall come along several other algorithms that are related to CG, and, therefore, we shall briefly discuss them too. However, for these algorithms no convergence results are presented; they are merely used to introduce CG in a most natural way. For the CG method itself of course full details are given.

9.1.1 Quadratic form

Solving system (7.1.1) (with A an SPD $n \times n$ matrix) corresponds with minimizing the following *quadratic functional* :

$$x^* = \arg\min_{x \in \mathbb{R}^n} f(x), \quad f(x) = \frac{1}{2} x^T A x - b^T x. \tag{9.1.1}$$

This can be seen as follows. For symmetric A we can compute

$$\nabla f(x) = Ax - b, \tag{9.1.2}$$

and so $\nabla f(x) = 0$ if and only if $Ax = b$. Furthermore, $\nabla^2 f = A$ is SPD. From the conditions of x being a minimum we may conclude that there is an unique minimizer x^* of $f(x)$, and this x^* is the solution to system (7.1.1). To visualize this, for an SPD matrix A the quadratic form $f(x)$ is some kind of paraboloid. By solving $\nabla f(x) = 0$ we thus search for the lowest point of the surface.

[1] http://www.math.psu.edu/shen_w/524/CG_lecture.pdf
[2] Pennsylvania State University

9.1.2 The method of Steepest Descent

The method of Steepest Descent is a line search method. As the name indicates a *line search* method searches for an optimum along a line; therefore, the method performs every iteration two steps: given the current approximate solution x^k,

1. find a direction p^k to move into (the line);
2. find out how far to move into that direction; i.e.
   (a) $\alpha_k = \arg\min f(x^k + \alpha p^k)$;
   (b) set $x^{k+1} = x^k + \alpha_k p^k$.

For the method of Steepest Descent the direction p^k is opposite to the direction in which f increases most quickly, that is, $p^k = -\nabla f(x^k)$, hence a gradient-based method. Note that because of equation (9.1.2) we have

$$-\nabla f(x^k) = b - Ax^k = r^k. \tag{9.1.3}$$

With such search directions the iterative scheme becomes

$$x^{k+1} = x^k + \alpha_k r^k. \tag{9.1.4}$$

To determine the stepsize α_k, we apply Taylor's theorem for multivariable functions[3] to $f(x^{k+1})$ with x^{k+1} given by (9.1.4). This yields[4]

$$f(x^{k+1}) = f(x^k) - \alpha_k (r^k)^T r^k + \frac{\alpha_k^2}{2} (r^k)^T A r^k. \tag{9.1.5}$$

By taking the directional derivative of (9.1.5) to α_k, i.e., $(\mathrm{d}/\mathrm{d}\alpha_k) f(x^{k+1})$, and setting the result to zero, we find

$$\alpha_k = \frac{(r^k)^T r^k}{(r^k)^T A r^k}. \tag{9.1.6}$$

In summary the method of Steepest Descent is defined by the following iterative scheme. Given x^0, for $k = 0, 1, 2, \ldots$ do

$$r^k = b - Ax^k, \tag{9.1.7a}$$

$$\alpha_k = \frac{\langle r^k, r^k \rangle}{\langle r^k, Ar^k \rangle}, \tag{9.1.7b}$$

$$x^{k+1} = x^k + \alpha_k r^k, \tag{9.1.7c}$$

where $\langle \cdot, \cdot \rangle$ is the Euclidean inner product. The computational effort of the method is dominated by matrix-vector products (MVs) which arise in computing the residuals and stepsizes, hence 2 MVs per iteration. Fortunately, one of the MVs can be eliminated (at the cost of

[3] *Multivariable Taylor's Theorem.* For $x, y \in \mathbb{R}^n$ and $t \in \mathbb{R}$ we have

$$f(x + ty) = f(x) + t(\nabla f(x))^T y + \frac{t^2}{2} y^T \nabla^2 f(x) y + \cdots . \qquad (*)$$

Note: $\nabla^2 f(x)$ is called the *Hessian*.

[4] In $(*)$ take $x = x^k$, $t = \alpha_k$, $y = r^k$, and substitute $\nabla f(x^k) = -r^k$, and $\nabla^2 f(x^k) = A$.

accuracy) as follows. By multiplying the equation for x^{k+1} by A and thereafter subtracting b we get

$$Ax^{k+1} - b = Ax^k - b + \alpha_k Ar^k,$$

hence

$$r^{k+1} = r^k - \alpha_k Ar^k. \tag{9.1.8}$$

In this way only Ar^k needs to be computed, we can store the result in say a vector q. Because of roundoff errors it is wise to compute the true residual, that is, using (9.1.3), after a couple of iterations, say *niter*. A slightly different way to safeguard a correct outcome, is to compute the true residual after the termination criterium (see below) is met, and to restart the method with the latest x^k if necessary.

About the memory requirements: Notice that x^k, r^k and α_k can be overwritten by their updates, so that only 1 matrix, A, and three vectors, x, r, q (recall $q = Ar$), need to be stored (and of course some scalars).

Finally, we need a measure to determine when to stop. A common termination criterium is to stop when $\|r^k\| < \epsilon \cdot \|b\|$, with ϵ a predefined tolerance, or when a given number of iterations, *maxiter*, is reached. Based on the preceding hints, the ready-to-implement code for the method of Steepest Descent is given by Algorithm 2.

Input: A, b, x, ϵ, *niter*, *maxiter*
Output: x
1 $r = b - Ax$;
2 $\rho = r^T r$;
3 **for** $k = 1, 2, \ldots, maxiter$ **do**
4 **if** $\rho < \epsilon^2 \|b\|$ **then**
5 break;
6 **end**
7 $q = Ar$; // MV
8 $\sigma = r^T q$; // dot product
9 $\alpha = \frac{\rho}{\sigma}$;
10 $x = x + \alpha r$; // AXPY
11 **if** *niter* | j **then**
12 $r = b - Ax$;
13 **else**
14 $r = r - \alpha q$; // AXPY
15 **end**
16 $\rho = r^T r$; // dot product
17 **end**

Algorithm 2: The method of Steepest Descent.

Because of the consecutive orthogonality of gradients, the method proceeds according to a "zig-zag"-pattern. Typically the method of Steepest Descent does multiple steps in the same direction, as if it adjusts earlier work. It would be much more efficient when the method searches only once in each particular direction, and moves in that direction just right so that

it does not have to come back. For example, in 2D the method would then find the exact solution in at most 2 steps.

9.1.3 The method of Conjugate Directions

Consider the iterative scheme

$$x^{k+1} = x^k + \alpha_k p^k, \tag{9.1.9}$$

where α_k is the stepsize in search direction p^k. To accomplish searching in non-interfering directions we want a new search direction p^k in some sense to be orthogonal to all earlier search directions $\{p^j\}_{j=0}^{k-1}$ so that the method leads us straight away to the solution x (thus without the "zig-zagging"). Therefore, we have to introduce a different form of orthogonality.

Definition 9.1.1. *A set of vectors $\{p^j\}_{j=0}^{k-1}$ are* conjugate *or* A-orthogonal *to an SPD $n \times n$ matrix A if $(p^i)^T A p^j = 0$, for all $i, j = 0, 1, 2, \ldots, k-1$, $i \neq j$.*

The method of Conjugate Directions uses conjugate directions instead of orthogonal directions. We shall show in a moment how the stepsizes α_j are determined and that the method converges in at most n steps towards the solution of system (7.1.1) and, equivalently, finds the minimizer of the quadratic form given by (9.1.1). Therefore, we need the following result.

Theorem 9.1.2. *For an SPD $n \times n$ matrix A the conjugate set $\{p^j\}_{j=0}^{n-1}$ forms an A-orthogonal basis for $\mathbb{R}^n$.*

Proof. We show that the set $\{p^j\}_{j=0}^{n-1}$ is a linearly independent set. Suppose not, thus there exist nonzero constants $a_j, j = 0, 1, 2, \ldots, n-1$, such that

$$a_0 p^0 + a_1 p^1 + \cdots + a_{n-1} p^{n-1} = 0.$$

Multiplying by A and taking the inner product with arbitrary p^i yields

$$a_0 (p^i)^T A p^0 + a_1 (p^i)^T A p^1 + \cdots + a_{n-1} (p^i)^T A p^{n-1} = 0,$$

which by the A-orthogonality, i.e., $(p^i)^T A p^j = 0$ for all $i \neq j$, reduces to $a_i (p^i)^T A p^i = 0$. By the positive definiteness of A we must have $(p^i)^T A p^i > 0$, which implies $a^i = 0$. This gives the desired contradiction. We conclude $\text{span}\{p^0, p^1, \ldots, p^{n-1}\} = \mathbb{R}^n$. □

Next, we introduce the following notation. Let

$$\mathcal{D}_k := \text{span}\{p^0, p^1, \ldots, p^{k-1}\} \quad \text{and} \quad \mathcal{X}_k := x^0 + \mathcal{D}_k.$$

Then, we have the following theorem.

Theorem 9.1.3. *Let $\mathcal{D}_n$ be an A-orthogonal basis for $\mathbb{R}^n$, and consider the sequence $\{x^j\}_{j=0}^{k}$ generated by the iterative scheme (9.1.9). Then $x^j = \arg\min_{x \in \mathcal{X}_j} f(x)$, for all $j = 1, 2, \ldots, k$.*

Proof. We proceed by induction to k. For $k = 1$ the result follows immediately from the definition. Suppose that for some $k = i$ we have (the induction hypothesis):

$$x^j = \arg\min_{x \in \mathcal{X}_j} f(x), \quad \text{for all } j = 1, 2, \ldots, i.$$

We have to show that if $x^{i+1} = x^i + \alpha_i p^i$ then $x^{i+1} = \arg\min_{x \in \mathcal{X}_{i+1}} f(x)$. For $x \in \mathcal{X}_{i+1}$ we can write $x = y + \alpha p^i$, with $y \in \mathcal{X}_i$ and $\alpha \in \mathbb{R}$. Then, using again the multivariable Taylor's Theorem (see footnote 3) we have

$$f(x + \alpha p^i) = f(y) + \alpha (p^i)^T (\nabla f(y)) + \frac{\alpha^2}{2} (p^i)^T A p^i. \tag{9.1.10}$$

Now, although the second term appears to couple minimizations w.r.t. α and y, it is actually not the case since $y - x^i \in \mathcal{D}_i$ and thus $y - x^i$ is A-orthogonal to p^i, hence

$$0 = (p^i)^T A(y - x^i) = (p^i)^T (Ay - b - (Ax^i - b)) \overset{(9.1.2),(9.1.3)}{=} (p^i)^T (\nabla f(y) + r^i),$$

or, equivalently, $(p^i)^T (\nabla f(y)) = -(p^i)^T r^i$. Substituting this result into (9.1.10) allows us to decouple the minimization problem as

$$\min_{x \in \mathcal{X}^{i+1}} f(x) = \min_{y \in \mathcal{X}_i} f(y) + \min_{\alpha \in \mathbb{R}} \big(-\alpha (p^i)^T r^i + \frac{\alpha^2}{2} (p^i)^T A p^i \big).$$

By the induction hypothesis the first term on the RHS is minimized by x^j, and the second term on the RHS is minimized by $\alpha = \alpha_i$ given by

$$\alpha_i = \frac{(p^i)^T r^i}{(p^i)^T A p^i}. \tag{9.1.11}$$

Thus $x^{i+1} = x^i + \alpha_i p^i$ minimizes $f(x)$ over $\mathcal{X}_{i+1}$. □

The theorem shows that the method of Conjugate Directions finds the solution x^* to system (7.1.1) in at most n steps.

Notice that by taking $p_i = r_i$ we get back the method of Steepest Descent.

Putting everything together the method of Conjugate Directions is given by the following iterative scheme. Given a conjugate set $\{p^k\}_{k=0}^{n-1}$, and initial guess x^0, for $k = 0, 1, 2, \ldots, n-1$ do

$$r^k = b - Ax^k, \tag{9.1.12a}$$

$$\alpha_k = \frac{\langle p^k, r^k \rangle}{\langle p^k, Ap^k \rangle}, \tag{9.1.12b}$$

$$x^{k+1} = x^k + \alpha_k p^k. \tag{9.1.12c}$$

In the method of Conjugate Directions we again have to compute 2 MVs per iteration: Ap^k and Ar^k, but in contrast to the method of Steepest Descent none can be eliminated. However, as the method takes at most n iterations, the solution is found in fewer iterations, and the total amount of work is likely smaller.

What lacks in the description of the method of Conjugate Directions is how the set of conjugate directions is generated, and, therefore, presenting a ready-to-implement algorithm is impossible. There are several options to generate a conjugate set of vectors, but we shall only consider that option that leads to the CG method. The CG method uses the residuals, those that are used in the method of Steepest Descent, to generate a conjugate set by applying the so-called Gram-Schmidt Conjugation method to them.

9.1.4 Gram-Schmidt Conjugation method

Suppose we have a set of n linearly independent vectors $u^0, u^1, \ldots, u^{n-1}$. For the moment take for the u^i's the coordinate axes. The Gram-Schmidt Conjugation method is then as follows:

1. Set $p^0 := u^0$;
2. For $k = 1, 2, \ldots, n-1$ compute $p^k = u^k - \sum_{j=0}^{k-1} \beta_{kj} p^j$ with $\beta_{kj} = \dfrac{(p^j)^T A u^k}{(p^j)^T A p^j}$.

To check that the set $\{p^k\}_{k=0}^{n-1}$ is indeed an A-orthogonal set is easily proved by induction.

Note that the method is quite expensive to implement, because it takes $\mathcal{O}(n^2)$ operations (for a full matrix-vector product) to compute one β_{kj}, and the whole set of search directions takes $\mathcal{O}(n^3)$ operations. Moreover, as a new search direction p_k depends on all previous search directions $\{p_j\}_{j=0}^{k-1}$, all search directions need to be stored. Also, due to roundoff errors the search directions may loose A-orthogonality. All together, using the coordinate axes to generate the search direction vectors is obviously not a very good choice.

9.1.5 Conjugate directions that lead to CG

In CG the conjugate directions are not determined beforehand, instead they are generated sequentially as the algorithm progresses. For this the residuals r_k, as they appear in the method of Steepest Descent, are used, for the simple reason that each iteration a new residual is computed, and the residuals work well for the method of Steepest Descent, so why not use them for CG? So, take $u^k = r^k$ in the Gram-Schmidt Conjugation method. This yields $p^0 = r^0$ and for $k > 1$,

$$p^k = r^k - \sum_{j=0}^{k-1} \frac{(p^j)^T A r^k}{(p^j)^T A p^j} p^j. \tag{9.1.13}$$

At first sight this does not improve things; however, we have the following theorem.

Theorem 9.1.4. *Let $\{p^j\}_{j=0}^{k-1}$ be a set of conjugate search directions generated by (9.1.13). Then,*

(i) $\mathcal{D}_k = \text{span}\{r^0, r^1, \ldots, r^{k-1}\}$;

(ii) $(r^i)^T r^j = 0$, *for all* $0 \le j < i \le k$ *(i.e., ℓ_2-orthogonal);*

(iii) $(p^k)^T r^j = (r^k)^T r^k$, *for all* $0 \le j \le k$;

(iv) The search direction p_k satisfies

$$p^k = r^k + \beta_{k-1} p^{k-1}, \quad \textit{with} \quad \beta_{k-1} = \frac{(r^k)^T r^k}{(r^{k-1})^T r^{k-1}}. \tag{9.1.14}$$

Proof. Part (i) follows immediately by construction, i.e., since the search directions are built from the residuals using (9.1.13) with $p^0 = r^0$, the subspace $\mathcal{D}_k = \text{span}\{p^0, p^1, \ldots, p^{k-1}\}$ is equal to $\text{span}\{r^0, r^1, \ldots, r^{k-1}\}$. For (ii) note that for $0 \le j < i \le k$ we have that $x^i + a r^j \in \mathcal{X}_i$,

for any $a \in \mathbb{R}$. From Theorem 9.1.3 we know that x^i is the unique minimizer of $f(x)$ over $\mathcal{X}_i$, i.e., $f(x^i) \leq f(x)$ for all $x \in \mathcal{X}_i$. Thus $g(a) := f(x^i + ar^j)$ on its turn is minimal when $a = 0$. Now, $\mathrm{d}g/\mathrm{d}a = 0$ if and only if

$$0 = \frac{\mathrm{d}f(x^i + ar^j)}{\mathrm{d}a}\bigg|_{a=0} = -(r^j)^T(b - Ax^i),$$

and hence by definition of residual we find $(r_j)^T r^i = 0$. For (iii) we note that $(p^j)^T r^k = 0$ for all $j < k$ because of part (i), and then we find for $j = k$:

$$(p^k)^T r^k \overset{(9.1.13)}{=} (r^k)^T r^k + \sum_{j=0}^{k-1} \frac{(p^j)^T A r^k}{(p^j)^T A p^j} (p^j)^T r_k = (r^k)^T r_k. \qquad (*)$$

For $j < k$ we observe that since $x^k \in \mathcal{X}_k$ and $x^j \in \mathcal{X}_j$ we have that $x^k - x6j \in \mathcal{D}_k$, and therefore, by the A-orthogonality, $0 = (p^k)^T A(x^k - x^j) = (p^k)^T(r^k - r^j)$. Hence $(p^k)^T r^j = (p^k)^T r^k$ for all $j < k$ and thus $(p^k)^T r^j = (r^k)^T r^k$ by (*). For (iv) we observe that the set $\{r^j\}_{j=0}^k$ forms an ℓ_2-orthogonal basis for $\mathcal{D}_{k+1}$ (this follows from the preceding parts). Then $p^k \in \mathcal{D}_{k+1}$ can be written as a linear combination of those vectors. Next we apply (iii) two times, i.e.,

$$\begin{aligned}\mathcal{D}_{k+1} \ni p^k = \sum_{j=0}^{k} \frac{(p^k)^T r^j}{(r^j)^T r^j} r^j &\overset{(iii)}{=} \sum_{j=0}^{k} \frac{(r^k)^T r^k}{(r^j)^T r^j} r^j = r^k + \frac{(r^k)^T r^k}{(r^{k-1})^T r^{k-1}} \sum_{j=0}^{k-1} \frac{(r^{k-1})^T r^{k-1}}{(r^j)^T r^j} r^j \\ &\overset{(iii)}{=} r^k + \frac{(r^k)^T r_k}{(r^{k-1})^T r^{k-1}} \sum_{j=0}^{k-1} \frac{(p^{k-1})^T r^j}{(r^j)^T r^j} r^j = r^k + \beta_{k-1} p^{k-1},\end{aligned}$$

as desired. □

Part (iv) of Theorem 9.1.4 thus allows us to compute the search directions in a very cheap manner, namely by 2 dot products, a division and one AXPY. Now all ingredients for CG are there.

9.1.6 The CG algorithm

Let us put everything together. We start from the iterative scheme of the method of Conjugate Directions, see equations (9.1.12). However, by Theorem 9.1.4, part (iii) we are allowed to replace $(p^k)^T r^k$ by $(r^k)^T r^k$ (this saves computational effort, see below), so that we get the iterative scheme: for $k = 0, 1, 2, \ldots$ do

$$\begin{aligned} r^k &= b - Ax^k, \\ \alpha_k &= \frac{\langle r^k, r^k \rangle}{\langle p^k, Ap^k \rangle}, \\ x^{k+1} &= x^k + \alpha_k p^k. \end{aligned}$$

Now note that we can again save one MV (at the cost of accuracy), just like in the case of the method Steepest Descent, by computing r^{k+1} differently. That is, by multiplying the equation for x^{k+1} by A and thereafter subtracting b we get

$$Ax^{k+1} - b = Ax^k - b + \alpha_k Ap^k,$$

and thus

$$r^{k+1} = r^k - \alpha_k A p^k. \tag{9.1.15}$$

Now the MV Ap^k occurs twice, and therefore we save the outcome in a vector $q = Ap^k$. Next we add formula (9.1.14) to the scheme to come up with the new search directions. So, all together, the CG algorithm is as follows. Given x^0, compute $r^0 = b - Ax^0$ and set $p^0 = r^0$, then for $k = 0, 1, 2, \ldots$ do

$$\alpha_k = \frac{\langle r^k, r^k \rangle}{\langle p^k, Ap^k \rangle} \tag{9.1.16a}$$

$$x^{k+1} = x^k + \alpha_k p^k, \tag{9.1.16b}$$

$$r^{k+1} = r^k - \alpha_k A p^k, \tag{9.1.16c}$$

$$\beta_k = \frac{\langle r^{k+1}, r^{k+1} \rangle}{\langle r^k, r^k \rangle}, \tag{9.1.16d}$$

$$p^{k+1} = r^{k+1} + \beta_k p^k. \tag{9.1.16e}$$

Note that an inner product of the form $\langle r^k, r^k \rangle$ shows up three times (2 times for β_k and 1 time for α_k). Hence by saving the outcome cleverly we actually have to compute the dot product only once. The CG algorithm is frequently slightly rearranged and put in a form like Algorithm 3 (ready-to-implement).

9.2 Storage and computational requirements for CG

9.2.1 Memory

Consider system (7.1.1). Suppose that the SPD $n \times n$ matrix A is a generated by a 5-point stencil, and therefore saved in the DIAG format. For that case, based on Algorithm 3, we have listed the variables that need to be stored and their memory requirements in Table 9.1.

Variable	Type	Description	Memory
A	Matrix	System matrix	$5n$
b	Vector	Right-hand side	n
x	Vector	Solution	n
r	Vector	Residual	n
p	Vector	Search direction	n
q	Vector	$q = Ap$	n
$\epsilon, niter, maxiter$	Scalar	Parameters	3
$\rho_{\text{new}}, \rho_{\text{old}}, \sigma, \alpha, \beta, j$	Scalar	Internal variables	6

Table 9.1: Storage requirements for the CG algorithm.

9.2.2 Flop count

We compute the amount of flops for the case that the matrix is generated by a 5-point stencil. For most CG iterations, according to Algorithm 3 the computational effort consists of one MV, two dot products and three AXPYs. Hence a total of $10n + 3 \cdot 2n + 2 \cdot 2n = 20n$ flops.

Input: A, b, x, ϵ, $niter$, $maxiter$
Output: x

$r = b - Ax$;
$\rho_{\text{new}} = r^T r$;
for $k = 1, 2, \ldots, maxiter$ **do**
    if $\rho_{\text{new}} < \epsilon^2 \|b\|$ **then**
        break;
    end
    if $k = 1$ **then**
        $p = r$;
    else
        $\beta = \dfrac{\rho_{\text{new}}}{\rho_{\text{old}}}$;
        $p = r + \beta p$; `// AXPY`
    end
    $q = Ap$; `// MV`
    $\sigma = p^T q$; `// dot product`
    $\alpha = \dfrac{\rho_{\text{new}}}{\sigma}$;
    $x = x + \alpha p$; `// AXPY`
    if $niter \mid j$ **then**
        $r = b - Ax$;
    else
        $r = r - \alpha q$; `// AXPY`
    end
    $\rho_{\text{old}} = \rho_{\text{new}}$;
    $\rho_{\text{new}} = r^T r$; `// dot product`
end

Algorithm 3: The CG algorithm.

9.3 Convergence analysis of CG

The ultimate goal of this section is to prove the upper bound

$$\|x^k - x\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|x^0 - x\|_A.$$

9.3.1 CG and the Krylov space

Definition 9.3.1. *The subspace* $\mathcal{K}^k(A; r^0) := \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\}$ *is called the* Krylov (sub-)space *of dimension* k *for matrix* A *and residual* r^0.

Let $\mathcal{P}_k$ denote the set of all polynomials of degree k. Then note that $y \in \mathcal{K}^k(A; r^0)$ can be written as $y = P(A)r^0$, with $P \in \mathcal{P}_{k-1}$.

Lemma 9.3.2. *It holds that* $\mathcal{D}_k = \mathcal{K}^k(A; r^0)$.

Proof. We use induction to k. The case $k = 1$ is clear. Assume that it holds for $k = i$ (induction hypothesis), i.e., by Theorem 9.1.4(i), we assume that $\mathcal{D}_i = \text{span}\{r^0, r^1, \ldots, r^{i-1}\} = \mathcal{K}^i(A; r^0)$. To show that it holds for $k = i + 1$ we thus have to show that $r^i \in \mathcal{K}^{i+1}(A; r^0)$. By the induction hypothesis we can write for $r^{i-1}, p^{i-1} \in \mathcal{D}_i$:

$$r^{i-1} = R_{i-1}(A)r^0 \quad \text{and} \quad p^{i-1} = P_{i-1}(A)r^0, \tag{*}$$

where R_{i-1} and P_{i-1} are polynomials of degree at most $i - 1$. Equation (9.4.3c) gives us

$$\begin{aligned} r^i &= r^{i-1} + \alpha_{i-1}Ap^{i-1} \\ &\overset{(*)}{=} R_{i-1}(A)r^0 + \alpha_{i-1}AP_{i-1}(A)r^0 \end{aligned}$$

which is clearly in $\mathcal{K}^{i+1}(A; r^0)$. □

Corollary 9.3.3. *It holds that* $\mathcal{D}_k = \text{span}\{Ae^0, A^2e^0, \ldots, A^ke^0\}$.

Proof. This follows directly from the relationship between the error and the residual, i.e, $Ae^j = -r^j$, see (8.3.1). □

Note that the CG iterates $x^k \in \mathcal{X}_k$ are picked from $x^0 + \mathcal{K}^k(A; r^0)$. Likewise the errors e^k are picked from $e^0 + \mathcal{K}^k(A; r^0)$. Therefore, CG belongs to the class of *Krylov subspace methods*.

9.3.2 CG and optimal polynomials

We saw that CG minimizes the quadratic functional (9.1.1). Now note that

$$\begin{aligned} \|e^k\|_A^2 = (e^k)^T Ae^k &= (x^k - x^*)^T A(x^k - x^*) \\ &= (x^k)^T Ax^k - 2(x^k)^T Ax^* + (x^*)^T Ax^* \\ &= (x^k)^T Ax^k - 2(x^k)^T b + (x^*)^T b \\ &= 2f(x^k) + \text{constant}, \end{aligned}$$

by recalling that $f(x) = \frac{1}{2}x^T Ax - b^T x$. Hence as CG generates an optimal x^k in each iteration, e^k is simultaneously minimized in the A-norm. Actually, some authors derive the CG algorithm by trying to minimize $\|e^k\|_A$ within $e^0 + \mathcal{K}^k(A; r^0)$.

Since $e^k \in e^0 + \mathcal{K}^k(A; r^0)$ the error e^k can be written as

$$e^k = P_k(A)e^0,$$

where

$$P_k(z) = \sum_{j=0}^{k} \gamma_j(k) z^j \quad \text{with} \quad p_k(0) = 1.$$

The CG algorithm finds thus an optimal polynomial $P_k \in \mathcal{P}_k$ with $P_k(0) = 1$, such that $\|e^k\|_A = \|P_k(A)e^0\|_A$ is minimized; i.e.,

$$\|e^k\|_A = \min_{P \in \mathcal{P}_k; P(0)=1} \|P(A)e^0\|_A. \tag{9.3.1}$$

We have the following lemma.

Lemma 9.3.4. *With $\sigma(A)$ being the spectrum of A, we have*

$$\|e^k\|_A \leq \min_{P \in \mathcal{P}_k; P(0)=1} \max_{\lambda \in \sigma(A)} |P(\lambda)| \|e^0\|_A. \tag{9.3.2}$$

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of orthonormal eigenvectors, and $\sigma(A) = \{0 < \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n\}$ be the spectrum of an SPD $n \times n$ matrix A. Write e^0 as a linear combination of the vectors in V, i.e.,

$$e^0 = \sum_{j=1}^{n} a_j v_j.$$

Then

$$\|e^0\|_A^2 = (e^0)^T A e^0 = \sum_{j=1}^{n} a_j^2 \lambda_j,$$

and by Lemma 6.1.6 we have

$$e^k = P(A)e^0 = \sum_{j=1}^{n} a_j P(\lambda_j) v_j \quad \Longrightarrow \quad \|e^k\|_A^2 = \sum_{j=1}^{n} a_j^2 \lambda_j P^2(\lambda_j).$$

Then,

$$\begin{aligned}
\|e^k\|_A^2 &= \min_{P \in \mathcal{P}_k; P(0)=1} \|P(A)e^0\|_A^2 \\
&= \min_{P \in \mathcal{P}_k; P(0)=1} \sum_{j=1}^{n} a_j^2 \lambda_j P^2(\lambda_j) \\
&\leq \min_{P \in \mathcal{P}_k; P(0)=1} \max_{\lambda \in \sigma(A)} P^2(\lambda) \sum_{j=1}^{n} a_j^2 \lambda_j \\
&= \min_{P \in \mathcal{P}_k; P(0)=1} \max_{\lambda \in \sigma(A)} P^2(\lambda) \|e^0\|_A^2.
\end{aligned}$$

By taking the square root we find the desired inequality. □

Rather than minimizing expression (9.3.2) over a finite number of points (the eigenvalues), we shall minimize the expression over the whole interval $[\lambda_1, \lambda_n]$. The polynomial P that does the trick is a so-called Chebyshev polynomial. We shall prove this in a moment. Let us first explain what Chebyshev polynomials generally look like.

9.3.3 Chebyshev polynomials

The Chebyshev polynomials $T_j(z)$ of the first kind are defined by the following recursion

$$T_j(z) = 2zT_{j-1}(z) - T_{j-2}(z), \quad \text{for } j \geq 2, \tag{9.3.3}$$

with $T_1(z) = z$, and $T_0(z) = 1$. The general solution to this recursion formula can be found as follows. For fixed z try solutions of the form $T_j(z) = (r(z))^j$. Substitution in (9.3.3) yields

$$(r(z))^{j-2}((r(z))^2 - 2zr(z) + 1) = 0.$$

Hence the general solution is given by

$$T_j(z) = c_1(r_1(z))^j + c_2(r_2(z))^j,$$

where $r_1(z)$ and $r_2(z)$ are roots of the characteristic equation

$$r^2 - 2zr + 1 = 0.$$

This equation is easily solved, either by the *abc*-formula, or noting that

$$0 = r^2 - 2zr + 1 \Longleftrightarrow (r - z)^2 = z^2 - 1 \Longleftrightarrow r_{1,2} = z \pm \sqrt{z^2 - 1}.$$

Hence

$$T_j(z) = c_1\big(z + \sqrt{z^2 - 1}\big)^j + c_2\big(z - \sqrt{z^2 - 1}\big)^j.$$

Using the boundary condition $T_0(z) = 1$ we find $c_1 + c_2 = 1$, and with the boundary condition $T_1(z) = z$ we find, substituting $c_2 = 1 - c_1$, that $2c_1\sqrt{z^2 - 1} - \sqrt{z^2 - 1} = 0$, so $c_1 = c_2 = \frac{1}{2}$. The Chebyshev polynomials are thus explicitly given by

$$T_j(z) = \frac{1}{2}\big\{\big(z + \sqrt{z^2 - 1}\big)^j + \big(z - \sqrt{z^2 - 1}\big)^j\big\}, \quad \text{for } j \geq 0. \tag{9.3.4}$$

Written out, the first Chebyshev polynomials are

$$\begin{aligned} T_0(z) &= 1, \\ T_1(z) &= z, \\ T_2(z) &= 2z^2 - 1, \\ T_3(z) &= 4z^3 - 3z, \\ T_4(z) &= 8z^4 - 8z^2 + 1. \end{aligned}$$

The Chebyshev polynomials can also be defined trigonometricly. Substitute $z = \cos(\theta)$, $\theta \in [-\pi, \pi]$ (and thus $\theta = \arccos(z)$) into (9.3.4) to get

$$T_j(\cos\theta) = \frac{1}{2}\big((\cos\theta + i\sin\theta)^j + (\cos\theta - i\sin\theta)^j\big).$$

Then applying De Moivre's formula[5] gives $T_j(\cos\theta) = \cos(j\theta)$, and thus

$$T_j(z) = \cos(j \arccos z). \tag{9.3.5}$$

is another definition. Note the following fact.

Proposition 9.3.5. *If $\xi_m = \cos(m\pi/j)$, $m = 0, 1, 2, \ldots, j$ then $T_j(\xi_m) = (-1)^m$.*

Proof. Indeed, for $m = 0, 1, 2, \ldots, j$ we readily compute

$$\begin{aligned} T_j(\xi_m) &= T_j(\cos(m\pi/j)) \\ &= \cos(j \arccos(\cos(m\pi/j))) \\ &= \cos(m\pi) \\ &= (-1)^m, \end{aligned}$$

as desired. □

Expression (9.3.5) shows that $|T_j(z)| \leq 1$ for all $z \in [-1, 1]$, and that they oscillate between 1 and -1, in fact, the higher j the more rapidly they oscillate on the interval $z \in [-1, 1]$. From expression (9.3.4) we see that the Chebyshev polynomials grow very rapidly outside the interval $z \in [-1, 1]$. On the interval $z \in [-1, 1]$ the Chebyshev polynomials have the property that they are smaller than any other polynomial of the same degree, which we shall prove in a moment.

9.3.4 A perfect polynomial

The desired upper bound on the error is found by taking for P the scaled kth degree Chebyshev polynomial T_k, which is given by

$$C_k(\lambda) = \frac{T_k\left(\dfrac{\lambda_n + \lambda_1 - 2\lambda}{\lambda_n - \lambda_1}\right)}{T_k\left(\dfrac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}\right)}. \tag{9.3.6}$$

Remark 9.3.6. *We have*

$$x = \frac{\lambda_n + \lambda_1 - 2\lambda}{\lambda_n - \lambda_1} \quad \Longleftrightarrow \quad \lambda = \frac{\lambda_1 - \lambda_n}{2} x + \frac{\lambda_1 + \lambda}{2}.$$

and hence $x \in [-1, 1] \Longleftrightarrow \lambda \in [\lambda_1, \lambda_n]$

Remark 9.3.7. *We still have $C_k(0) = 1$ thanks to a proper scaling (the denominator). Furthermore, as the numerator is bounded by 1 in absolute value, we have that*

$$\|C_k\|_\infty := \sup_{\lambda \in [\lambda_1, \lambda_n]} |C_k(\lambda)| = \left| C_k\left(\frac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}\right)\right|^{-1}.$$

The next lemma ensures there is no better polynomial than C_k.

[5] *De Moivre's formula.* For any complex number ω and integer n it holds that $(\cos\omega + i\sin\omega)^n = \cos(n\omega) + i\sin(n\omega)$.

Lemma 9.3.8. *For any $P_k \in \mathcal{P}_k$ with $P_k(0) = 1$ we have that*

$$\|C_k\|_\infty \leq \|P_k\|_\infty.$$

Proof. The proof is by contradiction. Set

$$C := \left[C_k\left(\frac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}\right)\right]^{-1} \in \mathbb{R}$$

for convenience. Further, let

$$\mu_m := \frac{\lambda_1 - \lambda_n}{2}\xi_m + \frac{\lambda_1 + \lambda}{2}, \quad \xi_m := \cos\left(\frac{m\pi}{k}\right), \quad m = 0, 1, 2, \ldots, k.$$

The nice property of those μ_m's is that $P_k(\mu_m) = (-1)^m C$, for all $m = 0, 1, 2, \ldots, k$ which we shall use in a moment (this follows immediately from Proposition 9.3.5 and Remark 9.3.6). Also note that $\mu_m \in [\lambda_1, \lambda_n]$. Let us now assume there exists $P_k \in \mathcal{P}_k$ with $P_k(0) = 1$ such that $|P_k(\lambda)| < |C|$ for all $\lambda \in [\lambda_1, \lambda_n]$. In particular this implies that

$$-|C| < P_k(\mu_m) < |C|, \quad m = 0, 1, 2, \ldots, k.$$

If $\text{sign}(C) = 1$ then

$$\begin{aligned} P_k(\mu_m) - C_k(\mu_m) &< 0, \qquad m \text{ even}, \\ P_k(\mu_m) - C_k(\mu_m) &> 0, \qquad m \text{ odd}, \end{aligned}$$

and if $\text{sign}(C) = -1$ then

$$\begin{aligned} P_k(\mu_m) - C_k(\mu_m) &> 0, \qquad m \text{ even}, \\ P_k(\mu_m) - C_k(\mu_m) &< 0, \qquad m \text{ odd}. \end{aligned}$$

So, regardless of the sign of C, the difference $P_k - C_k$ has a zero in every interval (μ_m, μ_{m+1}). Now there are m such intervals, hence m zeros. However, also $P_k(0) - C_k(0) = 0$. Hence the difference $P_k - C_k$ is a polynomial of degree k with at least $k + 1$ zeros, which is impossible, unless $P_k \equiv C_k$ but this leads to a contradiction. □

9.3.5 The upper bound for the error

We first prove the following result seperately.

Proposition 9.3.9. *For $x := \dfrac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}$ we have*

$$C_k(x) \geq \frac{1}{2}\big(x + \sqrt{x^2 - 1}\big)^k.$$

Proof. Note that since $0 < \lambda_1 < \lambda_n$ (as the matrix A is SPD) we have $x > 1$ by construction, which implies $\big(x - \sqrt{x^2 - 1}\big)^k > 1$. Therefore,

$$C_k(x) = \frac{1}{2}\{\big(x + \sqrt{x^2 - 1}\big)^k + \big(x - \sqrt{x^2 - 1}\big)^k\} \geq \frac{1}{2}\big(x + \sqrt{x^2 - 1}\big)^k,$$

and we are done. □

We have now everything in place to prove the main result.

Theorem 9.3.10. *An upper bound of the error is given by*

$$\|x^k - x\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|x^0 - x\|_A \tag{9.3.7}$$

Proof. By combining Lemma 9.3.2 and the optimality as proved in Lemma 9.3.8 of the Chebyshev polynomial $C_k(\lambda)$ which was defined in (9.3.6) we arrive at

$$\|x^k - x\|_A \leq \|C_k\|_\infty \|x^0 - x\|_A.$$

So what remains is to compute $\|C_k\|_\infty$ explicitly. Now this is easy:

$$\begin{aligned}
\|C_k\|_\infty = \left|C_k\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)\right|^{-1} &\leq 2\big(x+\sqrt{x^2-1}\big)^{-k} \quad \text{(by Proposition 9.3.9)} \\
&= 2\bigg(\frac{\big(x-\sqrt{x^2-1}\big)}{\big(x+\sqrt{x^2-1}\big)\big(x-\sqrt{x^2-1}\big)}\bigg)^k \\
&= 2\big(x-\sqrt{x^2-1}\big)^k \\
&= 2\left(\frac{\kappa+1}{\kappa-1} - \sqrt{\left(\frac{\kappa+1}{\kappa-1}\right)^2-1}\right)^k \\
&= 2\left(\frac{\kappa+1}{\kappa-1} - \frac{2\sqrt{\kappa}}{\kappa-1}\right) \\
&= 2\bigg(\frac{(\sqrt{\kappa}-1)^2}{(\sqrt{\kappa}-1)(\sqrt{\kappa}+1)})\bigg)^k \\
&= 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k.
\end{aligned}$$

And with this estimate for $\|C_k\|_\infty$ we find the desired upper bound for the error. □

9.4 Preconditioned Conjugate Gradient (PCG) method

We have seen in Section 9.3 that for solving system (7.1.1), i.e.,

$$Ax = b, \quad A \text{ being an SPD } n \times n \text{ matrix},$$

with CG, the rate of convergence of CG depends on the spectral condition number, $\kappa(A)$, of the matrix A, and the specific distribution of its eigenvalues. The idea behind *preconditioning* is to transform the original linear system into a system that

1. has the same solution or a solution from which the original solution can be easily recovered;

2. involves a matrix that has a more favorable spectrum, i.e., a smaller condition number, and/or, a "better" distribution of the eigenvalues.

For this purpose, consider a nonsingular matrix M. Then, by multiplying the system $Ax = b$ from the left with M^{-1}, we get

$$M^{-1}Ax = M^{-1}b, \tag{9.4.1}$$

which is called a *left preconditioned* system; likewise, by multiplying from the right by M^{-1}, we get a *right preconditioned* system. In order to preserve symmetry, we may consider a matrix M that is SPD also, so that we can factor it as $M = PP^T$, and then we solve

$$P^{-1}AP^{-T}y = P^{-1}b, \quad x = P^{-T}y, \tag{9.4.2}$$

a *central preconditioned* system.

Lemma 9.4.1. *The matrix $P^{-1}AP^{-T}$ is also SPD.*

Proof. For the symmetry: $(P^{-1}AP^{-T})^T = (P^{-T})^T A^T P^{-T} = P^{-1}AP^{-T}$. Further, for $x \neq 0$ we have $\langle x, P^{-1}AP^{-T}x\rangle = \langle P^{-T}x, AP^{-T}x\rangle = \langle y, Ay\rangle > 0$, hence $P^{-1}AP^{-T}$ is also positive definite. □

Although the matrices $M^{-1}A$ and $P^{-1}AP^{-T}$ are generally not the same, they share an important feature.

Lemma 9.4.2. *Let $M = PP^T$ be an SPD matrix. Then, the matrices $M^{-1}A$ and $P^{-1}AP^{-T}$ have the same spectrum.*

Proof. Since $\det(AB) = \det(A)\det(B)$ for nonsingular matrices A and B, we have for arbitrary $\lambda \in \sigma(P^{-1}AP^{-T})$:

$$\begin{aligned}
0 = \det(\lambda I - P^{-1}AP^{-T}) &\iff 0 = \det(P)\det(\lambda I - P^{-1}AP^{-T})\det(P^T) \\
&\iff 0 = \det(\lambda PP^T - A) \\
&\iff 0 = \det(M(\lambda I - M^{-1}A)) \\
&\iff 0 = \det(M)\det(\lambda I - M^{-1}A) \\
&\iff 0 = \det(\lambda I - M^{-1}A),
\end{aligned}$$

hence $\lambda \in \sigma(M^{-1}A)$: the eigenvalues are the same. □

The attendive reader will at this point remark that, in general the matrix $M^{-1}A$ in the left preconditioned system is not SPD, and hence CG cannot be applied. However, this issue is easily solved by using a different inner product than the Euclidean inner product in the CG algorithm. The following observation is key.

Lemma 9.4.3. *The matrix $M^{-1}A$ is self-adjoint in the M-inner product.*

Proof. The M-inner product is given by $\langle x, y\rangle_M := \langle Mx, y\rangle_2 = \langle x, My\rangle_2$ as M is SPD. Then,

$$\langle M^{-1}Ax, y\rangle_M = \langle MM^{-1}Ax, y\rangle_2 = \langle Ax, y\rangle_2 = \langle x, Ay\rangle_2 = \langle x, MM^{-1}Ay\rangle_2 = \langle x, M^{-1}Ay\rangle_M,$$

as desired. □

As pointed out in Section 9.3.2, what CG does is minimizing $\|e^k\|_A = \|x^k - x^*\|_A$. Now note that $\|x^k - x^*\|_A = \langle x^k - x^*, A(x^k - x^*)\rangle = \langle x^k - x^*, M^{-1}A(x^k - x^*)\rangle_M$. Hence, by replacing the Euclidean inner product by the M-inner product in the CG method the left preconditioned system (9.4.1) is solved.

For the left preconditioned system the original residuals $r^k = b - Ax^k$ are mapped to $z^k = M^{-1}r^k$. Accordingly, the new method is given by the following iterative scheme. Given x^0, compute $r^0 = b - Ax^0$, $z^0 = M^{-1}r^0$ and set $p^0 = z^0$, then for $k = 0, 1, 2, \ldots$ do

$$\alpha_k = \frac{\langle z^k, z^k\rangle_M}{\langle p^k, M^{-1}Ap^k\rangle_M} \tag{9.4.3a}$$

$$x^{k+1} = x^k + \alpha_k p^k, \tag{9.4.3b}$$

$$r^{k+1} = r^k - \alpha_k Ap^k, \tag{9.4.3c}$$

$$z^{k+1} = M^{-1}r^{k+1} \tag{9.4.3d}$$

$$\beta_k = \frac{\langle z^{k+1}, z^{k+1}\rangle_M}{\langle z^k, z^k\rangle_M}, \tag{9.4.3e}$$

$$p^{k+1} = r^{k+1} + \beta_k p^k. \tag{9.4.3f}$$

Note that since $\langle z^k, z^k\rangle_M = \langle r^k, z^k\rangle$ and $\langle p^k, M^{-1}Ap^k\rangle_M = \langle p^k, Ap^k\rangle$ the M-inner products do not have to be computed.

For the central preconditioned system (9.4.2) CG can readily be applied as the matrix $P^{-1}AP^{-T}$ is SPD (see Lemma 9.4.1).

Both methods are referred to as the *preconditioned Conjugate Gradient (PCG)* method. The version that uses central preconditioning is given in Algorithm 4 (version I). The version that uses left preconditioning is given in Algorithm 5 (version II). Although the two versions are a little different, by rearranging terms it can be shown that the methods are identical, that is, they generate the same sequences of iterates [20]. It depends on the type of preconditioner (the choice of M) which version of the PCG algorithm is most proficient. In Chapter 10 we discuss how to choose M and what version of PCG is most proficient for various type of preconditioners.

Input: A, P, b, x, ϵ, $niter$, $maxiter$
Output: x

$r = b - Ax$;
$r = P^{-1}r$;
$x = P^{T}x$;
$\rho_{\text{new}} = r^{T}r$;
for $k = 1, 2, \ldots, maxiter$ **do**
  if $\rho_{\text{new}} < \epsilon^2 \|b\|$ **then**
    break;
  end
  if $k = 1$ **then**
    $p = r$;
  else
    $\beta = \dfrac{\rho_{\text{new}}}{\rho_{\text{old}}}$;
    $p = r + \beta p$; // AXPY
  end
  $q = P^{-1}AP^{-T}p$; // MV
  $\sigma = p^{T}q$; // dot product
  $\alpha = \dfrac{\rho_{\text{new}}}{\sigma}$;
  $x = x + \alpha p$; // AXPY
  if $niter \mid j$ **then**
    $r = b - Ax$;
  else
    $r = r - \alpha q$; // AXPY
  end
  $\rho_{\text{old}} = \rho_{\text{new}}$;
  $\rho_{\text{new}} = r^{T}r$; // dot product
end
$x = P^{-T}x$;

Algorithm 4: Version I of the PCG algorithm (central preconditioning).

Input: A, M, b, x, ϵ, $niter$, $maxiter$
Output: x

$r = b - Ax$;
$z = M^{-1}r$;
$\rho_{\text{new}} = r^{T}z$;
for $k = 1, 2, \ldots, maxiter$ **do**
  if $\rho_{\text{new}} < \epsilon^2 \|b\|$ **then**
    break;
  end
  if $k = 1$ **then**
    $p = z$;
  else
    $\beta = \dfrac{\rho_{\text{new}}}{\rho_{\text{old}}}$;
    $p = z + \beta p$; // AXPY
  end
  $q = Ap$; // MV
  $\sigma = p^{T}q$; // dot product
  $\alpha = \dfrac{\rho_{\text{new}}}{\sigma}$;
  $x = x + \alpha p$; // AXPY
  if $niter \mid j$ **then**
    $r = b - Ax$;
  else
    $r = r - \alpha q$; // AXPY
  end
  $z = M^{-1}r$; // solve
  $\rho_{\text{old}} = \rho_{\text{new}}$;
  $\rho_{\text{new}} = r^{T}z$; // dot product
end

Algorithm 5: Version II of the PCG algorithm (left preconditioning).

Chapter 10

Preconditioners

In Section 9.4 the idea behind preconditioning was explained, and the PCG algorithm was derived for left and central preconditioning for a preconditioning matrix $M = PP^T$ but it was not specified yet how to choose this matrix M. So, let us continue from there.

What we want to achieve is that the spectrum of $M^{-1}A$ is more favorable than the spectrum of A, that is, we want the eigenvalues to be clustered around 1. If that were the only requirement, than taking $M = A^{-1}$ would be apparently the best choice because than $M^{-1}A = I$, hence all eigenvalues are 1. However, if we have A^{-1} available then we do not need to apply CG as we can compute $x = A^{-1}b$ right away. Using a preconditioner should ultimately lead to a reduction in work coming from a reduction in required number of iterations for CG to converge. The extra computational effort that comes with incorporating a preconditioner should thus be initiated accordingly and preferably should be as little as possible. By looking at the PCG code, this implies that the system $Mx = r$ should be relatively cheap to solve in case of PCG version II (possibly by factoring it as $M = PP^T$ first) or that $q = P^{-1}AP^{-T}p$ is cheap to compute in case of PCG version I.

10.1 Classical preconditioners

In this section we discuss the classical preconditioners. "Classical" because these preconditioners are based on the classical methods, or, what we call them throughout the report, basic iterative methods (BIMs), see Chapter 8. Recall that to derive the iterative scheme (8.1.2) we introduced a splitting $A = M - N$.

10.1.1 Diagonal scaling

By choosing $M = D$ we get *diagonal scaling*. The preconditioner is also frequently called *Jacobi preconditioner* as it is based on the Jacobi method (see Section 8.2.1). In the literature a common abbreviation for CG with diagonal scaling is SCG (the S comes from "scaling"), and we shall use it too in this report.

The preconditioner generously fulfills the requirement of being cheap to work with. The inverse is namely found by inverting the diagonal elements, i.e., if we write $M = D = \text{diag}(d_1, d_2, \ldots, d_n) = PP^T$, then $P^{-1} = \text{diag}(1/\sqrt{d_1}, 1/\sqrt{d_2}, \ldots, 1/\sqrt{d_n})$. Because P^{-1} is a diagonal matrix the product $\tilde{A} = P^{-1}AP^{-T}$ is cheap to compute. Moreover, the entries on the main diagonal of $\tilde{A}$ are all 1, and hence n multiplications can be saved. So version I of

PCG is here the best choice.

SCG is most effective for diagonally dominant matrices. By applying diagonal scaling to a diagonally dominant matrix, the matrix becomes a normalized variant with 1's on the main diagonal and off diagonal elements that have magnitudes much smaller than 1. By Gershgorin's Theorem we conclude that the eigenvalues become clustered around 1. However, for diagonally dominant matrices with constant diagonals, such as the system matrix that arises from Poisson's problem with constant coefficients, diagonal scaling is not efficient as the condition number does not change[1]. In other words, the more equal the coefficients in the matrix A are, the less efficient diagonal scaling will be.

From the viewpoint of parallel programming diagonal scaling is very promosing as it is embarrasingly parallel.

10.1.2 SSOR

For symmetric A the SSOR preconditioner is given by, see Table 8.1:

$$\begin{aligned} M(\omega) &= \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega L^T) \\ &= \frac{1}{2-\omega}\left(\frac{1}{\omega}D+L\right)\left(\frac{1}{\omega}\right)^{-1}\left(\frac{1}{\omega}+L\right)^T. \end{aligned} \tag{10.1.1}$$

The matrix M can be split as $M = PP^T$ with

$$P = \frac{1}{\sqrt{2-\omega}}\left(\frac{1}{\omega}+L\right)\left(\frac{1}{\omega}D\right)^{-\frac{1}{2}}. \tag{10.1.2}$$

Note that for a pentadiagonal matrix, the factor P is very sparse. Now version II of PCG is applied as follows. The preconditioning step is $Mz = PP^Tz = r$. This system can be solved by first solving $Py = z$ using forward substitution and then $P^Tz = y$ using backward substitution.

On a GPU forward and backward substitution are extremely inefficient operations, and results in a high number of unoccupied threads and a waste of computation power. A common strategy to parallelize these operations is to subdivide the matrices into k independent blocks and solve the uncoupled triangular systems simultaneously. Modification of the matrices lead to an increase in iterations of the CG algorithm, but for a small number of blocks $k \ll n$, n being the dimension of L, this number is not too large. Although this is a reasonable approach for multi-CPU systems, to address all resources of a GPU this number is too small; however, making the number k bigger will degenerate the system, and the preconditioner gets counterproductive.

[1] By assumption we can write $M = dI$, with I the $n \times n$ identity matrix and d some scalar. By Lemma 9.4.2 we find that

$$\det(\lambda I - A) = 0 \quad \Longleftrightarrow \quad \det((\lambda/d)I - M^{-1}A) = 0.$$

Hence if $\lambda \in \sigma(A)$ then $\lambda/d \in \sigma(M^{-1}A) = \sigma(P^{-1}AP^{-T})$. By definition of condition number κ we find

$$\kappa(M^{-1}A) = \frac{\lambda^{M^{-1}A}_{\max}}{\lambda^{M^{-1}A}_{\min}} = \frac{d\lambda^{A}_{\max}}{d\lambda^{A}_{\min}} = \frac{\lambda^{A}_{\max}}{\lambda^{A}_{\min}} = \kappa(A).$$

10.2 Preconditioners based on leaving out fill-in

In this section we discuss several preconditioners that are based on leaving out some (or all) of the fill-in elements that occur when a sparse SPD matrix A is factored as $A = GG^T$. In Section 7.3.2 we illustrated the fill-in phenomenon with the 1D Poisson equation. By approximating the Cholesky factor G by a (much) sparser matrix $K \approx G$, we hope for a overall reduction in computational effort in the PCG algorithm.

10.2.1 Incomplete Cholesky (IC)

The Incomplete Cholesky (IC) decomposition is one of the most popular methods to find such a lower triangular matrix $K \approx G$. Other similar methods are *modified Incomplete Cholesky (MIC)* and *relaxed Incomplete Cholesky (RIC)*. In this section we discuss briefly how the three decompositions can be computed and the relationships between them.

The IC preconditioner is found by taking for $K+K^T$ the same sparsity pattern as A itself, e.g., for a pentadiagonal matrix (arising from a 2D Poisson problem with a lexicographic numbering) with bandwidth $2N-1$, N being the number of grid points in either direction), only the main diagonal (0) and four outer diagonals $(\pm 1, \pm N)$ are kept for $K + K^T$; all other fill-in entries are set to zero. Hence the lower triangular matrix K only has three nonzero diagonals $(0, -1, -N)$. By computing KK^T we observe that two fill-in diagonals occur $(\pm(N-1))$, hence KK^T is given by a 7-point stencil. Since K is sparse and $KK^T \approx A$, the IC decomposition is a good choice as preconditioner in the PCG method.

The IC decomposition is often referred to as IC(0), because of the fact that in K no other diagonals are kept than those that belong to the structure of A itself. By saving m extra fill-in diagonals of G we obtain a matrix K with $m+3$ diagonals in the Poisson case. This decomposition is referred to as the IC(m) decomposition method.

Instead of computing the Cholesky factorization $M = GG^T$ we may instead compute

$$M = (D + L)D^{-1}(D + L)^T, \tag{10.2.1}$$

where L is a strictly lower triangular matrix and D a diagonal matrix. In case of the IC decomposition we thus require that $L_{ij} = 0$ for $i \neq j+1, j+p$. For the 2D Poisson problem we found that KK^T is given by a 7-point stencil. To find an algorithm for computing K, the matrix M is written out for these 7 diagonals. The computations are tedious and can be found in [28]; Appendix H. We skip this part and directly give the results. We want $M \approx A$, so a natural choice is to take L equal to the strictly lower triangular part of A. Hence L in (10.2.1) does not have to be computed. By doing so, it turns out that the main diagonal of M is given by

$$M_{ii} = A^2_{i,i-N} D^{-1}_{i-N,i-N} + A^2_{i,i-1} D^{-1}_{i-1,i-1} + D_{ii}, \tag{10.2.2}$$

and the fill-in entries are given by

$$M_{i,i-N+1} = A_{i,i-N} D^{-1}_{i-N,i-N} A_{i-N,i-N+1}, \tag{10.2.3a}$$

$$M_{i,i+N-1} = A_{i,i-1} D^{-1}_{i-1,i-1} A_{i-1,i+N-1}. \tag{10.2.3b}$$

So to be able to compute the entries of M we need to specify D. One choice is to ignore fill-in and use $M_{ii} = A_{ii}$. This yields

$$D_{ii} = A_{ii} - \frac{A^2_{i,i-N}}{D_{i-N,i-N}} - \frac{A^2_{i,i-1}}{D_{i-1,i-1}}. \tag{10.2.4}$$

This model for choosing L and D in Equation (10.2.1) is called the *Incomplete Cholesky decomposition* (IC)[2]. Another choice is to require equal row sums for A and M, i.e., $\sum_{j=1}^{n} A_{ij} = \sum_{j=1}^{n} M_{ij}$, leading to the recursive formula

$$D_{ii} = A_{ii} - \frac{A_{i,i-N}(A_{i,i-N} + A_{i-N,i-N+1})}{D_{i-N,i-N}} - \frac{A_{i,i-1}(A_{i,i-1} + A_{i-1,i+N-1})}{D_{i-1,i-1}}, \tag{10.2.5}$$

called the *modified Incomplete Cholesky* (MIC). The preceding choices for D can be combined into one scheme by requiring that

$$A_{ii} = M_{ii} + \omega \left(M_{i,i-N+1} + M_{i,i+N-1} \right), \tag{10.2.6}$$

where $0 \leq \omega \leq 1$ a *relaxation parameter*. This leads to

$$D_{ii} = A_{ii} - \frac{A_{i,i-N}(A_{i,i-N} + \omega A_{i-N,i-N+1})}{D_{i-N,i-N}} - \frac{A_{i,i-1}(A_{i,i-1} + \omega A_{i-1,i+N-1})}{D_{i-1,i-1}}, \tag{10.2.7}$$

which is known as the *relaxed Incomplete Cholesky* (RIC). Note that for $\omega = 0$ we get back IC and for $\omega = 1$ we get back MIC.

10.2.2 Repeated Red-Black (RRB)

Now we have discussed the Cholesky factorization and incomplete factorization methods as IC, MIC, and RIC, it is much easier to understand how the RRB-method works and how it can be applied as a preconditioner. However, at this point we shall only demonstrate how the RRB-method works; mathematical details be found in Part IV of this thesis, in the Master thesis by Elwin van 't Wout [28]. His work is partly based on a paper by Brand [4]. A brief but very clear discussion on the RRB-method can also be found in [14]; Section 8.13.

What the RRB-method basically does is making a decomposition

$$A = LDL^T + R, \tag{10.2.8}$$

with L a lower triangular matrix, D a block diagonal matrix, and R a matrix that contains the adjustments made during a so-called lumping procedure. What a lumping procedure basically does is making sure that we keep a 5-point stencil all the time during the factorization, i.e., the outer elements of a 9-point stencil are 'lumped' (read: added) to the other elements of the 5-point stencil.

The Repeated Red-Black method thanks its name to the way grid points of a 2D grid are colored and numbered. Suppose we have a grid of m by n grid points (nodes). The lower left corner is $(1, 1)$, the upper right corner (m, n). All "even" nodes are colored red, i.e, the nodes x_{ij} that satisfy $i + j = 0 \pmod 2$, and all "odd" nodes are colored black, i.e., the nodes with $i + j = 1 \pmod 2$. Then the black nodes are numbered sequentially and the process is repeated on the red nodes (or vice versa, there is no difference). Thus $mn/2$ nodes are numbered right away, and for the other $mn/2$ nodes we repeat the procedure, hence the word "repeated" in Repeated Red-Black.

If the red-black node numbering and black node elimination process is repeated k times we call the method the RRB-k method. Clearly, there is a maximum number of levels: each time the number of nodes is halved, and at some point only 1 node remains. If the total

[2]Confusingly, this is the same term as the Incomplete Cholesky decomposition before.

number of nodes were $N = mn$, and N were of the form $N = 2^\ell$, then the maximum number of levels is ℓ.

Let us see how the RRB-method works. Take the system matrix A corresponding to the 2D Poisson problem with Dirichlet boundary conditions on a square grid of 8×8 internal grid points. With a lexicographic ordering of the grid points the matrix A is pentadiagonal and has size 64×64. We shall compute the sparsity pattern of $L + D + L^T$ if we go all the way down until the final grid contains only 1 node.

Step 1: The 64 nodes are divided in 32 red (white) and 32 black (light-gray) nodes, see Figure 10.1(a). The black nodes are numbered sequentially starting from 1 up to 32. Also, the remaining 32 nodes are divided in two groups each consisting of 16 red nodes. The first group is numbered sequentially starting from 33 up to 48, see the figure.

Step 2: The process is repeated for the remaining 16 red nodes, see Figure 10.1(b). The 16 nodes are again divided into two groups of red and black nodes, having 8 nodes each. Again first the black nodes are numbered sequentially starting from 49 up to 56. Then half of the red nodes is numbered 57 up to 60.

Step 3: The process is again repeated for the remaining 4 nodes, see Figure 10.1(c). We find 2 black nodes with numbers 61 and 62 and one of the 2 red nodes is labeled node 63.

Step 4: Finally, the 1 node that remains in the 4th level, see Figure 10.1(d), is labeled node 64.

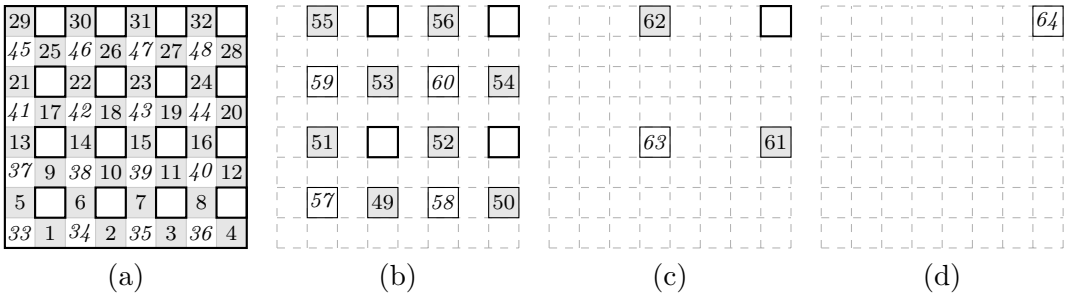


Figure 10.1: Recursive red-black numbering process for an 8×8 grid.

29	55	30	62	31	56	32	64
45	25	46	26	47	27	48	28
21	59	22	53	23	60	24	54
41	17	42	18	43	19	44	20
13	51	14	63	15	52	16	61
37	9	38	10	39	11	40	12
5	57	6	49	7	58	8	50
33	1	34	2	35	3	36	4

Figure 10.2: Complete RRB-numbering for an 8×8 grid.

Putting everything together we find the numbering as given in Figure 10.3 (on the left). The matrix corresponding to this numbering is given in Figure 10.3 (on the right).

29	55	30	62	31	56	32	64
45	25	46	26	47	27	48	28
21	59	22	53	23	60	24	54
41	17	42	18	43	19	44	20
13	51	14	63	15	52	16	61
37	9	38	10	39	11	40	12
5	57	6	49	7	58	8	50
33	1	34	2	35	3	36	4

RRB-numbering

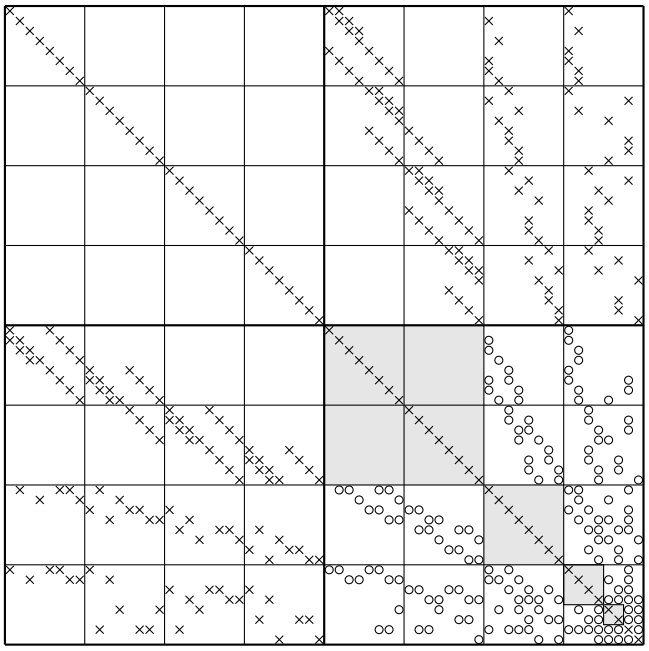


Figure 10.3: *On the left:* Numbering obtained with the RRB-method. *On the right:* Corresponding sparsity pattern of $L+D+L^T$. Entries of the original matrix after a red-black ordering are denoted by little crosses ($\times$), fill-in entries are denoted by circles ($\circ$), and the shaded areas indicate the positions where fill-in has been dropped.

The occurence of fill-in can be explained using graph theory. In the beginning we have a 5-point stencil, see Figure 19.10 (on the left). Elimination of the black nodes leads to a 9-point stencil (the red nodes now only depend on other red nodes), see Figure 19.10 (in the middle). This causes the occurence of fill-in and this is where the lumping procedure is used: the fill-in elements are added to the other elements so that a rotated 5-point stencil remains, see Figure 19.10 (on the right).

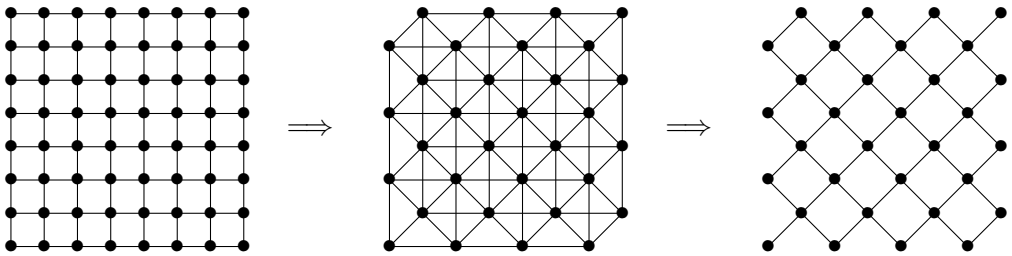


Figure 10.4: Elimination of the black nodes leads to fill-in and a 9-point stencil. A lumping procedure is used to obtain again a (rotated) 5-point stencil.

The RRB-(k)method can be used as preconditioner in the PCG algorithm version II.

During each iteration the preconditioning step $Mz = LDL^T z = r$ has to be solved. This is done in three steps:

1. $La = r$ is solved using forward substitution;
2. $b = D^{-1}a$;
3. $L^T z = b$ is solved using backward substitution.

Because of the sparsity structure of L the RRB-method offers opportunities to parallelize it, e.g., on the first level half of the computations can be done fully in parallel; unfortunately, however, with increasing level number, more and more overhead is introduced and on very coarse levels likely no longer all processors/cores have work to do (idling).

10.2.3 Incomplete Poisson (IP)

The forward and backward substitution steps that occur in the SSOR, IC and RRB preconditioners are inherently sequential and therefore not suitable for a massive parallel architecture such as the GPU. The solution is to find the inverse $M^{-1} \approx A^{-1}$ explicitly and use it directly in version II of the PCG algorithm. For general sparse A this can be achieved with the Sparse Approximate Inverse (SpAI) method. However, for the matrix resulting from discretization of Poisson's problem, there is a more straightforward approach [1] . The preconditioner that is found in this way is the so-called Incomplete Poisson (IP) preconditioner. Recently, Rohit Gupta implemented CG with the IP preconditioner (in combination with deflation) on the GPU and applied the obtained solver to bubbly flow problems [7].

The IP preconditioner is found as follows. Decompose the SPD matrix M as $M = D + L + L^T$, where L is the strictly lower part of M and D its diagonal. Set

$$K := I - LD^{-1}, \quad M^{-1} := KK^T. \tag{10.2.9}$$

Let us now consider the 2D Poisson problem on the unit square with Dirichlet boundary conditions, see the first test problem in Section 3.1. Using an equidistant grid with mesh size $h = 1/(N+1)$, where N is the number of internal grid points in the x- and y-direction, the system matrix A that results from a finite differences approach is given by the stencil

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix},$$

or, equivalenty, the ith row of A is given by

$$\begin{aligned} \text{row}_i(A) &= (a_{i-N}, a_{i-1}, a_i, a_{i+1}, a_{i+N}) \\ &= \frac{1}{h^2}(-1, -1, 4, -1, -1). \end{aligned} \tag{10.2.10}$$

Hence, for L, D^{-1}, and L^T we find

$$\begin{aligned} \text{row}_i(L) &= \frac{1}{h^2}(-1, -1, 0, 0, 0), \\ \text{row}_i(D^{-1}) &= \frac{1}{h^2}(0, 0, \frac{1}{4}, 0, 0), \\ \text{row}_i(L^T) &= \frac{1}{h^2}(0, 0, 0, -1, -1). \end{aligned} \tag{10.2.11}$$

Next, with K defined in (10.2.9) we readily compute

$$\begin{aligned} \text{row}_i(K) &= (\frac{1}{4}, \frac{1}{4}, 1, 0, 0), \\ \text{row}_i(K^T) &= (0, 0, 1, \frac{1}{4}, \frac{1}{4}). \end{aligned} \tag{10.2.12}$$

Note that the $1/h^2$ factor cancelled out. Finally, we find for $M^{-1} = KK^T$ the 7-point stencil

$$\begin{aligned} \text{row}_i(M^{-1}) &= (a_{i-N}, a_{i-N+1}, a_{i-1}, a_i, a_{i+1}, a_{i+N-1}, a_{i+N}) \\ &= (\frac{1}{4}, \frac{1}{16}, \frac{1}{4}, \frac{9}{8}, \frac{1}{4}, \frac{1}{16}, \frac{1}{4}). \end{aligned} \tag{10.2.13}$$

Fill-in has occured at the places a_{i-N+1} and a_{i+N-1}; however, compared to the other elements the numbers are rather small. Therefore, we may set them to zero and keep the 5-point stencil. We thus approximate M^{-1} by a matrix $\tilde{M}^{-1}$ that is given by the incomplete stencil

$$\text{row}_i(\tilde{M}^{-1}) = (\frac{1}{4}, 0, \frac{1}{4}, \frac{9}{8}, \frac{1}{4}, 0, \frac{1}{4}). \tag{10.2.14}$$

Note that $\tilde{M}^{-1}$ is still a symmetric matrix. From now on we will not write the $\sim$'s anymore. The preconditioner M^{-1} given by the stencil (10.2.14) is called the Incomplete Poisson (IP) preconditioner. We have thus explictly computed a matrix M^{-1} that can be used in version II of the PCG algorithm, and leads to a very cheap matrix-vector product.

For the 2D Poisson problem we find for an inner grid point

$$\begin{aligned} \text{row}_i(AM^{-1}) &= (a_{i-2N}, a_{i-N-1}, a_{i-N}, a_{i-N+1}, a_{i-2}, a_{i-1}, a_i, a_{i+1}, a_{i+2}, a_{i+N-1}, a_{i+N}, a_{i+N+1}, a_{i+2N}) \\ &= \frac{1}{h^2}(-\frac{1}{4}, -\frac{1}{2}, -\frac{1}{8}, -\frac{1}{2}, -\frac{1}{4}, -\frac{1}{8}, \frac{7}{2}, -\frac{1}{8}, -\frac{1}{4}, -\frac{1}{2}, -\frac{1}{8}, -\frac{1}{2}, -\frac{1}{4}). \end{aligned} \tag{10.2.15}$$

By multiplying the above stencil with $(2/7)h^2$ we get

$$\frac{2}{7}h^2 \cdot \text{row}_i(AM^{-1}) = (-\frac{1}{14}, -\frac{1}{7}, -\frac{1}{28}, -\frac{1}{7}, -\frac{1}{14}, -\frac{1}{28}, 1, -\frac{1}{28}, -\frac{1}{14}, -\frac{1}{7}, -\frac{1}{28}, -\frac{1}{7}, -\frac{1}{14}), \tag{10.2.16}$$

and we see that the matrix $(2/7)h^2AM^{-1}$ is quite close to the identity matrix. Footnote 1 says that $(2/7)h^2AM^{-1}$ and AM^{-1} have the same condition number. Thereby we can conclude that since the element-wise signed distance of the matrix $(7/2)h^2AM^{-1}$ to the identity matrix is much smaller than the element-wise signed distance of the original matrix A to the identity, the condition number of the former should be significantly smaller.

As the structure of the preconditioner M^{-1} is the same as the structure of the system matrix A, i.e., both are given by a 5-point stencil, and the nonzeros occur in the exact same places, the preconditioning step comes down to a similar SpMV that occurs already elsewhere in the CG algorithm, i.e., the matrix M^{-1} can be stored the same way as A is stored, and $z = M^{-1}r$ can be computed in the exact same way as $q = Ap$. The degree of parallelism is thus $n = N^2$, the number of rows of matrix A. The IP preconditioner is thus a very good candidate for the GPU.

Chapter 11

Deflation

In this chapter we shall discuss briefly the deflation technique. Deflation is some kind of preconditioning. Deflation can be used on top of ordinary preconditioning. Therefore, the term *two-level precondioning* is quite common.

11.1 Introduction

We have seen in Section 9.3 that the rate of convergence of CG depends on the spectral condition number κ; generally, *the larger κ the slower the convergence*, see Inequality (9.3.7). Recall that for a symmetric $n \times n$ matrix A the spectrum of A, $\sigma(A)$, can be ordered as

$$\lambda_{\min} = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n = \lambda_{\max},$$

and then, for an SPD matrix A the spectral condition number κ is defined to be

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

Apparently, κ can be made smaller by making $\lambda_{\min}$ larger, or in some way by "eliminating", say, the first $k < n$ smallest eigenvalues. This is how one may think of deflation in first instance: getting rid of the "bad" (= small) eigenvalues to get a more favorable spectral condition number.

But this is far from mathematically precise. Let us start to clarify things. Firstly, by "eliminating" small eigenvalues we actually mean *making them zero*. This is achieved by multiplying matrix A from the left by some matrix P. The spectrum of the singular matrix PA will then look like

$$\sigma(PA) = \{0, 0, \ldots, 0, \lambda_{k+1}^{PA}, \lambda_{k+2}^{PA}, \ldots, \lambda_n^{PA}\},$$

in which $\lambda_j^{PA} \approx \lambda_j$ for $j = k+1, \ldots, n$. Secondly, for singular systems we cannot compute the ratio $\lambda_{\max}/\lambda_{\min}$ as the smallest eigenvalue equals zero. Instead, for singular systems it turns out that the the *effective condition number* κ_{eff} defined by

$$\kappa_{\text{eff}} = \frac{\lambda_n^{PA}}{\lambda_{k+1}^{PA}},$$

is appropriate, which is thus the ratio of the biggest and smallest *nonzero* eigenvalues. Thirdly, a slightly different system, i.e.,

$$PA\tilde{x} = Pb. \tag{11.1.1}$$

must be solved; however, because of singularity of matrix PA, the solution $\tilde{x}$ is not unique. After remedying this latter issue, by choosing P appriopriately, we shall have $\kappa_{\text{eff}} < \kappa$ and hence CG will likely converge faster.

11.2 The deflation matrix

The matrix P that does the trick is usually defined as follows.

Definition 11.2.1. *Let $A \in \mathbb{R}^{n\times n}$ be an SPSD matrix, and let $Z \in \mathbb{R}^{n\times k}$, rank$(Z) = k$, $k \ll n$ be given. Then,*

1. *$E := Z^T AZ$ is called the* Galerkin matrix*;*
2. *$Q := ZE^{-1}Z^T$ is called the* correction matrix*;*
3. *$P := I - AQ$ is called the* deflation matrix*,*

or, written out,

$$P = I - AZ(Z^T AZ)^{-1}Z^T. \tag{11.2.1}$$

Remark 11.2.2. *$E \in \mathbb{R}^{k\times k}$ and $P, Q \in \mathbb{R}^{n\times n}$.*

The matrix Z in the definition above is called the *deflation subspace matrix* [23] and is chosen such that E^{-1} exists. The matrix Z can be written as $Z = [z_1, z_2, \dots z_k]$, in which the column vectors z_j $(j = 1, 2, \dots, k)$ of Z are called the *deflation vectors*. Later we shall come back how the deflation vectors z_j are constructed.

Usually, for the deflation matrix the letter P is taken, because of the following fact:

Lemma 11.2.3. *The matrix P defined by (11.2.1) is a projection.*

Proof. One must show $P^2 = P$, which follows directly by writing things out. □

11.3 Deflated Preconditioned Conjugate Gradients (DPCG)

The deflated is system (11.1.1) is solved with CG. However, as pointed out earlier, PA is a singular matrix and therefore the solution $\tilde{x}$ is not unique and may not be the solution to the system $Ax = b$. However, it can be shown [23] that the vector $P^T\tilde{x}$ is unique and well-defined, and satisfies $P^T\tilde{x} = P^T x$. For this reason we split x as

$$x = (I - P^T)x + P^T x. \tag{11.3.1}$$

The first term on the RHS can be rewritten as

$$\begin{aligned}(I - P^T)x &= (I - (I - AZ(Z^T AZ)^{-1}Z^T)^T)x \quad \text{(by definition of } P\text{)} \\ &= (I - (I - Z(Z^T AZ)^{-T}Z^T A))x \\ &= Z(Z^T AZ)^{-1}Z^T Ax \quad \text{(by using symmetry of } Z^T AZ\text{)} \\ &= Z(Z^T AZ)^{-1}Z^T b,\end{aligned} \tag{11.3.2}$$

and we see that this part thus does not depend on x. We can thus solve system (11.1.1) for $\tilde{x}$ using CG and then the solution to the original system $Ax = b$ is found via

$$\begin{aligned} x &= Z(Z^TAZ)^{-1}Z^Tb + P^Tb \\ &= \tilde{x} + Z(Z^TAZ)^{-1}Z^T(b - Sx). \end{aligned} \tag{11.3.3}$$

Moreover, because a preconditioner usually improves the rate of convergence of CG, we instead consider the preconditioned system

$$M^{-1}PA\tilde{x} = M^{-1}Pb, \tag{11.3.4}$$

where $M = GG^T \approx A$ is some preconditioner. Then version II of the PCG algorithm, see Algorithm 5 can be applied. This results in the following algorithm called the *deflated preconditioned Conjugate Gradient (DPCG)* algorithm, see Algorithm 6 (ready-to-implement).

Note that M occurs in a different expression than P and A, and therefore the deflation method can be used on top of the PCG algorithm for any kind of preconditioner M.

11.4 Choice of the deflation vectors

So far we did not specify how the k deflation vectors are chosen. A natural choice is to take for the deflation vectors the k eigenvectors that correspond to the k smallest eigenvalues. By doing so we refer to the method as *eigenvalue deflation*. However, computing the eigenvectors in advance costs too much computational effort, cancelling out the profit that we gain with deflation. A cheap method to construct the deflation vectors is so-called *subdomain deflation*.

In this method the domain Ω is divided in k disjoint subdomains Ω_i that cover the complete domain, i.e. $\Omega_i \cap \Omega_j = \varnothing$ for all $i \neq j$, and $\cup_{i=1}^{k} \Omega_i = \Omega$. An example is given in Figure 11.1.

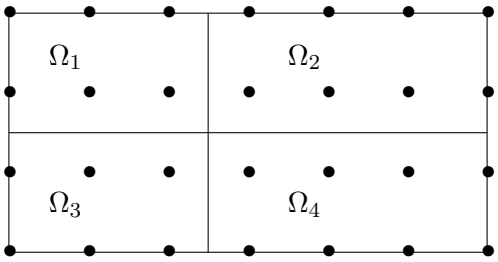


Figure 11.1: Domain divided in four rectangular subdomains.

The subdomain deflation vectors z_i for $i = 1, 2, \ldots, k$ are then defined via

$$(z_i)_j := \begin{cases} 1, & x_j \in \Omega_i, \\ 0, & x_j \in \Omega \setminus \Omega_i, \end{cases} \tag{11.4.1}$$

where x_j are the grid points of the domain.

Input: A, M, P, Z, b, x, ϵ, $niter$, $maxiter$
Output: x

$r = b - Ax$;
$r = Pr$;
$z = M^{-1}r$;
$\rho_{\text{new}} = r^T z$;
for $k = 1, 2, \ldots, maxiter$ **do**
  if $\rho_{\text{new}} < \epsilon^2 \|b\|$ **then**
    break;
  end
  if $k = 1$ **then**
    $p = z$;
  else
    $\beta = \frac{\rho_{\text{new}}}{\rho_{\text{old}}}$;
    $p = z + \beta p$; // AXPY
  end
  $q = PAp$; // MV
  $\sigma = p^T q$; // dot product
  $\alpha = \frac{\rho_{\text{new}}}{\sigma}$;
  $x = x + \alpha p$; // AXPY
  if $niter \mid j$ **then**
    $r = b - PAx$;
  else
    $r = r - \alpha q$; // AXPY
  end
  $z = M^{-1}r$; // solve
  $\rho_{\text{old}} = \rho_{\text{new}}$;
  $\rho_{\text{new}} = r^T z$; // dot product
end
$x = x + Z(Z^T AZ)^{-1} Z^T (b - Ax)$;

Algorithm 6: The deflated PCG algorithm (left preconditioning).

Chapter 12

The Multigrid (MG) method

12.1 Concepts of MG

Although the 1D Poisson's problem can be solved much more efficiently by a special class of solvers, we shall use it to demonstrate the basic principles behind the Multigrid (MG) method. A similar analysis can be done for the 2D Poisson problem, but this does not lead to more insight and the technical details are more involved. As we shall see the MG method is based on two observations, namely

1. Many basic iterative schemes (such as Weighted Jacobi, Gauss-Seidel, SSOR) have the so-called smoothing property, i.e., high frequency modes of the error are rapidly damped, whereas low frequency modes are poorly damped.

2. On a coarser grid the low frequency error modes become more oscillatory, so (by observation 1) low frequency error modes can be effectively reduced by treating them on a coarser grid. This is advantageous because (i) on a coarser grid the computations are cheaper, and (ii) on a coarse grid the convergence rate is marginally better.

The two observations motivate the design of a nested iterative scheme: the MG method. Our analysis is based on the document "Introduction to Multigrid Methods", by Alfio Borzì[1], the lecture sheets "A Multigrid Tutorial", by William L. Briggs[2], and the excellent books by Pieter Wesseling [27] and Gerard Meurant [14]. In the next sections the two concepts are illustrated and explained in more detail.

12.1.1 The smoothing property

Consider the 1D Poisson problem

$$\begin{aligned} -\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} &= f(x) \quad \text{in} \quad \Omega = (0,1), \\ u(0) &= u(1) = 0. \end{aligned} \tag{12.1.1}$$

The domain Ω is divided into $n+1$ equal intervals, so that there are $n+2$ grid points of which n are internal grid points (the unknowns). The grid spacing is $h = 1/(n+1)$, and each grid point is located at jh for $j \in \{0, 1, 2, \ldots, n+1\}$, see Figure 12.1.

[1] www.kfunigraz.ac.at/imawww/borzi/mgintro.pdf
[2] www.math.ust.hk/~mawang/teaching/math532/mgtut.pdf

Figure 12.1: Discretization of domain.

A finite difference approximation yields[3]

$$\begin{aligned} \frac{-v_{j-1}+2v_j-v_{j+1}}{h^2} &= f(x_j), \quad j \in \{1,2,\ldots,n-1\} \\ v_0 = v_{n+1} &= 0. \end{aligned} \tag{12.1.2}$$

where v_j denotes the approximation to the exact solution $u(x_j)$. By defining $v = (v_1, v_2, \ldots, v_n)^T$, and $f = (f_1, f_2, \ldots, f_n)^T = (f(x_1), f(x_2), \ldots, f(x_n))^T$ the above equation can be written as

$$Av = f, \tag{12.1.3}$$

where the $n \times n$ matrix A is given by the stencil $\frac{1}{h^2}[-1 \quad 2 \quad -1]$, i.e.

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}. \tag{12.1.4}$$

Let us first gather some properties of the system.

Lemma 12.1.1. *The eigenvectors ϕ^k and eigenvalues λ^k of the linear system (12.1.3) are given by*

$$\phi^k = (\sin(k\pi jh))_{j=1}^n, \quad \textit{and} \quad \lambda^k = \frac{1}{h^2}(2 - 2\cos(k\pi h)), \quad \textit{for } k = 1, 2, \ldots, n. \tag{12.1.5}$$

Proof. We have to solve $Av = \lambda v$, or, equivalently, $(A - \lambda I)v = 0$, leading to the recurrence relation

$$-v_{j-1} + (2 - \hat{\lambda})v_j - v_{j+1} = 0, \tag{12.1.6}$$

with $\hat{\lambda} = h^2\lambda$, and $v_0 = v_{n+1} = 0$. Substituting $v_j = r^j (r \neq 0)$ yields $-r^{j-1}(r^2 - (2-\hat{\lambda})r + 1) = 0$, and since $r^{j-1} \neq 0$ for all j we have to solve

$$(r^2 + (\hat{\lambda} - 2)r + 1) = 0. \tag{12.1.7}$$

[3] Use the Taylor expansions $u(x \pm h) = u(x) \pm hu'(x) + \frac{1}{2}h^2u''(x) \pm \frac{1}{6}h^3u'''(x) + \mathcal{O}(h^4)$. Then add $u(x+h)$ and $u(x-h)$ and rearrange terms, result:

$$-\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{-u(x-h) + 2u(x) - u(x+h)}{h^2} + \mathcal{O}(h^2).$$

A second-order accurate approximation is then found by neglecting the higher order (h^2) terms.

The discrimant of this quadratic equation is given by $D = (\hat{\lambda} - 2)^2 - 4$, which is a parabola with vertex 2 and zeros 0 and 4. Now observe that for all i the Gershgorin circles D_i of $h^2 A$ are given by (use Theorem 6.1.14): $a_{ii} = 2$ and $R_i = 1 + 1 = 2$, hence $D_i = D(a_{ii}, R_i) = (0, 4)$ for all $i = 1, 2, \ldots, n$. Hence, because all $\hat{\lambda} \in (0, 4)$ the discrimant D is negative, and the roots of (12.1.7) are conjugate complex. Therefore, substitute $r_1 = \exp(i\varphi)$ and $r_2 = \exp(-i\varphi)$ into (12.1.7) yielding $(2 - \hat{\lambda}) = \exp(i\varphi) + \exp(-i\varphi)$, or, equivalently, $\hat{\lambda} = 2 - 2\cos(\varphi)$. The general solution of (12.1.6) is thus $v_j = ar_1^j + br_2^j = a\exp(ij\varphi) + b\exp(-ij\varphi)$. With the boundary condition $v_0 = 0$ we find $a = -b$, which yields $v_j = 2ai\sin(j\varphi)$. Finally, with $v_{n+1} = 0$ we compute: $v_{n+1} = 2ai\sin((n+1)\varphi) = 0$ if and only if $\varphi = k\pi/(n+1) = k\pi h$, $k = 1, 2, \ldots, n$. Let $\phi^k = (v_1, v_2, \ldots, v_n)^T$. Then, by using $\lambda = \hat{\lambda}/h^2$ and $\varphi = k\pi h$, we find[4]

$$\phi^k = (\sin(k\pi jh))_{j=1}^n, \quad \text{and} \quad \lambda^k = \frac{1}{h^2}(2 - 2\cos(k\pi h)), \quad \text{for } k = 1, 2, \ldots, n,$$

as desired. □

Remark 12.1.2. *The eigenvectors ϕ^k, $k = 1, 2, \ldots, n$, are the discrete counterparts of the eigenfunctions $\varphi^k(x)$ of Poisson's equation, for $x \in [0, 1]$ we have $\varphi^k(x) = \sin(k\pi x), k \in \mathbb{N}\backslash\{0\}$.*

We have plotted $\varphi^k(x) = \sin(k\pi x)$ for $k \in \{1, 2, \ldots, 5\}$ and its discrete counterparts $\phi^k = (\sin(k\pi jh))_{j=1}^n$ for $n = 15$ in Figure 12.2.

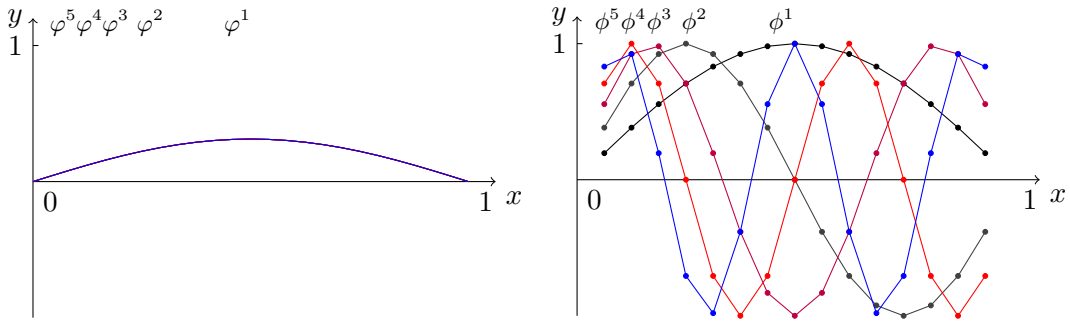


Figure 12.2: First 5 eigenfunctions $\varphi^k(x)$ of the Poisson equation (on the left), and its discrete counterparts for $n = 15$: eigenvectors $\phi^k = (\sin(k\pi jh))_{j=1}^{15}$, $k \in \{1, 2, \ldots, 5\}$ (on the right).

Lemma 12.1.3. *The matrix A given by (12.1.4) is an SPD matrix.*

Proof. Matrix A is symmetric, obviously, and from Lemma 12.1.1 it follows that all eigenvalues of A are positive. Then Lemma 6.1.11 can be applied and we are done. □

We now investigate what happens if we apply the Weighted Jacobi method (see Chapter 8 and Table 8.1) to the system (12.1.3). The iteration matrix of Weighted Jacobi is given by

$$Q_{\omega\text{JAC}} = I - \omega D^{-1} A. \tag{12.1.8}$$

[4]We have quietly used the fact that if v is an eigenvector of A, then also cv is an eigenvector of A, for any $c \in \mathbb{C}\backslash\{0\}$.

Lemma 12.1.4. *The eigenvalues μ^k of $Q_{\omega\text{JAC}}$ are given by*

$$\mu^k(\omega) = 1 - \frac{h^2\omega}{2}\lambda^k, \tag{12.1.9}$$

where λ^k are the eigenvalues of the system matrix A given by (12.1.4), see Lemma 12.1.1. Thus, written out, we have

$$\mu^k(\omega) = 1 - \omega(1 - \cos(k\pi h)) = 1 - 2\omega\sin^2(k\pi h/2). \tag{12.1.10}$$

Moreover, the eigenvectors of $Q_{\omega\text{JAC}}$ and A are the same.

Proof. Note that $D^{-1} = (h^2/2)I$. Then,

$$\begin{aligned} & (A - \lambda^k I)\phi^k = 0 \\ \iff \quad & \phi^k - \omega D^{-1}(A - \lambda^k I)\phi^k = \phi^k \\ \iff \quad & (I - \omega D^{-1}A)\phi^k = (1 - \frac{h^2\omega}{2}\lambda^k)\phi^k. \end{aligned}$$

We see that, if ϕ^k is an eigenvector of A with corresponding eigenvalue λ^k, then ϕ^k is also an eigenvector of $Q_{\omega\text{JAC}}$ with corresponding eigenvalue $\mu^k(\omega)$ given by (12.1.9). □

Remark 12.1.5. *The property that Weighted Jacobi preserves the eigenvectors is the reason that most introductions to MG at first instance analyze the properties of this BIM instead of better ones such as Gauss-Seidel; it makes the analysis much easier and more transparant.*

We can explicitly compute $Q_{\omega\text{JAC}}$ to be

$$Q_{\omega\text{JAC}} = \begin{bmatrix} 1-\omega & \frac{1}{2}\omega & & & & \\ \frac{1}{2}\omega & 1-\omega & \frac{1}{2}\omega & & & \\ & \ddots & \ddots & \ddots & & \\ & & \frac{1}{2}\omega & 1-\omega & \frac{1}{2} & \\ & & & \frac{1}{2}\omega & 1-\omega & \end{bmatrix}. \tag{12.1.11}$$

Recall that for $\omega = 1$ we get back the Jacobi method. From the structure of the iteration matrix, see (12.1.11), we see that in the Jacobi case an iteration is nothing more than taking the average over the neighbouring points. For $0 < \omega < 1$ a weighted average over three grid points is taken.

For $0 < \omega \leq 1$ we have $|\mu^k(\omega)| < 1$ for all $k = 1, 2, \ldots, n$. This is best seen by using expression (12.1.10). Hence $\rho(Q_{\omega\text{JAC}}) < 1$ (by definition of the spectral radius) and it is guaranteed that the method converges. More precisely, we have $\rho(Q_{\omega\text{JAC}}) = \lambda_1 = 1 - \omega(1 - \cos(\pi h)) = 1 - \mathcal{O}(h^2) \approx 1$ for small h. Thus, although the method converges, the convergence is very slowly because of $\rho(Q_{\omega\text{JAC}}) \approx 1$. However, we now analyze Weighted Jacobi in the context of MG, not as a method in itself, and therefore the overall convergence is not what we are primarily interested in; rather we are interested in how different frequencies are damped, or "smoothed".

Therefore, we distinguish (roughly) between

1. *Low frequencies* (LFs): the set of eigenvectors ϕ^k with $1 \leq k < \frac{1}{2}(n+1)$;

2. *High frequencies* (HFs): the set of eigenvectors ϕ^k with $\frac{1}{2}(n+1) \leq k \leq n$.

The error can be expanded in terms of eigenvectors; for an initial error $e^0 = x^0 - x$ we can write

$$e^0 = \sum_{k=1}^{n} \alpha_k \phi^k. \tag{12.1.12}$$

In this respect we say that the error consists of n *error modes*. If we were to apply Weighted Jacobi ν times, we would obtain

$$e^\nu = Q_{\omega\text{JAC}}^\nu e^0 = \sum_{k=1}^{n} \alpha_k Q_{\omega\text{JAC}}^\nu \phi^k = \sum_{k=1}^{n} \alpha_k \mu_k^\nu(\omega) \phi^k.$$

Hence, the kth error mode is reduced by a factor $\mu_k(\omega)$ in every iteration. In Figure 12.3 we have plotted $\mu_k(\omega)$ as function of k (the wavenumber) for several ω's.

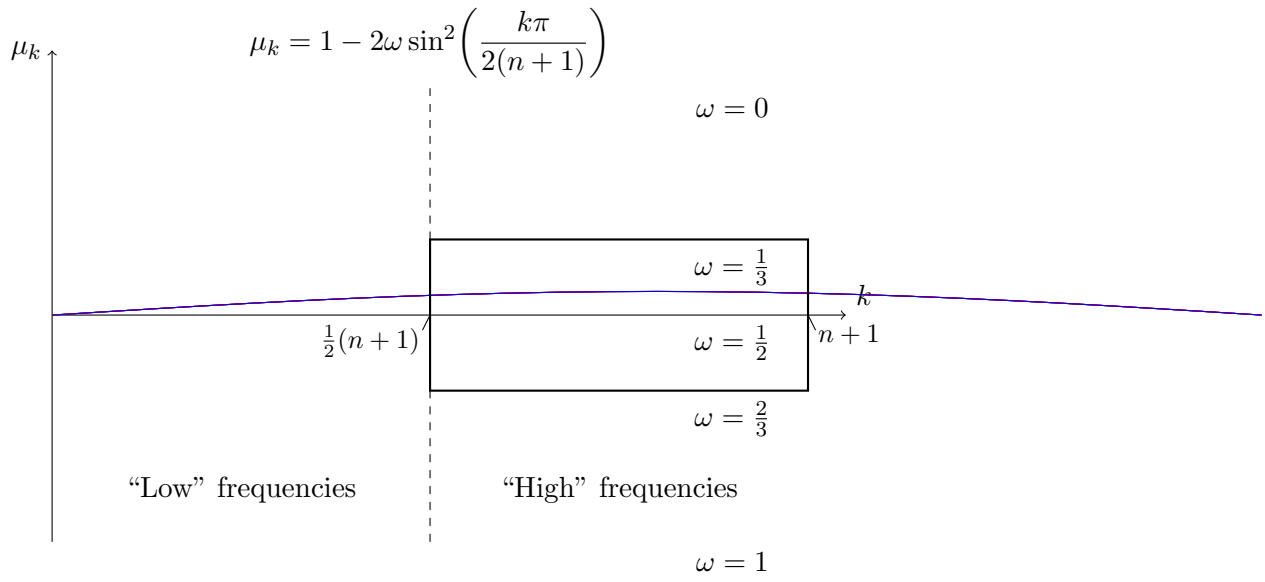


Figure 12.3: Eigenvalues $\mu_k(\omega)$ of Weighted Jacobi for the 1D Poisson problem for different weight factors ω.

From the figure we see that LFs cannot be damped efficiently, regardless the choice of ω (μ_1 is close to 1, always). However, for HFs the choice of ω can apparently make a difference with regard to the spectrum. For that, let us introduce the smoothing factor:

Definition 12.1.6. *The* smoothing factor*, denoted by μ, is the worst factor by which the HFs are damped per iteration for a scheme with iteration matrix Q, thus:*

$$\mu := \max\{|\mu_k| \mid \mu_k \in \sigma(Q), \tfrac{1}{2}(n+1) \leq k \leq n\}.$$

In case of our Weighted Jacobi method we find that the optimal ω is $\omega^* = \frac{2}{3}$ for which $\mu = \frac{1}{3}$. In Figure 12.3 this corresponds to the fat box. So, $\mu_k^\nu \leq \mu^\nu = (\frac{1}{3})^\nu$ for all $\frac{1}{2}(n+1) \leq k \leq n$, and this number becomes very small after just a few iterations.

As a numerical example, we consider system (12.1.1) with $f(x) = 12x(1-x)$ and $n = 15$. By elementary calculus, one readily computes that for this f the analytical solution to system (12.1.1) is given by $u(x) = x^3(x-2) + x$. We apply Weighted Jacobi with $\omega = \omega^* = \frac{2}{3}$ (optimal).

With $n = 15$ there are 15 eigenvectors, or eigenmodes, given by $\phi^k = (\sin(k\pi jh))_{j=1}^{15}, k = 1, 2, \ldots 15$, see Figure 12.2.

To demonstrate the damping, or "smoothing" effect as mentioned earlier, we take for the initial guess x^0 just some random vector, see the black curve (iter 0) in Figure 12.4 (on the right). Due to the highly oscillating nature of the initial error $e^0 = x^0 - x$, both the smooth and oscillatory modes deliver a good part to the error, that is, by writing e^0 like in (12.1.12) all α_k have a significant value, see the black curve (iter 0) in Figure 12.4 (on the left).

In Figure 12.4 the contribution of each error mode (on the left) to the error and the error itself (on the right) is plotted for 0 (initial), 1, 2, 3 and 4 iterations with optimal Weighted Jacobi. We see the oscillatory modes are rapidly damped ($|\alpha_k|$ become zero rapidly for $8 \leq k \leq 15$), whereas the smooth modes are poorly damped, causing the error to become very rapidly a smooth curve, this is the so-called "smoothing effect", but also causing the error slowly to converge towards zero. In Figure 12.5 the approximate solution x^i is plotted for $i = 0$ (initial guess), 1, 2, 3 and 4 (on the left) and for $i = 10, 20, 30, 40, 50, 100$ and 1000 (on the right). We see that the approximate solution very rapidly becomes a smooth curve, but also that it takes several 100 iterations to converge towards the analytical solution due to the poor damping of the smooth error modes.

12.1.2 Exploiting coarse grids

In the previous section we saw how relaxation schemes effectively damp out oscillatory modes, but that smooth modes are hardly influenced at all. At first sight this may seem a limitation; however, we can use the smoothing property to good advantage by dealing with them on coarser grids. A smooth error mode namely appears to be more oscillatory on a coarser grid! Let us first explain what we exactly mean with a "fine" and a "coarse" grid.

For convenience, assume that the number of internal grid points n is odd, e.g., $n = 2^k - 1$ for some positive integer k. With *fine grid* Ω_h we mean the original grid, i.e., for the 1D model Poisson problem,

$$\Omega_h := \{x \in \mathbb{R} \mid x = x_j = jh, h = 1/(n+1), 0 \leq j \leq n+1\},$$

so

$$\Omega_h \ : \ x_0, x_1, x_2, \ldots, x_{n+1}.$$

Set $H = 2h$, hence a two times bigger grid spacing than the fine grid; accordingly, the *coarse grid* Ω_H is given by

$$\Omega_H := \{x \in \mathbb{R} \mid x = x_j = 2jh, h = 1/(n+1), 0 \leq j \leq (n+1)/2\},$$

so

$$\Omega_H \ : \ x_0, x_2, x_4, \ldots, x_{n+1}.$$

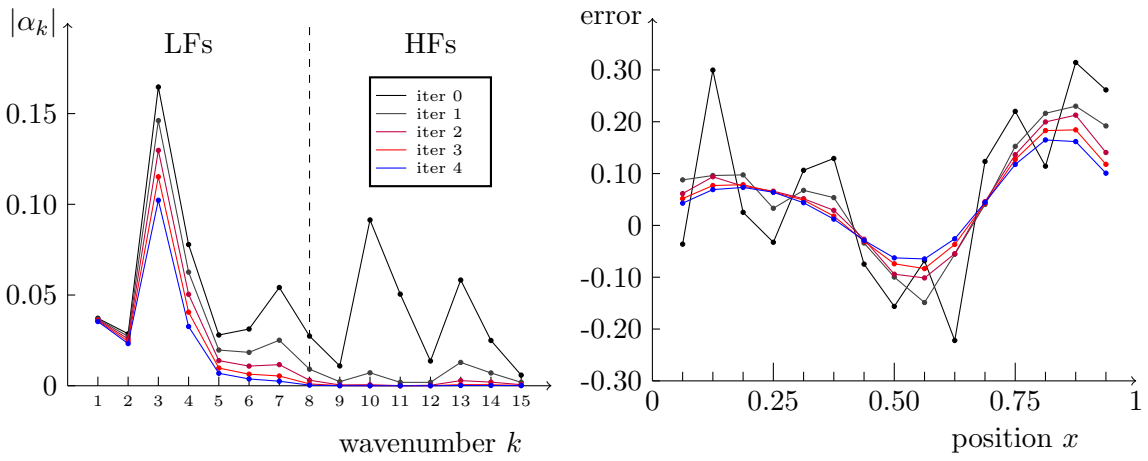


Figure 12.4: Weighted Jacobi applied to the 1D Poisson problem (12.1.1) with $f(x) = 12x(1-x)$ on a grid with $n = 15$ internal grid points. *On the left:* The contribution of the eigenvectors (there are 15) to the error after 0 (initial error), 1, 2, 3 and 4 iterations with optimal Weighted Jacobi ($\omega^* = \frac{2}{3}$). We see that the contribution of the HFs rapidly decreases, while the contribution of the LFs are much slower damped. *On the right:* The error after 0, 1, 2, 3 and 4 iterations with optimal Weighted Jacobi. We see that the highly oscillating pattern, consisting of equal contributions of LF modes and HF modes, is very rapidly reduced to a pattern which consists mainly of LF modes, this is the so-called "smoothing effect". As we saw on the left, the LF modes are poorly damped, causing the error very slowly to converge towards zero.

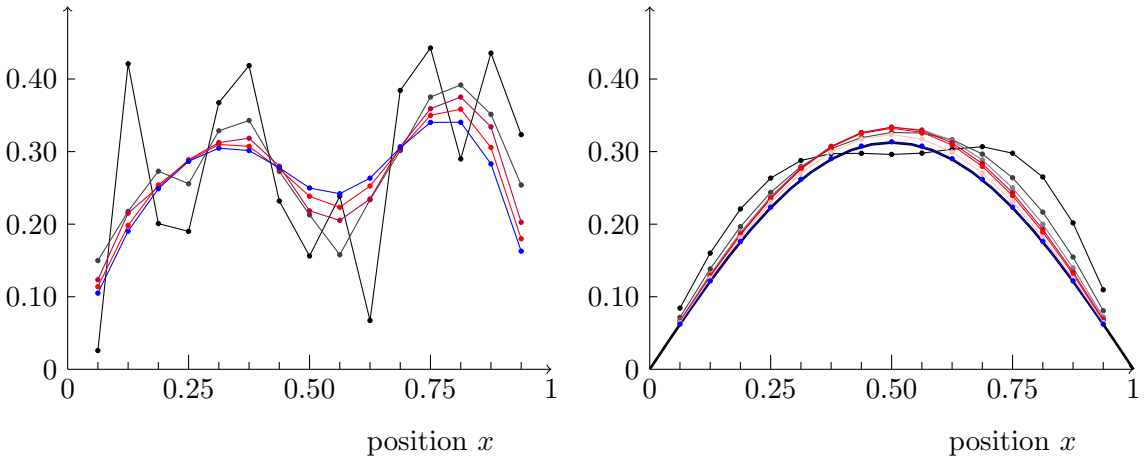


Figure 12.5: Weighted Jacobi applied to the 1D Poisson problem (12.1.1) with $f(x) = 12x(1-x)$ on a grid with $n = 15$ internal grid points. *On the left*: The approximate solution after 0 (initial guess), 1, 2, 3 and 4 iterations with optimal Weighted Jacobi ($\omega^* = \frac{2}{3}$). The highly oscillating approximation is very rapidly reduced to a smooth approximation thanks to the smoothing property of Weighted Jacobi. *On the right:* The approximate solution after 10, 20, 30, 40, 50, 100 and 1000 iterations. Due to the slow damping of LF error modes it takes Weighted Jacobi several 100 iterations to approximate the analytical solution (the fat black curve on the interval $[0, 1]$) accurately enough.

Now consider again the 1D Poisson problem (12.1.1) for $n = 15$. On the fine grid, Ω_h, there are 15 eigenmodes, i.e., just as many as there are internal grid points. With $H = 2h$ the coarse grid, Ω_H, has 7 internal grids points, and there are accordingly 7 eigenmodes. For Ω_h the eigenmodes ϕ_h^k are given by

$$\phi_h^k = (\sin(k\pi jh))_{j=1}^{15}, \quad k = 1, 2, \ldots 15,$$

whereas for Ω_H the eigenmodes ϕ_H^k are given by

$$\phi_H^k = (\sin(k\pi jH))_{j=1}^{7}, \quad k = 1, 2, \ldots 7.$$

We see that the smooth eigenmodes are preserved since

$$\phi_{h,2j}^k = \sin\left(\frac{k\pi(2j)}{n+1}\right) = \sin\left(\frac{k\pi j}{(n+1)/2}\right) = \phi_{H,j}^k$$

for $k = 1, 2, \ldots, 7$. For $k > (n+1)/2$, ϕ_h^k are invisible on ω_H because

$$\phi_{h,2j}^k = \sin\left(\frac{k\pi(2j)}{n+1}\right) = -\sin\left(\frac{2\pi j(n+1-k)}{n+1}\right) = -\sin\left(\frac{\pi(n+1-k)j}{(n+1)/2}\right) = -\phi_{H,j}^{n+1-k}.$$

This phenomenon is called *aliasing*, see Figure 12.6.

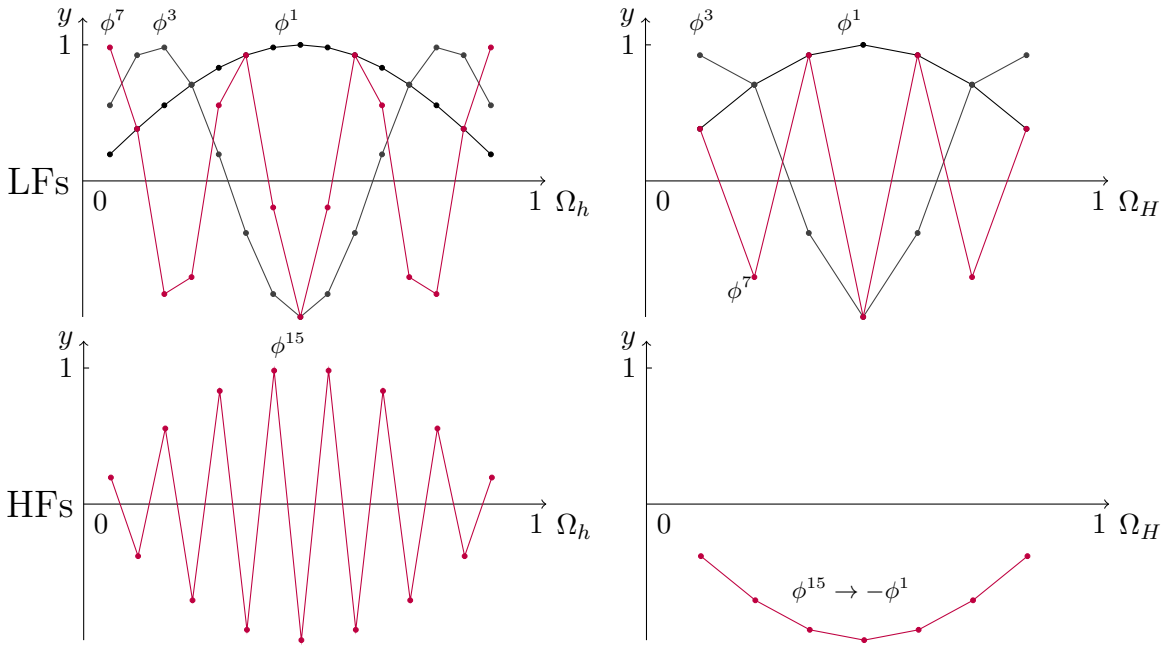


Figure 12.6: Smooth eigenmodes (LFs) are preserved on the coarse grid, oscillatory modes (HFs) become aliased.

What we see is that on the coarse grid the smooth mode appears to be relatively higher in frequency. For example, the eigenmode ϕ^3 on the fine grid is the 3th out of a total of 15, hence 1/5 the way up the spectrum. However, on the coarse grid ϕ^3 is the 3th out of a total of 7, hence 3/7 the way up the spectrum. Hence relaxation on a smooth mode will be more effective when performed on a coarser grid.

12.2 Two-grid (TG) method

The problem to be solved (12.1.3) lives on the fine grid Ω_h, let us write

$$A_h u_h = f_h. \tag{12.2.1}$$

As pointed out earlier, we want to approximate this problem on a coarser grid; therefore, on Ω_H we formulate the following problem:

$$A_H u_H = f_H. \tag{12.2.2}$$

Here A_H is called the *coarse grid matrix* and in a moment we shall discuss how we can get this A_H. Let us first explain how we map from the fine grid to the coarse grid and vice versa.

12.2.1 Restriction and prolongation

Let

$$U_h : \Omega_h \longrightarrow \mathbb{R} \quad \text{and} \quad U_H : \Omega_H \longrightarrow \mathbb{R}$$

be the sets of fine and coarse grid functions, respectively. To go from the fine grid to the coarse grid we use a *restriction operator*

$$R_h^H : \Omega_h \longrightarrow \Omega_H, \tag{12.2.3}$$

and to map from the coarse grid to the fine grid we use a *interpolation* or *prolongation operator*

$$P_H^h : \Omega_H \longrightarrow \Omega_h. \tag{12.2.4}$$

Then, for $u_h \in U_h$ and $u_H \in U_H$ we have

$$u_H = R_h^H u_h \quad \text{and} \quad u_h = P_H^h u_H.$$

The prolongation operation P_H^h is defined by linear interpolation, i.e., by writing $u_h = (u_{h,1}, u_{h,2}, \ldots, u_{h,n})^T$ and $u_H = (u_{H,1}, u_{H,2}, \ldots, u_{H,(n-1)/2})^T$, thus only taking the unknowns into consideration, we have $u_{h,1} = \frac{1}{2}u_{H,1}$, $u_{h,n} = \frac{1}{2}u_{H,(n-1)/2}$, and

$$\begin{aligned} u_{h,2j} &= u_{H,j}, \quad \text{for } j = 1, 2, \ldots, (n-1)/2, \\ u_{h,2j+1} &= \tfrac{1}{2}(u_{H,j} + u_{H,j+1}), \quad \text{for } j = 1, 2, \ldots, (n-3)/2. \end{aligned}$$

Hence, written as a matrix, $P_H^h \in \mathbb{R}^{n \times (n-1)/2}$ is given by

$$P_H^h = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \ldots & & & \\ 1 & 0 & 0 & \ldots & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & \ldots & & \varnothing & \\ 0 & 1 & 0 & \ldots & & & \\ 0 & \frac{1}{2} & \frac{1}{2} & \ldots & & & \\ 0 & 0 & 1 & \ldots & & & \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ & & & \ldots & 1 & 0 & 0 \\ & & & \ldots & \frac{1}{2} & \frac{1}{2} & 0 \\ & & & \ldots & 0 & 1 & 0 \\ & \varnothing & & \ldots & 0 & \frac{1}{2} & \frac{1}{2} \\ & & & \ldots & 0 & 0 & 1 \\ & & & \ldots & 0 & 0 & \frac{1}{2} \end{bmatrix}. \tag{12.2.5}$$

Remark 12.2.1. *Note that P_H^h has full rank, hence* $\text{Null}(P_H^h) = \varnothing$.

An example of prolongation is given in Figure 12.7.

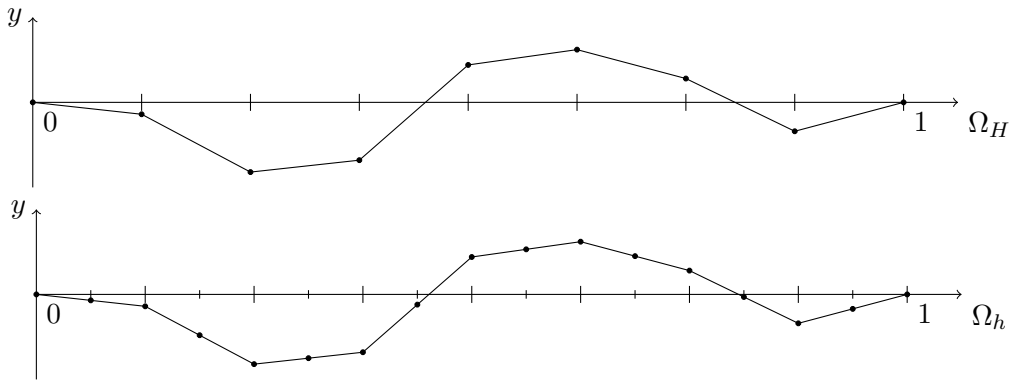


Figure 12.7: Prolongation by linear interpolation.

For the restriction operator there are several options. The simplest one is so-called *injection* which is nothing more than taking the value on the coarse grid to be the same as the corresponding value on the fine grid, and forget about the grid points that are not part of the coarse grid. So, for $u_h = (u_{h,1}, u_{h,2}, \ldots, u_{h,n})^T \in U_h$ and $u_H = (u_{H,1}, u_{H,2}, \ldots, u_{H,(n-1)/2})^T \in U_H$ the injection method in 1D is given by

$$v_{H,j} = v_{h,2j}, \quad \text{for } j = 1, 2, \ldots, (n-1)/2.$$

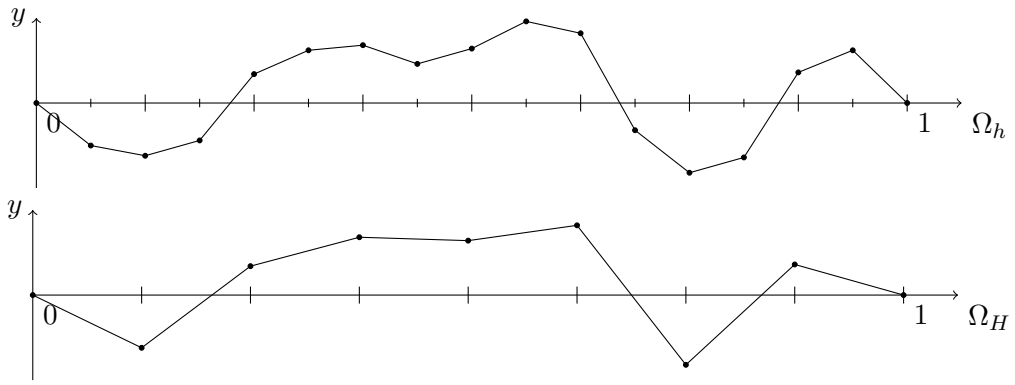


Figure 12.8: Restriction by injection.

An example of restriction by injection is given in Figure 12.8. Restriction by *full weighting* is a second method; in 1D it is given by

$$v_{H,j} = \tfrac{1}{4}(v_{h,2j-1} + 2v_{h,j} + v_{h,2j+1}), \quad \text{for } j = 1, 2, \ldots, (n-1)/2.$$

An example of restriction by full weighting is given in Figure 12.9.

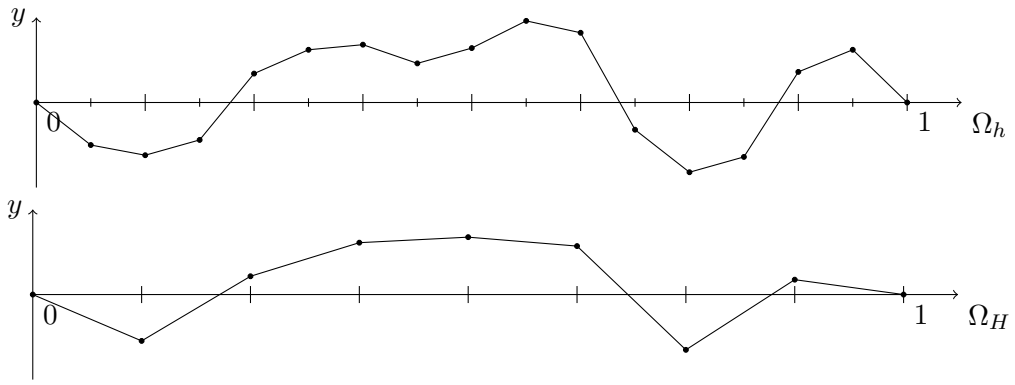


Figure 12.9: Restriction by full weighting.

Written out as a matrix, $R_h^H \in \mathbb{R}^{(n-1)/2\times n}$ is given by

$$R_h^H = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & \dots & & & & & & \\ 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & \dots & & & & \varnothing & & \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \dots & & & & & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & & & \dots & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ & & \varnothing & & & & \dots & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 \\ & & & & & & \dots & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

Remark 12.2.2. *Note that* $\operatorname{rank}(R_h^H) = (n-1)/2$, *hence* $\operatorname{nullity}(R_h^H) = (n+1)/2$.

Remark 12.2.3. *Note that* $P_H^h = 2(R_h^H)^T$. *Although not essential, this relationship, also called* variational condition *is highly useful as we shall see later on.*

Let us now focus on how to obtain the coarse grid matrix A_H. There are basically two ways [14, 27] to choose A_H, namely

1. Discretization of the PDE on the coarse grid, which is often referred to as *discretization coarse grid approximation (DCA)*;
2. The *Galerkin coarse grid approximation (GCA)* by letting A_H given by

$$A_H = R_h^H A_h P_H^h. \tag{12.2.6}$$

Remark 12.2.4. *Equation (12.2.6) is usually referred to as the* Galerkin condition.

We should check the following claim.

Lemma 12.2.5. *A_H given by (12.2.6) is symmetric.*

Proof. Indeed,

$$(A_H)^T = (R_h^H A_h P_H^h)^T = (P_H^h)^T A_h^T (R_h^H)^T = 2R_h^H A_h \cdot \tfrac{1}{2} P_H^h = R_h^H A_h P_H^h = A_H,$$

where we have used the symmetry of A_h and the relationship $P_H^h = 2(R_h^H)^T$. □

In case of our 1D Poisson problem, it is easily checked that the DCA and GCA method lead to the same result[5], namely

$$A_H = \frac{1}{(2h)^2}[-1 \quad 2 \quad -1]. \tag{12.2.7}$$

12.2.2 Pre- and post-smoothing

Let us formalize the idea of smoothing. Henceforth we shall denote the smoothing process by

$$u = S(u, A, b, \nu). \tag{12.2.8}$$

This notation means that ν steps are performed on the system $Au = b$ with some smoothing method S, i.e., by writing $A = M - N$ the smoothing process may be represented as

$$u^{i+1} = Su^i + M^{-1}b, \tag{12.2.9}$$

where $S = M^{-1}N$ is the iteration matrix. For an initial approximation u^0 after ν iterations we have

$$u^\nu = S^\nu u^0 + R(\nu)b, \tag{12.2.10}$$

where

$$R(\nu) = \sum_{j=1}^{\nu-1} S^j M^{-1} = (S^{\nu-1} + S^{\nu-2} + \cdots + I)M^{-1}. \tag{12.2.11}$$

This is easily checked by writing things out. All this is performed in one iteration, therefore u is consecutively overwritten, and the indices i are omitted in (12.2.8).

12.2.3 The TG algorithm

Using the notations introduced in the preceding sections, the two-grid (TG) method is given by the following iterative scheme.

Step	Mathematics	Description
1.	$u_h = S(u_h, A_h, f_h, \nu_1)$	Pre-smoothing on fine grid
2.	$r_h = f_h - A_h u_h$	Computing residual
3.	$r_H = R_h^H r_h$	Restriction
4.	Solve $A_H e_H = r_H$	Solving exactly on coarse grid
5.	$e_h = P_H^h e_H$	Prolongation
6.	$u_h = u_h + e_h$	Coarse-grid correction
7.	$u_h = S(u_h, A_h, f_h, \nu_2)$	Post-smoothing on fine grid

Table 12.1: The two-grid (TG) method.

Like any method the TG method can be put in the form (8.1.2) for some iteration matrix $Q = Q_{\mathrm{TG}}$.

[5] For the DCA method substitute $h = 2h$ into (12.1.2), and for the GCA method just straightforwardly compute $A_H = R_h^H A_h P_H^h$.

Lemma 12.2.6. *The iteration matrix Q_{TG} of the TG method is given by*

$$Q_{\mathrm{TG}} = S^{\nu_2}(I - P_H^h A_H^{-1} R_h^H A_h) S^{\nu_1}. \tag{12.2.12}$$

Proof. First consider only Steps 2 to 6 (thus no pre- and post-smoothing). We then find

$$\begin{aligned}
u_h &= u_h + e_h \quad \text{(by Step 6)} \\
&= u_h + P_H^h e_H \quad \text{(by Step 5)} \\
&= u_h + P_H^h A_H^{-1} r_h \quad \text{(by Step 4)} \\
&= u_h + P_H^h A_H^{-1} R_h^H r_h \quad \text{(by Step 3)} \\
&= u_h + P_H^h A_H^{-1} R_h^H (f_h - A_h u_h) \quad \text{(by Step 2)} \\
&= (I - P_H^h A_H^{-1} R_h^H A_h) u_h + P_H^h A_H^{-1} R_h^H f_h \\
&= Q_{\mathrm{CGC}} u_h + P_H^h A_H^{-1} R_h^H f_h,
\end{aligned}$$

where we introduced the matrix Q_{CGC}, the *coarse grid correction matrix*, which is thus given by

$$Q_{\mathrm{CGC}} = I - P_H^h A_H^{-1} R_h^H A_h. \tag{12.2.13}$$

Next we apply equation (12.2.10) two times and forget about the terms that do not contribute to the iteration matrix. In the end post-smoothing is used, hence $u_h = S^{\nu_2} u_h$, according to (12.2.10). Before that, the course grid correction is applied, hence $u_h = S^{\nu_2} Q_{\mathrm{CGC}} u_h$, with Q_{CGC} given by (12.2.13), and even before that, in the beginning, pre-smoothing is used, hence $u_h = S^{\nu_2} Q_{\mathrm{CGC}} S^{\nu_1} u_h = Q_{\mathrm{TG}} u_h$, with Q_{TG} indeed given by (12.2.12), as desired. □

Schematically the TG method can be presented as in Figure 12.10.

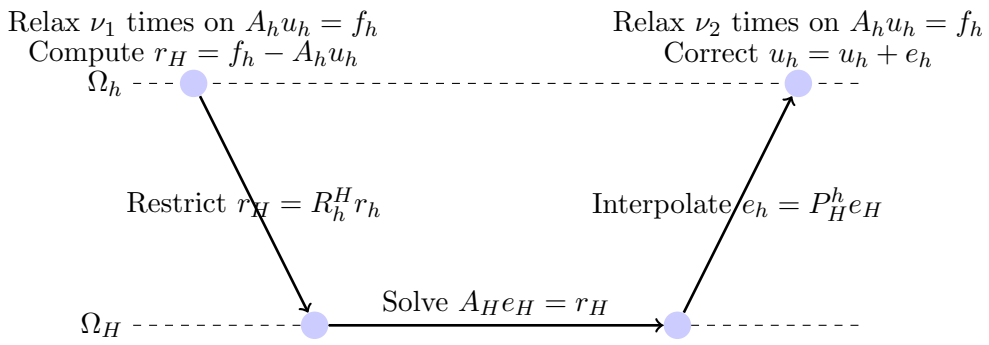


Figure 12.10: The TG method.

12.3 The MG algorithm

Until now we solved the problem $Au = f$ using only two grids with the TG method, see Table 12.1. However, it is not realistic that we can solve Step 4, i.e., $A_H e_H = r_H$, in the TG method exactly, because the amount of work is about 4 times less, but for large n the problem can still be quite large. The solution is simple: we apply the TG method recursively to Step 4.

To be able to make this mathematically precise, we need some new notation. Assume that the number of grid points in x- and y-direction is $n = 2^k - 1$ for some nonnegative integer k. This choice of n allows a maximum number of k grids. However, usually it is not practical to go all the way down to a grid with only 1 grid point. So instead assume that we have $L + 1$ grids. We generate a set of grids: $(\Omega_\ell)_{\ell=0}^{L} = \{\Omega_0, \Omega_1, \ldots, \Omega_L\}$, where Ω_0 denotes the coarsest grid, and Ω_L the finest grid. The meshsizes of the grids are halved each time starting from $h_L = h$. Now let

1. A_ℓ be the approximation of A on Ω_ℓ;
2. f_ℓ be the corresponding RHS on Ω_ℓ;
3. $R_\ell^{\ell-1} : \Omega_\ell \longrightarrow \Omega_{\ell-1}$ be the restriction operator;
4. $P_\ell^{\ell+1} : \Omega_{\ell+1} \longrightarrow \Omega_\ell$ be the prolongation operator; and
5. S_ℓ be the iteration matrix of the smoothing method on Ω_ℓ.

We write $S(\tilde{u}_\ell, u_\ell, A_\ell, f_\ell, \nu)$ to express that we perform ν smoothing iterations on the system $A_\ell u_\ell = f_\ell$. Although the numbers ν can be different on each grid, let us take for now a fixed number ν_1 of pre-smoothing steps and a fixed number ν_2 of post-smoothing steps.

Using the preceding notation the Multigrid algorithm is given by the following scheme.

```
1  MG(u_ℓ, A_ℓ, f_ℓ, ℓ);
2  if ℓ = 0 then
3      Solve A_ℓ u_ℓ = f_ℓ;
4  else
5      u_ℓ = S(u_ℓ, A_ℓ, f_ℓ, ν_1);              // Pre-smoothing
6      r_{ℓ-1} = R_ℓ^{ℓ-1}(f_h − A_ℓ u_ℓ);          // Coarse grid
       residual
7      A_{ℓ-1} = R_ℓ^{ℓ-1} A_ℓ P_ℓ^{ℓ+1};      // Coarse grid matrix
       (GCA)
8      for i = 1, 2, ..., γ_ℓ do
9          MG(u_{ℓ-1}, A_{ℓ-1}, f_{ℓ-1}, ℓ − 1); // Recursive call
10     end
11     u_ℓ = u_ℓ + P_ℓ^{ℓ+1} u_{ℓ-1}; // Coarse grid correction
12     u_ℓ = S(u_ℓ, A_ℓ, f_ℓ, ν_2);          // Post-smoothing
13 end
```

Algorithm 7: Recursive Multigrid algorithm.

A Multigrid cycle depends on γ_ℓ, ν_1 and ν_2. The first parameter says how many times the same cycle is repeated. This is best illustrated some examples. Take L grids, $\gamma_\ell = 1$ for all $\ell = 0, 1, 2, \ldots, L$, and $\nu_1 = \nu_2 \neq 0$. This type of cycle is called, for obvious reasons, a V-cycle, see Figure 12.11. To be able to visualize other type of cycles we take $L = 3$, thus 4 grids. When $\gamma_\ell = 2$ for all $\ell = 0, 1, 2, 3$, and $\nu_1 = \nu_2 \neq 0$, we get a W-cycle, see Figure 12.12 (on the left), and for $\gamma_\ell = 1$ for all $\ell = 0, 1, 2, 3$, and $\nu_1 = 0$, $\nu_2 \neq 0$ we get a so-called sawtooth-cycle, see Figure 12.12 (on the right).

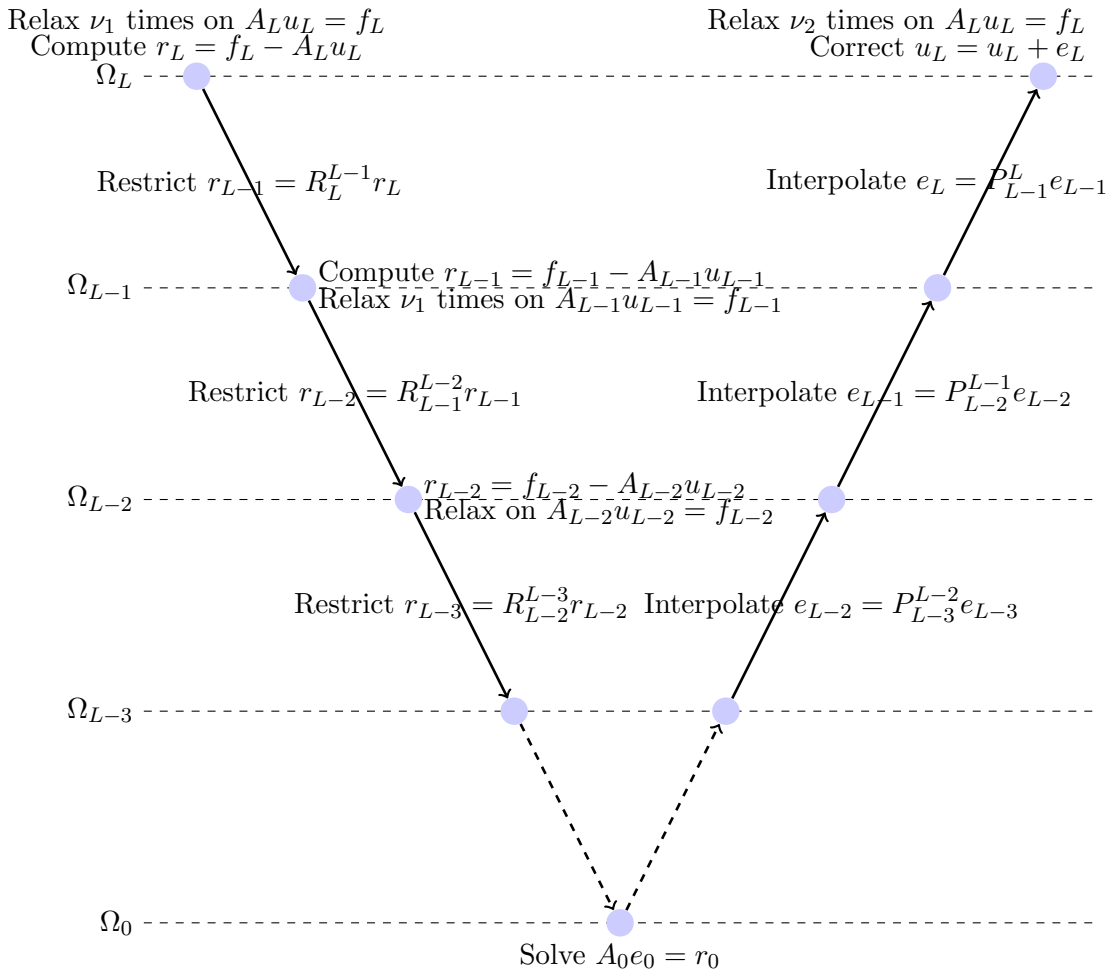


Figure 12.11: A Multigrid V-cycle.

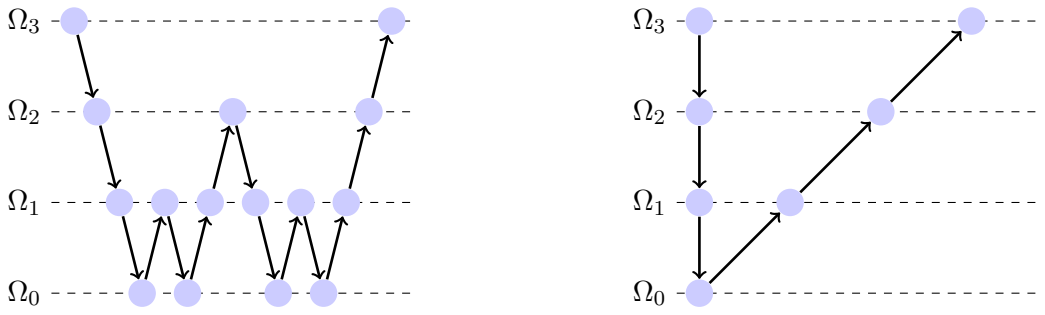


Figure 12.12: A Multigrid W-cycle (left) and sawtooth-cycle (right).

Part III

SCIENTIFIC COMPUTING WITH CUDA

Chapter 13

GPU architecture

We discuss briefly a GPU's architecture as it is necessary to understand how to develop efficient CUDA software. The next discussion is far from complete and only focusses on the most important notions for us; for full details please consult the NVIDIA CUDA C Programming Guide version 4.0 [17] (this is the latest version in which the new Fermi architecure is included also). Also the book written by Kirk and Hwu [12] contains lots of information on this subject. We shall briefly mention the differences between the latest and older generation GPUs. We shall use the terms GPU and device interchangeably.

13.1 Architecture category

A GPU belongs to the architecture category of single-input multiple-data (SIMD) processors, which basically means that many processors do the same computations for different data in parallel. While each processor may run at ordinary speeds, say 1 GHz, thousands of these processors integrated in one device yield extreme crunch power. In the context of a GPU we rather speak about single-input multiple-threads (SIMT). What a thread is, is discussed below.

13.2 How work is executed on the GPU

Before we can discuss the architecture of a GPU, we must already say something about the way a program on the GPU is executed and how work is scheduled. It basically works as follows. When running CUDA software the program will on its way invoke one or multiple so-called *kernels*. A kernel in fact is a portion of the program that is executed on the GPU and can be isolated into a C language function that is executed as many times as there are threads. A *CUDA thread* is thus one instance of a job that has to be done in parallel, and for the job the thread uses its own data, hence an SIMD or SIMT architecture.

The threads are organized by the programmer by defining a *grid* and making a division of the grid in *thread blocks* or just *blocks*, see Figure 13.1. Each block consists of a batch of threads, and can be a 1D, 2D or 3D object. The maximal number of threads which is allowed depends on the compute capability, see Section 13.3.

The blocks are divided amongst the physical processors of the GPU: the *streaming multiprocessors*, see Section 13.4. Threads inside a block are grouped into *warps*. A warp consists of typically of 32 threads with consecutive thread indices. The *scheduler* that picks up threads

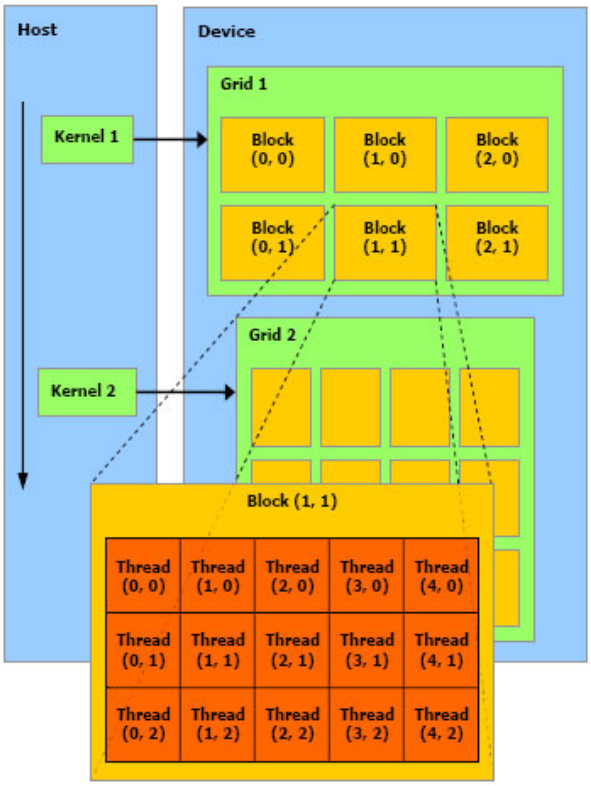


Figure 13.1: The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks.

for execution does so in granularity of a warp. A thread (resp., block) is identified by its *thread index* (resp., *block index*), which is the thread (resp., block) number within the block (resp., grid). Index is commonly abbreviated to ID.

13.3 Compute capability

Every CUDA compatible GPU is tagged with a 2 digit number: the *compute capability*. The number expresses to what degree the GPU can perform different tasks and operations and indicates the availability and amount of different resources on the device, e.g., whether the device can perform double precision arithmetic, the amount of registers available, the maximal number of threads allowed per block, the throughput of arithmetic instructions for different variable formats, etc.

More precisely: the compute capability number consists of a *major revision number* and a *minor revision number*, notation: c.x. The major revision number (c = 1 or 2) indicates the core architecture, and the minor revision number indicates smaller improvements with respect to core architecture and available resources and newly added features. At the moment of writing the latest architecture is the so-called *Fermi architecture* with compute capability 2.0 or 2.1, e.g., the GeForce GTX 580. The Fermi architecture often requires algorithms to be (completely) redesigned such that optimal performance is achieved. The reason for this

is that for Fermi GPUs the register bandwidth is significantly higher than before, see also Section 15.2.6.

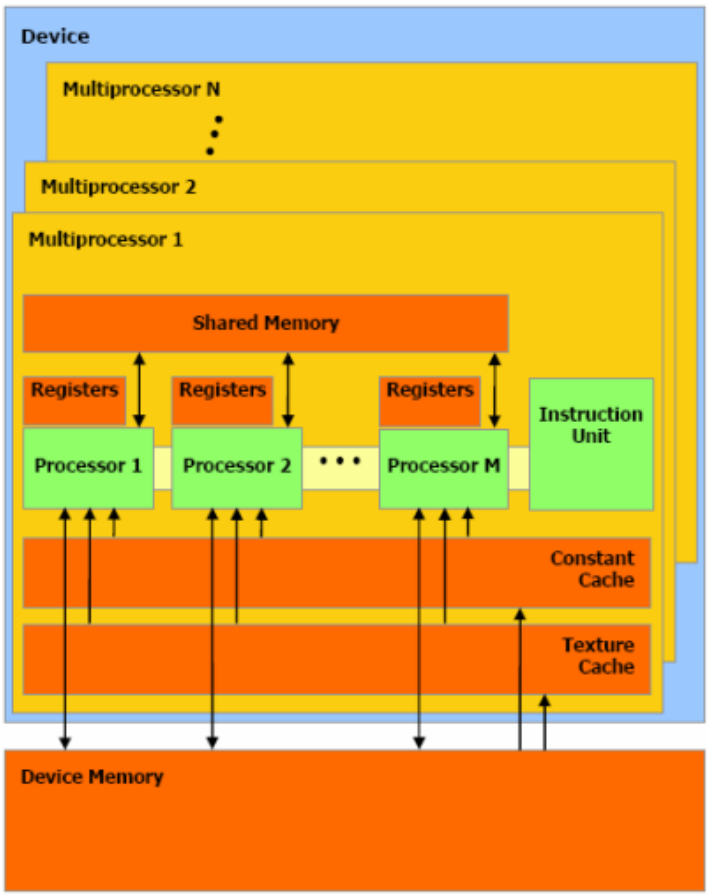


Figure 13.2: Layout of a GPU. The basic entity is a Streaming Multiprocessor (SM), that has its own cache and shared memory and that contains a number $(8 - 48)$ streaming processors (SPs) that work synchronously (SIMD).

13.4 Physical processors

Consider Figure 13.2. A GPU is equipped with a number of *streaming multiprocessors* (SMs). The number of SMs varies from 1 (Quadro NVS 295) up to, say, 4×30 (Tesla C1070). Each SM contains a number of *streaming processors* (SPs) also called *CUDA cores* or *Shader Units* (SUs). The amount of SPs depends on the *compute capability* of the GPU. For older GPUs, i.e., GPUs of compute capability 1.x, an SM consists of 8 SPs, whereas for GPUs of compute capability 2.0 an SM consists of 32 SPs, and GPUs of compute capability 2.1 have 48 SPs per SM. Furthermore, an SM has an *Instruction Unit* (IU) also called *warp scheduler* which provides the instructions and schedules the warps. Older architectures (1.x) indeed have 1 IU, but the SMs of the Fermi have in fact 2 IUs; one takes care of the even thread IDs while the other takes care of the odd thread IDs.

13.5 Memory hierarchy

Again consider Figure 13.2. A GPU has different layers of memory. The main memory is the *global memory* or *device memory* (DRAM). It is the memory with the biggest capacity, namely up to 6GB (Tesla C2070), but is also the slowest ($400 - 600$ cycles latency). Parts of the DRAM are dedicated to *constant memory* and *texture memory* which are read-only. Within each SM the constant memory and texture memory are accessible through *constant cache* and *texture cache* respectively.

Each SM contains some *shared memory*. Shared memory is very fast memory ($30 - 50$ cycles latency), but it is also very scarce. GPUs of compute capability 1.x have 16 banks of 1KB each, whereas GPUs of compute capability 2.0 and 2.1 have 32 banks with a total capacity of 48KB.

Further each SM has a set of *registers*. Registers are the fastest memory type, although shared memory without bank conflicts can compete with the older type of registers (compute capability 1.x) [17]. The latest architecture (Fermi) GPUs have a set of registers that are significantly faster than shared memory: up to $8\times$ according to Volkov [24]. GPUs of compute capability 1.0 and 1.1 have 8K (= 8192) registers, GPUs of compute capability 1.2 and 1.3 have 16K registers, and GPUs of compute capability 2.0 and 2.1 have 32K registers. Although the numbers seem okay, they usually are the limiting factor in GPU program designing.

Further, the GPU has some cache (not depicted in the figure). With older architectures the GPU has texture cache only; with the Fermi architecture the GPU has 768 KB L2 cache and (adjustable) 16 up to 48 KB L1 cache per SM.

The memory types come with a set of rules which make the GPU harder to program than the CPU. In Sections 15.1.3 and 15.1.4 we discuss two very important notions: memory coalescing and bank conflicts.

Chapter 14

CUDA C programming environment

By this point you may possibly wonder what CUDA actually stands for since we have used the abbreviation already several times in the previous sections. CUDA stands for "Compute Unified Device Architecture" and basically is an extension to the C programming language which offers developers a relatively easy environment to program the GPU. CUDA was introduced somewhere around november 2006, and from that moment programming on the GPU for scientific purposes really took off.

At the moment of writing there are already quite some libraries available with implementations of, e.g., BLAS, Conjugate Gradients (CG), and FFT, on the GPU, so programming from scratch is no longer really necessary if your problem can be put in a form with lots of library function calls. Examples of libraries are CUBLAS, CUFFT, MAGMA BLAS, CUSP, Iterative CUDA.

In the next sections we briefly discuss the structure of programming with CUDA. For more information we refer to the books [12] and [21].

14.1 Thread organization

In Section 13.2 we already introduced *kernels* and we saw how *threads* are organized. Now it is time to provide more details on this subject. When invoking a kernel one must specify the dimension of the *grid*, i.e., the number of *blocks*, and the dimension of the blocks, i.e., the number of threads per block. In CUDA this is achieved via:

```
kernel <<< dimGrid, dimBlock >>> (...);
```

Both dimGrid and dimBlock are of the dim3−format; both grids and blocks can be 1D, 2D or 3D objects ($x-$, $y-$, $z-$direction). The number of blocks and threads allowed depends on the compute capability of the device, see Section 13.3. Between brackets we put the parameters and data required by the kernel, e.g., in case of a matrix-vector product, we specify two input arrays, say A and x, and an output array, say y, also we must specify the dimension of matrix A, say n for an $n \times n$ matrix. The dimensions of the grid and blocks can be retrieved from the kernel's code via statements like: Gx = gridDim.x, and By = blockDim.y.

The index of the blocks and threads in $x-, y-$ and $z-$direction can be retrieved via statements like: by = blockIdx.y, and tz = threadIdx.z. By combining the block and thread indices we can assign to each thread in a particular block a *unique* global $x-$

and $y-$index, e.g., for a 2D grid consisting of 2D blocks: `threadIDx = blockDim.x * blockIdx.x + threadIdx.x` and `threadIDy = blockDim.y * blockIdx.y + threadIdx.y`. Likewise we can compute a unique global index when one prefers to use just one running index. This comes in handy when we want each thread to do different work.

Chapter 15

Strategies for a fast implementation

For details on the most of this matter one may consult the NVIDIA CUDA C Programming Guide [17], the excellent book by Kirk and Hwu [12] or just google for the Dr. Dobbs "CUDA, Supercomputing for the Masses"-articles on the Internet. We confine ourselves by just listing keywords and giving a brief discussion on each so that the reader is fully aware of what techniques are available.

15.1 General strategies

In this section we discuss some basic, frequently used techniques to get fast `CUDA` code.

15.1.1 Library functions

This technique is obvious: by using library functions rather than building own kernels, one has the advantage that as soon as a new version of the specific library is released one may instantenously get a faster program (after compiling) without doing any effort. The increase in performance is due to clever redesigning of the algorithms and taking in account the latest GPU architecures when implementing. A good example is the `CUBLAS` library which contains GPU implementations of the so-called BLAS (Basic Linear Algebra Subprograms). The implementations of the routines in the latest releases, `CUBLAS` v3.2 and `CUBLAS` v4.0, are much faster than in earlier releases.

15.1.2 Optimal tiling

Tiling, as the most general and important technique, is widely used for optimization in `CUDA` programs. Tiling basically means dividing the problem into blocks, or tiles, and assigning them to the different streaming multiprocessors (SMs) of the GPU. The goal is to do the tiling in such a way that the GPU's resources are addressed as optimal as possible. A basic example to illustrate tiling is the matrix-matrix (or matrix-vector) multiplication: a different tiling may yield a big increase (or decrease) in performance. An important factor here is how the data is actually stored (row-wise, column-wise). For optimal tiling of GPU programs one must have a very good insight in both the problem and the architecture of the GPU.

15.1.3 Global memory and memory coalescing

First of all, one should always strive for transferring big chunks of data instead of transferring many small data chuncks as the former is usually faster. The reason for this is that the global memory latency is in the order of 400-600 cycles of computing time which is very high.

Further, to the global memory applies the notion of so-called coalesced memory. The global memory bandwidth is highest when the global memory accesses can be coalesced within a half-warp, e.g., for 16 threads in a half-warp the consecutive 4-byte words must fall within 64-byte memory boundaries, and the 16 threads must access the words in sequence: the kth thread in the half-warp must access the kth word.

The penalty for non-coalesced memory transactions varies according to the actual size of the data type and architecture of the device, hence depends on the compute capability. Basically it holds that for older architectures (compute capability 1.0 and 1.1) non-coalesced memory transactions are serialized into 16 transactions, for newer architecturesit only results in two memory transactions instead of 16. However, in both cases performance is degraded.

If memory coalescing is not possible right away, one may consider to pad the data with zeros such that the data is aligned along 64-byte boundaries, or, for example, one may consider to use textures, see Section 15.2.3. In Section 20.2.6 a small case study is presented to illustrate the importance of coalesced memory. This study also shows how textures can be used to increase throughput.

15.1.4 Shared memory and bank conflicts

Shared memory is on chip memory and is therefore significantly faster than the global memory. Shared memory is slower than registers and is best used for communication between the treads. The shared memory is divided amongst so-called memory banks that are equal in size. Each memory bank holds a successive word and so consecutive accesses by consecutive threads are very fast. Bank conflicts occur when there are requests from multiple threads for the same memory bank. In that case the memory access are serialized. If all threads request data from one bank then a broadcast-mechanism is used and all threads recieve the data at the same time, hence no serialization. In Figure 15.1 on the left an example of shared memory bank conflicts is given.

15.1.5 Sum reduction

Sum reduction is discussed in more detail in Section 17.2. For now: sum reduction basically is adding together n numbers (preferably n is a power of 2) in a parallel and recursive way. Although there are different manners, we shall see in Section 17.2 how sum reduction is implemented most efficiently on the GPU. In Figure 15.1 on the right the concept of sum reduction without warp divergence and bank conflicts is already shown; see Section 17.2 how this scheme is derived. Not only can sum reduction be benificial to compute a large sum faster it also improves the accuracy with which the sum is computed significantly compared with a straightforward sequential computation.

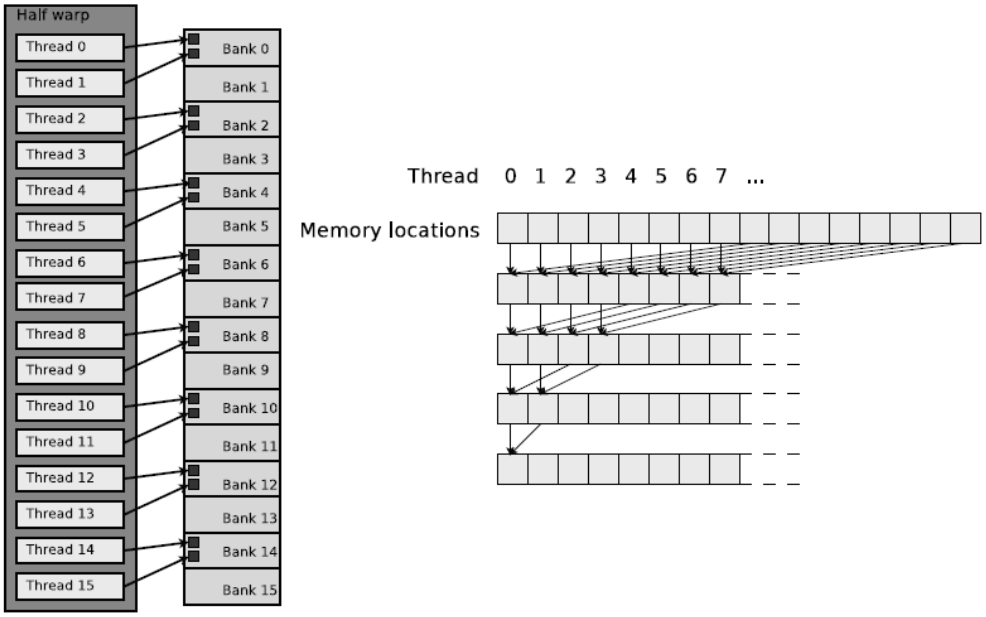


Figure 15.1: *On the left:* bank conflicts. *On the right:* sum-reduction without divergence and bank conflicts.

15.2 Advanced strategies

Below follows an enumeration of somewhat more advanced techniques. Depending on the device's architecture the following techniques may increase performance significantly, but it may also occur that no performance gain is obtained.

15.2.1 Pointers

In `C` is it slightly faster to march through arrays via pointers rather then by indexing. So too in `CUDA`. Usually it will be slightly faster because this allows direct memory manipulation; however, sometimes the differences can be so small that thousands of iterations are required to even notice them. By running two instances of the same kernel, one with and one without pointers, you can see whether they contribute to the kernel's performance or not.

15.2.2 Page-locked memory

As opposed to ordinary pageable host memory the runtime environment also offers the possibility to allocate so-called page-locked or pinned memory. Page-locked means that the memory pages are locked, thus that physical adresses remain unchanged and that the memory will not being swapped out by the operating system. The main benefit is that page-locked memory can be up to twice as fast as ordinary host memory. If in addition the memory is allocated as "write-combining" even higher bandwidth can be achieved, for more details consult [17]. However, page-locked memory is very scarce and should be used sparingly since the host may slow down or even hang.

15.2.3 Textures

In case memory reads from the global memory cannot be done in a coalesced manner, one should consider reading data through texture fetches. Texture cache reference latency is about the same as the latency of global memory, but offers benefits. The texture memory is namely cached in texture cache, and optimized for 2D spatial locality, which means that when data is fetched, nearby data is already been cached, since it is quite likely that this data is also wanted shortly.

Fujimoto published in 2008 a very interesting article [5] (source code included!) about a —at least for that time— superior matrix-vector algorithm in terms of overall performance. It outperformed NVIDIA's `CUBLAS` 1.1 SGEMV routine easily and did not show the huge performance drops which were typical for the SGEMV routine for matrices with dimensions not being a multiple of 16. The implementation uses 2D textures in combination with a float4 storage format for matrix A. The float4-format was used because a single float4 read is faster than four separate float reads.

In Section 20.2.6 a small case study is presented to show how textures can be used to greatly enhance a `CUDA` kernel's throughput.

15.2.4 Loop unrolling

An excellent thesis on loop unrolling is "Optimal Loop Unrolling for GPGPU Programs" by G. Murthy [16]. In this thesis it is investigated how to properly unroll loops in GPU programs; this is no so easy.

As the name indicates unrolling (or unwinding) a loop means that the loop is written out a couple of times, say n times (accordingly the loop counter is incremented by an n times bigger value), such that its performance is increased. This increase in performance may come from a reduction in dynamic instruction count (think of instructions that control the loop, such as pointer arithmetic, the each-iteration-check whether the loop can be exited, and compare and branch instructions); from the fact that the compiler's scheduler can better schedule the instructions thanks to the availability of additional independent instructions, improving instruction level parallelism (ILP), hiding pipeline and memory access latencies; and from the fact that the compiler has opportunities to exploit register usage and memory hierarchy localty.

However, by unrolling a loop too agressively, that is, too many times, actually the performance may decrease for several reasons, e.g., overflow of instruction cache leading to instruction cache misses (the code has become too big in size), increased register pressure spilling registers to memory, or inlining code is affected or no longer possible.

One basically has two options to unroll a loop: do it manually (static unrolling), or let the compiler take care of it (dynamic unrolling). For the latter: in `C` one uses `#pragma unroll` if the loop size is known at compile time, and `#pragma unroll n` when the loop size is variable, here n denotes the times the loop is unrolled; if the loop size is not a multiple of n the compiler automatically generates code to compute the remaining part. By the way, a `C` compiler automatically unrolls loops where it is beneficial to do so, so you can just omit this in plain `C` code; however, in the case of programming with `CUDA` with the NVCC compiler one has to put these statements in the code to force the compiler to unroll the loop. Moreover, to control the register pressure one may compile the code with the flag `--maxregcount=N` to limit the number of registers per thread to N.

15.2.5 Better performance at lower occupancy

Vasily Volkov came recently some very interesting observations, which go against earlier beliefs. It was namely commonly agreed and advocated through CUDA manuals and programming guides that it is best to have as many threads as possible alive per multiprocessor or per thread block, thus to maximize occupancy. It was believed that this is the only way to cover memory latencies. However, Volkov showed in several presentations that by maximizing occupancy you may lose performance, and actually that faster codes run at lower occupancy. A nice illustration is how Volkov manages to double the SDK matrix-matrix multiply example by minor modifications. See for example [24] and [25].

15.2.6 Registers versus shared memory

Another fallacy that Vasily Volkov observed was the belief that shared memory without bank conflicts is as fast as registers. In fact this is still stated in the NVIDIA CUDA C Programming Guide 3.2 [17], p92. According to Volkov [24], the bandwidth of shared memory is at least 3 times lower than the register bandwidth for GPUs with compute capability 1.x, and at least 6 times lower than the register bandwidth with the Fermi architecture.

Hence to get close to the device's Peak performance Volkov concludes that one must exploit register usage rather than shared memory for computations. This inherently requires low occupancy to preserve the registers for actual computations rather than "wasting" them for creating instances of the same variable for each thread. Key to obtain lower occupancy but still doing the same amount of work overall is to compute multiple outputs per thread.

15.2.7 Overlapping communication and computation

With the latest GPU architectures it is possible to overlap communication between the host and the device and computations on the device. This is possible as modern devices have two types of processing units: units that handle memory operations and units that do actual computations. Consider the following simple example. Suppose we have two data objects A and B, and a kernel that does some computations with A and B. Further assume that copying the data and performing the calculations take an equal amount of time.

In Figure 15.2 a straightforward approach is shown. First A and B are copied from the host to the device, then the kernel works on A and B, and in the end the data is copied back from the device to the host.

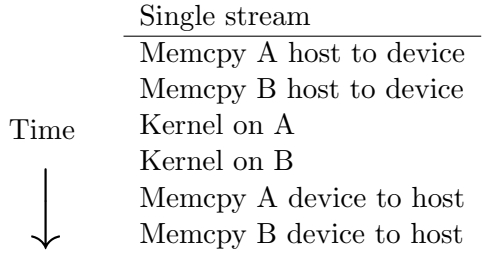


Figure 15.2: Using a single stream.

If each job takes 1 time unit, the total runtime of the program would be 6 time units. In Figure 15.3 a better approach is shown. We create two so-called *streams*. Stream 0 will take care of the data transfers, while stream 1 takes care of invoking the kernel. As modern GPUs allow memory transfers and computations at the same time, communication and computation can be overlapped. The total time is now only 4 time units, hence a performance gain of 50%.

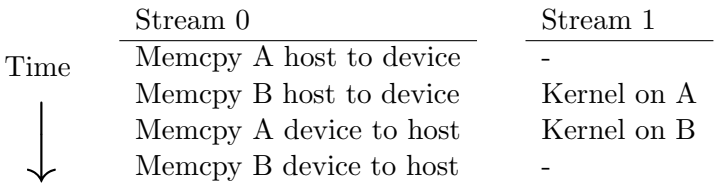


Figure 15.3: With overlapping.

In Part IV, Section 20.2.8 we come back to this technique. In the IPDIAG-solver some computations and communication are overlapped, saving some computation time.

15.2.8 Concurrent kernels

Like the idea of overlapping communication and computation, concurrent kernels can save execution time by running more kernels parallely. This only works if both kernels do not address all resources of the GPU already; if one (or both) kernel(s) already ask(s) for all the GPUs resources the kernels are either serialized or the hardware resources are divided over the kernels. Hence concurrent kernel launches give no sudden doubling of compute power; rather it is just a scheduling convenience.

Chapter 16

Measuring and optimizing performance

16.1 Performance measures

The most frequently used and measurable metric of an algorithm is its speed or execution time. Below we discuss the two derivated quantaties to indicate performance that are used most often.

16.1.1 Floprate

In case of floating point operations (multiplications and additions) the notion of *flops* or *flop/s* is frequently used. Flops stands for floating point operations per second. The amount of flops is computed via:

$$\text{flops} = \frac{\#\text{floating point operations}}{\text{execution time}},$$

where execution time is in seconds (s). Because modern computers can perform million of flops, we shall use Mflops, Gflops and even Tflops: 1 Tflops = 10^3 Gflops, 1 Gflops = 10^3 Mflops, and 1 Mflops = 10^6 flops. Thus the higher the amount of flops the faster the algorithm.

Example: floprate for an SpMV Let us calculate the floprate for the no textures SpMV kernel using StC, StS, StW in Section 20.2.6 on the GeForce GTX 580 in case of a 2048×2048 grid. According to Table 17.2 it takes the kernel 649 μs. Computing the sparse matrix-vector product for a sparce $n \times n$ matrix it takes $9n$ flops (for each row of the matrix: 5 multiplications and 4 adds). Hence the floprate in Gflops is:

$$\text{flops} = \frac{9 \cdot 2048^2}{10^9 \cdot 649 \cdot 10^{-6}} = 58.16 \text{ Gflops}.$$

It is interesting to compare the achieved number of flops by the algorithm with the theoretical Peak of the GPU, that is the maximal amount of flops the device can deliver. In practice we can only reach a fraction of the Peak performance, in fact it is already satisfactory if the sustained rate of the algorithm is just 10% of Peak. Modern CPUs have a Peak between 5-100 Gflops (Core i7 @ 3.2 GHz: 70 Gflops) whereas high-end GPUs have possibly

more than one Tflops on board (Tesla C2070: 1.030 Tflops, GeForce GTX 580: 1.580 Tflops). These numbers apply to single-precision computations only; double-precision is significantly slower (actually 2 times in case of the latest GPU architectures, on older GPU architectures even 4 or 8 times slower).

Example: Peak versus sustained floprate The sustained rate of flops of the SpMV kernel above is much smaller than GTX 580's theoretical Peak, namely 58.16 versus 1,580 Gflops (the efficiency is thus a "poor" 3.6%). This indicates the SpMV kernel is bandwidth bound.

16.1.2 Throughput

Besides flops the notions of bandwidth and throughput is frequently used to investigate the performance. The terms are frequently used interchangeably although, originally, they are conceptual different. This may cause confusion and, therefore, we shall try to be clear from the very start by defining them as follows. The given interpretations will be used throughout the report.

Bandwidth is the maximum rate at which data can be transferred, thus a theoretical amount (depending on hardware), whereas *effective bandwidth*, or *throughput*, is the actual number of transferred bits per time, thus a real amount (depending on application). Furthermore, additionaly, we would like to introduce the term *useful throughput*. Simply putted: the useful throughput is the amount of data per time read that "can be used (usefully)", and is, thus, a fraction of the throughput, i.e., useful throughput $\leq$ throughput. Note that, when all data that passes the device is actually used, throughput and useful throughput are just the same. Since all these performance numbers are typically in the order of 10^9 bytes per second, we use GB/s. Note that it makes a small difference whether we take 10^9 or 1024^3. We will use 10^9 throughout the report.

Let us now see how to compute bandwidth and throughput for the global memory for a NIVIDA device that runs a `CUDA` kernel.

The (Peak theoretical) global memory bandwidth in GB/s of a NVIDIA device can be computed via:

$$\text{global memory bandwidth} = \frac{\text{memory clock rate} \cdot \text{memory bus width} \cdot \text{data rate}}{10^9},$$

where the memory clock rate is given in Hertz (Hz) and the interface width in bytes (B). The data rate indicates the speed of the global memory, e.g., in case of DDR (which is standard) we have data rate = 2.

Example: global memory bandwidth GeForce GTX 580 The GTX 580 runs at 2.010 GHz = $2.01 \cdot 10^9$ Hz and its memory bus width is 384-bit. This information can be obtained by running `./deviceQuery` that comes with the NVIDIA SDK library. Hence

$$\text{global memory bandwidth} = \frac{(2.01 \cdot 10^9) \cdot (384/8) \cdot 2}{10^9} = 192.96 \text{ GB/s}.$$

To compute the throughput of a `CUDA` kernel we can use the formula

$$\text{throughput} = \frac{B_r + B_w}{10^9 \cdot \text{execution time}},$$

where the execution time is in seconds (s) and B_r is the number of bytes (B) read from and B_w the number of bytes (B) written to the global memory. Let us give an example.

Example: vector-update (AXPY) Say we have written a `CUDA` kernel that computes $y \leftarrow \alpha x + y$, where $\alpha \in \mathbb{R}$ and $x, y \in \mathbb{R}^{2^{20}}$. Further, the kernel is a single-precision (`float`) implementation. When running the kernel we observe with the NVIDIA profiler that the vector-update is computed in 160 μs (the time taken by the device to run the kernel). The throughput is computed as follows. The number of bytes that must be read from the global memory is: $B_r = 2^{20} \cdot 2 \cdot 4$ $(= 8,388,608)$ since there are 2^{20} values of x and 2^{20} values of y and each takes 4 bytes. Likewise, $B_w = 2^{20} \cdot 1 \cdot 4$. Hence the throughput is: $(2^{20} \cdot 2 \cdot 4 + 2^{20} \cdot 4)/(10^9 \cdot 160 \cdot 10^{-6}) = 78.64$ GB/s.

16.2 Timing of GPU tasks

To be able to compute the floprate or throughput we need the execution time of the kernel. Below we briefly discuss what different options we have to do our timing.

16.2.1 Wall-clock timing

In Linux the wall-clock time can be measured through the function `gettimeofday()`. Although the function returns the current time in seconds and microseconds, the resolution of the measurements is usually in the order of milliseconds, depending on the environment (hardware, software). The following `C++` code can be used:

```
#include <sys/time.h>

int main()
{
   struct timeval start, stop;
   float time;

   gettimeofday(&start, NULL);

   // insert code that you want to time right here

   gettimeofday(&stop,  NULL);
   time = (stop.tv_sec - start.tv_sec) +
      1e-6 * (stop.tv_usec - start.tv_usec)

   return 0;
}
```

Because of the low accuracy for tasks that barely take a few milliseconds, usually one inserts a for-loop and runs the task many times, say 20 or even 1000 times if necessary, and computes the average. This is a basic trick to increase timing accuracy. However, it may occur that the compiler is so "smart" that it "sees" that nothing changes in the for-loop and just runs the loop once, whilst the time is still being divided by the number of iterations, resulting in very small and bogus numbers.

16.2.2 GPU events

Typical C++/CUDA code to time your CUDA kernels is the following.

```
int main()
{
   cudaEvent_t start_event, stop_event;
   float time_event;

   cudaEventCreate(&start_event);
   cudaEventCreate(&stop_event);

   cudaEventRecord(start_event, 0);

   // insert GPU code that you want to time right here

   cudaEventRecord(stop_event, 0);
   cudaEventSynchronize(stop_event);

   cudaEventElapsedTime(&time_event,
      start_event, stop_event);
   time_event /= 1e3;

   cudaEventDestroy(start_event);
   cudaEventDestroy(stop_event);

   return 0;
}
```

Again to increase accuracy you may consider to do the CUDA kernel multiple times. Also notice the synchronizing statement which is necesaary to ensure that all threads are finished when retrieving the stop time. (By the way, you should also check all CUDA statements for possible errors.)

16.2.3 NVIDIA profiler

Instead of incorporating timing functions in your CUDA program yourself, you may also consider to use NVIDIA's profiler. Typically this profiler can be found in: `../cuda/computeprof/bin`, and in Linux you can run it via `./computeprof`.

Although there are manuals available, the program is very user friendly and you should be able to figure things out by yourself.

The NVIDIA profiler comes also very handy to obtain properties and specifications of your own CUDA kernels or to get insight in library routines.

16.3 Throughput and coalesced memory — two little studies

To show how important coalesced memory is for optimal throughput we have conducted two experiments, namely:

1. copy data with a stride;
2. copy data with a shift.

Although similar studies can already be found in NIVIDIA's "CUDA Best Practices Guide" [18], we have chosen to incorporate these studies in this report as coalesced memory is a key concept in fast CUDA code. We want to make sure that the reader really understands what coalesced memory is and how throughput depends on "the degree of coalescence". Moreover, in [18] results for the Fermi architectures are missing.

16.3.1 Copy with a stride

A first quite common situation is that data is read from the global memory using a stride. For example, in case of a red-black numbering applied in a CG solver we have to do first some computations for the red nodes only followed by computations for the black nodes only. Assumed that the red and black nodes lie alternately in memory (*rbrbrb*...), computations are done with stride 2.

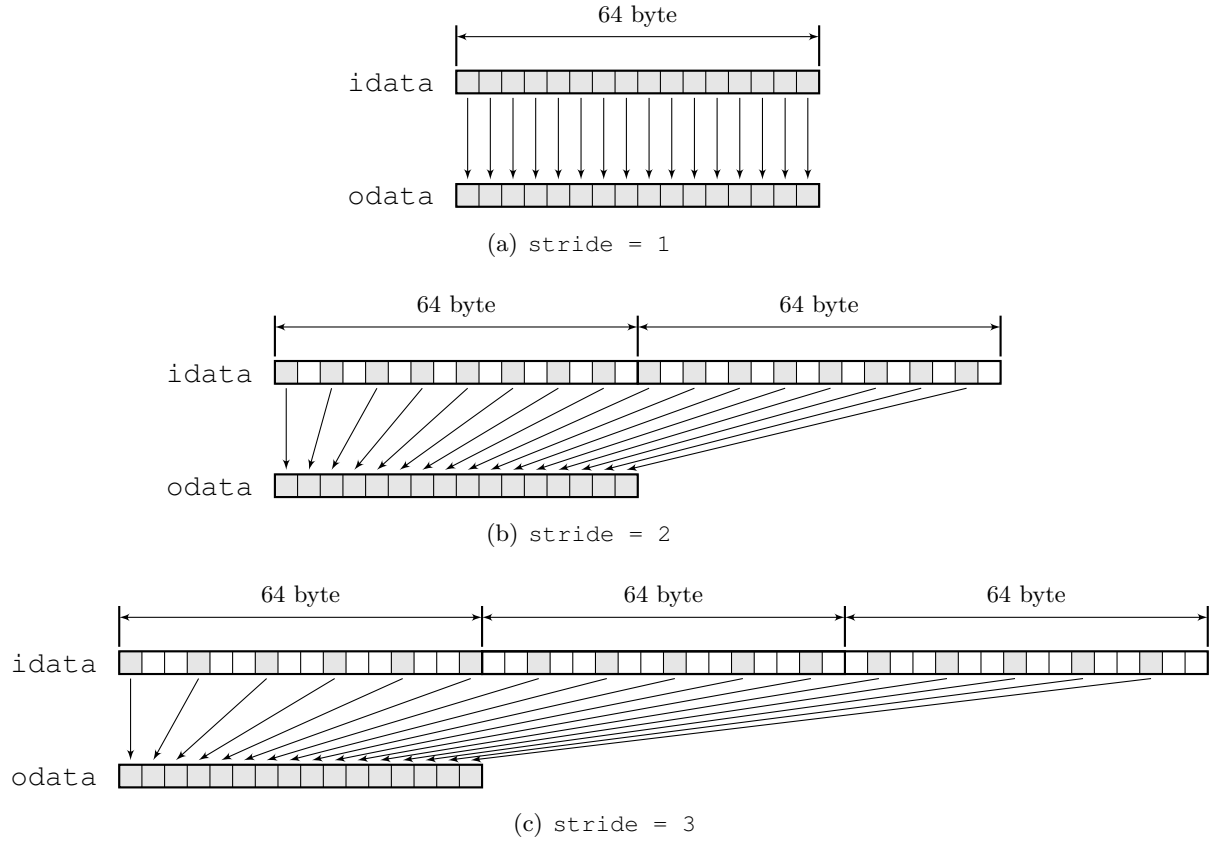


Figure 16.1: Copying of data with a stride.

To see that computations with a stride lead to poor performance, we do a simple experiment. Consider the following CUDA kernel.

```
__global__ void kernel_stride(float *odata, float *idata, unsigned int stride)
{
    unsigned int threadID = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    odata[threadID] = idata[threadID * stride];
}
```

The kernel is used to copy data with a stride, see Figure 16.1. Given an input array `idata` consisting of single-precision numbers (`floats`) the kernel copies data to an output array `odata`. Each thread takes care of one element in the output array `odata`.

In the figure the 64-byte boundaries are indicated. We see that when `stride = 1` that all elements read from `idata` lie directly next to each other in the global memory. In that case a single 64-byte memory transaction is used by each half-warp to access the data. This is optimal. All data that is read is relevant, and "can be used usefully". Therefore the throughput equals the overall throughput.

In case `stride = 2` the targeted data will be read from the global memory with a single 128-byte memory transaction. This is still fast; however, as we see half of the data cannot be used and thus means wasted throughput. In case `stride = 3` the targeted data will be read from the global memory with a 128-byte plus a 64-byte memory transaction. Although the overall throughput remains high the throughput degrades with increasing stride as more and more elements cannot be used. At some point (`stride = 16`) all memory reads will be done using 32-byte memory fetches; the minimum transaction size of the device is 32 bytes. So, for each thread 32 bytes will be fetched while only 4 bytes will be used, resulting in only 1/8th of the throughput relative to fully coalesced memory.

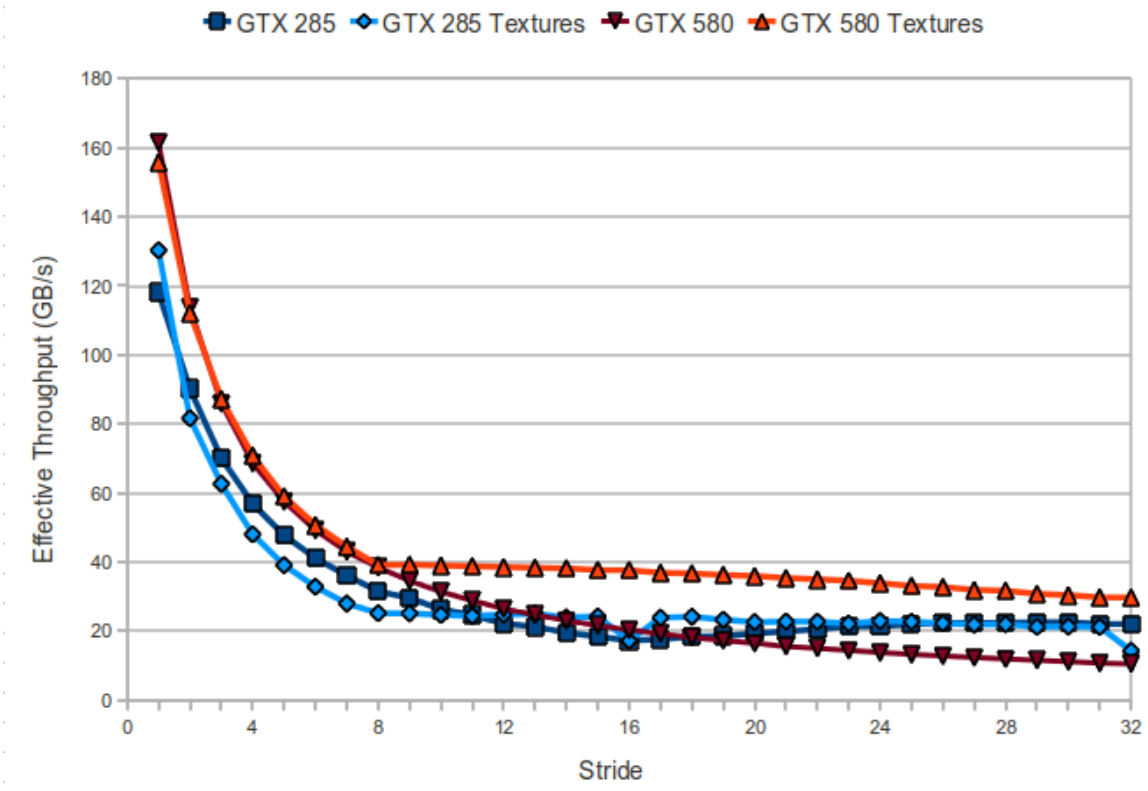


Figure 16.2: Throughput for the stride experiment.

In Figure 16.2 the throughput is plotted as function of the stride. Although for the previous discussion sufficient is to plot results for a stride up to 16 (from that point on

only/mostly 32-byte transactions will be used) we have chosen to plot results up to a stride of 32. There are two reasons for this. Firstly, in case of the latest architectures (Fermi, compute capability 2.x), the story presented above is a little bit more complicated as these architectures have (L1) cache; however, this is beyond the scope of this report. Secondly, we have also investigated how textures can be used to increase throughput and in that throughput keeps varying (decreasing) for strides larger than 16.

We see that in all cases the throughput decreases gradually with increasing stride. Although in some cases techniques can be used to overcome this throughput diminishing effect, e.g., by using shared memory, one should always strive for as much as possible coalesced memory.

16.3.2 Copy with a shift

Another common situation is that data must be read from the global memory with a shift, e.g., in case of a 5-point stencil the direct neighbours of a node may be required to compute the new value in that center node. To investigate the effect we use the following CUDA kernel.

```
__global__ void kernel_shift(float *odata, float *idata, unsigned int shift)
{
    unsigned int threadID = bx * Bx + tx;

    odata[threadID] = idata[threadID + shift];
}
```

Note the great similarity with the stride kernel above. In Figure 16.3 we have depicted what the kernel does.

For architectures with compute capability 1.3 the number of memory transactions issued for a half-warp of threads depends on the shift and whether the warp is even- or odd-numbered and throughput is correspondindly affected. In Figure 16.4 the throughput is plotted against increasing shift. For shifts of 0 or 16, each half-warp results in a single 64-byte memory transaction leading to highest throughput. For shifts of 1 through 7 or 9 through 15, even-numbered warps result in a single 128-byte transaction and odd-numbered warps result in two transactions: one 64-byte and one 32-byte transaction. Consequently, the throughput is slightly less. For shifts of 8, even-numbered warps result in one 128-byte transaction and odd-numbered warps result in two 32-byte transactions. The two 32-byte transactions, rather than a 64- and a 32-byte transaction, are responsible for the little blip at shifts of 8 and 24 in the figure. For architectures with compute capability 2.x (Fermi) memory transactions have always the width of a full cacheline which is 128 byte, or 32 byte is the L1 cache was disabled. Correspondinly, the throughput is almost constant for increasing stride.

The figure also shows that by using textures the throughput for the GTX 285 (compute capability 1.3) can be significantly boosted and becomes almost constant for varying stride. However, for the GTX 580 (compute capability 2.0) the throughput becomes slightly less by using textures.

So what we learn from this little study is that shifted global memory reads do not seriously harm the throughput; in case of older architectures we can use textures and in case of the latest architectures (Fermi) throughput is not affected at all.

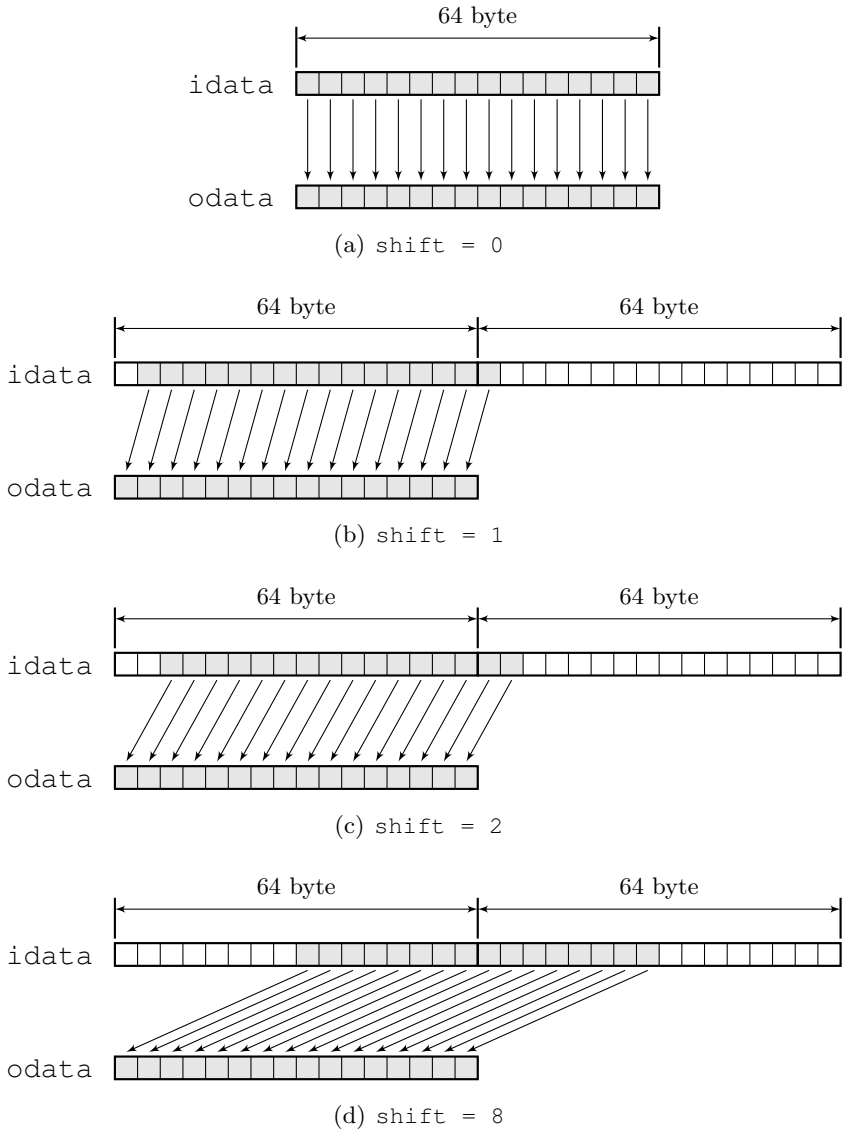


Figure 16.3: Copying of data with a shift.

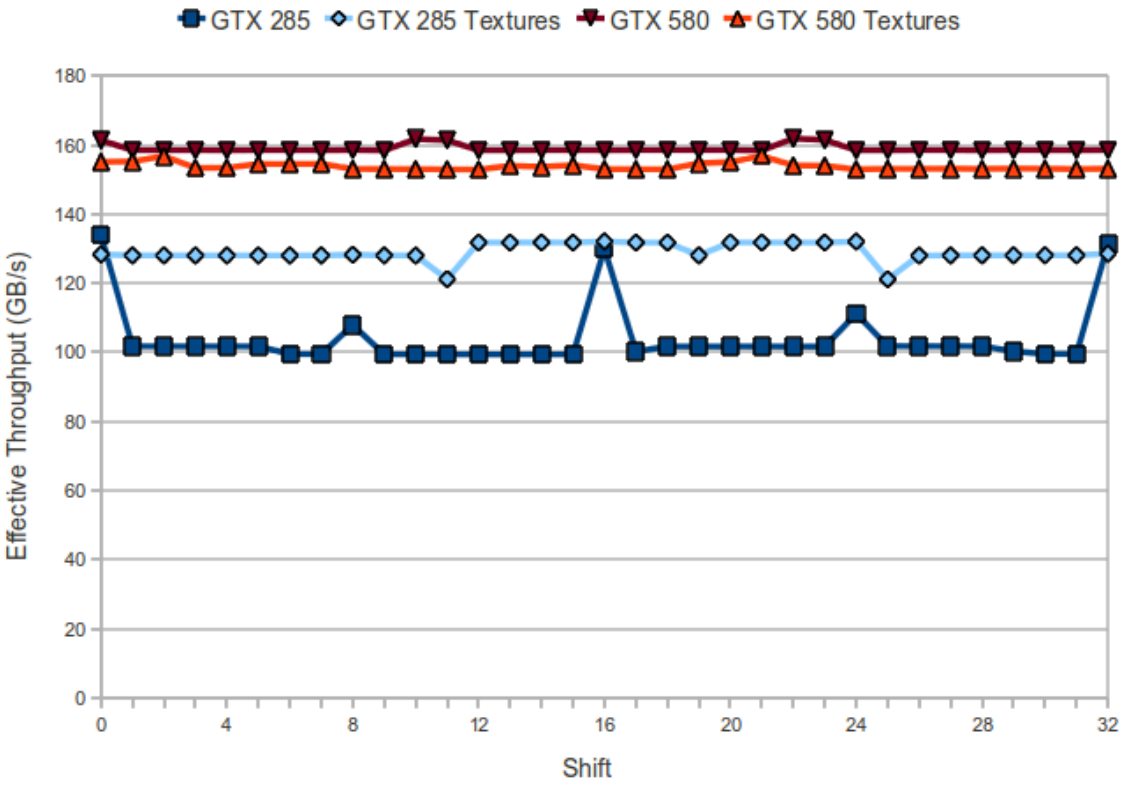


Figure 16.4: Throughput for the shift experiment.

16.4 Measuring speed up and Amdahl's law

The speed up S of parallel code compared to sequential code is computed via

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}},$$

where T_{seq} is the best time of sequential versions of the algorithm, and T_{par} is the time that the parallel version takes to do the job. Now in every program there is an intrinsically sequential part. With *Amdahl's law* we can estimate to what extent we may succeed to speed up code by parallelizing parts of it.

Amdahl's law is as follows: if f is the fraction of the algorithm that has to be executed sequentially and $1 - f$ is the fraction that is parallelizable, then if p processors are available the speed up S is bound by

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{T_{\text{seq}}}{fT_{\text{seq}} + (1-f)T_{\text{seq}}/p} = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}.$$

Hence, even if an infinite amount of processors p were available (so far scalability allows it) we would never reach a speed up more than $1/f$. Let us put this in perspective: say that only 5% of our code has to be performed sequentially, and thus 95% can be made parallel (which seems pretty good!), than unfortunately, the maximal speed up we can attain is "only": $1/0.05 = 20$ times. Always. Even if the parallel part runs at tremendous speeds and those 95% of computations can be done in almost zero time.

Although this seems so clear that mistakes are unthinkable, still researchers frequently misinterpret timing results. Commonly the following mistake is made: assume that an algorithm, say Conjugate Gradients (CG), is such that 98% of the time is consumed by matrix-vector products. Matrix-vector products lend themselves well for parallelization. Suppose that we succeed to speed up the matrix-vector products by a factor 1000, that is, computing them in parallel goes 1000 times faster then doing them sequentially. Did we now made CG 1000 times faster? No, unfortunately not; we have to apply Amdahl here: say that fully sequential CG takes 1 second for a particular problem size. Now assume 98% of the time is made faster, i.e. 0.98 seconds previously will now only take 0.00098 seconds. Hence the parallel execution time is: sequential time + parallel time $= 0.02 + 0.00098 = 0.02098$ seconds, hence the parallel version of CG is "only" $1/0.02098 = 47.7$ times faster than the sequential version. Note that this is close to the upper bound of: $1/f = 1/0.02 = 50$.

Chapter 17

Two important basic CUDA kernels

17.1 Sparse Matrix-vector products (SpMVs)

17.1.1 Introduction

In the CG algorithm one of the most time-consuming operations is the sparse matrix-vector product, abbreviated SpMV. In this section we discuss how an SpMV can be efficiently implemented in CUDA in the case that CG is applied for solving a system $Ax = b$ in which the matrix A is given by a 5-point stencil. For example, discretization of the Poisson problem on a 2D domain leads to such a 5-point stencil. In 2008 NVIDIA released a paper [3] in which Bell and Garland studied the performance of SpMVs for different storage schemes. Their source code was made public in the form of the CUSP library.

At this point one may wonder why we are going to spend time (and even a complete section) to a problem that can be considered "solved". One may argue that there are already efficient libraries-routines available (CUSP) that can be —*snap*— called just like that, or perhaps that the SpMV is "such a trivial" operation that does not belong in a report in which an advanced CUDA solver is the main subject.

We see this way different, actually we consider this section as absolutely indispensable. We believe that a throughout understanding of how an SpMV is computed on the GPU and seeing for yourself why it is done as it is done, is key in developing efficient CUDA code your own. Even more important, as the CG algorithm in the RRB-solver operates on the 1st Schur complement rather than on the system matrix itself, the SpMV takes a really different form than the "standard" one, so actually a library-routine cannot be used!

17.1.2 The DIA storage scheme

There are many different ways for storing a sparse matrix in the computer's memory, depending on where the nonzeros are in the matrix. Popular storage schemes are: the *coordinate* (COO) format, the *compressed sparse row* (CSR) format, the *diagonal* (DIA) format, the *ELLPACK* (ELL) format, and so on. Each format has its own benefits over other formats. For a matrix A given by a 5-point stencil, i.e., a matrix that contains only 5 nonzero diagonals, the DIA format is the most proficient. In Figure 17.1 an example is given for a 4×3 2D grid.

The places of nonzero elements in the matrix A follow from the dependencies of a grid point on its N, E, S and W neighbours (using the compass). Correspondingly, the main diagonal elements are indicated by a "c" (coming from "center"), and super-superdiagonal

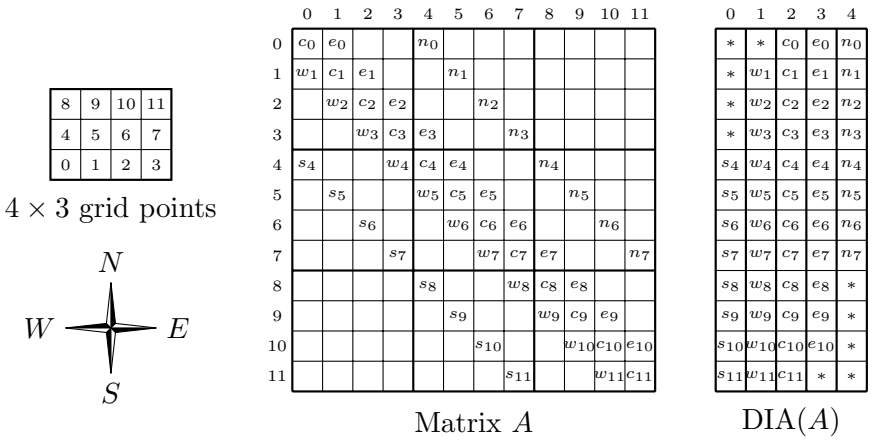


Figure 17.1: The diagonal (DIA) format for a matrix A resulting from discretization on a 4×3 grid.

elements are indicated by a "n" (coming from "north"), etcetera. The matrix A must be read as follows. For the ith node, the matrix A tells us (see the ith row) on what neighbours node i depends (the columns where we find a nonzero), and by how much the node depends on that neighbour is expressed in the particular coefficient s_i, w_i, e_i and n_i, of course node i can also depend on itself; this is just coefficient c_i.

On the right in Figure 17.1 it is shown how matrix A can be stored as 5 vectors of length 12. It is common use to pad the vectors (with zeros) so that on row i we find all dependencies for node i. The padding is indicated with the stars ($*$).

We know that the matrix A resulting from the discretization of the Poisson equation is a symmetric matrix, which means that, for all i, $e_i = w_{i+1}$ and $n_i = w_{i+N_x}$ (N_x being the dimension of the grid in the x-direction). Therefore, it is not necessary to store all 5 diagonals; we may just store the center, west and south (or the center, east and north) stencil which saves 2/5 memory. We have also seen that the matrix S resulting from the Variational Boussinesq model is symmetric as well (in fact it is SPD), so the problem is fully determined when just 3 diagonals are stored. However, we shall see in a moment that saving all diagonals can be beneficial when it comes down to performance. On the other hand, if memory is scarce one should offer some performance to obtain a good reduction in required memory.

One may have noticed that storing the nonzero diagonals of A is not sufficient; we also need some information where these diagonals are. Therefore, DIA(A) is accompanied with a vector that contains offsets, namely

$$\text{offsets} = [-N_x \quad -1 \quad 0 \quad 1 \quad N_x],$$

where N_x is the size of the grid in the x-direction.

17.1.3 Computation of an SpMV in case of a 5-point stencil

Let us now investigate how multiplying matrix A by a vector x is computed in terms of DIA(A). Consider Figure 17.2.

The small triangles point out what elements need to be multiplied and added. Because of padding the diagonals with 0's it is obviously unnecessary to invoke `if`-statements to ensure

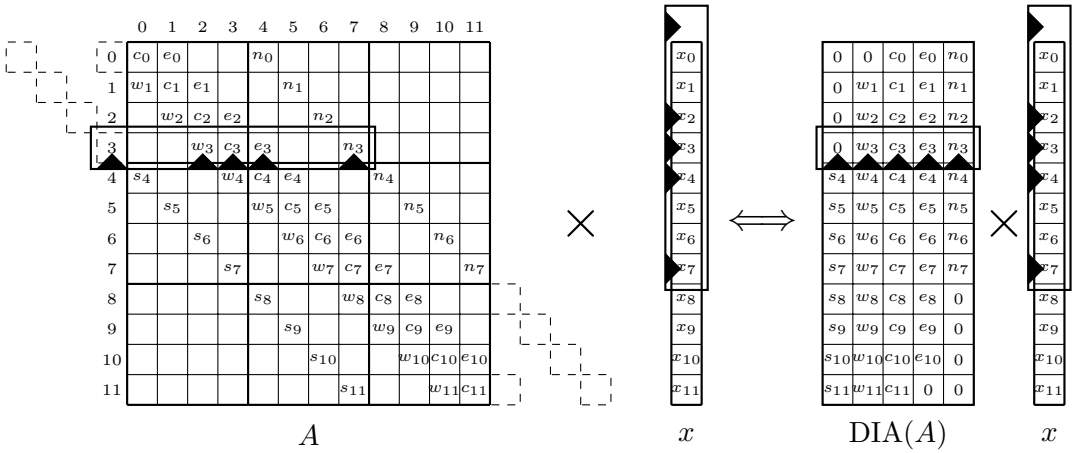


Figure 17.2: Multiplying matrix A by a vector x (SpMV).

that data from DIA(A) is read correctly. However, if the vector x is stored straightforwardly, that is, without any padding, for the first and final rows corresponding elements for each of the stencils s, s, n, e may not exist. This is indicated by that the bold rectangle partly falls outside vector x. Thus to ensure that the SpMV is computed correctly we have to incorporate an `if`-statement that checks whether a corresponding element in vector x exists. It won't take you long to figure out that suitable `C++` code to compute the SpMV can be:

```
float sum;
unsigned int column;

for (unsigned int row = 0; row < n; ++row) {
    sum = 0;

    for (unsigned int i = 0; i < 5; ++i) {
        column = row + offset[i];

        if (column >= 0 && column < n)
            sum += DIA[n * i + row] * x[column];
    }
}
```

in which it is assumed that all diagonals are stored in one big (linear) array; e.g., to address the west dependency for row `row` we search in the array at the location `n + row`, where `n` is the size of the matrix (here thus $n = 12$). In case of storing StC, StW and StS in `stC`, `stW` and `stS`, respectively, we can use the `C++` code:

```
float sum;
unsigned int column;

for (unsigned int row = 0; row < n; ++row) {
    // south
    column = row + offset[0];
    if (column >= 0)
        sum  = stS[row] * x[column];
```

```
    // west
    column = row + offset[1];
    if (column >= 0)
        sum += stW[row] * x[column];

    // center
    column = row + offset[2];
    sum += stC[row] * x[column];

    // east
    column = row + offset[3];
    if (column < n)
        sum += stW[row - 1] * x[column];

    // north
    column = row + offset[4];
    if (column < n)
        sum += stS[row - Nx1] * x[column];
}
```

This implementation has the advantage of less memory requirements since the west array is reused to compute the east contribution and the south array is reused to compute the north contribution. As we will see in a moment this memory advantage may come at a (small) price in CUDA.

17.1.4 Hints for an optimal CUDA implementation

With the presented C++ code snippets it is rather easy to come up with a first CUDA implementation of the SpMV routine. The `for`-loop is replaced by parallel threads; instead of marching through the elements of vector x we let each thread take care of computing a single element in the vector x. This makes sense as computing each row takes an equal amount of work and time in the presented sequential code. The first thing we have to make is a CUDA grid of n threads to deal with the vector x of length n.

As we learned a CUDA grid consists of thread blocks, and each thread block contains an equal amount of threads. Typical numbers for the number of threads per block are 64, 128, 256, but also numbers like 160 are frequently used. Why this many threads and not, say, 31? Well, recall that the numbers must obey the "optimal 16 spacing (64 bytes)"-rule (or even better, 128 bytes) to ensure that all reads from global memory are coalesced and thus optimal throughput is obtained.

So, if we have n elements to compute we need $\lceil n/Bx \rceil$ thread blocks. For n being a multiple of Bx (thus a multiple of e.g., 64, 128, 256) we need n/Bx thread blocks in which all threads will be busy; in case Bx does not divide n there will be one thread block with some idle threads.

We introduce a variable `threadID` to make sure that each thread points to a unique element as follows:

```
unsigned int threadID = blockIdx.x * blockDim.x + threadIdx.x;
```

Moreover, before starting the computations we must turn off some threads using an `if`-statement:

```
if (threadID < n);
```

which makes sure that only threads will do work for which work is really available. In case we store *stC*, *stW* and *stS* only a first implementation of the SpMV routine in CUDA may be:

```
__global__ void SpMV(float *y,
                     const float *stC, const float *stS, const float *stW,
                     const float *x,
                     const unsigned int Nx1, const unsigned int Nx2)
{
    unsigned int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int n = Nx1 * Nx2;

    float sum = 0;

    if (threadID < n)
    {
        sum = stC[threadID]            * x[threadID];       // center

        if (threadID - Nx1 >= 0)
            sum += stS[threadID]       * x[threadID - Nx1]; // south
        if (threadID - 1   >= 0)
            sum += stW[threadID]       * x[threadID - 1];   // west
        if (threadID + 1   <  n)
            sum += stW[threadID + 1]   * x[threadID + 1];   // east
        if (threadID + Nx1 <  n)
            sum += stS[threadID + Nx1] * x[threadID + Nx1]; // north

        y[threadID] = sum;
    }
}
```

The above routine thus computes the SpMV $y = Ax$ where $A \in \mathbb{R}^{n \times n}$ using three stencils only: *stC*, *stW* and *stS*. In Table 17.1 and Table 17.2 the throughput and performance is reported for different n on the GeForce GTX 285 and GeForce GTX 580, respectively.

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	103	22.9	112.0
500	1,000	500,000	204	22.1	107.8
1,024	1,024	1,048,576	393	24.0	117.4
1,000	1,500	1,500,000	660	20.5	100.0
2,048	2,048	4,194,304	1572	24.0	117.4

Table 17.1: Throughput and performance for different n in case of using 3 stencils (GeForce GTX 285, 256 threads per block).

We mentioned earlier that the memory advantage may come at a (small) price. Let us confirm that right now. In case that we want to use all five stencils *StC*, *StN*, *StE*, *StW*, *StS* suitable `CUDA` code may be:

```
__global__ void SpMV(float *y,
                     const float *stC, const float *stS, const float *stW,
                     const float *stE, const float *stN, const float *x,
                     const unsigned int Nx1, const unsigned int Nx2)
{
```

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	44	53.6	262.1
500	1,000	500,000	83	54.2	265.1
1,024	1,024	1,048,576	166	56.9	277.9
1,000	1,500	1,500,000	237	57.0	278.5
2,048	2,048	4,194,304	649	58.2	284.4

Table 17.2: Throughput and performance for different n in case of using 3 stencils (GeForce GTX 580, 256 threads per block). Note that the throughput is higher than the device's bandwidth; about 250 GB/s versus a theoretical limit of 193 GB/s, see Section 16.1.2. This is due the usage of faster L1 cache

```
    unsigned int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int n = Nx1 * Nx2;

    float sum = 0;

    if (threadID < n)
    {
        sum = stC[threadID]      * x[threadID];       // center

        if (threadID - Nx1 >= 0)
            sum += stS[threadID] * x[threadID - Nx1]; // south
        if (threadID - 1   >= 0)
            sum += stW[threadID] * x[threadID - 1];   // west
        if (threadID + 1   <  n)
            sum += stE[threadID] * x[threadID + 1];   // east
        if (threadID + Nx1 <  n)
            sum += stN[threadID] * x[threadID + Nx1]; // north

        y[threadID] = sum;
    }
}
```

This code leads to the following results on the GTX 285 and GTX 580:

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	97	24.3	118.9
500	1,000	500,000	190	23.7	115.8
1,024	1,024	1,048,576	371	25.4	124.4
1,000	1,500	1,500,000	597	22.6	110.6
2,048	2,048	4,194,304	1487	25.4	124.1

Table 17.3: Throughput and performance for different n in case of using 5 stencils (GeForce GTX 285, 256 threads per block).

We notice a slight increase in performance, about 6%, for the GTX 285. This comes from the fact that in the previous code not all global memory reads for the stencils were coalesced, leading to wasted throughput; on even older architectures (e.g., compute capability 1.0) this effect is much more noticable, but nowadays it is unlikely that one uses a device older than a GTX 285 (compute capability 1.3) to perform scientific computations. In case of the GTX 580,

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	49	48.1	235.4
500	1,000	500,000	91	49.5	241.8
1,024	1,024	1,048,576	185	49.8	243.5
1,000	1,500	1,500,000	259	52.1	254.8
2,048	2,048	4,194,304	732	51.6	252.1

Table 17.4: Throughput and performance for different n in case of using 5 stencils (GeForce GTX 580, 256 threads per block). Note that the throughput is higher than the device's bandwidth; about 250 GB/s versus a theoretical limit of 193 GB/s, see Section 16.1.2. This is due the usage of faster L1 cache.

a Fermi device, we actually see a slight decrease in performance. This comes from the fact that when using three stencils only, more data can be read from (the much faster) L1 cache as there are only three different arrays.

To get a real significant boost in performance on the GTX 285 we should use *textures*. We observe that the elements in vector x that are accessed lie close to each other in memory, especially the elements `x[threadID - 1]`, `x[threadID]` and `x[threadID + 1]`. We have pointed out in Section 15.2.3 that using textures can boost the throughput as nearby data is pre-cached in the *texture cache* when a particular element is retrieved from the global memory. Let us give it a try. We rewrite our `CUDA` kernel into the following.

```
__global__ void SpMV(float *y,
                     const float *stC, const float *stS, const float *stW,
                     const float *stE, const float *stN, const float *x,
                     const unsigned int Nx1, const unsigned int Nx2)
{
    unsigned int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int n = Nx1 * Nx2;

    float sum = 0;

    if (threadID < n)
    {
        sum = stC[threadID]      * tex1Dfetch(texRef, threadID);       // center

        if (threadID - Nx1 >= 0)
            sum += stS[threadID] * tex1Dfetch(texRef, threadID - Nx1); // south
        if (threadID - 1   >= 0)
            sum += stW[threadID] * tex1Dfetch(texRef, threadID - 1);   // west
        if (threadID + 1   <  n)
            sum += stE[threadID] * tex1Dfetch(texRef, threadID + 1);   // east
        if (threadID + Nx1 <  n)
            sum += stN[threadID] * tex1Dfetch(texRef, threadID + Nx1); // north

        y[threadID] = sum;
    }
}
```

We see that elements in the vector x are no longer straightforwardly loaded from global memory but via textures. The command

```
tex1Dfetch(texRef, threadID);
```

accesses the element that lies at location `threadID` in the array x. `texRef` is the (1D) texture bound to the (1D) array x. Of course this must be done prior to the kernel invocation:

```
cudaBindTexture(NULL, texRef, x);
SpMV <<< nblocks, NUM_THREADS >>> (y, stC, stS, stW, stE, stN, Nx1, Nx2);
```

It is common use to define the number of threads aforehand (pre-processor) via something like `#define NUM_THREADS 256`. The number of threads blocks, `nblocks`, is computed on the spot. Let us discuss by how much the performance is increased. In Table 17.5 and Table 17.6 we have listed the results.

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	82	28.8	140.7
500	1,000	500,000	140	32.1	157.1
1,024	1,024	1,048,576	275	34.3	167.7
1,000	1,500	1,500,000	388	34.8	170.1
2,048	2,048	4,194,304	1039	36.3	177.6

Table 17.5: Throughput and performance for different n in case of using 5 stencils using textures for vector x (GeForce GTX 285, 256 threads per block). Note that the throughput is higher than the device's bandwidth; about 170 GB/s versus a theoretical limit of 158 GB/s. This is due the usage of faster texture cache.

$Nx1$	$Nx2$	n	Time (μs)	Performance (Gflops/s)	Throughput (GB/s)
512	512	262,144	50	47.2	230.7
500	1,000	500,000	93	48.4	236.6
1,024	1,024	1,048,576	186	50.7	248.1
1,000	1,500	1,500,000	263	51.3	251.0
2,048	2,048	4,194,304	736	51.3	250.7

Table 17.6: Throughput and performance for different n in case of using 5 stencils using textures for vector x (GeForce GTX 580, 256 threads per block). Note that the throughput is higher than the device's bandwidth; about 250 GB/s versus a theoretical limit of 193 GB/s. This is due the usage of faster texture cache.

We observed for serveral kernels throughput numbers that are higher than the bandwidth of the device. This is caused by L1 cache (Fermi) and/or texture cache. Exploiting textures yields a performance boost of about 45% for the GTX 285 and we see that using textures for the GTX 580 yield no real difference, because L1 cache benefit is traded against the L2 cache benefit. Summarizing, generally, that is, when our CUDA code must do well on all architectures, using textures is a technique not to forget, and whenever we observe that nearby data is accessed consecutively, we may try the "textures-trick".

Finally we would like to mention that our GTX 285 numbers are just as good as the (single precision) numbers presented in [3] (they used about the same device: a GeForce GTX 280). We have thus shown that for getting good (read: optimal) performance CUDA kernels can be remarkably simple of nature! In fact, in [3], the CUDA code they used is (in essence) the same as our code. This is not surprising as much more clever code is almost impossible. However, we may try to write code that does not contain any `if`-statements (so to avoid thread divergence), e.g., by using padding. This may save up to another 5% in computation time (but also requires some extra storage).

17.2 Work efficient parallel sum reduction

17.2.1 Introduction

Suppose we have an array x of n elements $x_i (i = 1, 2, \ldots, n)$. The goal is compute the sum

$$\sum_{i=1}^{n} x_i := x_1 + x_2 + \cdots + x_n$$

as fast as possible in parallel. Computing a (big) sum is for example part of computing an inner (dot) product, i.e., for real vectors x and y of length n the dot product, denoted by $\langle x, y \rangle$, is computed as

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

Computing the sum sequentially is trivial; we just loop over all elements and each time add the value to the sum previously computed. In C++ we could use something like

```
float sum = 0;
for (unsigned int i = 0; i < n; ++i)
    sum += x[i];
```

for a single-precision array `x` consisting of `n` elements. We see that the total number of flops required is n. It is obvious that this number of flops is optimal.

We now show a parallel algorithm that is *work efficient*, that is, the parallel algorithm computes the sum also using a total of n flops, no any flops more than that. For ease, suppose we have an array x of n elements such that n is a power of 2, i.e., $n = 2^d$ for some $d > 0$. A suitable work efficient sum reduction algorithm in C++ could be

```
for (unsigned int i = 0; i < log(n)/log(2); ++i)
    for (unsigned int k = 0; k < n; k += pow(2,i+1))
        x[k] += x[k + pow(2,i)];
```

where `pow(2,i)` is some implementation for 2^i, and note that `log(n)/log(2)` actually computes $d = \log_2 n$. It is common to speak about the *step complexity*, the step complexity of the above sum reduction algorithm is thus $\mathcal{O}(\log n)$. Of course the above code is still sequential code, but actually the inner loop may be performed by n/k processors in parallel. What we see is that at each of the d stages two elements are added, and, moreover, the higher the level the less of these pairs remain (the spacing is increased by a factor 2 each level up), so in case of a parallel performed inner loop the number of active processors decreases by a factor 2 each time we go a level up. It is easily found that the *work complexity* of this algorithm is $\mathcal{O}(n)$, or more precisely $n-1$, thus work efficient, as

$$\sum_{i=0}^{\log_2 n-1} \frac{n}{2^{i+1}} = \sum_{i=0}^{d-1} \frac{2^d}{2^{i+1}} = \sum_{i=0}^{d-1} 2^{d-i-1} = \sum_{j=0}^{d-1} 2^j = 2^d - 1 = n - 1.$$

In Figure 17.3 the algorithm is applied to an array x of length $n = 16$.

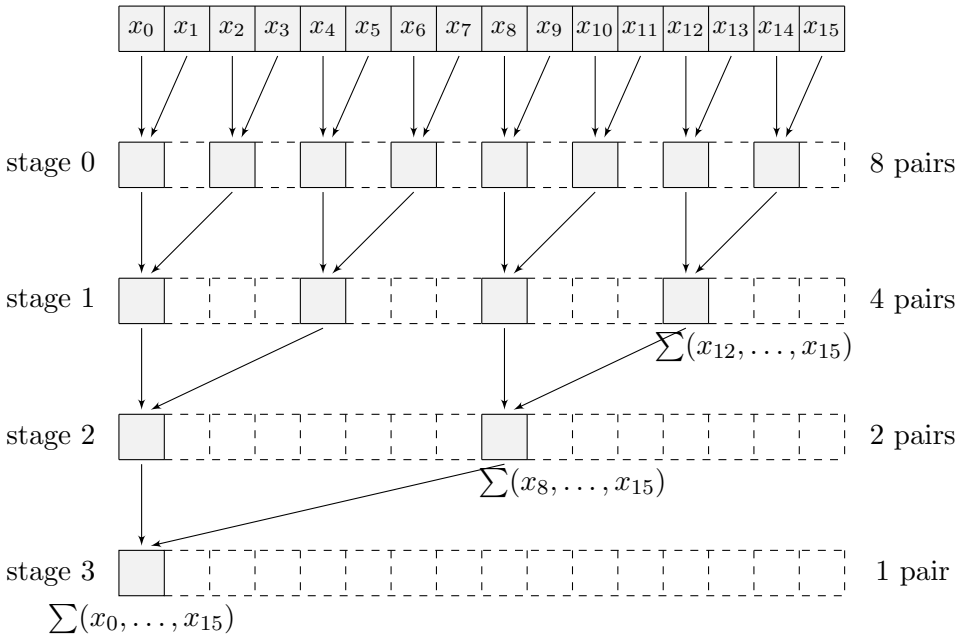


Figure 17.3: Work efficient sum reduction algorithm for $n = 16$.

17.2.2 Time and cost efficiency of the parallel sum reduction algorithm

The difference between 1 processor and p processors in parallel performing the same sum reduction above is of course expressed in terms of computing time. Here the notion *time complexity* comes into play. For 1 processor the time complexity is obviously $\mathcal{O}(n)$, whereas if we have as many physical processors as there are elements, so when $p = n$, then the time complexity is just $\mathcal{O}(\log n)$. In case $p < n$ we follow a slightly different strategy.

For $p < n$ (usually $p \ll n$) physical processors in parallel the idea is to let in the first stage the p processors each sum up $\lceil n/p \rceil$ elements whereafter the remaining p partial sums are summed up using the parallel sum reduction algorithm above, which takes another $\lceil \log p \rceil$ stages. So on total the time complexity becomes $\mathcal{O}(n/p + \log p)$.

More important is to look at the *cost efficiency* of the algorithm. A parallel algorithm is said to be cost efficient if its asymptotic running time multiplied by the number of processers involved in the computation is comparable to the running time of the best sequential algorithm. The cost of the algorithm is thus defined as

$$\text{cost} = \#\text{processors} \times \text{running time per processor}.$$

So, what cost efficiency basically tells us is how efficient we use the hardware. Looking at our parallel sum reduction algorithm we want to obtain a cost of $\mathcal{O}(n)$ since this is the time complexity of the sequential algorithm above, which is also best. Given that there are enough processors available, we may decide to use as many processors as there are elements, so taking $p = n$. We have seen that in case $p = n$ the time complexity of the parallel sum reduction is $\mathcal{O}(\log n)$. As there are $\mathcal{O}(n)$ processors, this choice leads to a cost of: $\mathcal{O}(n) \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$, thus not cost efficient, and thus, in some way, "spilling hardware". A better choice is to pick $p = n/\log n$ processors, in which case each processor does $\mathcal{O}(\log n)$ sequential work. Then all $\mathcal{O}(n/\log n)$ processors cooperate for $\log n$ stages leading to a cost of $\mathcal{O}(n \log n) \times \mathcal{O}(\log n) = \mathcal{O}(n)$, which is cost efficient. So, the "optimal" number of processors that we use for the parallel sum reduction algorithm would be $n/\log n$. Keep this in mind, because we will come back to it (and use it) in the next section.

17.2.3 Hints for an optimal `CUDA` implementation

From Figure 17.3 one problem becomes already obvious when implementing the parallel sum reduction algorithm in `CUDA` : the higher we go up the tree the more processors become idle, or probably better said, cannot perform useful computations anymore. We shall see in a moment that besides this "standard" problem in parallelizing algorithms there are more problems due to the GPU's architecture.

By studying the algorithm and the fact that the computations should be performed on the GPU as much as possible, it becomes evident that we should exploit the shared memory of the GPU. The shared memory is the fastest way that threads can use to communicate data. The idea is to first load a chunk of (slow) global memory into the (fast) shared memory whereafter a set of threads compute the sum over the elements contained in this chunk. So somewhere in the beginning of a sum reduction's kernel we should find code like

```
__global__ void sum_kernel(float *x, ... )
{
    unsigned int loc = ...
    unsigned const int threadID = ...
```

```
    __shared__ REAL sm[CHUNK_SIZE];

    sm[threadID] = x[loc];
    __syncthreads();

    ...
}
```

What this code does is determining a specific location `loc` of data in the larger array `x`, whereafter a bunch of threads, `BUNCH_SIZE` many, each having unique number `threadID`, together load a chunk of data from the global memory array `x` into the shared memory `sm`. In the end we find the synchronization command `__syncthreads()` to make sure that all threads are finished loading data before the actual sum reduction computations are started. However, in a moment we shall see that there is a more efficient (much faster) mechanism to load data from global memory into shared memory yielding more throughput, namely by letting each thread do more work. At this point you may already observe that this may have something to do with cost efficiency.

Before delving into more implementation details, let us first return to the hardware specific problems that occur when implementing the parallel sum reduction in CUDA. For that we need a somewhat bigger example than in Figure 17.3, so see Figure 17.4 for the first phases of a four times larger sum. In this figure we have also depicted the global to shared memory transfer as discussed earlier.

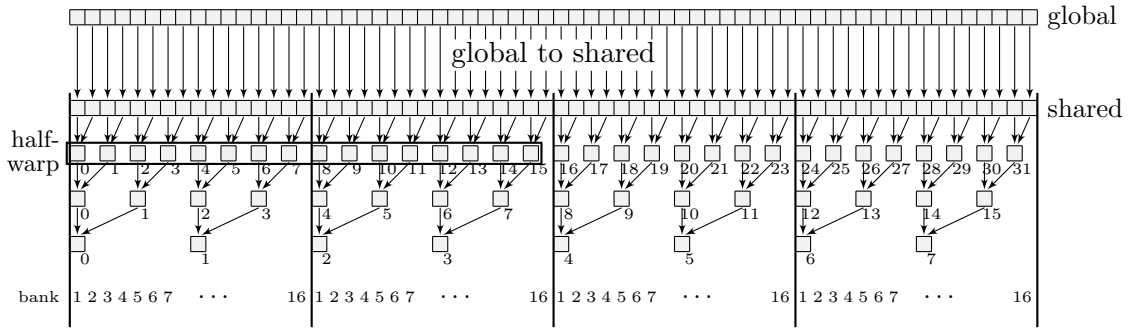


Figure 17.4: Work efficient sum reduction algorithm using 64 threads. However, the algorithm is yet efficient on the GPU's hardware because of shared memory bank conflicts or warp divergence.

In Figure 17.4 we have illustrated the phenomenon 'shared memory bank conflicts'. As we know the shared memory consists of multiple banks, see Section 13.5, i.e., we have learned that devices with compute capability 1.x have 16 banks and devices with compute capability 2.0 and 2.1 have 32 banks, which are interleaved with a granularity of 32bit (so byte $0-3$ fall in bank 1, $4-7$ in bank 2, ..., byte $64-69$ again in bank 1, etcetera). In Figure 17.4 we have depicted a device with 16 banks which thus means that 16 consecutive 4-byte words each fall in a different bank. Bank conflicts arise if multiple threads in the same half-warp (= 16 threads) access different words in the same bank. In the figure we see that if 32 threads (= 2 half-warps) were assigned to the data as indicated, there would be quite some bank conflicts; for example, thread 0 and thread 8 would both read from bank 1. The same holds for thread 1 and thread 9; they would both read from bank 2, etcetera. In the next stage there would

be even more bank conflicts, see the figure; in that case threads 0, 4, 8 and 12 would read different 4-byte words from bank 1. To avoid bank conflicts we may force half of the threads in the half-warp to do nothing, e.g., by using `if`-statements; however in that case we will face warp divergence, that is, threads in the same half-warp are instructed to do different things, in our sum reduction algorithm this would lead to idle threads. So, summarizing, we either face bank conflicts or warp divergence if we were to straightforwardly port the proposed sum reduction algorithm to `CUDA`. Fortunately, the solution to overcome both is pretty simple.

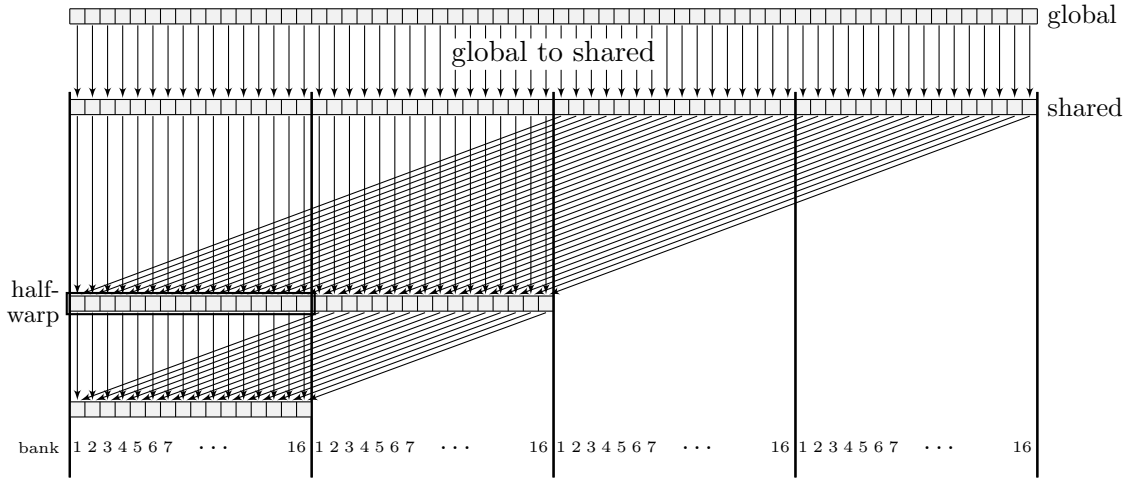


Figure 17.5: Work efficient sum reduction algorithm using 64 threads. This version of the algorithm is optimal for `CUDA`, because there are no shared memory bank conflicts and, except from the last half-warp (= 16 threads), there is no warp divergence.

Consider Figure 17.5. This figure shows how the sum reduction algorithm should be rephrased so that no bank conflicts occur, and warp divergence is reduced to a minimum. What we see is that each thread in the half-warp reads from a different memory bank, so no bank conflicts at all. Also, the warp scheduler is able to schedule half-warps in which all threads are busy except from the final stages in which the last 16 elements are cumulated. For the 64 elements example in Figure 17.5 the warp scheduler would schedule only 3 half-warps to come up with the total sum, whereas in Figure 17.4 the number of half-warps would be many more when trying to avoid bank conflicts.

The heart of the sum reduction algorithm should thus contain code like:

```
for (unsigned int k = CHUNK_SIZE / 2; k > 0; k >>= 1) {
    if (threadID < k)
        sm[threadID] += sm[threadID + k];
    __syncthreads();
}
```

which does what we have explained above. Note that in the last runs warp divergence occurs. Warp divergence cannot be overcome; however, we can avoid synchronization where it is no longer needed (since instructions are SIMD synchronous within a warp) by unrolling the last warp (= 32 threads). Moreover, we may decide to let threads do useless work so that also `if`-statements are no longer needed as well. The following code snippet contains these small improvements:

```
for (unsigned int k = size / 2; k > 32; k >>= 1) {
    if (threadID < k)
        sm[threadID] += sm[threadID + k];
    __syncthreads();
}
if (threadID < 32) {
    sm[threadID] += sm[threadID + 32];
    sm[threadID] += sm[threadID + 16];
    sm[threadID] += sm[threadID + 8];
    sm[threadID] += sm[threadID + 4];
    sm[threadID] += sm[threadID + 2];
    sm[threadID] += sm[threadID + 1];
}
```

In the end of the sum reduction we have to write the result back to the global memory, it is natural to let the first thread do that, i.e., the thread with `threadID = 0`.

Finally we come back to the issue how to read from global memory as efficient as possible, that is, how should employ the threads with work so that maximal throughput is obtained. In Section 17.2.2 we have seen how we can alter the cost of the parallel sum reduction algorithm by varying the number of processors assigned to the task. Although threads are not the same as physical processors, or streaming processors (SPs) in the case of a GPU, the notion of cost efficiency can stear us how to use the hardware as optimal as possible. When there are more threads than SPs the warp scheduler is forced to launch blocks of threads sequentially. So to minimize time we should let each thread do an optimal amount of work, targeting a kernel that is overall as most cost efficient as possible. Threads are launched in blocks, and each streaming multiprocessor (SM) can launch one or multiple thread blocks. The "optimal" number of threads within each block to sum up m elements, where m is a fraction of the total array of length n, is something like $m/\log m$ as we pointed out in Section 17.2.2. Or, turned around, given that each thread blocks contains p threads, there is an "optimal" number of elements that each thread should sum up sequentially before the reduction stages are initiated, i.e., the mechanism that we have called the parallel sum reduction algorithm. The moral of this section is that we should let each thread do more work to get more throughput and thus a faster kernel. The "optimal" number is more or less found by trial and error and depends on the specific hardware.

The `CUDA` kernel for the parallel sum reduction should thus rather start with loading data from the global memory into shared memory as follows:

```
__global__ void sum_kernel(float *x, ... )
{
    unsigned int loc = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned const int threadID = loc;

    __shared__ REAL sm[NUM_THREADS];

    float sum = 0;
    for (unsigned int i = 0; i < OPTIMAL_NUM_ELTS; ++i) {
        sum += x[loc];
        loc += OFFSET;
    }

    sm[threadID] = sum;
    __syncthreads();
```

This piece of code is used as follows: given an array `x` and a number of thread blocks `NUM_BLOCKS`, each of the `NUM_THREADS` threads accumulates `OPTIMAL_NUM_ELTS` elements and stores the result in shared memory at location `threadID`. We feel that a figure is needed to make things easier to understand, so see Figure 17.6. It must be said: in this figure the problem is very small and the number of threads and things like that is far from realistic; however, the figure contains what is essential, and you can scale things up to arbitrary large sizes. In the figure it is supposed that we have 4 thread blocks (so `NUM_BLOCKS = 4`), each consisting of 4 threads (so `NUM_THREADS = 4`). The total array `x` has length 64, and hence each thread is going to sum up 4 elements (so `OPTIMAL_NUM_ELTS = 4` which is assumed to be the number that does the trick).

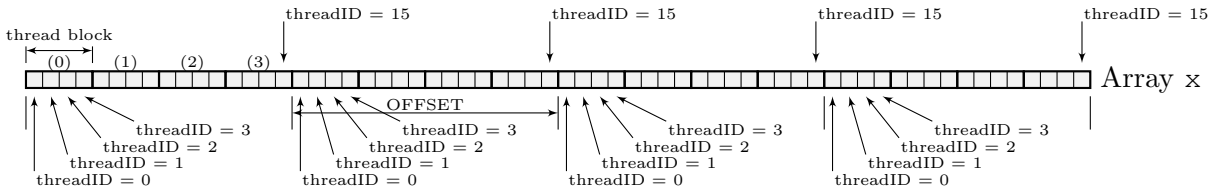


Figure 17.6: Mechanism for a cost efficient parallel sum reduction algorithm and maximal throughput.

Although there are different choices for reading data from global memory in the code snippet it is assumed that each thread reads data elements that lay `OFFSET` apart in global memory. Of course, the striding should be such that the memory reads occur all coalesced, e.g., `OFFSET` should be a multiple of 16 (= 64 bytes in case of floats). We see that each thread (of a total of $4 \cdot 4 = 16$ threads) is assigned to a unique spot in the first part of array `x` with the line `loc = blockIdx.x * blockDim.x + threadIdx.x;` Each thread has a unique label `threadID` which is constant. `blockDim.x` returns the size of the thread block, hence the number `NUM_THREADS` which is 4. Likewise, `blockIdx.x` and `threadIdx.x` return the location of a particular thread block in the `CUDA` grid and the location of a particular thread in the thread block, respectively. Note that for this example the kernel would be invoked by something like

```
sum_kernel <<< 4, 4 >>> (x, ... );
```

In the `for`-loop at each iteration the location where a thread points to is updated; in the next iteration the thread should read data at `loc + OFFSET` which is the next element to be added. The other obvious choice is to let the very first thread take care of the elements x_0, x_4, x_8, x_{12}. In this figure this will not work because they lay just 4 spaces (= 16 byte) apart which goes against the "optimal 16 spacing (64 byte)" rule for coalesced global memory reads; however, for thread blocks that contain a multiple of 16 threads (which is a more realistic and quite typical number for `CUDA` programming), this way of addressing data is the other good option to march through the array and pick up elements.

For complete source codes and optimal sum reduction kernels one can study the NVIDIA GPU Computing SDK's example code "reduction" which incorporates all aforementioned hints and even more techniques to get the fastest sum reduction algorithm in `CUDA`. This code belongs to the study performed by Mark Harris, see [8]. To see what order of speed ups we may think of by using the various hints we listed above, we refer to Harris' results for a sum reduction counting 4 million (2^{22}) elements. By taking all his suggested improvements into

account (all of ours plus some more) we can achieve a speed up of a factor $20\times$(!) compared to a straightforward (naive) implementation.

Part IV

PCG SOLVERS

Chapter 18

General comments that apply to all the PCG solvers in the `lin_wacu` software

18.1 Termination criterium

The termination criterium in all[1] the PCG solvers (RRB, Cg4, Nop, CUDA) that are used in the `lin_wacu` software is a relative criterium based on the preconditioned residual $z_i = M^{-1}r_i$, namely: stop the iterative process when

$$\langle r_i, z_i \rangle_2 = \|r_i\|^2_{M^{-1}} \leq (\|r_0\|^2_{M^{-1}} + 1) \cdot (psitol)^2.$$

In `C++` this is implemented as follows.

```
// Earlier computed: rho = <r, z>, where r = b - S1*x and z comes from solving Mz = r

// stop criterium
REAL stop = (rho + 1) * FloatUtils::square(tolerance);

// here the CG iterations start
while (rho > stop) && (iter < MAXITER))
{
    ...
}
```

A while loop is used to determine when to stop. The iterative process is stopped either when the termination criterium is fulfilled or the maximal number of iterations is exceeded.

[1] We have to point out something. For most solvers (Nop, IPDIAG, Cg4) the dot product is computed over all elements in r and z. As there are n unknows the dot product is computed by

$$\text{dot} = \sum_{j=1}^{n} r_j \cdot z_j.$$

However, for the RRB-solver only half of the nodes are used, namely the first level red nodes only, which leads to an initial preconditioned residual that is about half in magnitude. However, this is done consistently throughout the RRB-solver and therefore it does not make any difference. All solvers reduce the initial residual by a factor $1/(psitol)^2$, e.g., when $psitol$ =1e-5 the initial residual is reduced by a factor 10^{10}.

We see that the criterium is of the form: $\|r_i\|_{M^{-1}} \leq \varepsilon \|r_0\|_{M^{-1}}$. The relative criterium has the property that it is scaling invariant which implies that mesh-refinement does not lead to a more stringent criterium.

Using $\|r_0\|_{M^{-1}}$ rather than $\|r_0\|_2$ is logical as $\|r_0\|_2$ is not directly available in an efficient PCG implementation. In Figure 18.1 we have plotted $\langle r_0, z_0 \rangle_2$, $\|r_0\|_2^2$ and $\|b\|_2^2$ for the first 1000 time steps in the Plymouth 1.5M test problem.

We see that $\|r_0\|_{M^{-1}}$ is close to $\|r_0\|_2$, and, as the field gets filled with more and more waves, the right-hand side b in $S\psi = b$ becomes more and more "difficult" to solve for, which translates in a larger Euclidean norm.

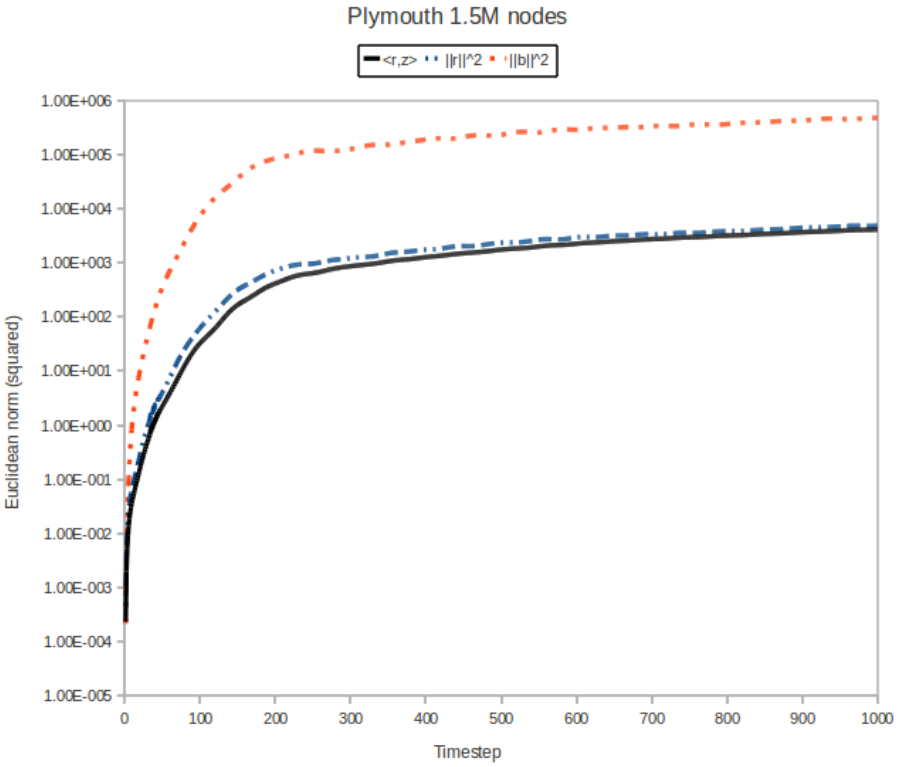


Figure 18.1: Evolution of $\langle r_0, z_0 \rangle_2$, $\|r_0\|_2^2$ and $\|b\|_2^2$ for the first 1000 time steps in the Plymouth 1.5M nodes test problem.

At the moment the tolerance allowed, i.e., *psitol* is given per test problem in the corresponding `.par` files. Typically we have: *psitol* = 1e-5 (most test problems: lin_wacu, IJssel, Plymouth, Port Presto) or *psitol* = 2e-6 (open sea).

As the required number of CG-iterations directly depends on *psitol*, one may consider taking a larger *psitol* to get a faster solver. However, one must be careful by doing this as at some point the problem will not be solved correctly anymore across time and we will observe that the wave pattern starts deviating from the true solution and it may even explode.

Chapter 19

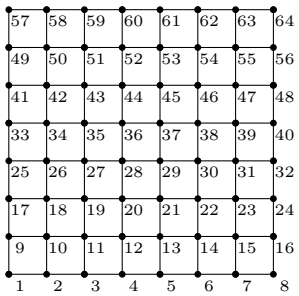
The C++ and CUDA RRB-SOLVER

19.1 RRB-solver basic concepts

In the next sections we shall explain the underlying concepts of the RRB-solver. We shall do this in a very graphical and intuitive way (lots of figures) to make sure that the reader really understands, and sees what is going on. Once the basic concepts are well understood, we can gradually increase the number of details and introduce the underlying mathematics. However, it never gets really difficult, as we have tried to avoid difficult details as much as possible. As a consequence the next sections are not complete or mathematically precise. For a complete mathematical desription we would like to refer to [4]. Also the Master thesis from Elwin van 't Wout [28] and the internal document "The RRB-preconditioner" (only available at MARIN) contain additional valuable information.

19.1.1 Repeated Red-Black numbering

The fundamental underlying idea of the RRB-solver is applying a red-black numbering repeatedly, hence the name RRB (Repeated Red-Black). Let us see how this works. Consider a grid with 8×8 unknowns. Although it is unrealistically small, it is a perfect size to illustrate the RRB-numbering.



(a) As actual points.

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

(b) As a 2D-array.

Figure 19.1: An 8×8 grid with lexicographic numbering.

Consider Figure 19.1 which shows the 8×8 grid with the grid points, or *nodes*, drawn as actual points (on the left) and as squares (on the right). We prefer the squares, because we

would like you to think of a grid as a two-dimensional array from the very start. The figure also shows the natural, or *lexicographic*, numbering of the nodes.

The first time we apply a red-black numbering to the 8×8 grid we get Figure 19.2. First the black nodes are numbered (1-32, the colored squares), and then the red nodes (33-64, the white squares). Of course one can also start with the red nodes and thereafter the black ones. It does not matter, but we have to make a choice and be consistent throughout the method.

29	*61*	30	*62*	31	*63*	32	*64*
57	25	*58*	26	*59*	27	*60*	28
21	*53*	22	*54*	23	*55*	24	*56*
49	17	*50*	18	*51*	19	*52*	20
13	*45*	14	*46*	15	*47*	16	*48*
41	9	*42*	10	*43*	11	*44*	12
5	*37*	6	*38*	7	*39*	8	*40*
33	1	*34*	2	*35*	3	*36*	4

Figure 19.2: Basic red-black numbering for the 8×8 grid.

We see that by doing so we have numbered all nodes already, so for the RRB-numbering there should be a difference. The difference is that, after we have numbered all the black nodes, we do not number all the red nodes, but just half of them, see Figure 19.3.

29		30		31		32	
45	25	*46*	26	*47*	27	*48*	28
21		22		23		24	
41	17	*42*	18	*43*	19	*44*	20
13		14		15		16	
37	9	*38*	10	*39*	11	*40*	12
5		6		7		8	
33	1	*34*	2	*35*	3	*36*	4

Figure 19.3: First level RRB-numbering for the 8×8 grid. The empty spots form the 2nd level.

The 1st level consists of all nodes, the 2nd level is formed by the empty spots (16 pieces for the 8×8 example). We see that the next level is, depending on what you mean, 2 or 4 times coarser. The number of nodes in either direction (x or y) is about 2 times smaller; the total number of nodes in the 2nd level is about 4 times smaller than in the 1st level. We write "about" because only for "perfect" grids, that is, grids with dimensions that are a power of 2, the next level is indeed 4 times coarser; for "less perfect" grids, that is, grids that have an x- or y-dimension not being a power of 2, the next level is approximately 4 times coarser. For example, for a 17×17 grid the 1st level has 289 nodes, whereas the 2nd level has $8 \times 8 = 64$ nodes (check this for yourself), so 289 / 64 = 4.5 times smaller. Note also that, because of the 2nd level of the 17×17 grid is 8×8 nodes, from that level on the "4 times coarser"-rule applies to all next grids.

Next we apply the numbering technique to the 2nd level. By doing so we are left with empty spots which define the 3th level (4 in the 8×8 example), and again we apply the numbering technique. We continue "as long as the grid allows". How far that is shall we, for grids with arbitrary dimensions, determine in Section 19.1.3. In Figure 19.4 we have shown

the complete numbering process for the 8×8 grid. Clearly, an 8×8 grid allows 4 levels (64, 16, 4, 1 nodes).

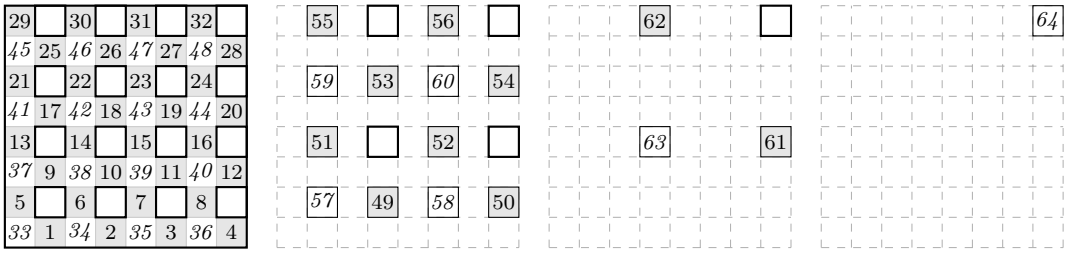


Figure 19.4: Recursive red-black numbering process for an 8×8 grid.

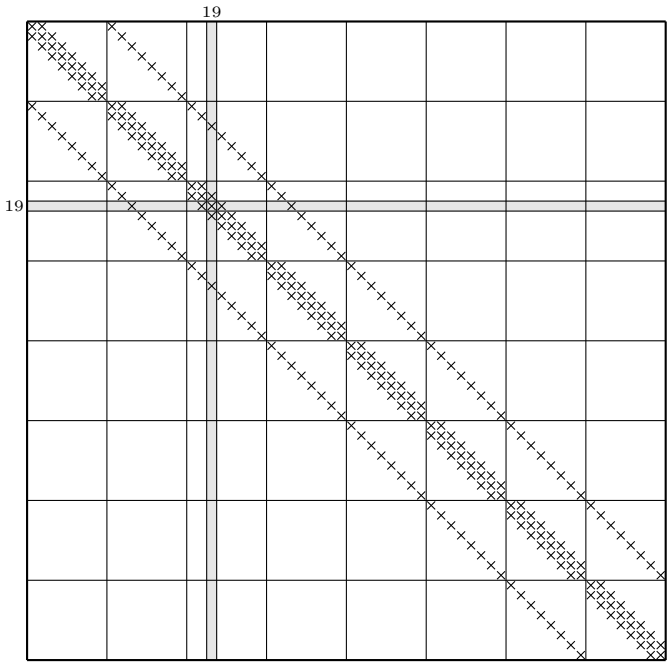
Combining all levels yields Figure 19.5.

29	55	30	62	31	56	32	64
45	25	46	26	47	27	48	28
21	59	22	53	23	60	24	54
41	17	42	18	43	19	44	20
13	51	14	63	15	52	16	61
37	9	38	10	39	11	40	12
5	57	6	49	7	58	8	50
33	1	34	2	35	3	36	4

Figure 19.5: Complete RRB-numbering for an 8×8 grid.

19.1.2 Effect of the RRB-numbering on the sparsity pattern of matrix S

The matrix S in our system $S\psi = b$ is given by a 5-point stencil, see Section 2.4. This means that a node depends on its direct north, east, south and west neighbours. In Figure 19.6 on the right we have indicated for node 19 its dependencies according to a 5-point stencil. Its neighbours are nodes 11, 18, 20 and 27. On the left in Figure 19.6 the corresponding sparsity pattern of matrix S is shown. A matrix with such a sparsity pattern is called a *pentadiagonal* matrix. On the left in the figure we have also highlighted the row and column that correspond to node 19. The locations where crosses ($\times$) are drawn are just the nodes on which a particular node (the cross on the main diagonal) depends.



57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

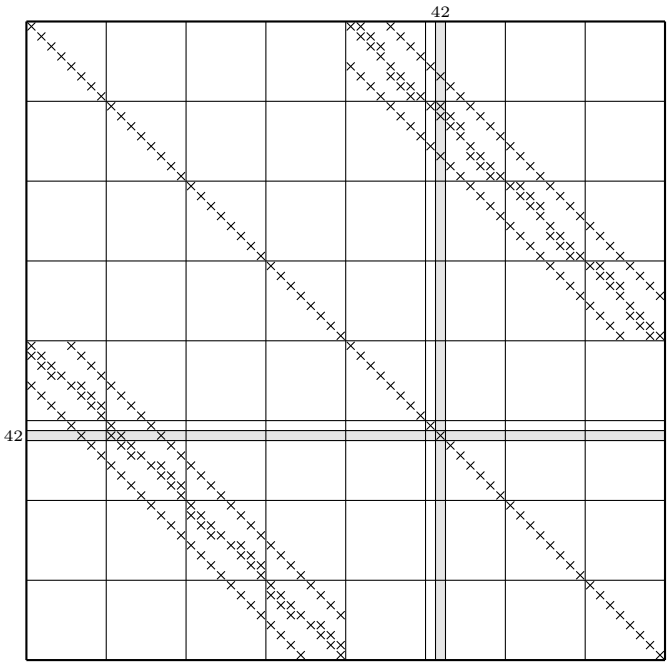
Figure 19.6: Sparsity pattern of $S \in \mathbb{R}^{64\times 64}$ when the basic red-black numbering is applied.

When a basic red-black numbering is used, we get a quite different sparsity pattern of S due to reordering, see Figure 19.7. Of course the node that is called node 19 with a lexicographic numbering, is still at the same location in the physical domain. However, due to the red-black numbering node 19 is now called node 42 (which is a red node), and has direct neighbours 6, 9, 10 and 14 (which are all black nodes). We see that red nodes only depend on black nodes and vice versa. This independency on the other color translates to a 2×2 block structure for matrix S: we can write

$$S = \begin{bmatrix} D_b & S_{br} \\ S_{rb} & D_r \end{bmatrix},$$

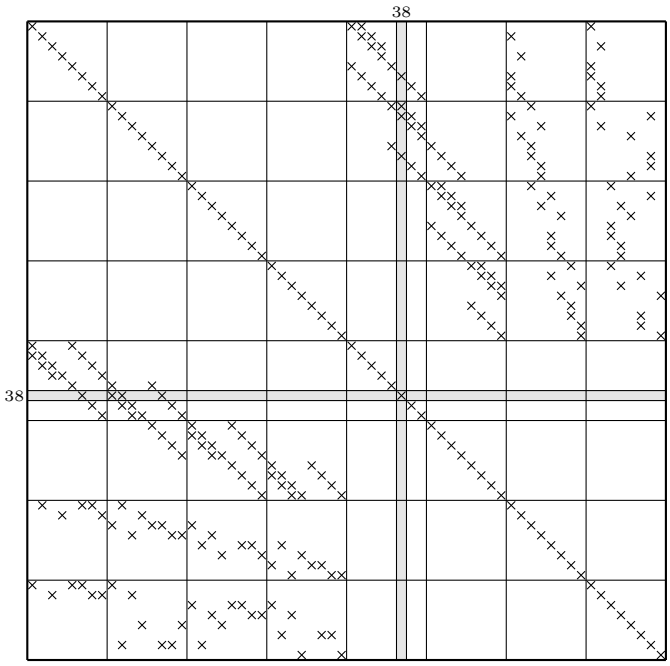
where "r" indicates the red nodes, and "b" the black nodes. Further, D_b and D_r are diagonal matrices and $S_{br} = S_{rb}^T$ are matrices with 4 diagonals; not precisely "4 diagonals", but this is a good way to describe them, see Figure 19.7.

When the RRB-numbering is applied to the 8×8 example we get Figure 19.8.



29	*61*	30	*62*	31	*63*	32	*64*
57	25	*58*	26	*59*	27	*60*	28
21	*53*	22	*54*	23	*55*	24	*56*
49	17	*50*	18	*51*	19	*52*	20
13	*45*	14	*46*	15	*47*	16	*48*
41	9	*42*	10	*43*	11	*44*	12
5	*37*	6	*38*	7	*39*	8	*40*
33	1	*34*	2	*35*	3	*36*	4

Figure 19.7: Sparsity pattern of $S \in \mathbb{R}^{64\times64}$ when the basic red-black numbering is applied. The matrix S becomes a 2×2 block matrix.



29	55	30	62	31	56	32	64
45	25	46	26	47	27	48	28
21	59	22	53	23	60	24	54
41	17	42	18	43	19	44	20
13	51	14	63	15	52	16	61
37	9	38	10	39	11	40	12
5	57	6	49	7	58	8	50
33	1	34	2	35	3	36	4

Figure 19.8: Sparsity pattern of $S \in \mathbb{R}^{64\times64}$ when the RRB-numbering is applied.

19.1.3 Maximal number of levels

In this section we shall determine how many levels a grid consisting of $N_x \times N_y$ nodes allows. That is, we are going to determine the maximal number of levels so that in the coarsest grid only 1 node remains.

Perfect dimensions

In the case that N_x and N_y are powers of 2 we say that the grid has perfect dimensions. We say "perfect" because for such grids the CUDA RRB-solver offers maximal performance.

The number n is a power of 2 when it can be written as $n = 2^k$ for $k \in \mathbb{N}$. We have $n = 2^k$ if and only if $k = \log_2 n$, or, for arbitrary base b (comes in handy for hand calculators), $k = \log_b n / \log_b 2$. For example, if $n = 512$ then $k = \log_{10} 512 / \log_{10} 2 = 9$. This means that we can divide the number n consecutively k times by 2 until 1 remains. And if we were to write down the numbers we would write down $k + 1$ different powers of 2. For example: $512, 256, 128, 64, 32, 16, 8, 4, 2, 1$ (9+1 = 10 numbers).

Let us now apply this to our $N_x \times N_y$ grid. If both N_x and N_y are a power of 2, i.e., $N_x = 2^{k_1}$ and $N_y = 2^{k_2}$, then we have to divide N_x k_1 times by 2 and N_y k_2 times until in both directions the number 1 remains, hence 1 node. So if we want to have just 1 node in the final level, we have to coarsen the grid $\max\{k_1, k_2\}$ times.

For example, for a 128×512 grid we would find the following levels:

level 1:	128	×	512	nodes
level 2:	64	×	256	nodes
level 3:	32	×	128	nodes
level 4:	16	×	64	nodes
level 5:	8	×	32	nodes
level 6:	4	×	16	nodes
level 7:	2	×	8	nodes
level 8:	1	×	4	nodes
level 9:	1	×	2	nodes
level 10:	1	×	1	nodes

So, although in the x-direction the grid has already become of size 1 after just 7 coarsening steps, the y-direction takes 9 coarsening steps so that 9+1 = 10 steps are necessary to generate a level with just 1 node.

Arbitrary dimensions

Let us now discuss how many levels there will be for a grid with arbitrary dimensions N_x and N_y. We have seen in Section 19.1.1 the mechanism behind the RRB-numbering. It was explained that the "odd/odd" red nodes in a certain level form the next level (actually we stated that the "even/even" nodes are being numbered, so the "odd/odd" nodes remain).

We observe that if N_x is even than there are $\frac{1}{2}N_x$ next level red nodes in the x-direction, and if N_x is odd than there are $\frac{1}{2}(N_x - 1)$ next level red nodes in the x-direction. The former number can be either even or odd, whereas the latter number is always even. Of course the same holds for N_y and the y-direction.

Now note that as soon as the number of next level red nodes has become a power of 2 the number of levels from that point on can be determined according to the perfect dimensions rules. Let us take an example: $N_x = 117, N_y = 33$. These numbers lead to:

level 1:	117	×	33	nodes
level 2:	58	×	16	nodes
level 3:	29	×	8	nodes
level 4:	14	×	4	nodes
level 5:	7	×	2	nodes
level 6:	3	×	1	nodes
level 7:	1	×	1	nodes

So by construction we find that 7 levels are allowed. In this example it is also nicely illustrated how the not so perfect number 33 becomes after 1 coarsening step a perfect number, namely 16. The pattern that we observe makes perfectly sense when we write the number as a binary number. Let us write

$$N_x = c_k \cdot 2^{n_k} + c_{k-1} \cdot 2^{n_{k-1}} + \cdots + c_2 \cdot 2^2 + c_1 \cdot 2^1 + c_0 \cdot 2^0,$$

where $c_k = 1$, $c_j \in \{0, 1\}$ $(j = 0, 1, 2, \ldots, k-1)$, and k refers to the highest power of 2 that can be substracted from N_x. For example,

$$117 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0,$$

so $117_2 = 1110101$. Then the number of nodes in the next level is $1110101 >> 1$, where $>>$ stands for bitshift by 1 to the right. Here: $1110101 >> 1 = 111010 = 58_2$. So the x-dimension of the 2nd level would be 58 nodes. Also, $33_2 = 100001$, and $100001 >> 1 = 10000 = 16_2$, so the 2nd level would consist of 58×16 nodes, just like we "computed" above.

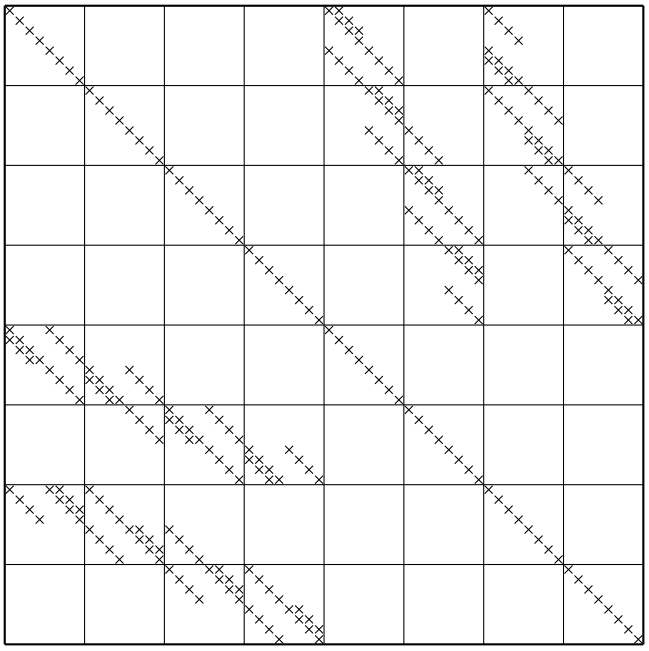
By looking at the numbers as binary numbers the maximal number of levels is easily found: the maximum number of levels, $k_{\max}$, is 1 plus the k that refers to the largest power of 2 that can be substracted from $\max\{N_x, N_y\}$. As a formula:

$$k_{\max} = 1 + \lfloor(\log_2(\max\{N_x, N_y\}))\rfloor.$$

For the 117×33 example: $k_{\max} = 1 + \lfloor(\log_2(\max\{117, 33\}))\rfloor = 1 + \lfloor(\log_2 117\rfloor = 1 + \lfloor 6.870\rfloor = 7$. Note that the formula for $k_{\max}$ also holds for "perfect" dimensions. For the 128×512 example in Section 19.1.3: $k_{\max} = 1 + \lfloor(\log_2(\max\{128, 512\}))\rfloor = 1 + \lfloor(\log_2 512\rfloor = 1 + \lfloor 9\rfloor = 10$.

19.1.4 The RRB-k method

In the previous sections it was assumed that we go all the way down to a level that has only 1 node left. This is not mandatory; we can at any point in the coarsening process decide to stop. The level at which we stop the process is called level k, hence the RRB-k method. On level k the remaining nodes are numbered naturally. In Figure 19.9 we have shown the RRB-numbering and corresponding sparsity pattern of S for the 8×8 example in case we stop at level 2.



29	*61*	30	*62*	31	*63*	32	*64*
45	25	46	26	47	27	48	28
21	*57*	22	*58*	23	*59*	24	*60*
41	17	42	18	43	19	44	20
13	*53*	14	*54*	15	*55*	16	*56*
37	9	38	10	39	11	40	12
5	*49*	6	*50*	7	*51*	8	*52*
33	1	34	2	35	3	36	4

Figure 19.9: Sparsity pattern of $S \in \mathbb{R}^{64\times 64}$ and corresponding numbering for the RRB-2 method applied to the 8×8 example.

The RRB-method is used to compute an incomplete factorization $M = LDL^T$ which approximates S. The idea of stopping earlier, i.e., at level k, is that we want to solve the solution in the remaining nodes more accurately or even exactly. Suppose that at level k there are $N_{x,k} \times N_{y,k}$ nodes. From these nodes a matrix $E \in \mathbb{N}^{n\times n}$, where $n = N_{x,k}N_{y,k}$. By using the natural numbering the matrix E becomes a symmetric pentadiagonal matrix. More accurate solvers are more expensive, so that n should be considerably smaller than the original matrix S which has dimensions $N_xN_y \times N_xN_y$ in order to keep a fast solver.

On level k we have to solve a system like

$$Ex = b,$$

where x is formed by the level k nodes of z and b by the level k nodes of r in the problem $Mz = r$. Because E is a pentadiagonal matrix the most proficient method (on a sequential platform) would be to apply a Complete Cholesky factorization (see Section 7.3.3) to E, i.e., $E = GG^T$, where G is a lower triangular matrix, and then use forward and backward substitution to solve $Ex = b$:

1. Set $y := G^Tx$ and solve $Gy = b$ using forward substitution;
2. Solve $G^Tx = y$ using backward substitution.

A different approach can be to compute the inverse of E directly, and just compute $x = E^{-1}b$. This seems bad at first glance but on a parallel platform computing a full matrix-vector product may be as fast or even faster than using forward and backward substitution as the latter two methods are inherently sequential. For more information on the RRB-k method we refer to [28].

19.1.5 PCG for half of the nodes

What the name of the solver does not reveal is that the RRB-solver is a Conjugate Gradient (CG) solver. More precisely, a preconditioned version of the CG algorithm is used: the Preconditioned Conjugate Gradient (PCG) algorithm. Moreover, the PCG algorithm operates on only half of the total number of nodes, i.e., only the red nodes. Let us start with explaining how we get this reduction of a factor 2 in number of unknowns.

In Section 19.1.2 we have seen that with the basic red-black numbering the matrix S kan be written as a 2×2 block matrix. If we indicate the red nodes with "r" and the black nodes with "b" we may put the system

$$S\psi = b$$

into the form:

$$\begin{bmatrix} D_b & S_{br} \\ S_{rb} & D_r \end{bmatrix} \begin{bmatrix} \psi_b \\ \psi_r \end{bmatrix} = \begin{bmatrix} b_b \\ b_r \end{bmatrix}.$$

Herein are D_r and D_b diagonal matrices and $S_{rb} = S_{br}^T$ are matrices with 4 diagonals.

Next we apply so-called *Gaussian elimination* to "get rid off" all black nodes. This yields:

$$\begin{bmatrix} D_b & S_{br} \\ 0 & D_r - S_{rb}D_b^{-1}S_{br} \end{bmatrix} \begin{bmatrix} \psi_b \\ \psi_r \end{bmatrix} = \begin{bmatrix} b_b \\ b_r - S_{rb}D_b^{-1}S_{br} \end{bmatrix}. \tag{19.1.1}$$

The matrix $S_1 := D_r - S_{rb}D_b^{-1}S_{br}$ is called the *1st Schur complement* and is given by a 9-point stencil, the vector $b_1 := b_r - S_{rb}D_b^{-1}b_b$ is the corresponding right-hand side. In Figure 19.10 we see, using graph representation, how Gaussian elimination leads to this 9-point dependency.

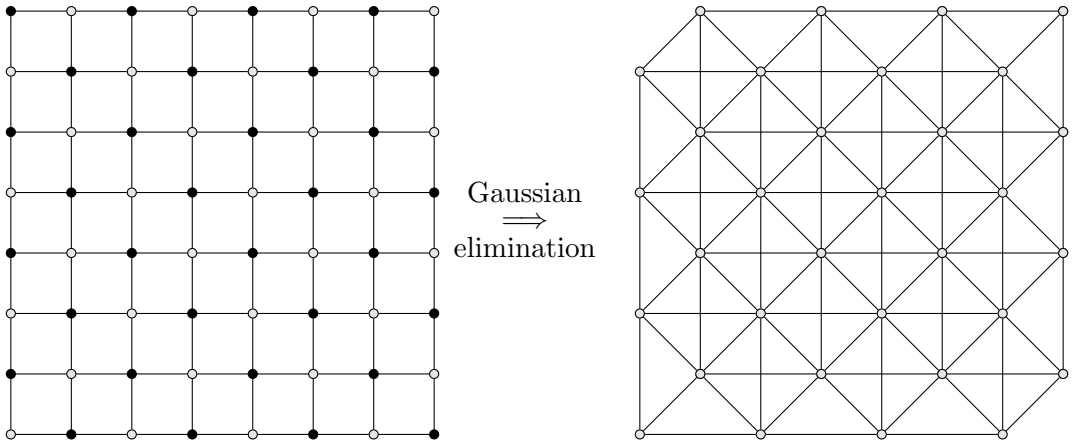


Figure 19.10: Elimination of the black nodes leads to fill-in and a 9-point stencil.

We observe that if we were to know the solution in all red nodes (ψ_r), we would be able to compute the solution in all black nodes (ψ_b) by means of a simple substitution. Thanks to the red-black numbering we thus only have to compute the solution for about half of the nodes (only the red nodes). Solving the sub-system

$$S_1\psi_r = b_1 \tag{19.1.2}$$

with $b_1 := b_r - S_{rb}D_z^{-1}S_{br}$ is done iteratively, namely with the Conjugate Gradient (CG) method. The "engine" of the RRB-solver is thus the CG algorithm applied to half of the nodes.

But there is a catch. As the CG algorithm operates on half of the notes, at first glance we may believe that we gain a factor 2 reduction in computing time. Unfortunately, this is not true. Indeed the number of unknows is reduced by a factor 2, but the system involves no longer a 5-point stencil but a more expensive 9-point stencil. For example, the matrix-vector product which occurs in every iteration of the CG algorithm becomes more expensive; the number of flops per node increases from 9 for the 5-point to 17 for the 9-point stencil. Other operations such as vector updates (AXPYs) and dot products (DOTs) have become two times cheaper. If we were to count the overall number of flops for the RRB-solver we would see a decrease in number of flops.

Moreover, a reordering of the nodes (in our case thus a red-black reordering) typically means an increase in number of CG iterations required for the CG algorithm to converge. However, the RRB-solver does not use the plain CG algorithm, but the preconditioned Conjugate Gradient algorithm (PCG), which increases the rate of convergence. The preconditioner that is used is the RRB method.

If everything goes well, the CG algorithm will provide us the solution in the red nodes, ψ_r. Given this vector ψ_r we can compute the solution in the black nodes, i.e., the vector ψ_b, by means of a simple substitution as follows. From (19.1.1) we see that

$$D_b\psi_b + S_{br}\psi_r = b_b \quad \Longleftrightarrow \quad \psi_b = D_b^{-1}(b_b - S_{br}\psi_r).$$

So, summarizing, we do the following:

1. Compute $b_1 = b_r - S_{rb}D_b^{-1}b_b$;
2. Apply CG to system (19.1.2), i.e., $S_1\psi_r = b_1$, result: ψ_r;
3. Compute ψ_b via $\psi_b = D_b^{-1}(b_b - S_{br}\psi_r)$.

19.2 The ideas behind the CUDA RRB-solver

As we have seen earlier in Part III one of the most, if not, the most important notion to get really fast CUDA code is the notion of *coalesced memory*. For our RRB-solver not different: we want all the global memory reads and stores to be "as coalesced as possible". Maximal throughput is achieved only if all memory transactions go coalesced-wise. However, due to one of the basis principles behind the RRB-solver, i.e., computations on consecutively coarser red-black grids, it seems grit is already thrown in the machine. How are we going to overcome global memory reads with increasing offsets due to coarsening? This is the main subject of this section: clever tricks to get maximal throughput.

19.2.1 Clever storage of the data: the $r_1/r_2/b_1/b_2$-storage format

Throughout this section we shall demonstrate everything at the hand of a small example: a region that leads to a grid containing 75×40 nodes, see Figure 19.11. The rectangle shows how data is stored in a 2D array-format. Throughout the `lin_wacu` code the class `Array2D` is used, a very efficient custom-built C++ class exploiting pointers that let us store

and manipulate 2D arrays in a very cheap manner. Next to the figure a compass is drawn, the so-called "host compass". In a moment we shall also meet the so-called "device compass". We distinguish between two orientations due to the fact that data is copied wrongly from the host to the device.

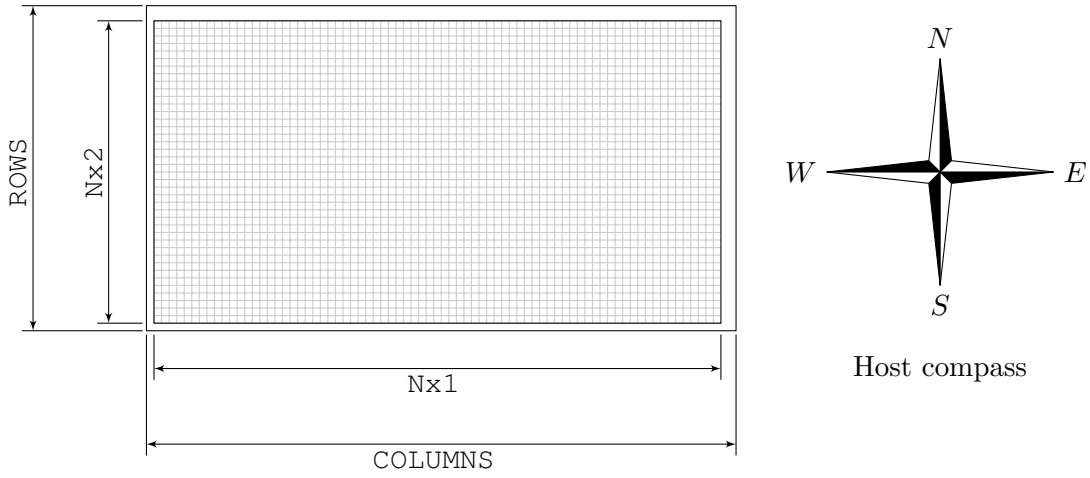


Figure 19.11: An example how a grid is stored in the `lin_wacu` software. For the storage the `Array2D` class is used. The grid has `Nx1` nodes in the x-direction and `Nx2` nodes in the y-direction. In the south and west there is a layer of "ghost nodes"; in the north and east are two layers of extra nodes. The overall dimensions of the grid are thus: `COLUMNS = Nx1 + 3` by `ROWS = Nx2 + 3` nodes. Here: `Nx1 = 75`, `Nx2 = 40`.

The data which we are interested in is indicated by the gray cells. The relevant data is thus surrounded by layers of "ghost nodes". By thinking in advance how `CUDA` will work with the data, we figured that we should copy the data to the device in a not-straightforward manner, i.e., we are going to embed the array in a somewhat larger array, see Figure 19.12.

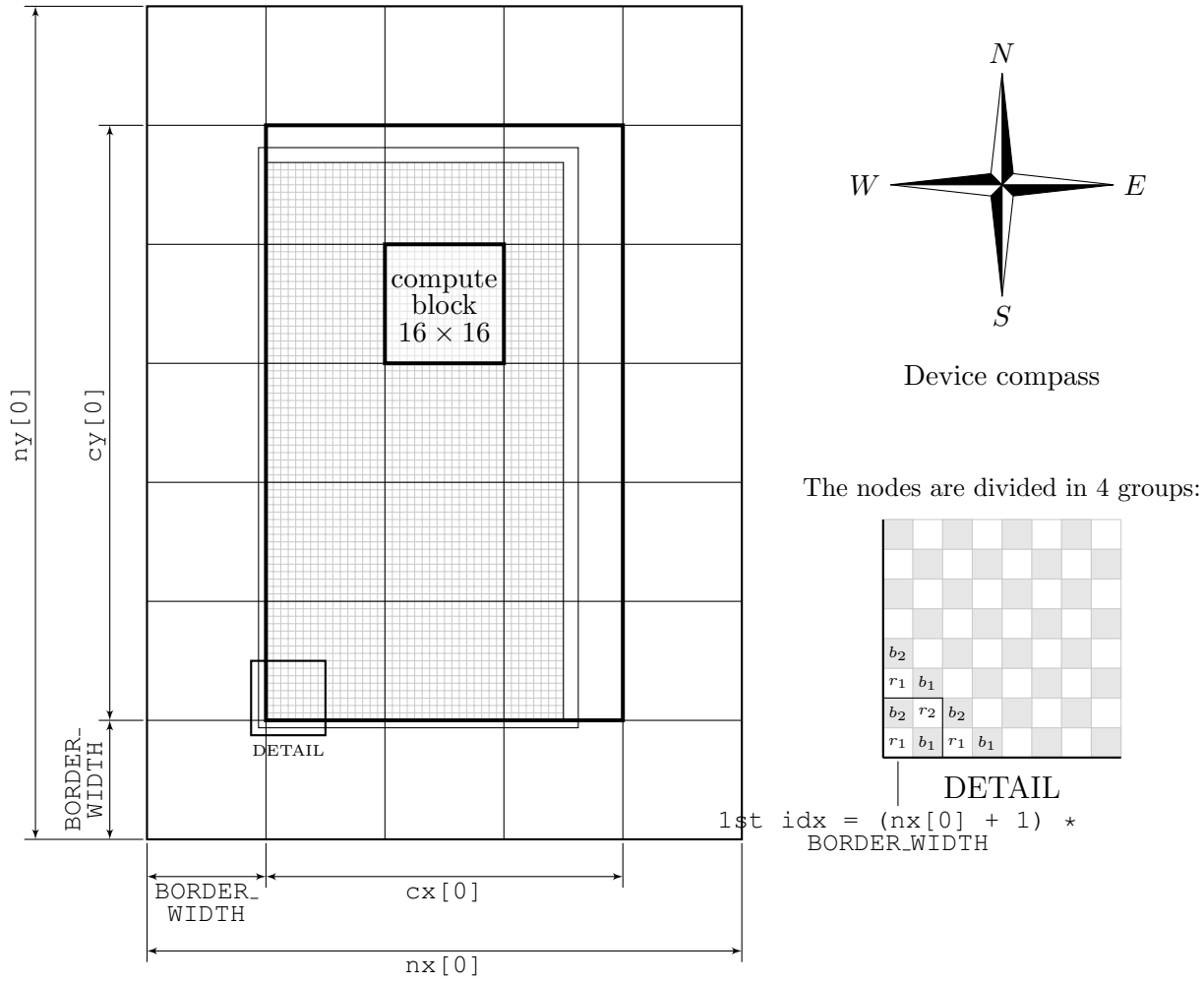


Figure 19.12: The original grid is embedded in a somewhat larger grid that offers better parallellization possibilities and more efficient code. In the bottom-right corner is a $r_1/r_2/b_1/b_2$-numbering is shown.

Consider Figure 19.12. First note that the array is mirrored in some sense. This is due to the row-wise/column-wise storage of data. Furthermore, we see how the original grid is copied into a grid with "wide borders", namely `BORDER_WIDTH` wide, an even number. We have taken `BORDER_WIDTH = 16` which makes that data is stored according to the "optimal 16-spacing (= 64 byte)" rule. The new array is `nx[0]` by `ny[0]`. Also a so-called *compute-block* is indicated. A compute-block is the basic identity with which `CUDA` will work. The compute-block is square: 16×16 elements or 32×32 elements (depending on the GPU). Henceforth we shall assume that the compute-block consists of 16×16 elements but everything also holds for compute-blocks with 32×32 elements. Later on we shall use `DIM_COMPUTE_BLOCK` as size of the compute-block.

It should be noted that "elements" is not equal to "threads" per se: The `CUDA` code will be such that each division of the 16×16 block into smaller thread-blocks is allowed; e.g., 2 thread-blocks of 16×8 threads each or 16 thread-blocks consisting of 4×4 threads, which

won't be optimal, but this division is also allowed.

Finally in the bottom-right corner a special numbering of the nodes is suggested: the data is divided into four distinct groups: the r_1-nodes (even/even red nodes), the b_1-nodes (the odd/even black nodes), the r_2 nodes (the odd/odd red nodes), and the b_2-nodes (the even/odd black nodes). With "even/odd" we mean: the first coordinate is even; the second coordinate is odd.

The idea is now, especially since the CG algorithm operates on the red nodes only, to restore the data according to this $r1/r2/b1/b2$-storage format, See Figure 19.13.

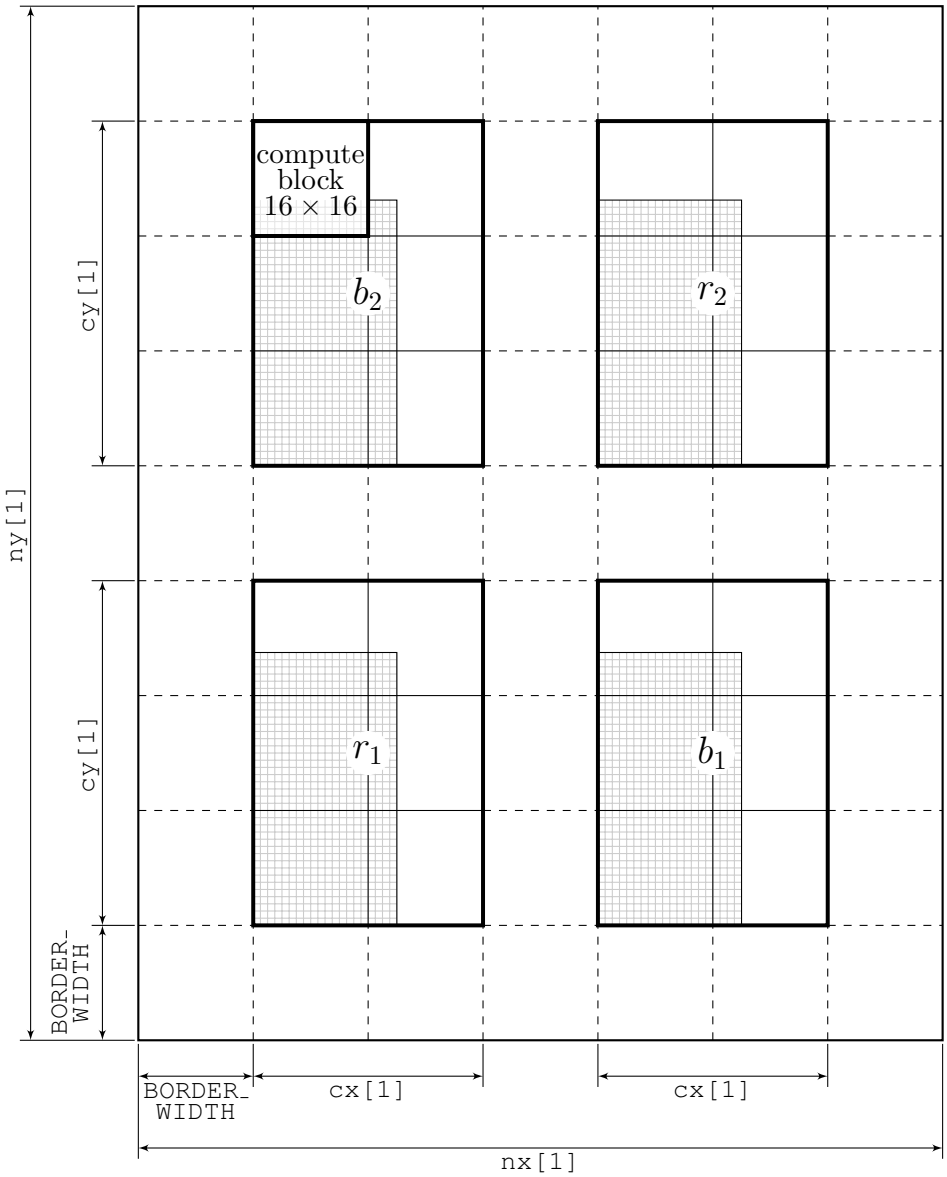


Figure 19.13: Restoring the first grid according to a $r_1/r_2/b_1/b_2$-storage format.

Why saving the data in this different format? Well, by doing so we will get much more coalesced memory transactions. For example, imagine we have to do some computations

for the red nodes only in the first level, e.g., a vector update (AXPY), than by using the $r_1/r_2/b_1/b_2$-storage format we can read all data fully coalesced from the global memory using 64 bytes long chunks of memory, which is optimal. This is just why the compute-blocks consist of 16×16 elements (or 32×32 with 128 byte chunks)! If also nearby data is required, say the direct neighbours according to a 5-point stencil, i.e., 4 black nodes, then again we can read those for 50% from the global memory in a coalesced manner. Namely, for r_1 (or r_2) the b_2 and b_1 nodes can be read coalesced; the other two neighbours must be read in a non-coalesced manner. In some cases we may use textures (which uses caches) to decrease accessing time for such "nasty" elements.

The figure also shows that we do some useless work due to padding (the white parts), but for larger grids the amount of useless work is neglicable. The fat borders prevent inclusion of `if`-statements. Although this usually saves only about 5%, it is noticeable.

19.2.2 Recursively applying the $r_1/r_2/b_1/b_2$-storage format

Now the key observation is that the nodes in the next coarser red-black grid are just the r_2-nodes! Therefore, we can recursively apply this reordering! In Figure 19.14 and Figure 19.15 this coarsening process is shown.

In Figure 19.15 all relevant data is contained in four 16×16 blocks, which have just the dimensions of the basic computing entities: the compute-blocks consisting of 16×16 elements. At this level we stop the process as only 16×16 CUDA threads are active and the problem now fits completely in shared memory or in the cache of the GPU on 1 streaming multiprocessor (SM). 1 thread-block with 16×16 threads on 1 SM will therefore take care of the last few levels.

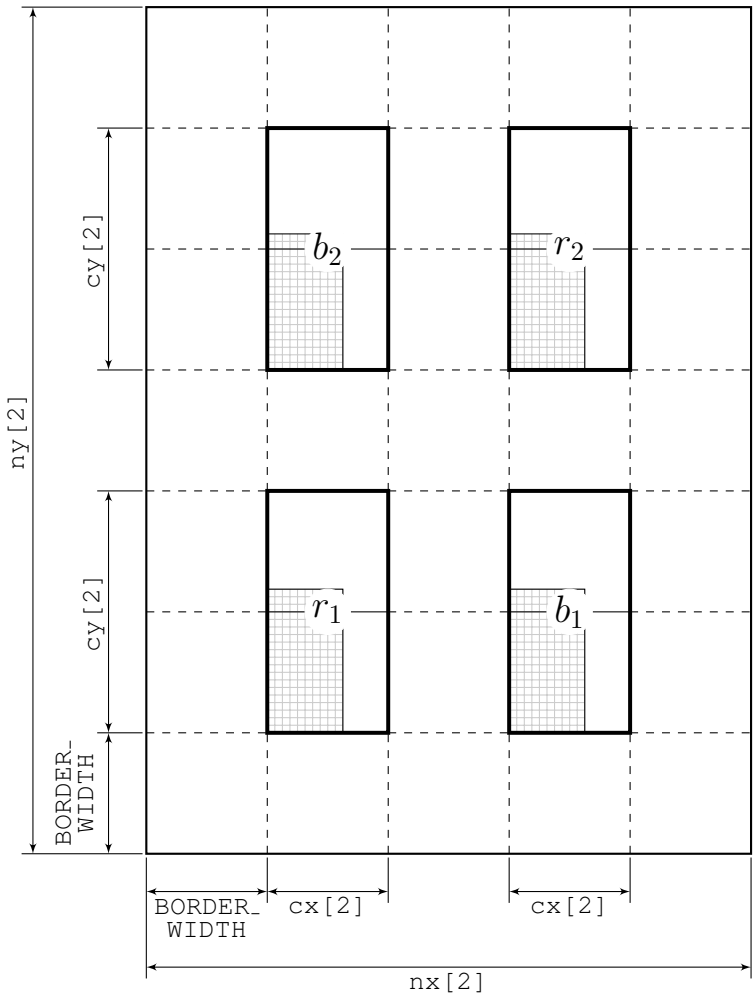


Figure 19.14: The second CUDA grid. Note how the grids systematically become smaller. From this picture it is also clear that grids that have dimensions that are a power of 2 (i.e., 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, etcetera) are optimal for the proposed method consisting of 16×16 compute-blocks.

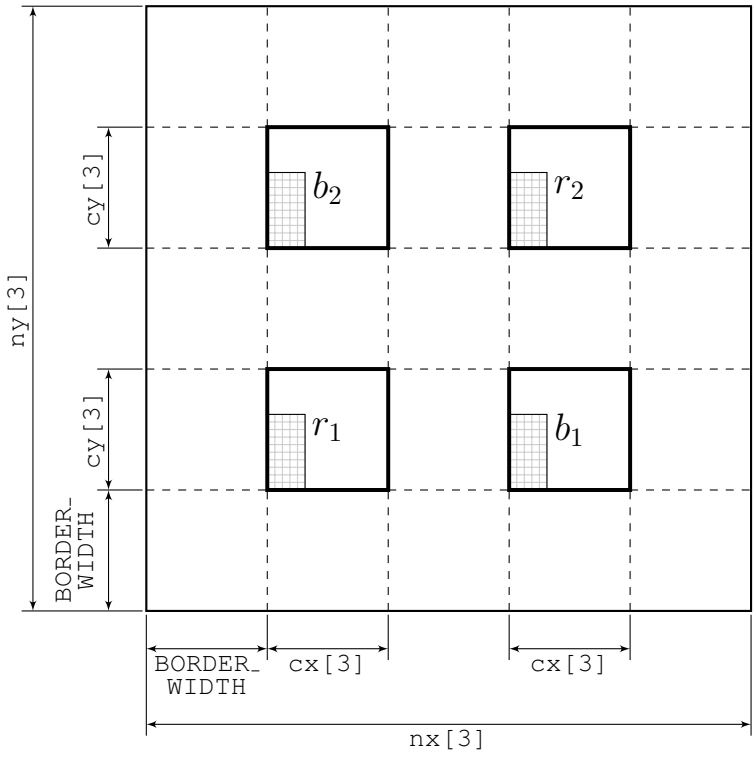


Figure 19.15: The third and final CUDA grid in case of the 75×40 example. 1 thread-block having 16×16 (or 32×32) threads will take care of this remaining grid by exploiting cached global memory.

19.2.3 Thread organization

Figure 19.16 shows how the threads are organized for all kernels; however, the size of thread-blocks may differ per kernel, depending on what division delivers the most throughput.

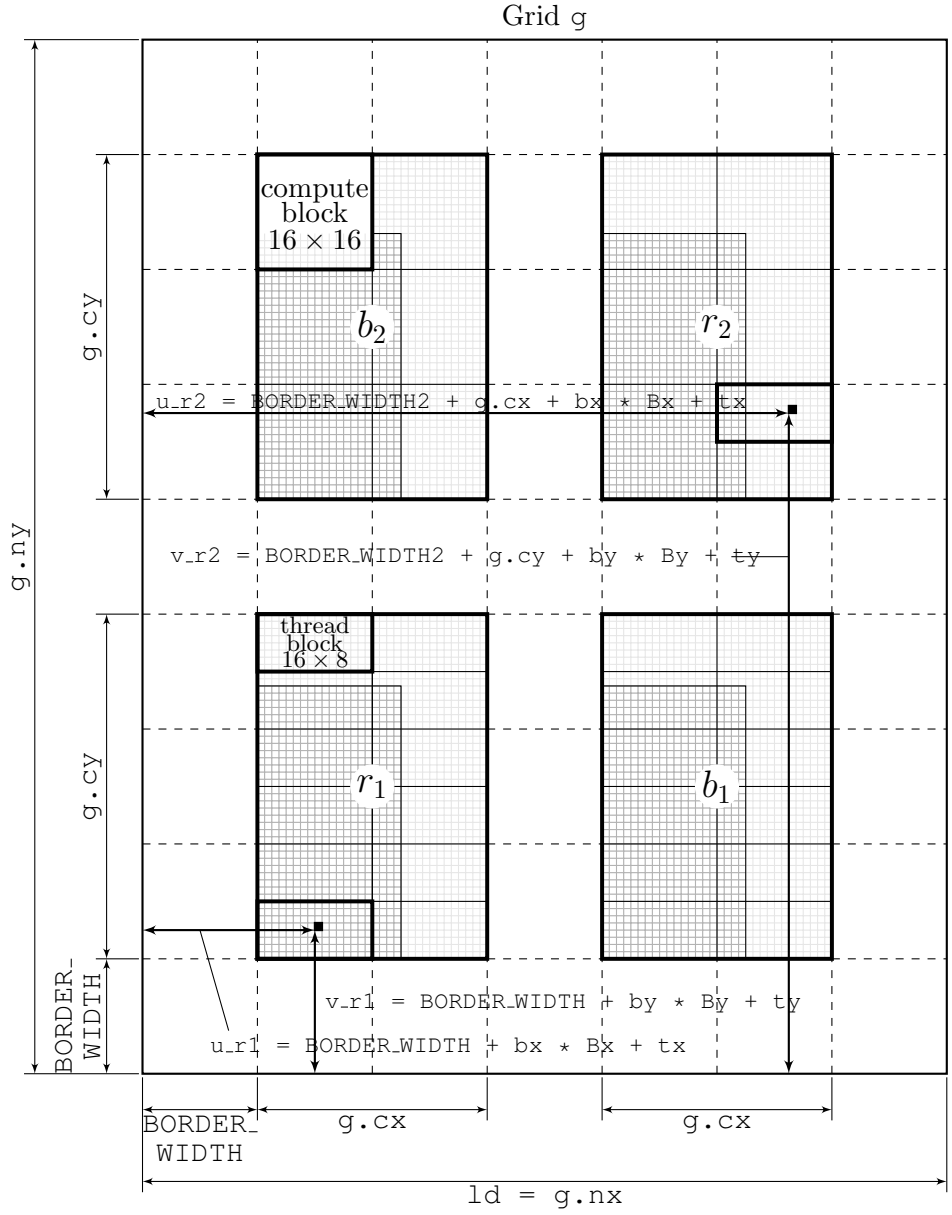


Figure 19.16: Assigning unique array elements to threads. In this picture it is assumed that the dimensions of the compute-block are set to 16×16. Further, `BORDER_WIDTH = 16` and the compute-block is divided into two thread-blocks of size 16×8, hence `Bx = 16` and `By = 8`. The grid consists of 6 compute-blocks, hence 12 thread-blocks: `bx = 0,1` and `by = 0,1,2`. For two array elements, one r_1- and one r_2-element, it is shown which thread handles it (the little black squares). For this particular elements we have (check it yourself): `bx = 0, by = 0, tx = 8, ty = 4` for the r_1-node and `bx = 1, by = 1, tx = 10, ty = 4` for the r_2-element.

19.3 General comments on implementation

We briefly comment on our CUDA implementation. This is necessary as in the next sections we are going to explain the kernels at the hand of the actual implementation.

Matrix S is given by three stencils, namely StC (the center stencil), StW (the west stencil) and StS (the south stencil). The east and north stencils are not needed as the matrix S is symmetric and therefore the east stencil, StE, can be expressed in terms of StW, and the north stencil, StN, can be expressed in terms of StS.

To avoid really difficult bookkeeping and to increase transparancy of the code, the preconditioning matrix M in the current C++ version is build with all five stencils. Our CUDA implementation also uses five stencils, but in principle it is possible to write code that uses only three stencils.

On top of the five stencils, four additional stencils (or two in case of just three stencils) are needed since the 1st Schur complement, namely $StNE$ (north-east), $StSE$ (south-east), $StSW$ (south-west) and $StNW$ (north-west).

In the CUDA implementation we have therefore declared a structure `Grid` which will contain the 9 stencils (structure members). The reason why a structure is used rather than a class, is that CUDA does not support classes. `Grid` is defined as a global variable as follows:

```
struct Grid {
    REAL *cc;
    REAL *nn;
    REAL *ee;
    REAL *ss;
    REAL *ww;
    REAL *ne;
    REAL *se;
    REAL *sw;
    REAL *nw;
    unsigned int nx;
    unsigned int ny;
    unsigned int cx;
    unsigned int cy;
};
```

The symbol `REAL` is either `float` or `double`. This is decided by the preprocessor directives

```
#ifdef USEDOUBLE
#define REAL double
#else
#define REAL float
#endif
```

in the headerfile "`defines.h`". So, if `USEDOUBLE` is defined (either via a preprocessor directive or adding `-DUSEDOUBLE` in the Makefile, double-precision (`double`) is used; if not, single-precision (`float`) is used throughout the code. Further, the structure has the structure members `nx, ny, cx` and `cy`. Herein refers, for example, `nx` to the dimension of the grid in the x-direction (the leading dimension) and `cy` to the height of the computing area.

We can use this structure by declaring a variable via `Grid g;`. Then, for example, the north-east stencil is retrieved with `g.ne`, and the leading dimension `ld` is retrieved with `ld = g.nx`.

19.4 Determining the sizes of the levels

In Section 19.1.3 we have seen how many levels there are for grids with arbitrary dimensions N_x by N_y nodes. In Section 19.1.3 it was remarked that in CUDA at a certain point we stop the coarsening process, namely as soon as the remaining number of nodes fit in the basic computing unit: the compute-block which is 16×16 or 32×32 elements large (depending on the GPU). In this section we shall provide C++ source code that computes the number of levels and the corresponding sizes.

19.4.1 Introduction

When arrays with the Array2D format are copied from the host to the device, the rows and columns are interchanged. This means that when the original grid has size Nx1 (x) by Nx2 (y) on the host, on the device it arrives as a Nx2 (x) by Nx1 (y) array. In Section 19.2.1 it was explained that the original data is embedded in a somewhat bigger array. The reason to do this was that we always strive for coalesced memory transactions. The first task is to determine the dimensions of this embedding grid.

19.4.2 The embedding grid

Consider Figure 19.12. The relevant data is contained in a computing area with dimensions cx[0] by cy[0] elements. Around this computing area wide borders are attached with width BORDER_WIDTH. The overall dimensions of the embedding grid are nx[0] by nx[1] elements. We have that nx[0] = cx[0] + BORDER_WIDTH2 where BORDER_WIDTH2 = 2 * BORDER_WIDTH. To make sure that the relevant data is copied into the embedding grid as intended the CUDA function cudaMemcpy2D() is used. To determine how large the computing area (cx[0] by cy[0]) should be we compute how many compute-blocks (with dimensions DIM_COMPUTE_BLOCK by DIM_COMPUTE_BLOCK) are needed in either direction to contain the data. This is done with the C++ code:

```
cx[0] = DIM_COMPUTE_BLOCK * (int)ceil((float)Nx2 / DIM_COMPUTE_BLOCK);
cy[0] = DIM_COMPUTE_BLOCK * (int)ceil((float)Nx1 / DIM_COMPUTE_BLOCK);
```

Note that we use Nx2 for the x-direction and Nx2 for the y-direction because of the row/column interchange. Accordingly, nx[0] and ny[0] can be computed as follows:

```
nx[0] = m_cx[0] + BORDER_WIDTH2;
ny[0] = m_cy[0] + BORDER_WIDTH2;
```

19.4.3 The $r_1/r_2/b_1/b_2$-grids

The next task it to determine how many different $r_1/r_2/b_1/b_2$-levels we need and also how large they should be. Let us first focus on how many levels there are needed. In Section 19.2.2 it was explained that we stop as soon as the number of remaining elements fit in one compute-block. Therefore we use a while-loop to check whether we can stop yet:

```
int k = 1;
while (cx[k-1] > DIM_COMPUTE_BLOCK || cy[k-1] > DIM_COMPUTE_BLOCK)
{
   ...
}
```

The first $r_1/r_2/b_1/b_2$-grid is called grid `1`. It remains to give a relationship between the dimensions of the computing areas of consecutive grids. It won't take you long to figure out that the following code does the trick:

```
cx[k] = DIM_COMPUTE_BLOCK * (int)ceil((float)(cx[k-1] / 2)
            / DIM_COMPUTE_BLOCK);
cy[k] = DIM_COMPUTE_BLOCK * (int)ceil((float)(cy[k-1] / 2)
            / DIM_COMPUTE_BLOCK);
```

Accordingly, the overall dimenions of each grid are given by

```
m_nx[k] = 2 * cx[k] + BORDER_WIDTH + BORDER_WIDTH2;
m_ny[k] = 2 * cy[k] + BORDER_WIDTH + BORDER_WIDTH2;
```

The term `BORDER_WIDTH` corresponds with the fact that the r_1, r_2, b_1 and b_2 parts of the array are seperated by a strokes with width `BORDER_WIDTH`. Everything combined an appriopriate `C++` class function for `CudaRrbSolver` is given in Listing 19.1.

Listing 19.1: Class function determineGrids().

```
1  void CudaRrbSolver::determineGrids(unsigned int Nx1, unsigned int Nx2)
2  {
3      m_cx[0] = DIM_COMPUTE_BLOCK * (int)ceil((float)Nx2 / DIM_COMPUTE_BLOCK);
4      m_cy[0] = DIM_COMPUTE_BLOCK * (int)ceil((float)Nx1 / DIM_COMPUTE_BLOCK);
5
6      m_nx[0] = m_cx[0] + BORDER_WIDTH2;
7      m_ny[0] = m_cy[0] + BORDER_WIDTH2;
8
9      int k = 1;
10     while (m_cx[k-1] > DIM_COMPUTE_BLOCK || m_cy[k-1] > DIM_COMPUTE_BLOCK)
11     {
12         m_cx[k] = DIM_COMPUTE_BLOCK * (int)ceil((float)(m_cx[k-1] / 2)
13                     / DIM_COMPUTE_BLOCK);
14
15         m_cy[k] = DIM_COMPUTE_BLOCK * (int)ceil((float)(m_cy[k-1] / 2)
16                     / DIM_COMPUTE_BLOCK);
17
18         m_nx[k] = 2 * m_cx[k] + BORDER_WIDTH + BORDER_WIDTH2;
19         m_ny[k] = 2 * m_cy[k] + BORDER_WIDTH + BORDER_WIDTH2;
20         ++k;
21     }
22     m_numLevels = k - 1;
23 } // determineGrids
```

Note that, for example, `nx` is replaced by `m_nx`. This means that the variable `m_nx` has become a member variable of the class `CudaRrbSolver`. The total number of $r_1/r_2/b_1/b_2$-levels is called `m_numLevels`. Let us illustrate the algorithm at the hand of a small example.

19.4.4 An example

Let us take `Nx1 = 75` and `Nx2 = 40`. Assume `DIM_COMPUTE_BLOCK = 16`. For the embedding grid we find: $cx[0] = 16 \cdot \lceil f40/16 \rceil = 16 \cdot \lceil 2.5 \rceil = 16 \cdot 3 = 48$. Herein is / the integer division operator and f the float-type cast. Likewise, $cy[0] = 16 \cdot \lceil f75/16 \rceil = 16 \cdot \lceil 4.6875 \rceil = 16 \cdot 5 = 80$. Hence $nx[0] = 48 + 32 = 80$ and $ny[0] = 80 + 32 = 112$.

For the first $r_1/r_2/b_1/b_2$-grid we compute $cx[1] = 16 \cdot \lceil f(40/2)/16 \rceil = 16 \cdot \lceil 1.25 \rceil = 16 \cdot 2 = 32$ and $cy[1] = 16 \cdot \lceil f(75/2)/16 \rceil = 16 \cdot \lceil 2.34375 \rceil = 16 \cdot 3 = 48$. Hence $nx[1] = 2 \cdot 32 + 16 + 32 = 112$ and $ny[1] = 2 \cdot 48 + 16 + 32 = 144$. By going through all computations we find:

level k	$cx[k]$		$cy[k]$	$nx[k]$		$ny[k]$
embedding (= 0):	48	×	80	80	×	112
level 1:	32	×	48	112	×	144
level 2:	16	×	32	80	×	112
level 3:	16	×	16	80	×	80

19.5 Memory requirements

In this section we shall give indications on how much memory is required for solving a problem consisting of `Nx1` by `Nx2` nodes on the GPU. The $r_1/r_2/b_1/b_2$-storage format offers great advantages regarding performance, however, we have to pay a price: extra memory is needed. We shall give estimates how much memory is required for the complete `CUDA` RRB-solver as well as the amount of extra memory that comes with introduction of the $r_1/r_2/b_1/b_2$-storage format.

19.5.1 A list of all data objects

To be able to give memory estimates we have put together a list that contains all data objects that take a significant amount of memory space. The list is based on the actual `CUDA` implementation.

Variable	Description	Storage format
`*m_prec`	preconditioner stencils	repeated $r_1/r_2/b_1/b_2$
`m_orig`	original stencils	single $r_1/r_2/b_1/b_2$
`*m_dcc`	center stencil	standard$^{(*)}$
`*m_dss`	south stencil	standard$^{(*)}$
`*m_dww`	west stencil	standard$^{(*)}$
`*m_dX`	vector x	standard
`*m_dB`	vector b	standard
`*m_dXr1r2b1b2`	vector x	single $r_1/r_2/b_1/b_2$
`*m_dYr1r2b1b2`	vector y	single $r_1/r_2/b_1/b_2$
`*m_dBr1r2b1b2`	vector b	single $r_1/r_2/b_1/b_2$
`*m_dPr1r2b1b2`	vector p	single $r_1/r_2/b_1/b_2$
`*m_dQr1r2b1b2`	vector q	single $r_1/r_2/b_1/b_2$
`*m_dRr1r2b1b2`	vector r	single $r_1/r_2/b_1/b_2$
`**m_dZr1r2b1b2`	vector z	repeated $r_1/r_2/b_1/b_2$
`*m_drem`	for dot products	standard

Note $(*)$: Only 1 of the stencils exists at the time. After the stencil is copied to the device, the stencil is restored in the $r_1/r_2/b_1/b_2$-storage format, the original stencil is deleted right after it.
It appears that only the vectors x and b are stored twice in two different formats. Further, for creating the preconditioner stencils in the $r_1/r_2/b_1/b_2$-storage format it is necessary to have at the same moment the stencil available in the original, standard format. Compared to a

straightforward CUDA PCG solver we thus need 3 extra arrays in the device's global memory that have the dimension of the original problem. Moreover, some of the variables require already some extra memory space because of the repeated $r_1/r_2/b_1/b_2$-storage format.

19.5.2 Extra memory requirements for the repeated $r_1/r_2/b_1/b_2$-storage format — an estimate

Let us start by stating the following result:

$$\sum_{i=1}^{\infty} \frac{1}{4^i} = 1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots = \frac{4}{3}.$$

What does this infinite series has to do with the memory? Well, imagine that we say that the original grid takes 1 unit of memory (a unit may be 6.3 MB, 1.7 GB, basically any number of bytes). Then, if we are going to store the r_2 nodes into a new grid, the 2nd grid, consisting of about 1/4th of the total number of nodes, we need a grid that takes 1/4 units of memory. The 3th level takes 1/16th units of memory, and so on. We see that in the ideal case only a factor 4/3 more memory is needed for using the repeated $r_1/r_2/b_1/b_2$-storage format compared to using only the original grid. We write "in the ideal case" because of the extra borders, one can imagine that for small problems the amount of extra memory can be much larger than that factor 4/3. So with the ideal case we mean very large grids, so that the width of the borders become negligable compared to the computing area.

19.5.3 Memory requirements for a 1.5M node test problem

The laboratorium version of the CUDA RRB-solver can provide valuable nformation while it runs. By setting the switches DISPLAY_GRIDS = 1 and DISPLAY_MEMORYUSAGE = 1 the CUDA RRB-solver will generate a .txt-file with information on the grid levels and the amount of allocated memory on the device. With the switch DISPLAY_PROPERTIES also the device is listed on which the experiments run. Below we have included the generated output file for the 1.5M Plymouth test problem when the aforementioned switches are turned on.

```
Display grid is enabled [DISPLAY_GRIDS = 1]
============================================================
Problem size (given): 1250 x 1200

Grid level         nx      ny              cx      cy
============================================================
 1               1264    1328             608     640
 2                688     688             320     320
 3                368     368             160     160
 4                240     240              96      96
 5                176     176              64      64
 6                112     112              32      32
============================================================

Display device props is enabled [DISPLAY_PROPERTIES = 1]

The \texttt{CUDA} RRB-solver will run on:

Device name:                    GeForce GTX 580
============================================================
```

```
Total global memory:            1572032 KB
Shared memory per block:        48 KB
Number of registers:            32768
Warp size in threads:           32
Max. nr. threads per block:     1024
Compute capability:             2.0
Number of SMs:                  16
Number of SPs per SM:           32


Display memory usage is enabled[DISPLAY_MEMORYUSAGE = 1]
============================================================

1) Allocating memory for preconditioner (9 stencils):
Grid level         nx      ny           memory (B)
============================================================
 1               1264    1328             60429312
 2                688     688             17040384
 3                368     368              4875264
 4                240     240              2073600
 5                176     176              1115136
 6                112     112               451584
                                TOTAL:   85985280 Bytes
                        RUNNING TOTAL:   85985280 Bytes

2) Allocating memory for vectors x and b:
Name               nx      ny           memory (B)
============================================================
vector x         1248    1312              6549504
vector b         1248    1312              6549504
                                TOTAL:   13099008 Bytes
                        RUNNING TOTAL:   99084288 Bytes

3) Allocating memory for CG vectors (r1/r2/b1/b2 storage):
Name               nx      ny           memory (B)
============================================================
vector x         1264    1328              6714368
vector b         1264    1328              6714368
vector p         1264    1328              6714368
vector q         1264    1328              6714368
vector r         1264    1328              6714368
vector y         1264    1328              6714368
                                TOTAL:   40286208 Bytes
                        RUNNING TOTAL:  139370496 Bytes

4) Allocating memory for vector z(r1/r2/b1/b2 storage):
Grid level         nx      ny           memory (B)
============================================================
 1               1264    1328              6714368
 2                688     688              1893376
 3                368     368               541696
 4                240     240               230400
 5                176     176               123904
 6                112     112                50176
                                TOTAL:    9553920 Bytes
                        RUNNING TOTAL:  139370496 Bytes
```

```
5) Allocating memory for original stencils:
                    nx      ny           memory (B)
============================================================
center           1264    1328              6714368
north            1264    1328              6714368
east             1264    1328              6714368
south            1264    1328              6714368
west             1264    1328              6714368
                                 TOTAL:   33571840 Bytes
                         RUNNING TOTAL:  172942336 Bytes

6) Allocating memory for partial dot products:
        array size:       380                 1520 Bytes
                         RUNNING TOTAL:  172943856 Bytes
============================================================

7) Some memory must be left for handling stC, stW, stS:
                    nx      ny           memory (B)
============================================================
buffer           1248    1312              6549504

============================================================
                         SUPER TOTAL:    179493360 Bytes
============================================================
```

We observe that we need for the preconditioner about 86 MB memory for the repeated $r_1/r_2/b_1/b_2$-storage format. For 9 stencils in the standard format we would need about $\#\text{stencils} \cdot Nx1 \cdot Nx2 \cdot 4$ bytes $= 9 \cdot 1250 \cdot 1200 \cdot 4 = 54$ MB. So, in practice we need about a factor $86/54 = 1.6$ more storage space rather than the factor $4/3$ ($= 1.33$). In the next section we present a list with the memory requirements for many more different problem sizes.

19.5.4 An overview: memory versus problem size

In the table below the memory requirements for all test problems are listed.

Test problem	N_x	N_y	Total memory (MB)	repeated $r_1/r_2/b_1/b_2$ versus standard
Poisson 65k	256	256	10.0	2.1×
Poisson 262k	512	512	33.5	1.7×
Poisson 590k	768	768	71.3	1.6×
Poisson 1M	1,024	1,024	121.5	1.5×
Poisson 1.6M	1,280	1,280	187.3	1.5×
Poisson 2.4M	1,536	1,536	264.6	1.5×
Poisson 3.2M	1,792	1,792	356.6	1.5×
Poisson 4.2M	2,048	2,048	460.9	1.4×
IJssel 100k	500	200	18.4	2.6×
IJssel 200k	800	250	29.8	2.1×
IJssel 500k	1,000	500	64.1	1.7×
IJssel 1M	1,600	625	122.9	1.7×
IJssel 1.5M	1,500	1,000	179.2	1.6×
Plymouth 100k	400	250	16.7	2.4×
Plymouth 200k	500	400	30.0	2.1×
Plymouth 500k	800	625	65.9	1.8×
Plymouth 1M	800	1,250	125.6	1.7×
Plymouth 1.5M	1,200	1,250	179.5	1.7×
Port Presto 100k	400	250	16.7	2.4×
Port Presto 200k	500	400	30.0	2.1×
Port Presto 500k	800	625	65.9	1.8×
Port Presto 1M	1,000	1,000	121.5	1.6×
Port Presto 1.5M	1,200	1,250	179.5	1.6×

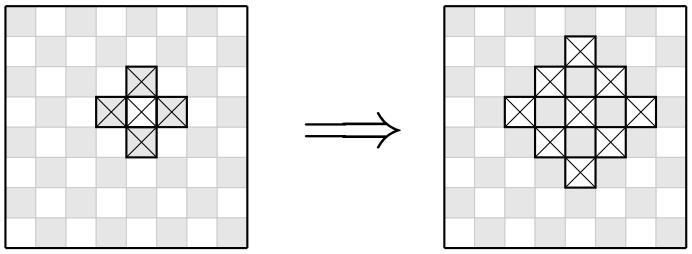
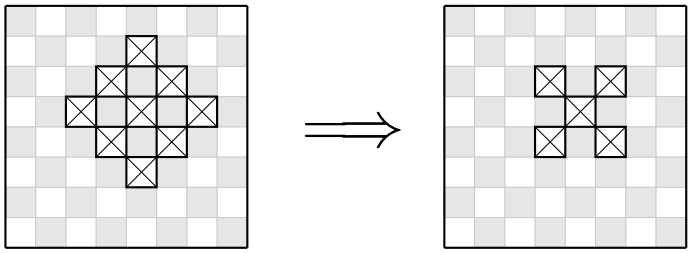
In this table the total memory is the memory requirement to store all data objects as listed in Section 19.5.1. The far most right column shows the ratio of storing the preconditioner matrix with the repeated $r_1/r_2/b_1/b_2$-storage format versus saving only the 9 stencils on the very first (finest) level. We see that for very large problems, i.e., the 2048×2048 node Poisson problem we see that the ratio approaches the theoretical factor 4/3, recall Section 19.5.1.

19.6 Constructing the preconditioning matrix M

In this section it is explained how the preconditioning matrix $M = LDL^T$ is constructed.

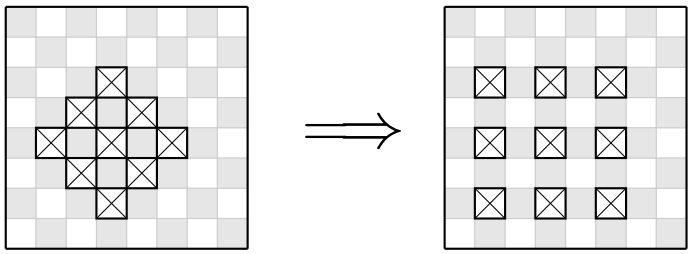
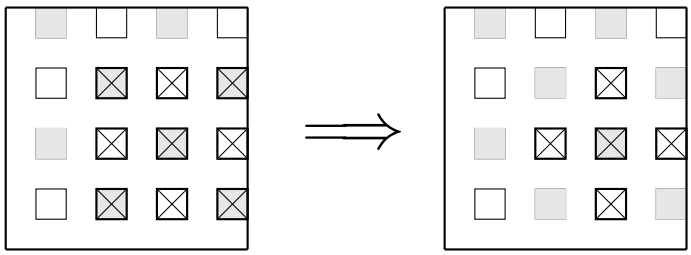
19.6.1 Algorithm

The preconditioning matrix M is constructed iteratively, that is, *level-wise*, as follows. The stencils StC, StW and StS that define the matrix S are copied into the three arrays `ccc`, `cww` and `css`. Further, the west and south stencil are used to make `cee` and `cnn`, and four additional arrays, `cne, cse, csw, cnw`, are created. Then the following steps are performed in the indicating order:

Level	Phase	Description
1st	Phase 2a	elimination of black nodes (5-point → 9-point) using the 5-point stencils for surrounding black nodes  for both r_1- and r_2-nodes
1st	Phase 2b	lumping (9-point → 5-point)  only for r_1-nodes

level	Phase	description

1st	Phase 2c	memory efficiency
		only for r_1-nodes: `cnn` $\leftarrow\!\shortmid$ `cnw`
		`cee` $\leftarrow\!\shortmid$ `cne`
		`cww` $\leftarrow\!\shortmid$ `csw`
		`css` $\leftarrow\!\shortmid$ `cse`

1st	Phase 3	elimination of 1st level red nodes using the 5-point stencil from Phase 2b  only for r_2-nodes (= nodes in 2nd level)
2nd	Phase 1	lumping (9-point $\rightarrow$ 5-point)  only for black nodes in 2nd level
2nd	Phase 2a	see above (repeated on $2\times$ coarser grid)

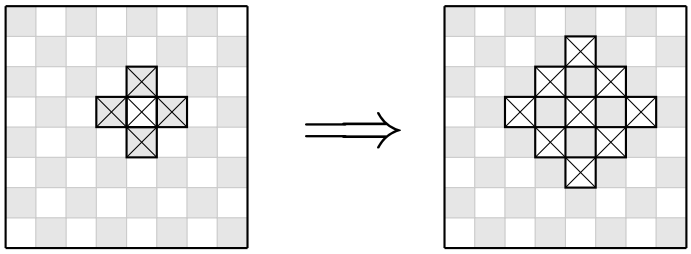
In the end `ccc, cnn, cee, css, cww` are divided by `ccc`.

level	Phase	description

1st	Phase 4	dividing by main diagonal
		for all nodes in 1st level: `ccc` $\leftarrow\!\shortmid$ 1
		`cnn` $\leftarrow\!\shortmid$ `cnn` / `ccc`
		`cee` $\leftarrow\!\shortmid$ `cee` / `ccc`
		`css` $\leftarrow\!\shortmid$ `css` / `ccc`
		`cww` $\leftarrow\!\shortmid$ `cww` / `ccc`

19.6.2 Phase 2a: elimination of black nodes

In Phase 2a the black nodes are eliminated. By doing so the 5-point stencil becomes a 9-point stencil.



The black nodes are eliminited with Gaussian elimination. When Gaussion elimination is applied to the 1st level, the resulting 9-point stencil defines the matrix S_1 which is called the 1st Schur complement, see Section 19.1.5. Using stencil notation the computation of $D_r - S_{rb}D_b^{-1}S_{br}$ translates to:

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N_b & 0 & 0 \\ 0 & W_b & C_r & E_b & 0 \\ 0 & 0 & S_b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{center}} - \frac{N_b}{C_b^N} \underbrace{\begin{bmatrix} 0 & 0 & N_r^N & 0 & 0 \\ 0 & W_r^N & C_b^N & E_r^N & 0 \\ 0 & 0 & S_r^N & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{north}} - \frac{E_b}{C_b^E} \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & N_r^E & 0 \\ 0 & 0 & W_r^E & C_b^E & E_r^E \\ 0 & 0 & 0 & S_r^E & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{east}}$$

$$- \frac{S_b}{C_b^S} \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N_r^S & 0 & 0 \\ 0 & W_r^S & C_b^S & E_r^S & 0 \\ 0 & 0 & S_b^S & 0 & 0 \end{bmatrix}}_{\text{south}} - \frac{W_b}{C_b^W} \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & N_r^W & 0 & 0 & 0 \\ W_r^W & C_b^W & E_r^W & 0 & 0 \\ 0 & S_r^W & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{west}}$$

$$= \begin{bmatrix} 0 & 0 & -\frac{N_b N_r^N}{C_b^N} & 0 & 0 \\ 0 & -\frac{N_b W_r^N}{C_b^N} - \frac{W_b N_r^W}{C_b^S} & 0 & -\frac{N_b E_r^N}{C_b^N} - \frac{E_b N_r^E}{C_b^E} & 0 \\ -\frac{W_b W_r^W}{C_b^S} & 0 & C_r - \frac{N_b S_r^N}{C_b^N} - \frac{E_b W_r^E}{C_b^E} - \frac{S_b N_r^S}{C_b^S} - \frac{W_b E_r^N}{C_b^S} & 0 & -\frac{E_b E_r^E}{C_b^E} \\ 0 & -\frac{S_b W_r^S}{C_b^S} - \frac{W_b S_r^N}{C_b^S} & 0 & -\frac{E_b S_r^E}{C_b^E} - \frac{S_b E_r^S}{C_b^S} & 0 \\ 0 & 0 & -\frac{S_b S_b^N}{C_b^N} & 0 & 0 \end{bmatrix}.$$

In C++ this can be implemented as follows.

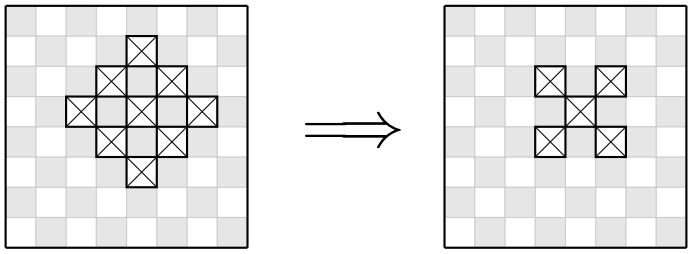
```
// r1 nodes:
for (j = st; j <= Nx2; j += tw) {
    for (i = st; i <= Nx1; i += tw) {
        // north
        cne[i][j] -=   cnn[i][j] * cee[i][j+st] / ccc[i][j+st];
        cnw[i][j] -=   cnn[i][j] * cww[i][j+st] / ccc[i][j+st];
        ccc[i][j] -=   cnn[i][j] * css[i][j+st] / ccc[i][j+st];
        cnn[i][j]  = - cnn[i][j] * cnn[i][j+st] / ccc[i][j+st];

        // south
        cse[i][j] -=   css[i][j] * cee[i][j-st] / ccc[i][j-st];
        csw[i][j] -=   css[i][j] * cww[i][j-st] / ccc[i][j-st];
        ccc[i][j] -=   css[i][j] * cnn[i][j-st] / ccc[i][j-st];
        css[i][j]  = - css[i][j] * css[i][j-st] / ccc[i][j-st];

        // east
        cne[i][j] -=   cee[i][j] * cnn[i+st][j] / ccc[i+st][j];
        cse[i][j] -=   cee[i][j] * css[i+st][j] / ccc[i+st][j];
        ccc[i][j] -=   cee[i][j] * cww[i+st][j] / ccc[i+st][j];
        cee[i][j]  = - cee[i][j] * cee[i+st][j] / ccc[i+st][j];

        // west
        cnw[i][j] -=   cww[i][j] * cnn[i-st][j] / ccc[i-st][j];
        csw[i][j] -=   cww[i][j] * css[i-st][j] / ccc[i-st][j];
        ccc[i][j] -=   cww[i][j] * cee[i-st][j] / ccc[i-st][j];
        cww[i][j]  = - cww[i][j] * cww[i-st][j] / ccc[i-st][j];
    }
}
```

For the r_2-nodes similar code can be used (just change the initializers of the for-loops into `i = tw` and `j = tw`). Check for yourself that the code snippet above indeed computes the elements in the 9-point stencil above. The code is close to the actual implementation in C++. Note that lots of terms occur multiple times, which motivates to introduce extra variables to store common values. This will increase readability of the code, and, perhaps, may even increase performance.

In CUDA Phase 2a, Phase 2b and Phase 2c are combined into 1 kernel. In Listing 19.2 the corresponding CUDA code can be found.

19.6.3 Phase 2b: lumping

In Phase 2b the 9-point stencil is reduced to a 5-point stencil by using a method which is known as lumping.



Basically lumping is nothing more than removing stencil dependencies by adding the respective coefficients to (an)other coefficient(s). Although there are lots of different possibilities, according to [28] it is best to use the lumping strategy that corresponds to the Modified Incomplete Cholesky decomposition (with $\omega = 1$). If the 9-point stencil from Step 2a is notated as

$$\begin{bmatrix} 0 & 0 & NN_r & 0 & 0 \\ 0 & NW_r & 0 & NE_r & 0 \\ WW_r & 0 & CC_r & 0 & EE_r \\ 0 & SW_r & 0 & SE_r & 0 \\ 0 & 0 & SS_r & 0 & 0 \end{bmatrix},$$

than the 5-point stencil after lumping is

$$\begin{bmatrix} NW_r & 0 & NE_r \\ 0 & CC_r^* & 0 \\ SW_r & 0 & SE_r \end{bmatrix},$$

where CC_r^* is given by

$$CC_r^* := CC_r + NN_r + EE_r + SS_r + WW_r.$$

Suitable C++ code would be:

```
// r1 nodes:
for (j = st; j <= Nx2; j += tw) {
    for (i = st; i <= Nx1; i += tw) {
        ccc[i][j] = ccc[i][j] +  // center
                    cnn[i][j] +  // north
                    cee[i][j] +  // east
                    css[i][j] +  // south
                    cww[i][j];   // west
    }
}
```

In CUDA Phase 2a, Phase 2b and Phase 2c are combined into 1 kernel. In Listing 19.2 the corresponding CUDA code can be found.

19.6.4 Phase 2c: memory efficiency

In this step a clever trick is used which saves memory. For the r_1-nodes the north-west stencils ($StNW$) for those nodes are saved in the north stencils (StN), the north-east stencils ($StNE$) in the east stencils ($StEE$), the south-east stencils ($StSE$) in the south stencils ($StSS$) and the south-west stencils ($StSW$) in the west stencils ($StWW$). By doing so the stencils $StNE$, $StSE$, $StSW$ and $StNW$ can be deleted as soon as the preconditioning matrix M is made. The CG-part of the solver will only use the remaining 5 stencils. Corresponding C++ code is:

```
// r1 nodes:
for (j = st; j <= Nx2; j += tw) {
    for (i = st; i <= Nx1; i += tw) {
        cnn[i][j] = cnw[i][j];
        cee[i][j] = cne[i][j];
        css[i][j] = cse[i][j];
        cww[i][j] = csw[i][j];
    }
}
```

In CUDA Phase 2a, Phase 2b and Phase 2c are combined into the kernel `prec::kernel_prec2()`. In Listing 19.2 the corresponding CUDA code can be found.

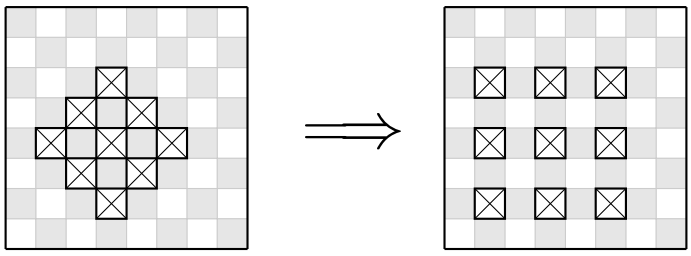
Listing 19.2: CUDA kernel prec::kernel_prec2().

```
1  template <class T>
2  __global__ void kernel_prec2(Grid g)
3  {
4      int ld = g.nx;
5
6      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7      int v_r1 = BORDER_WIDTH + by * By + ty;
8      int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
9      int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
11     int v_b1 = BORDER_WIDTH + by * By + ty;
12     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
13     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
14
15     int loc_r1 = ld * v_r1 + u_r1;
16     int loc_r2 = ld * v_r2 + u_r2;
17     int loc_b1 = ld * v_b1 + u_b1;
18     int loc_b2 = ld * v_b2 + u_b2;
19
20     T sum_cc = 0;
21     T sum_ne = 0;
22     T sum_nw = 0;
23     T sum_nn = 0;
24     T sum_se = 0;
25     T sum_sw = 0;
26     T sum_ss = 0;
27     T sum_ee = 0;
28     T sum_ww = 0;
29
30     const T val_nn_b1 = __FETCH_NN(u_b1, v_b1, ld);
31     const T val_ee_b1 = __FETCH_EE(u_b1, v_b1, ld);
32     const T val_ss_b1 = __FETCH_SS(u_b1, v_b1, ld);
33     const T val_ww_b1 = __FETCH_WW(u_b1, v_b1, ld);
```

```
34
35      const T val_nn_b2 = __FETCH_NN(u_b2, v_b2, ld);
36      const T val_ee_b2 = __FETCH_EE(u_b2, v_b2, ld);
37      const T val_ss_b2 = __FETCH_SS(u_b2, v_b2, ld);
38      const T val_ww_b2 = __FETCH_WW(u_b2, v_b2, ld);
39
40      const T val_cc_b1 = __FETCH_CC(u_b1, v_b1, ld);
41      const T val_cc_b2 = __FETCH_CC(u_b2, v_b2, ld);
42
43      T a, b;
44
45      // r1 nodes:
46      a = 1 / val_cc_b2;
47      b = g.nn[loc_r1] * a;
48      sum_ne  = g.ne[loc_r1] - b * val_ee_b2;
49      sum_cc  = g.cc[loc_r1] - b * val_ss_b2;
50      sum_nw  = g.nw[loc_r1] - b * val_ww_b2;
51      sum_nn  =              - b * val_nn_b2;
52
53      a = 1 / __FETCH_CC(u_b2, v_b2-1, ld);
54      b = g.ss[loc_r1] * a;
55      sum_se  = g.se[loc_r1] - b * g.ee[loc_b2 - ld];
56      sum_sw  = g.sw[loc_r1] - b * g.ww[loc_b2 - ld];
57      sum_cc -=                b * g.nn[loc_b2 - ld];
58      sum_ss  =              - b * g.ss[loc_b2 - ld];
59
60      a = 1 / val_cc_b1;
61      b = g.ee[loc_r1] * a;
62      sum_ne -=                b * val_nn_b1;
63      sum_se -=                b * val_ss_b1;
64      sum_cc -=                b * val_ww_b1;
65      sum_ee  =              - b * val_ee_b1;
66
67      a = 1 / __FETCH_CC(u_b1-1, v_b1, ld);
68      b = g.ww[loc_r1] * a;
69      sum_nw -=                b * __FETCH_NN(u_b1-1, v_b1, ld);
70      sum_sw -=                b * __FETCH_SS(u_b1-1, v_b1, ld);
71      sum_cc -=                b * __FETCH_EE(u_b1-1, v_b1, ld);
72      sum_ww  =              - b * __FETCH_WW(u_b1-1, v_b1, ld);
73
74      g.cc[loc_r1] = sum_cc + sum_nn + sum_ee + sum_ss + sum_ww;
75      g.nn[loc_r1] = sum_ne;
76      g.ee[loc_r1] = sum_se;
77      g.ww[loc_r1] = sum_nw;
78      g.ss[loc_r1] = sum_sw;
79      g.ne[loc_r1] = sum_ne;
80      g.se[loc_r1] = sum_se;
81      g.sw[loc_r1] = sum_sw;
82      g.nw[loc_r1] = sum_nw;
83
84      // r2 nodes:
85      a = 1 / __FETCH_CC(u_b1, v_b1+1, ld);
86      b = g.nn[loc_r2] * a;
87      sum_ne  = g.ne[loc_r2] - b * g.ee[loc_b1 + ld];
88      sum_cc  = g.cc[loc_r2] - b * g.ss[loc_b1 + ld];
89      sum_nw  = g.nw[loc_r2] - b * g.ww[loc_b1 + ld];
90      sum_nn  =              - b * g.nn[loc_b1 + ld];
```

```
91
92      a = 1 / val_cc_b1;
93      b = g.ss[loc_r2] * a;
94      sum_se  = g.se[loc_r2] - b * val_ee_b1;
95      sum_sw  = g.sw[loc_r2] - b * val_ww_b1;
96      sum_cc -=                b * val_nn_b1;
97      sum_ss  =              - b * val_ss_b1;
98
99      a = 1 / __FETCH_CC(u_b2+1, v_b2, ld);
100     b = g.ee[loc_r2] * a;
101     sum_ne -=                b * __FETCH_NN(u_b2+1, v_b2, ld);
102     sum_se -=                b * __FETCH_SS(u_b2+1, v_b2, ld);
103     sum_cc -=                b * __FETCH_WW(u_b2+1, v_b2, ld);
104     sum_ee  =              - b * __FETCH_EE(u_b2+1, v_b2, ld);
105
106     a = 1 / val_cc_b2;
107     b = g.ww[loc_r2] * a;
108     sum_nw -=                b * val_nn_b2;
109     sum_sw -=                b * val_ss_b2;
110     sum_cc -=                b * val_ee_b2;
111     sum_ww  =              - b * val_ww_b2;
112
113     g.cc[loc_r2] = sum_cc;
114     g.nn[loc_r2] = sum_nn;
115     g.ss[loc_r2] = sum_ss;
116     g.ee[loc_r2] = sum_ee;
117     g.ww[loc_r2] = sum_ww;
118     g.ne[loc_r2] = sum_ne;
119     g.se[loc_r2] = sum_se;
120     g.sw[loc_r2] = sum_sw;
121     g.nw[loc_r2] = sum_nw;
122 }
```

19.6.5 Phase 3: elimination of the red nodes which are not in the next level



Using stencil notation the elimination of the red nodes which are not in the next level is given by:

$$\underbrace{\begin{bmatrix} 0 & 0 & NN_r & 0 & 0 \\ 0 & NW_r & 0 & NE_r & 0 \\ WW_r & 0 & CC_r & 0 & EE_r \\ 0 & SW_r & 0 & SE_r & 0 \\ 0 & 0 & SS_r & 0 & 0 \end{bmatrix}}_{\text{center}} - \frac{NE_r}{CC_r^{NE}} \underbrace{\begin{bmatrix} 0 & 0 & NW_r^{NE} & 0 & NE_r^{NE} \\ 0 & 0 & 0 & CC_r^{NE} & 0 \\ 0 & 0 & SW_r^{NE} & 0 & SE_r^{NE} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{north-east}}$$

$$- \frac{SE_r}{CC_r^{SE}} \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & NW_r^{SE} & 0 & NE_r^{SE} \\ 0 & 0 & 0 & CC_r^{NE} & 0 \\ 0 & 0 & SW_r^{SE} & 0 & SE_r^{SE} \end{bmatrix}}_{\text{south-east}}$$

$$- \frac{SW_r}{CC_r^{SW}} \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ NW_r^{SW} & 0 & NE_r^{SW} & 0 & 0 \\ 0 & CC_r^{SW} & 0 & 0 & 0 \\ SW_r^{SW} & 0 & SE_r^{SW} & 0 & 0 \end{bmatrix}}_{\text{south-west}}$$

$$- \frac{NW_r}{CC_r^{NW}} \underbrace{\begin{bmatrix} NW_r^{NW} & 0 & NE_r^{NW} & 0 & 0 \\ 0 & CC_r^{NW} & 0 & 0 & 0 \\ SW_r^{NW} & 0 & SE_r^{NW} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{north-west}}$$

$$= \begin{bmatrix} NW_r^* & 0 & NN_r^* & 0 & NE_r^* \\ 0 & 0 & 0 & 0 & 0 \\ WW_r^* & 0 & CC_r^* & 0 & EE_r^* \\ 0 & 0 & 0 & 0 & 0 \\ SW_r^* & 0 & SS_r^* & 0 & SE_r^* \end{bmatrix},$$

where

$$\begin{aligned}
NW_r^* &:= -\frac{NW_r NW_r^{NW}}{CC_r^{NW}}, \\
NE_r^* &:= -\frac{NE_r NE_r^{NE}}{CC_r^{NE}}, \\
SE_r^* &:= -\frac{SE_r SE_r^{SE}}{CC_r^{SE}}, \\
SW_r^* &:= -\frac{SW_r SW_r^{SW}}{CC_r^{SW}}, \\
NN_r^* &:= NN_r - \frac{NE_r NW_r^{NE}}{CC_r^{NE}} - \frac{NW_r NE_r^{NW}}{CC_r^{NW}}, \\
EE_r^* &:= EE_r - \frac{NE_r SE_r^{NE}}{CC_r^{NE}} - \frac{SE_r NE_r^{SE}}{CC_r^{SE}}, \\
SS_r^* &:= SS_r - \frac{SE_r SW_r^{SE}}{CC_r^{SE}} - \frac{SW_r SE_r^{SW}}{CC_r^{SW}}, \\
WW_r^* &:= WW_r - \frac{SW_r NW_r^{SW}}{CC_r^{SW}} - \frac{NW_r SW_r^{NW}}{CC_r^{NW}}, \\
CC_r^* &:= CC_r - \frac{NE_r SW_r^{NE}}{CC_r^{NE}} - \frac{SE_r NW_r^{SE}}{CC_r^{SE}} - \frac{SW_r NE_r^{SW}}{CC_r^{SW}} - \frac{NW_r SE_r^{NW}}{CC_r^{NW}}.
\end{aligned}$$

In C++ these computations can be implemented as follows:

```
for (j = 0; j <= Nx2; j += tw) {
    for (i = 0; i <= Nx1; i += tw) {
        // north-east
        cnn[i][j] -=   cne[i][j] * cnw[i+st][j+st] / ccc[i+st][j+st];
        cee[i][j] -=   cne[i][j] * cse[i+st][j+st] / ccc[i+st][j+st];
        ccc[i][j] -=   cne[i][j] * csw[i+st][j+st] / ccc[i+st][j+st];
        cne[i][j]  = - cne[i][j] * cne[i+st][j+st] / ccc[i+st][j+st];

        // north-west
        cnn[i][j] -=   cnw[i][j] * cne[i-st][j+st] / ccc[i-st][j+st];
        cww[i][j] -=   cnw[i][j] * csw[i-st][j+st] / ccc[i-st][j+st];
        ccc[i][j] -=   cnw[i][j] * cse[i-st][j+st] / ccc[i-st][j+st];
        cnw[i][j]  = - cnw[i][j] * cnw[i-st][j+st] / ccc[i-st][j+st];

        // south-west
        css[i][j] -=   csw[i][j] * cse[i-st][j-st] / ccc[i-st][j-st];
        cww[i][j] -=   csw[i][j] * cnw[i-st][j-st] / ccc[i-st][j-st];
        ccc[i][j] -=   csw[i][j] * cne[i-st][j-st] / ccc[i-st][j-st];
        csw[i][j]  = - csw[i][j] * csw[i-st][j-st] / ccc[i-st][j-st];
```

```
        // south-east
        css[i][j] -=   cse[i][j] * csw[i+st][j-st] / ccc[i+st][j-st];
        cee[i][j] -=   cse[i][j] * cne[i+st][j-st] / ccc[i+st][j-st];
        ccc[i][j] -=   cse[i][j] * cnw[i+st][j-st] / ccc[i+st][j-st];
        cse[i][j]  = - cse[i][j] * cse[i+st][j-st] / ccc[i+st][j-st];
    }
}
```

In CUDA this translates to the code in Listing 19.3.

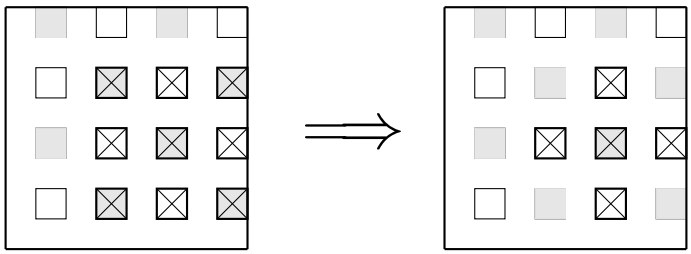
Listing 19.3: CUDA kernel prec::kernel_prec3().

```
1  template <class T>
2  __global__ void kernel_prec3(Grid g)
3  {
4      int ld = g.nx;
5
6      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7      int v_r1 = BORDER_WIDTH + by * By + ty;
8      int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
9      int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11     int loc_r2 = ld * v_r2 + u_r2;
12
13     T sum_cc = 0;
14     T sum_ne = 0;
15     T sum_nw = 0;
16     T sum_nn = 0;
17     T sum_se = 0;
18     T sum_sw = 0;
19     T sum_ss = 0;
20     T sum_ee = 0;
21     T sum_ww = 0;
22
23     T a, b;
24
25     const T val_cc_r1 = 1 / __FETCH_CC(u_r1, v_r1, ld);
26
27     a = 1 / __FETCH_CC(u_r1+1, v_r1+1, ld);
28     b = g.ne[loc_r2] * a;
29     sum_nn  = g.nn[loc_r2] - b * __FETCH_NW(u_r1+1, v_r1+1, ld);
30     sum_ee  = g.ee[loc_r2] - b * __FETCH_SE(u_r1+1, v_r1+1, ld);
31     sum_cc  = g.cc[loc_r2] - b * __FETCH_SW(u_r1+1, v_r1+1, ld);
32     sum_ne  =              - b * __FETCH_NE(u_r1+1, v_r1+1, ld);
33
34     a = 1 / __FETCH_CC(u_r1, v_r1+1, ld);
35     b = g.nw[loc_r2] * a;
36     sum_nn -=                b * __FETCH_NE(u_r1, v_r1+1, ld);
37     sum_ww  = g.ww[loc_r2] - b * __FETCH_SW(u_r1, v_r1+1, ld);
38     sum_cc -=                b * __FETCH_SE(u_r1, v_r1+1, ld);
39     sum_nw  =              - b * __FETCH_NW(u_r1, v_r1+1, ld);
40
41     a = val_cc_r1;
42     b = g.sw[loc_r2] * a;
43     sum_ss  = g.ss[loc_r2] - b * __FETCH_SE(u_r1, v_r1, ld);
44     sum_ww -=                b * __FETCH_NW(u_r1, v_r1, ld);
45     sum_cc -=                b * __FETCH_NE(u_r1, v_r1, ld);
46     sum_sw  =              - b * __FETCH_SW(u_r1, v_r1, ld);
```

```
47
48      a = 1 / __FETCH_CC(u_r1+1, v_r1, ld);
49      b = g.se[loc_r2] * a;
50      sum_ss -=                b * __FETCH_SW(u_r1+1, v_r1, ld);
51      sum_ee -=                b * __FETCH_NE(u_r1+1, v_r1, ld);
52      sum_cc -=                b * __FETCH_NW(u_r1+1, v_r1, ld);
53      sum_se  =              - b * __FETCH_SE(u_r1+1, v_r1, ld);
54
55      g.cc[loc_r2] = sum_cc;
56      g.ne[loc_r2] = sum_ne;
57      g.nw[loc_r2] = sum_nw;
58      g.nn[loc_r2] = sum_nn;
59      g.se[loc_r2] = sum_se;
60      g.sw[loc_r2] = sum_sw;
61      g.ss[loc_r2] = sum_ss;
62      g.ee[loc_r2] = sum_ee;
63      g.ww[loc_r2] = sum_ww;
64  }
```

19.6.6 Phase 1: lumping

In Phase 1 the 9-point stencil is reduced to a 5-point stencil by using a method which is known as lumping.



Basically lumping is nothing more than removing stencil dependencies by adding the respective coefficients to (an)other coefficient(s). Although there are lots of different possibilities, according to [28] it is best to use the lumping strategy that corresponds to the Modified Incomplete Cholesky decomposition (with $\omega = 1$). If the 9-point stencil from Step 3 is notated as

$$\begin{bmatrix} NW_b & NN_r & NE_b \\ WW_r & CC_b & EE_r \\ SW_b & SS_r & SE_b \end{bmatrix},$$

than the 5-point stencil after lumping is

$$\begin{bmatrix} 0 & NN_r & 0 \\ WW_r & CC_b^* & EE_r \\ 0 & SS_r & 0 \end{bmatrix},$$

where CC_b^* is given by

$$CC_b^* := CC_b + NE_b + SE_b + SW_b + NW_b.$$

Suitable C++ code would be:

```
if (level > 1) // skip first level
{
    // b1 nodes:
    for (j = st; j <= Nx2; j += tw) {
        for (i = tw; i <= Nx1; i += tw) {
            ccc[i][j] = ccc[i][j] +  // center
                        cne[i][j] +  // north-east
                        cse[i][j] +  // south-east
                        csw[i][j] +  // south-west
                        cnw[i][j];   // north-west
        }
    }
    // b2 nodes:
    for (j = tw; j <= Nx2; j += tw) {
        for (i = st; i <= Nx1; i += tw) {
            ccc[i][j] = ccc[i][j] +  // center
                        cne[i][j] +  // north-east
                        cse[i][j] +  // south-east
                        csw[i][j] +  // south-west
```

```
                        cnw[i][j];   // north-west
        }
    }
}
```

In CUDA this translates to the code in Listing 19.4.

Listing 19.4: CUDA kernel prec::kernel_prec1().

```
1  template <class T>
2  __global__ void kernel_prec1(Grid g)
3  {
4      int ld = g.nx;
5
6      int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
7      int v_b1 = BORDER_WIDTH + by * By + ty;
8      int u_b2 = BORDER_WIDTH + bx * Bx + tx;
9      int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11     int loc_b1 = ld * v_b1 + u_b1;
12     int loc_b2 = ld * v_b2 + u_b2;
13
14     g.cc[loc_b1] = g.cc[loc_b1] +
15                    g.ne[loc_b1] +
16                    g.nw[loc_b1] +
17                    g.se[loc_b1] +
18                    g.sw[loc_b1];
19
20     g.cc[loc_b2] = g.cc[loc_b2] +
21                    g.ne[loc_b2] +
22                    g.nw[loc_b2] +
23                    g.se[loc_b2] +
24                    g.sw[loc_b2];
25 }
```

19.6.7 Phase 4: dividing by main diagonal

To get the factorization $M = LDL^T$ it remains to divide the stencils (which define L) by the diagonal D. In C++ this can be done as follows:

```
for (i = 0; i <= Nx1; ++i) {
    for (j = 0; j <= Nx2; ++j) {
        float fac = 1.0 / ccc[i][j];
        ccc[i][j]  = fac;
        cnn[i][j] /= fac;
        css[i][j] /= fac;
        cee[i][j] /= fac;
        cww[i][j] /= fac;
    }
}
```

Because of the repeated $r_1/r_2/b_1/b_2$-storage format in CUDA we have to do this level-wise. For each grid we only divide the r_1-nodes by the center stencil, see Listing 19.5.

Listing 19.5: CUDA kernel prec::kernel_prec4().

```
1  template <class T>
2  __global__ void kernel_prec4(Grid g)
3  {
4      int ld = g.nx;
5
6      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7      int v_r1 = BORDER_WIDTH + by * By + ty;
8      int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
9      int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
11     int v_b1 = BORDER_WIDTH + by * By + ty;
12     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
13     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
14
15     int loc_r1 = ld * v_r1 + u_r1;
16     int loc_r2 = ld * v_r2 + u_r2;
17     int loc_b1 = ld * v_b1 + u_b1;
18     int loc_b2 = ld * v_b2 + u_b2;
19
20     T fac;
21
22     // r1 nodes:
23     fac = 1 / g.cc[loc_r1];
24     g.nn[loc_r1] *= fac;
25     g.ss[loc_r1] *= fac;
26     g.ee[loc_r1] *= fac;
27     g.ww[loc_r1] *= fac;
28     g.cc[loc_r1]  = fac;
29
30     // r2 nodes:
31     fac = 1 / g.cc[loc_r2];
32     g.nn[loc_r2] *= fac;
33     g.ss[loc_r2] *= fac;
34     g.ee[loc_r2] *= fac;
35     g.ww[loc_r2] *= fac;
36     g.cc[loc_r2]  = fac;
37
```

```
38      // b1 nodes:
39      fac = 1 / g.cc[loc_b1];
40      g.nn[loc_b1] *= fac;
41      g.ss[loc_b1] *= fac;
42      g.ee[loc_b1] *= fac;
43      g.ww[loc_b1] *= fac;
44      g.cc[loc_b1]  = fac;
45
46      // b2 nodes:
47      fac = 1 / g.cc[loc_b2];
48      g.nn[loc_b2] *= fac;
49      g.ss[loc_b2] *= fac;
50      g.ee[loc_b2] *= fac;
51      g.ww[loc_b2] *= fac;
52      g.cc[loc_b2]  = fac;
53  }
```

19.6.8 The final level

As explained earlier the final grid has a dimension of 16×16 or 32×32 nodes, i.e., the size of the compute-block. The final level is therefore handled by 1 SM of the GPU. Because of its small dimensions, the problem fits completely in the global memory's cache. All previous preconditioner kernels are combined in a single kernel, `prec::kernel_precfinal()`. The complete code is given in Listing 19.6. Notice the `__syncthreads()` statements, these are really importance; without them the complete RRB-solver does not function properly.

Listing 19.6: CUDA kernel prec::kernel_precfinal().

```
1  template <class T>
2  __global__ void kernel_precfinal(Grid g)
3  {
4      int ld = g.nx;
5
6      int loc;
7
8      int st = 1;
9      int tw = 2;
10     int u, v;
11
12     T a, b;
13
14     for (int nt = DIM_COMPUTE_BLOCK / 2; nt >= 1; nt >>= 1)
15     {
16         // phase 1: for all black points
17         if (tx < nt) {
18             u = (st - 1) + tw * tx;
19             v = (tw - 1) + tw * ty;
20         } else {
21             u = (tw - 1) + tw * (tx - nt);
22             v = (st - 1) + tw * ty;
23         }
24         __syncthreads();
25         loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
26                   + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
27         if (ty < nt && tx < nt << 1) {
```

```
28              g.cc[loc] = g.cc[loc] +
29                          g.ne[loc] +
30                          g.nw[loc] +
31                          g.se[loc] +
32                          g.sw[loc];
33          }
34          __syncthreads();
35
36          // phase 2: for all red points
37          if (tx < nt) {
38              u = (st - 1) + tw * tx;
39              v = (st - 1) + tw * ty;
40          } else {
41              u = (tw - 1) + tw * (tx - nt);
42              v = (tw - 1) + tw * ty;
43          }
44          loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
45                   + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
46
47          if (ty < nt && tx < nt << 1) {
48              a = 1 / g.cc[loc + st * ld];
49              b = g.nn[loc] * a;
50              g.ne[loc] -=   b * g.ee[loc + st * ld];
51              g.cc[loc] -=   b * g.ss[loc + st * ld];
52              g.nw[loc] -=   b * g.ww[loc + st * ld];
53              g.nn[loc]  = - b * g.nn[loc + st * ld];
54
55              a = 1 / g.cc[loc - st * ld];
56              b = g.ss[loc] * a;
57              g.se[loc] -=   b * g.ee[loc - st * ld];
58              g.sw[loc] -=   b * g.ww[loc - st * ld];
59              g.cc[loc] -=   b * g.nn[loc - st * ld];
60              g.ss[loc]  = - b * g.ss[loc - st * ld];
61
62              a = 1 / g.cc[loc + st];
63              b = g.ee[loc] * a;
64              g.ne[loc] -=   b * g.nn[loc + st];
65              g.se[loc] -=   b * g.ss[loc + st];
66              g.cc[loc] -=   b * g.ww[loc + st];
67              g.ee[loc]  = - b * g.ee[loc + st];
68
69              a = 1 / g.cc[loc - st];
70              b = g.ww[loc] * a;
71              g.nw[loc] -=   b * g.nn[loc - st];
72              g.sw[loc] -=   b * g.ss[loc - st];
73              g.cc[loc] -=   b * g.ee[loc - st];
74              g.ww[loc]  = - b * g.ww[loc - st];
75          }
76          __syncthreads();
77
78          // phase 3: for red points not in next level
79          if (ty < nt && tx < nt) {
80              g.cc[loc] = g.cc[loc] +
81                          g.nn[loc] +
82                          g.ee[loc] +
83                          g.ss[loc] +
84                          g.ww[loc];
```

```
85              g.nn[loc] = g.ne[loc];
86              g.ee[loc] = g.se[loc];
87              g.ww[loc] = g.nw[loc];
88              g.ss[loc] = g.sw[loc];
89          }
90          __syncthreads();
91
92          // phase 4: for red points in next level
93          u = (tw - 1) + tw * tx;
94          v = (tw - 1) + tw * ty;
95          loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
96                    + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
97          if (ty < nt && tx < nt) {
98              a = 1 / g.cc[loc + st * ld + st];
99              b = g.ne[loc] * a;
100             g.nn[loc] -=   b * g.nw[loc + st * ld + st];
101             g.ee[loc] -=   b * g.se[loc + st * ld + st];
102             g.cc[loc] -=   b * g.sw[loc + st * ld + st];
103             g.ne[loc]  = - b * g.ne[loc + st * ld + st];
104
105             a = 1 / g.cc[loc + st * ld - st];
106             b = g.nw[loc] * a;
107             g.nn[loc] -=   b * g.ne[loc + st * ld - st];
108             g.ww[loc] -=   b * g.sw[loc + st * ld - st];
109             g.cc[loc] -=   b * g.se[loc + st * ld - st];
110             g.nw[loc]  = - b * g.nw[loc + st * ld - st];
111
112             a = 1 / g.cc[loc - st * ld - st];
113             b = g.sw[loc] * a;
114             g.ss[loc] -=   b * g.se[loc - st * ld - st];
115             g.ww[loc] -=   b * g.nw[loc - st * ld - st];
116             g.cc[loc] -=   b * g.ne[loc - st * ld - st];
117             g.sw[loc]  = - b * g.sw[loc - st * ld - st];
118
119             a = 1 / g.cc[loc - st * ld + st];
120             b = g.se[loc] * a;
121             g.ss[loc] -=   b * g.sw[loc - st * ld + st];
122             g.ee[loc] -=   b * g.ne[loc - st * ld + st];
123             g.cc[loc] -=   b * g.nw[loc - st * ld + st];
124             g.se[loc]  = - b * g.se[loc - st * ld + st];
125         }
126         __syncthreads();
127
128         st = tw;
129         tw *= 2;
130     }
131 }
```

19.7 Solving $Mz = r$

During the CG algorithm per iteration step the preconditioning step $Mz = r$ has to be solved for z. The preconditioner matrix M can be written as

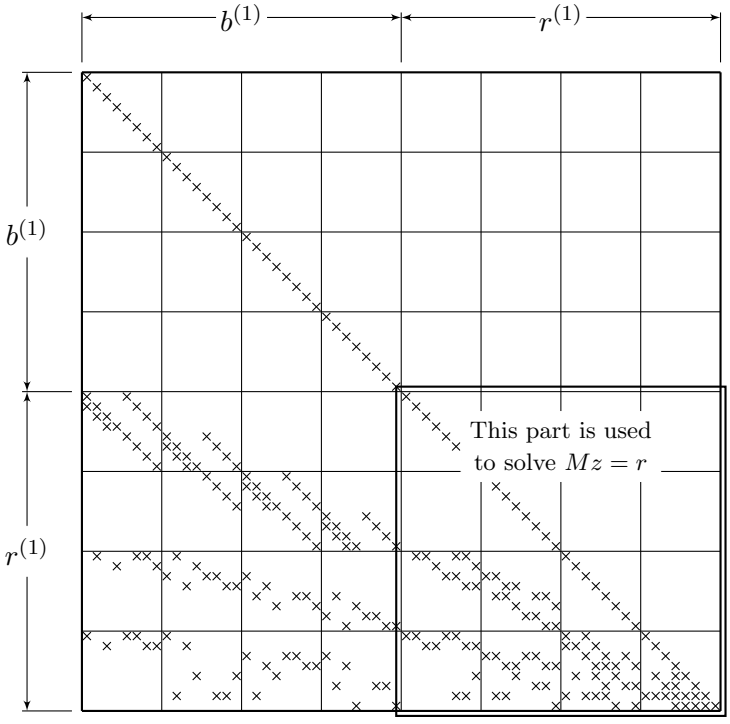
$$M = LDL^T$$

so that solving $Mz = r$ can be done in three steps as follows. Set $y := L^T z$ and $x := DL^T z = Dy$, then:

1. solve $Lx = r$ using forward substitution;
2. compute $y = D^{-1}x$;
3. solve $L^T z = y$ using backward substitution.

19.7.1 Preliminary work

In Figure 19.17 the Cholesky factor L is shown when the RRB-method is applied to matrix $S \in \mathbb{R}^{64\times 64}$ resulting from a 8×8 grid.



29	55	30	62	31	56	32	64
45	25	46	26	47	27	48	28
21	59	22	53	23	60	24	54
41	17	42	18	43	19	44	20
13	51	14	63	15	52	16	61
37	9	38	10	39	11	40	12
5	57	6	49	7	58	8	50
33	1	34	2	35	3	36	4

RRB-numbering

Figure 19.17: Cholesky factor L for matrix $S \in \mathbb{R}^{64\times 64}$ resulting from a 8×8 grid.

As emphasized earlier the CG algorithm in the RRB-solver operates on the red nodes only. In Figure 19.17 the first level red nodes are indicated by $r^{(1)}$. Using the $r^{(1)}$-nodes only means that only the indicated framed part of L is used to solve $Mz = r$. In Figure 19.18 we have zoomed in on the $r^{(1)}$-part of L. On the right in this figure the $r^{(1)}$-nodes are indicated.

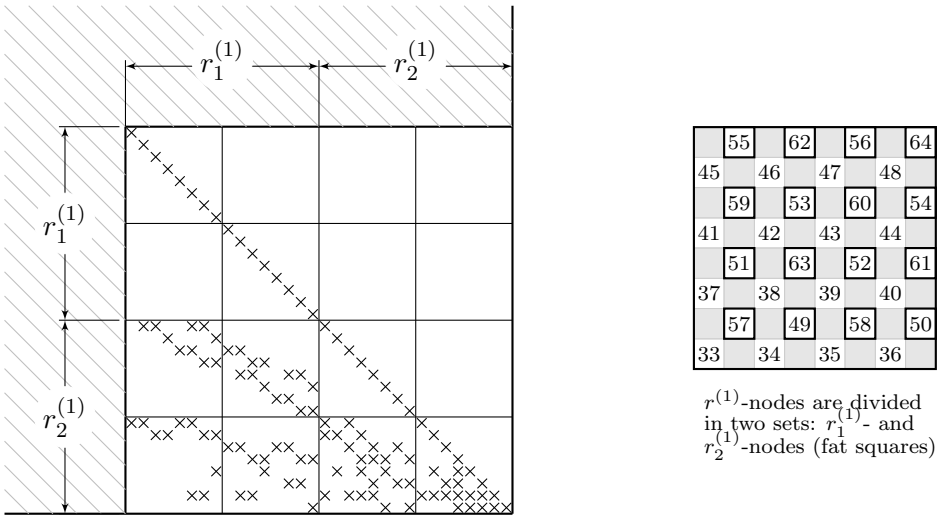


Figure 19.18: The first grid is divided into a set of red nodes and black nodes. The red nodes in turn are divided into 2 sets: $r_1^{(1)}$-nodes and $r_2^{(1)}$. The next level is always formed by the set of r_2-nodes, here: the second level is formed by the $r_2^{(1)}$-nodes. Those nodes are indicated on the right with fat boxes.

According to the $r1/r2/b1/b2$ numbering we have that the $r^{(1)}$-nodes are divided into two sets: $r_1^{(1)}$-nodes and $r_2^{(1)}$-nodes. For full clarity: the superscript () thus indicates the level we are at, and the subscript relates to the $r1/r2/b1/b2$ numbering.

The $r_2^{(1)}$-nodes together form the second level. We have seen that they are again divided in a set of black nodes and a set of red nodes. The second level is indicated in Figure 19.19.

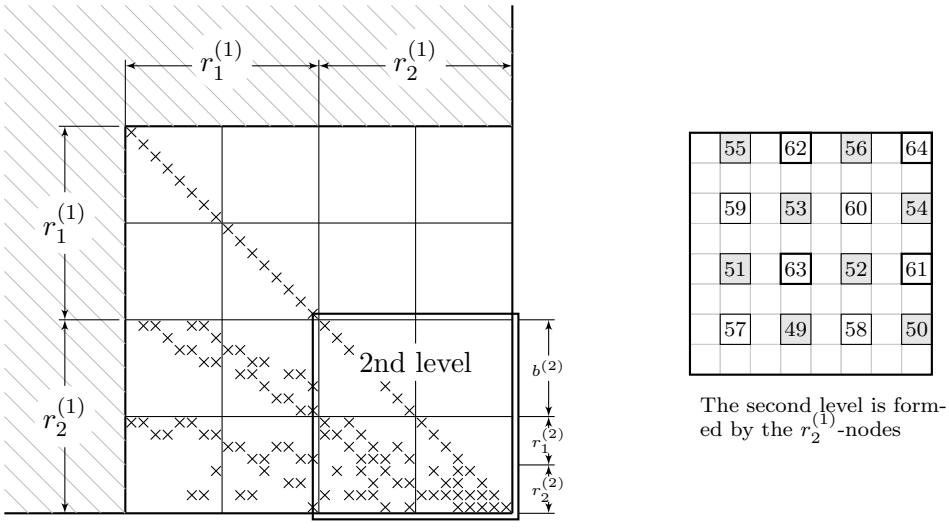


Figure 19.19: The second grid is again divided into a set of red nodes and black nodes. The red nodes consist of the $r_1^{(2)}$-nodes and $r_2^{(2)}$-nodes. The third level is formed by the $r_2^{(2)}$-nodes which are indicated on the right with fat boxes.

In Figure 19.20 we have zoomed in on the second level in which it can be seen how a third grid is formed.

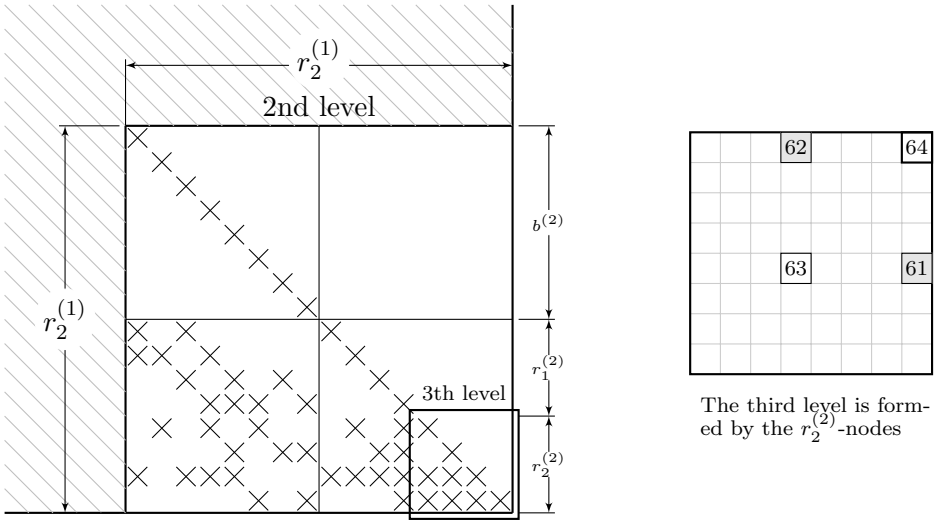


Figure 19.20: The third grid is again divided into a set of red nodes and black nodes. The red nodes consists of the $r_1^{(3)}$-nodes and $r_2^{(3)}$-nodes. The fourth and final level is formed by the single $r_2^{(3)}$-node which is indicated on the right with a fat box.

19.7.2 Step 1: Solving $Lx = r$

The first step is to solve $Lx = r$ for x using forward substitution. This is done level-wise using two steps:

- Phase 1: Updating r_2-nodes using r_1-nodes in the same level;
- Phase 2: Updating r_1- and r_2-nodes using b_1- and b_2-nodes in the same level.

The forward substitution is thus like:

$$\text{Phase 1} \to \text{Phase 2} \to \text{Phase 1} \to \text{Phase 2} \to \cdots \to \text{Phase 1} \to \text{Phase 2} \to \text{Phase 1},$$

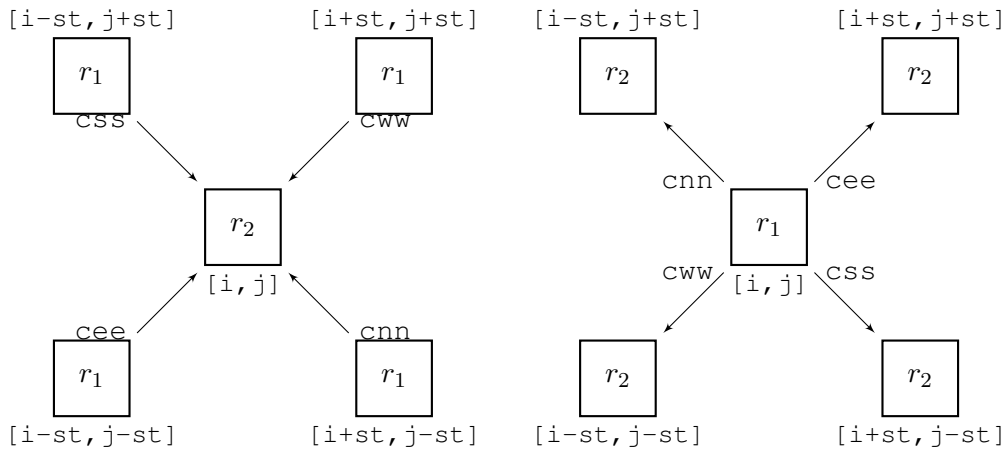
so starting with Phase 1 and finishing with Phase 1. Below both phases are discussed in more detail.

Phase 1

From Figure 19.18 it appears that r_2-nodes depend on r_1-nodes according to a rotated 5-point stencil. For example, r_2-node 59 depends on the r_1-nodes 41, 42, 45 and 46. Now note that the rotated 5-point stencils shows up every level.

Consider Figure 19.21 which shows the 5-point stencil more precisely. In Figure 19.21(a) we have shown how it would be straightforwardly done with a two-dimensional array storage format seen from the viewpoint of r_2-nodes; in Figure 19.21(b) we have shown how it would be done seen from the viewpoint of r_1-nodes. The latter version is the one used in the current

implementation of the C++ RRB-solver which uses the Array2D-class for storage. For the first level we have that tw = 1, i.e., the surrounding nodes lie directly around the node which is updated, for the second level we have tw = 2, for the third level tw = 4, etcetera. Hence the stride is increasing by a factor 2 each time we go a level up. Verify this for yourself by going through Figure 19.18 to Figure 19.20 again (the parts on the right).



(a) C++ version A (b) C++ version B

Figure 19.21: The rotated 5-point stencil for Phase 1. *On the left:* from the viewpoint of r_2-nodes. *On the right:* from the viewpoint of r_1-nodes.

In case of version A in C++ the new value for the r_2-node can be computed by using the following computation:

```
for (int i = tw; i <= Nx1; i += tw) {
    for (int j = tw; j <= Nx2; j += tw) {
        x[i,j] -= cww[i+st,j+st] * x[i+st,j+st] +  // north-east
                  cnn[i+st,j-st] * x[i+st,j-st] +  // south-east
                  cee[i-st,j-st] * x[i-st,j-st] +  // south-west
                  css[i-st,j+st] * x[i-st,j+st];   // north-west
    }
}
```

In case of version B, thus the version that the current C++ RRB-solver uses, we have

```
for (int i = st; i <= Nx1; i += tw) {
    for (int j = st; j <= Nx2; j += tw) {
            float val = x[i][j]; // r1-value
            x[i+st][j+st] -= cee[i][j] * val;  // north-east
            x[i+st][j-st] -= css[i][j] * val;  // south-east
            x[i-st][j-st] -= cww[i][j] * val;  // south-west
            x[i-st][j+st] -= cnn[i][j] * val;  // north-west
    }
}
```

In CUDA, thanks to the $r1/r2/b1/b2$ storage format, the stride is eliminated; at each level surrounding nodes lie — in some sense — directly around the node that is updated. Moreover,

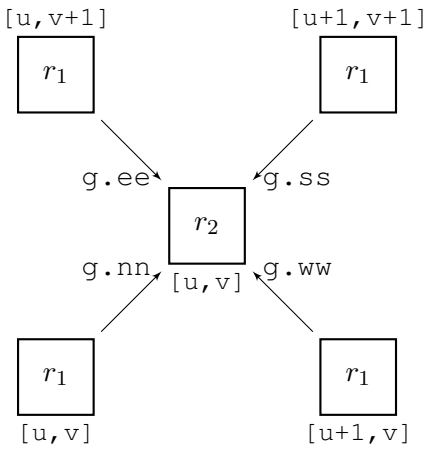


Figure 19.22: The rotated 5-point stencil in CUDA for Phase 1.

the nodes lie in the global memory in a more favourable manner. In Figure 19.22 we have shown how it is done in the CUDA RRB-solver.

Rather than `i,j` we use `u,v` as this is exactly the notation in our CUDA implementation. Now note that, given the r_2-node we are interested in is located at `[u,v]`, the south-west neighbour is also stored at `[u,v]`, and thus lies in a perfect way in the device's global memory. The north-west neighbour is stored at `[u,v+1]`, and because of the fact that the pitch of each $r1/r2/b1/b2$-grid is a multiple of 16 or 32, also this neighbour lies perfectly in the device's global memory. The north-east and south-east neighbours lie slightly worse in the device's global memory, namely at `[u+1,v]` and `[u+1,v+1]`, respectively. The x-coordinate causes the memory reads to be uncoalesced. However, as we have seen in Section 16.3 a copy with a shift is not so bad. Actually, for the new architectures (compute capability 2.x) copying with a shift does not lead to less throughput, see Figure 16.4 (red/triangles) and for older devices (compute capability 1.x) textures can be used to boost throughput, see Figure 16.4 (blue/squares). So, summarizing, no throughput is wasted in Phase 1.

The x-values may be fetched through textures as follows. Say we want to read for all values r_2 their corresponding north-east value (which are r_1-nodes) from the global memory. By looking at, for example, Figure 19.16, we see that the x- and y-coordinate should be computed as:

```
int u_r1 = BORDER_WIDTH + blockIdx.x * blockDim.x + tx;
int v_r1 = BORDER_WIDTH + blockIdx.y * blockDim.y + ty;
```

Then, the north-east x-value is read from the global memory via:

```
float value = tex2D(texRef, u_r1+1, v_r1+1);
```

For the other three neighbours, similar commands can be used. If we were to read the x-values directly from the global memory, thus without any textures, we would go from 2D coordinates `[u,v]` to a single (1D) coordinate. (Recall that, if we do nothing special, everything in the device's global memory is actually stored as a 1D-array.) We just compute

```
int loc_r1 = v_r1 * ld + u_r1;
```

where `ld` is the leading dimension of the array, that is, `g.nx`. So, in this case the north-east x-value is read from the global memory via:

```
float value = x[loc_r1 + ld + 1];
```

In as similar way the stencil-values can be read from the global memory (which are stored in `g.nn`, `g.ee`, `g.ss` and `g.ww`). So, in `CUDA`, the `C++` code snippet above would become:

```
x[loc_r2] -= g.ss[loc_r1+ld+1] * x[loc_r1+ld+1] +  // north-east
             g.ww[loc_r1+1   ] * x[loc_r1+1   ] +  // south-east
             g.ee[loc_r1+ld  ] * x[loc_r1+ld  ] +  // north-west
             g.nn[loc_r1     ] * x[loc_r1     ];   // south-west
```

or, using textures,

```
x[loc_r2] -= g.ss[loc_r1+ld+1] * tex2D(texRef, u_r1+1, v_r1+1) +  // north-east
             g.ww[loc_r1+1   ] * tex2D(texRef, u_r1+1, v_r1  ) +  // south-east
             g.ee[loc_r1+ld  ] * tex2D(texRef, u_r1  , v_r1+1) +  // north-west
             g.nn[loc_r1     ] * tex2D(texRef, u_r1  , v_r1  );   // south-west
```

Next, we have seen that the r_2-nodes define the next coarser grid, see Section 19.7.1 and, for example, Figure 19.18. So, from the updated values for r_2-nodes computed in this phase the next (coarser) $r_1/r_2/b_1/b_2$-grid should be formed. We let each thread compute a unique location in the new grid. An idea is to let the even/even threads deal with the new r_1-nodes, the odd/odd threads deal with the new r_2-nodes, the odd/even threads deal with the new b_1-nodes, and the even/odd threads deal with the new b_2-nodes, see Figure 19.23. Corresponding code is:

```
int loc_new = ldd * (BORDER_WIDTH + blockIdx.y * (BlockDim.y >> 1))
                  + (threadIdx.y & 1) * (BORDER_WIDTH + gg.cy) * ldd
                  +  BORDER_WIDTH + bx * (BlockDim.x >> 1)
                  + (threadIdx.x & 1) * (BORDER_WIDTH + gg.cx)
                  + (threadIdx.y >> 1) * ldd + (threadIdx.x >> 1);
```

where `ldd` is the leading dimension of the next grid, that is, `gg.nx`, where `>>` stands for a bit shift to the right, e.g., `8 >> 1 = 4`, and where `&` replaces the `%`-sign (modulo), which is generally faster. For example, `16 & 1` is just the same as `16 % 2` (which is 0).

The cleverness in the `CUDA` implementation of the solver part is thus that multiple grids can be used, each offering optimal throughput (instead of just one grid like in the `C++` implementation), and that using these grids come without a (significant) time penalty (overhead). Of course some extra time is spend to compute and fill the next grid, but this is completely justified as in the next level again full throughput is achieved (instead of a huge performance drop due to reading and writing data with a stride). Phase 1 corresponds with `CUDA` kernel `solv::kernel_solv1`. There are three flavours that are almost the same. We have shown the version that is used most often in Listing 19.7.

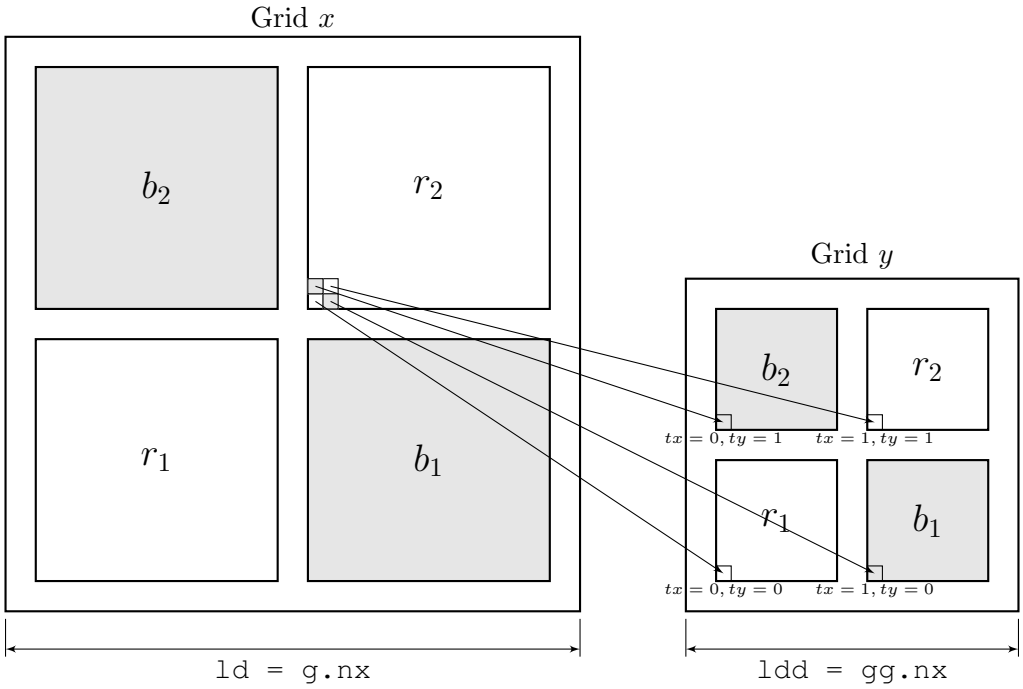


Figure 19.23: Dividing the r_2-nodes in grid x over the four groups of nodes in grid y.

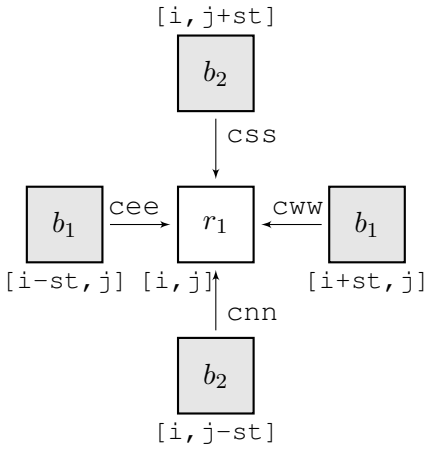
Listing 19.7: CUDA kernel solv::kernel_solv1v2().

```
1  template <class T>
2  __global__ void kernel_solv1v2(T *y, const T *x, const Grid gg,
3      const Grid g)
4  {
5      int ld = g.nx;
6      int ldd = gg.nx;
7
8      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
9      int v_r1 = BORDER_WIDTH + by * By + ty;
10     int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
11     int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
12
13     int loc_r1 = ld * v_r1 + u_r1;
14     int loc_r2 = ld * v_r2 + u_r2;
15
16     const T x_r1up1vp1 = __FETCH_XX(u_r1+1, v_r1+1, ld);
17     const T x_r1up1    = __FETCH_XX(u_r1+1, v_r1  , ld);
18     const T x_r1vp1    = __FETCH_XX(u_r1  , v_r1+1, ld);
19     const T x_r1       = __FETCH_XX(u_r1  , v_r1  , ld);
20
21     int loc_new = ldd * (BORDER_WIDTH + by * (By >> 1))
22                     + (ty & 1) * (BORDER_WIDTH + gg.cy) * ldd
23                     +  BORDER_WIDTH + bx * (Bx >> 1)
24                     + (tx & 1) * (BORDER_WIDTH + gg.cx)
25                     + (ty >> 1) * ldd + (tx >> 1);
26
27     T sum = x[loc_r2] - g.ss[loc_r1 + ld + 1] * x_r1up1vp1  // north-east
```

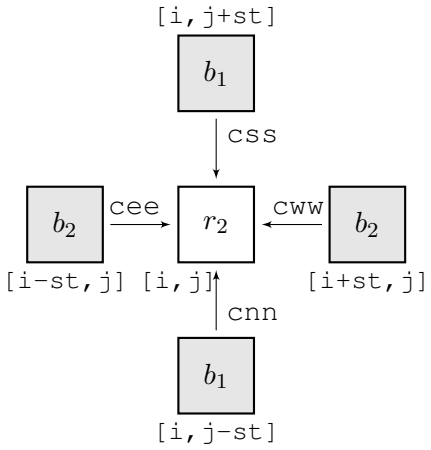
```
28                          - g.ww[loc_r1 + 1     ] * x_r1up1     // south-east
29                          - g.nn[loc_r1         ] * x_r1        // south-west
30                          - g.ee[loc_r1 + ld    ] * x_r1vp1;    // north-west
31
32      y[loc_new] = sum;
33  }
```

Phase 2

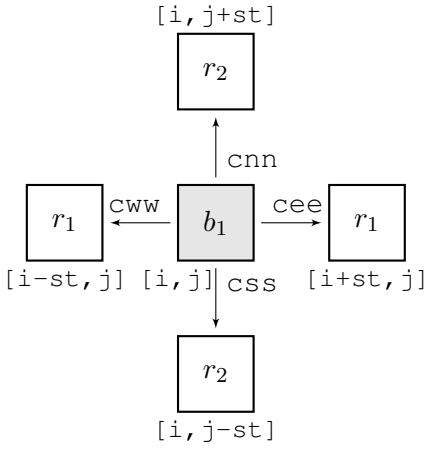
In Phase 2 the r_1- and r_2-nodes are updated by the b_1- and b_2-nodes as can be seen from Figure 19.19. The red nodes depend on the black nodes via a 5-point stencil. For example, node 57 (which is a r_1-node) depends on nodes 49 and 51, and node 63 (which is a r_2-node) depends on nodes 49, 51, 52 and 53. In Figure 19.24 this 5-point stencil is shown in more detail.



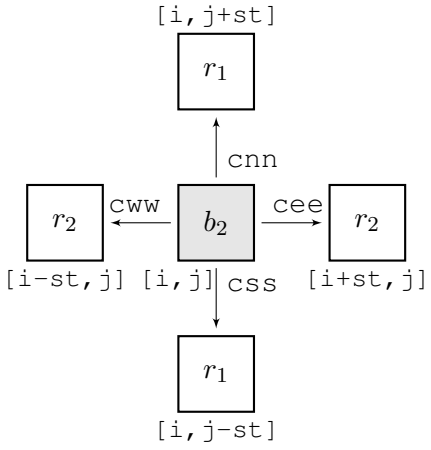
(a) C++ version A, r_1-nodes



(b) C++ version A, r_2-nodes



(c) C++ version B, r_1-nodes



(d) C++ version B, r_2-nodes

Figure 19.24: The 5-point stencil that is used in Phase 2 in C++.

In case of version A in C++ the new value for the r_1- and r_2-nodes may be computed by using the following computations:

```
// r1 nodes:
for (int i = st; i <= Nx1; i += tw) {
    for (int j = st; j <= Nx1; j += tw) {
        x[i,j] -= css[i,j+st] * x[i,j+st] +  // north
                  cww[i+st,j] * x[i+st,j] +  // east
                  cnn[i,j-st] * x[i,j-st] +  // south
                  cee[i-st,j] * x[i-st,j];   // west
    }
}

// r2 nodes:
for (int i = tw; i <= Nx1; i += tw) {
    for (int j = tw; j <= Nx1; j += tw) {
        x[i,j] -= css[i,j+st] * x[i,j+st] +  // north
                  cww[i+st,j] * x[i+st,j] +  // east
                  cnn[i,j-st] * x[i,j-st] +  // south
                  cee[i-st,j] * x[i-st,j];   // west
    }
}
```

Herein is `st = 1` and `tw = 2` in the beginning. Then, each time we go a level up, `st` and `tw` are updated via `st = tw` and `tw *= 2`. So, for the first level we have `st = 1, tw = 2`, for the second level `st = 2, tw = 4`, for the third level `st = 4, tw = 8`, etcetera. Verify for yourself that by doing so indeed, for all levels, the r_1- and r_2-nodes are addressed by the loop counters `i,j`.

In case of version B, see Figure 19.24(c) and Figure 19.24(d), and this is the way how it is actually implemented in the current C++ RRB-solver, we have:

```
// b1 nodes:
for (int i = tw; i <= Nx1; i += tw) {
    for (int j = st; j <= Nx2; j += tw) {
        float val = x[i][j]; // b1-value
        x[i][j+st] -= cnn[i][j] * val; // north
        x[i+st][j] -= cee[i][j] * val; // east
        x[i][j-st] -= css[i][j] * val; // south
        x[i-st][j] -= cww[i][j] * val; // west
    }
}

// b2 nodes:
for (int i = st; i <= Nx1; i += tw) {
    for (int j = tw; j <= Nx2; j += tw) {
        float val = x[i][j]; // b2-value
        x[i][j+st] -= cnn[i][j] * val; // north
        x[i+st][j] -= cee[i][j] * val; // east
        x[i][j-st] -= css[i][j] * val; // south
        x[i-st][j] -= cww[i][j] * val; // west
    }
}
```

To see how we implement Phase 2 in CUDA we combine the two 5-point stencils for the r_1- and r_2-nodes into a somewhat bigger stencil like the one shown in Figure 19.25. Note that

the north neighbour of r_1 is the same black b_2-node as the west neighbour of r_2, likewise, the east neighbour of r_1 is the same black b_1-node as the south neighbour of r_1. By handling both the r_1-nodes and r_2-nodes in the one CUDA kernel less different data has to retrieved from the global memory which improves throughput.

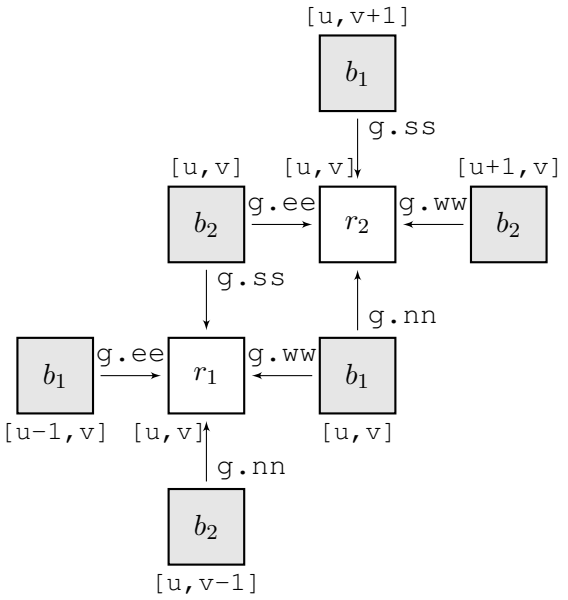


Figure 19.25: Two 5-point stencils are combined for Phase 2 in CUDA.

Much of the data lies in a perfect coalesced manner in the global memory. The x-values for r_1, r_2, and the x-values for the north b_2-neighbour of r_1 (which is thus the west b_2-neighbour of r_2) and east b_1-neighbour of r_1 (which is thus the south b_1-neighbour of r_2), all have coordinates `[u,v]`. Further, the x-values for the south b_2-neighbour of r_1 and the north neighbour of r_2 have coordinates `[u,v+1]`, and with our stride that is a multiple of 16 or 32, also these values lie in a perfect coalesced manner in the global memory. Only the x-values for the west b_1-neighbour and east b_2-neighbour lie slightly worse in the device's global memory. The same holds for the stencils `g.nn`, `g.ee`, `g.ss` and `g.ww`; most of the data lie perfectly fine in the global memory.

For the data that lies slightly worse in the global memory, that is, with a shift of 1, textures can be used to increase throughput on older devices. On devices with the newest architecture this is unnecessary, see Section 16.3 and in particular Figure 16.4.

In CUDA the two double for-loops above are replaced by the following code:

```
float x_b1 = x[loc_b1]; // common b-values for r1 and r2
float x_b2 = x[loc_b2];

x[loc_r1] -= g.ss[loc_b2   ] * x_b2 +          // north
             g.ww[loc_b1   ] * x_b1 +          // east
             g.nn[loc_b2-ld] * x[loc_b2-ld] +  // south
             g.ee[loc_b1-1 ] * x[loc_b1-1 ];   // west

x[loc_r2] -= g.ss[loc_b1+ld] * x[loc_b1+ld] +  // north
             g.ww[loc_b2+1 ] * x[loc_b2+1 ] +  // east
```

```
            g.nn[loc_b1   ] * x_b1 +          // south
            g.ee[loc_b2   ] * x_b2;           // west
```

where, for example, `loc_b1` is, for each thread, a unique b_1-location in the grid `g` and vector `x`:

```
int u_b1 = 2 * BORDER_WIDTH + g.cx + blockIdx.x * BlockDim.x + threadIdx.x;
int v_b1 = BORDER_WIDTH + blockIdx.y * BlockDim.y + threadIdx.y;

int loc_b1 = v_b1 * ld + u_b1; // ld = leading dimension of the grid/vector
```

Verify for yourself that the above lines are correct. When textures are used the r_1-part would become something like:

```
float x_b1 = tex2D(texRef, u_b1, v_b1); // common b-values for r1 and r2
float x_b2 = tex2D(texRef, u_b2, v_b2);

x[loc_r1] -= g.ss[loc_b2   ] * x_b2 +                        // north
             g.ww[loc_b1   ] * x_b1 +                        // east
             g.nn[loc_b2-ld] * tex2D(texRef, u_b2, v_b2-1) + // south
             g.ee[loc_b1-1 ] * tex2D(texRef, u_b1-1, v_b1);  // west
```

Phase 2 corresponds with CUDA kernel `solv::kernel_solv2`, see Listing 19.8.

Listing 19.8: CUDA kernel solv::kernel_solv2().

```
 1 template <class T>
 2 __global__ void kernel_solv2(T *x, const Grid g)
 3 {
 4     int ld = g.nx;
 5
 6     int u_r1 = BORDER_WIDTH + bx * Bx + tx;
 7     int v_r1 = BORDER_WIDTH + by * By + ty;
 8     int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
 9     int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
11     int v_b1 = BORDER_WIDTH + by * By + ty;
12     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
13     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
14
15     int loc_r1 = ld * v_r1 + u_r1;
16     int loc_r2 = ld * v_r2 + u_r2;
17     int loc_b1 = ld * v_b1 + u_b1;
18     int loc_b2 = ld * v_b2 + u_b2;
19
20     const T x_b1    = __FETCH_XX(u_b1,   v_b1, ld);
21     const T x_b1um1 = __FETCH_XX(u_b1-1, v_b1, ld);
22     const T x_b2    = __FETCH_XX(u_b2,   v_b2, ld);
23     const T x_b2up1 = __FETCH_XX(u_b2+1, v_b2, ld);
24     const T x_b2vm1 = __FETCH_XX(u_b2, v_b2-1, ld);
25     const T x_b1vp1 = __FETCH_XX(u_b1, v_b1+1, ld);
26
27     x[loc_r1] -= g.ss[loc_b2]      * x_b2 +    // north
28                  g.ww[loc_b1]      * x_b1 +    // east
29                  g.nn[loc_b2 - ld] * x_b2vm1 + // south
30                  g.ee[loc_b1 - 1 ] * x_b1um1;  // west
31
```

```
32      x[loc_r2] -= g.ss[loc_b1 + ld] * x_b1vp1 + // north
33                   g.ww[loc_b2 + 1 ] * x_b2up1 + // east
34                   g.nn[loc_b1]      * x_b1 +    // south
35                   g.ee[loc_b2]      * x_b2;     // west
36  }
```

19.7.3 Step 2: Solving $y = D^{-1}x$

This part of the preconditioner step is very easy. As D is a diagonal matrix, y is found by dividing all elements of x with a corresponding value in D. Actually, during the construction of the preconditioner at some point we have inverted all main diagonal elements, i.e., $d_{ii} \rightarrow 1/d_{ii}$, so that y is not found by dividing all elements of x by an element in D, but by multiplying all elements of x by an element in D. Generally, multiplying numbers can be done (much) faster than dividing numbers. The preconditioner step is performed just once whereas solving the system $Mz = r$ is done every iteration in the CG part of the RRB-solver. Hence, inverting D in the preconditioner phase saves valuable time later on in the CG-part.

The red nodes in the first level are used for solving $Mz = r$; correspondingly, $y = D^{-1}x$ requires to multiply each red node in x with the corresponding diagonal element in D. To save memory x is overwritten by y, so we actually solve $x = D^{-1}x$. In C++ we can do this as follows:

```
for (int i = 1; i <= Nx1; i += 2) {
    for (int j = 1; j <= Nx2; j += 2) {
        y[i  ][j  ] *= ccc[i  ][j  ]; // r1 nodes
        y[i-1][j-1] *= ccc[i-1][j-1]; // r2 nodes
    }
}
```

where `ccc` thus represents the diagonal matrix D.

In our CUDA implementation we do this differently. We have to do this differently, because of the fact that we use multiple levels. We have seen that at the end of Phase 1 in Step 1, i.e., solving $Lx = r$, the results are written to the next (coarser) level for x, so the previous level is not aware of any updated data. Therefore, we have to do the step $x = D^{-1}x$ locally at each level, and thus we have to integrate Step 2 into Step 1.

At each level, when Phase 1 is finished we update x in the r_1-nodes, and when Phase 2 is finished we update x in the b_1- and b_2-nodes. Vector x is not updated in the r_2-nodes as the r_2-nodes carry over to a next grid; only in the end when just 1 node remains, for this node its x-value has to be multiplied with the corresponding diagonal element. So, one of the CUDA kernels contains the lines

```
x[loc_b1] *= g.cc[loc_b1]);
x[loc_b2] *= g.cc[loc_b2]);
```

and another contains the line

```
x[loc_r1] *= g.cc[loc_r1]);
```

The two CUDA kernels are `solv::kernel_solv5` and `solv::kernel_solv6`, see Listing 19.9 and Listing 19.10.

Listing 19.9: CUDA kernel solv::kernel_solv5().

```
1 template <class T>
2 __global__ void kernel_solv5(T *x, const Grid g)
3 {
4     int ld = g.nx;
5
6     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
7     int v_b1 = BORDER_WIDTH + by * By + ty;
8     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
9     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11    int loc_b1 = ld * v_b1 + u_b1;
12    int loc_b2 = ld * v_b2 + u_b2;
13
14    x[loc_b1] *= g.cc[loc_b1];
15    x[loc_b2] *= g.cc[loc_b2];
16 }
```

Listing 19.10: CUDA kernel solv::kernel_solv6().

```
1 template <class T>
2 __global__ void kernel_solv6(T *x, const Grid g)
3 {
4     int ld = g.nx;
5
6     int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7     int v_r1 = BORDER_WIDTH + by * By + ty;
8
9     int loc_r1 = ld * v_r1 + u_r1;
10
11    x[loc_r1] *= g.cc[loc_r1];
12 }
```

19.7.4 Step 3: Solving $L^T z = y$

In Step 1 we go from fine grids (many nodes) to coarse grids (few nodes) and forward substitution is used. In Step 3 we go the other way around: we go from coarse grids to fine grids and backward substitution is used. In case of the 8×8 grid the coarsest level, that is, level 4, contains 1 node. This value is used to update the 4 nodes in level 3. Then, the updated 4 nodes in level 3 are used to update the 16 nodes in level 2, and, finally, the updated 16 nodes in level 2 are used to update the red nodes in the first level, level 1. So, where Step 1 can be seen as a "converging"-phase, Step 3 can be seen as a "diverging"-phase. Just like solving $Lx = r$ solving $L^T z = y$ is done iteratively using two steps:

- Phase 4: Updating r_1-nodes using r_2-nodes in the same level;
- Phase 3: Updating b_1- and b_2-nodes using r_1- and r_2-nodes in the same level.

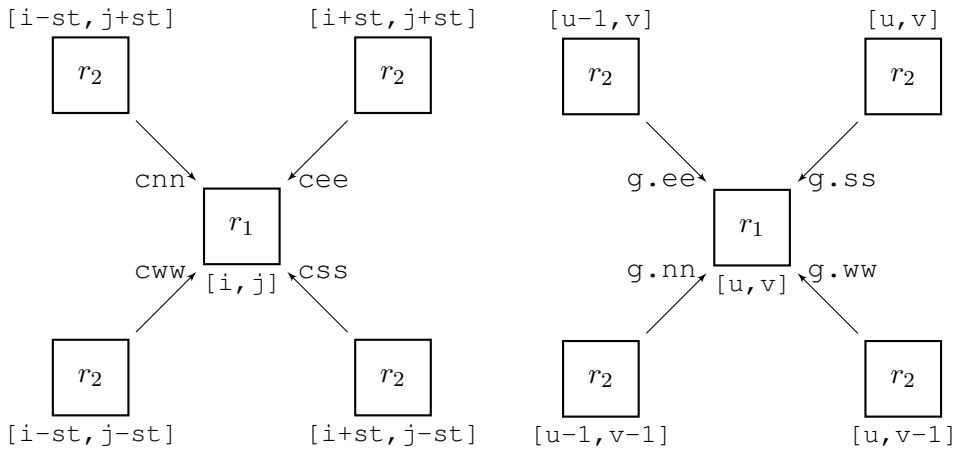
Intentionally we have put Phase 4 first. The backward substitution is like:

Phase 4 → Phase 3 → Phase 4 → Phase 3 → ⋯ → Phase 4 → Phase 3 → Phase 4,

so starting with Phase 4 and finishing with Phase 4. Below both phases are discussed in more detail.

Phase 4

Phase 4 is the backward counterpart of Phase 1. In this phase the r_2-nodes are used to update the r_1-nodes in the same level. For example, in level 2 of the 8×8 example, node 60 depends on nodes 61, 62, 63 and 64, see Figure 19.19. Of course the r_1-nodes depend on the r_2-nodes according to same rotated 5-point stencil as in Phase 1. In Figure 19.26 we have depicted this 5-point stencil from the viewpoint of r_1-nodes.



(a) C++ version (b) CUDA version

Figure 19.26: The 5-point stencil that is used in Phase 4 in C++ and CUDA.

In Figure 19.26(a) we have depicted the 5-point stencil that belongs to the current C++ implementation. It won't take you long to figure out that suitable C++ code for this stencil is:

```
for (int i = m_st; i <= Nx1; i += tw) {
    for (int j = m_st; j <= Nx2; j += tw) {
        x[i][j] = x[i][j] - cee[i][j] * x[i+st][j+st]  // north-east
                          - css[i][j] * x[i+st][j-st]  // south-east
                          - cww[i][j] * x[i-st][j-st]  // south-west
                          - cnn[i][j] * x[i-st][j+st]; // north-west
    }
}
```

In Figure 19.26(b) we have depicted the 5-point stencil that belongs to the CUDA implementation of Phase 4. Note the difference stencils between the C++ and CUDA version. The differences have to do with how data was copied to the GPU, recall the host and device compass. Suitable CUDA code is:

```
x[loc_r1] -= g.nn[loc_r1] * x[loc_r2     ] +  // north-east
             g.ee[loc_r1] * x[loc_r2-ld  ] +  // south-east
             g.ss[loc_r1] * x[loc_r2-ld-1] +  // south-west
             g.ww[loc_r1] * x[loc_r2-1   ];   // north-west
```

Phase 4 corresponds with CUDA kernel `solv::kernel_solv4`, see Listing 19.11.

Listing 19.11: CUDA kernel solv::kernel_solv4().

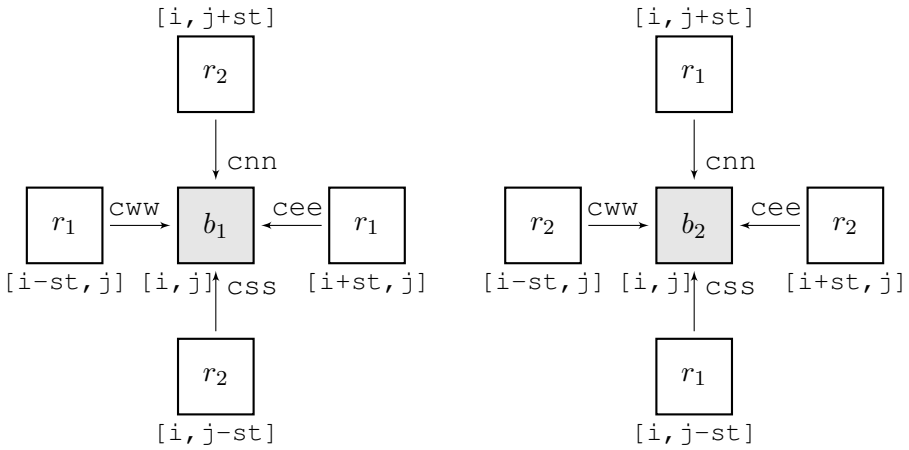
```
 1 template <class T>
 2 __global__ void kernel_solv4(T *x, const Grid g)
 3 {
 4     int ld = g.nx;
 5
 6     int u_r1 = BORDER_WIDTH + bx * Bx + tx;
 7     int v_r1 = BORDER_WIDTH + by * By + ty;
 8     int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
 9     int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11     int loc_r1 = ld * v_r1 + u_r1;
12
13     const T x_r2       = __FETCH_XX(u_r2,   v_r2,   ld);
14     const T x_r2vm1    = __FETCH_XX(u_r2,   v_r2-1, ld);
15     const T x_r2um1    = __FETCH_XX(u_r2-1, v_r2,   ld);
16     const T x_r2um1vm1 = __FETCH_XX(u_r2-1, v_r2-1, ld);
17
18     x[loc_r1] -= g.nn[loc_r1] * x_r2 +       // north-east
19                  g.ee[loc_r1] * x_r2vm1 +    // south-east
20                  g.ss[loc_r1] * x_r2um1vm1 + // south-west
21                  g.ww[loc_r1] * x_r2um1;     // north-west
22 }
```

Phase 3

Phase 3 is the backward counterpart of Phase 2. In this phase the b_1- and b_2-nodes are updated by the r_1- and r_2-nodes in the same level. For example, in level 2 of the 8×8 example, see Figure 19.19, the b_1-node 52 depends on the red nodes 58, 60 (r_1-nodes) and 61, 63 (r_2-nodes). Of course the black nodes depend on the red nodes according to same 5-point stencil as in Phase 2. In Figure 19.27 we see how it is done in the C++ version.

Corresponding C++ code is:

```
for (int i = tw; i <= Nx1; i += tw) {
    for (int j = st; j <= Nx2; j += tw) {
        x[i][j] -= cnn[i][j] * x[i   ][j+st] +  // north
                   cee[i][j] * x[i+st][j   ] +  // east
                   css[i][j] * x[i   ][j-st] +  // south
                   cww[i][j] * x[i-st][j   ];   // west
    }
}

for (int i = st; i <= Nx1; i += tw) {
    for (int j = tw; j <= Nx2; j += tw) {
        x[i][j] -= cnn[i][j] * x[i   ][j+st] +  // north
                   cee[i][j] * x[i+st][j   ] +  // east
                   css[i][j] * x[i   ][j-st] +  // south
```



(a) C++ version, b_1-nodes (b) C++ version, b_2-nodes

Figure 19.27: The 5-point stencil that is used in Phase 3 in C++.

```
                cww[i][j] * x[i-st][j   ];   // west
    }
}
```

In Figure 19.28 we have shown the extended stencil that is used in the CUDA implementation. Again, some red nodes are common to b_1 and b_2. Without textures CUDA code would be something like:

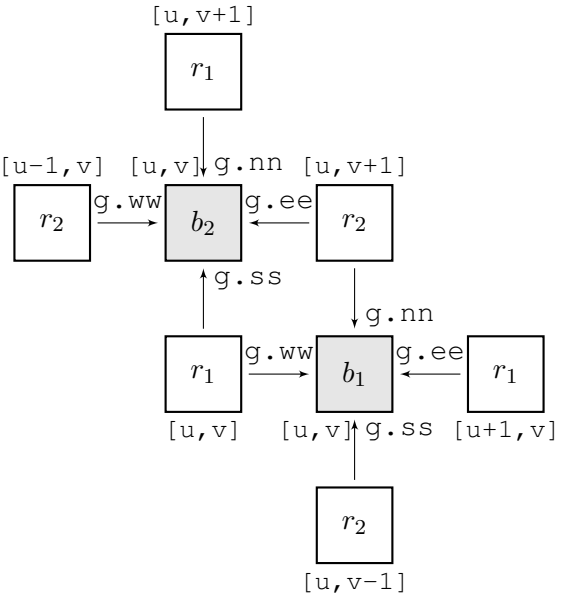


Figure 19.28: The 5-point stencil that is used in Phase 3 in CUDA.

```
float x_r1 = x[loc_r1]; // common r-values for b1 and b2:
```

```
float x_r2 = x[loc_r2];

// b1 nodes:
x[loc_b1] = x[loc_b1] - g.nn[loc_b1] * x_r2          // north
                      - g.ee[loc_b1] * x[loc_r1+1 ]  // east
                      - g.ss[loc_b1] * x[loc_r2-ld]  // south
                      - g.ww[loc_b1] * x_r1;         // west

// b2 nodes:
x[loc_b2] = x[loc_b2] - g.nn[loc_b2] * x[loc_r1+ld]  // north
                      - g.ee[loc_b2] * x_r2          // east
                      - g.ss[loc_b2] * x_r1          // south
                      - g.ww[loc_b2] * x[loc_r2-1 ]; // west
```

Next, all results computed in the current level are substituted into the r_2-part of the grid that is 1 level higher (which has 4 times more nodes). So, from the 4 seperate data blocks in the current grid we make 1 big data block that corresponds to the r_2-part of the next higher grid. Each thread must write four values to the next grid (one r_1-, one r_2-, one b_1- and one b_2-value). After the new value of b_1 is computed by using the computation above, each thread writes its computed value to a b_1-location in the r_2-part of the next grid, say `y` which has leading dimension `ldd`. Likewise, after the new value of b_2 is computed using the computation above, the thread writes the results to a b_2-location in the r_2-part of the next grid. In our computations above we have also used the x-values for r_1- and r_2-nodes so these can be written also to the next higher grid. Study Listing 19.12 yourself and see how we go from $r1/r2/b1/b2$ back to the r_2-part in the next higher grid.

Listing 19.12: CUDA kernel solv::kernel_solv3().

```
1  template <class T>
2  __global__ void kernel_solv3(T *y, const T *x, const Grid gg,
3      const Grid g)
4  {
5      int ld = g.nx;
6      int ldd = gg.nx;
7
8      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
9      int v_r1 = BORDER_WIDTH + by * By + ty;
10     int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
11     int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
12     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
13     int v_b1 = BORDER_WIDTH + by * By + ty;
14     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
15     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
16     int loc_b1 = ld * v_b1 + u_b1;
17     int loc_b2 = ld * v_b2 + u_b2;
18
19     int loc = ldd * (BORDER_WIDTH2 + gg.cy + 2 * by * By)
20                   + BORDER_WIDTH2 + gg.cx + 2 * bx * Bx;
21     int loc_new;
22
23     const T x_r1    = __FETCH_XX(u_r1,   v_r1, ld);
24     const T x_r2    = __FETCH_XX(u_r2,   v_r2, ld);
25     const T x_r1up1 = __FETCH_XX(u_r1+1, v_r1, ld);
26     const T x_r2vm1 = __FETCH_XX(u_r2, v_r2-1, ld);
27     const T x_r1vp1 = __FETCH_XX(u_r1, v_r1+1, ld);
28     const T x_r2um1 = __FETCH_XX(u_r2-1, v_r2, ld);
29
30     T sum;
31
32     // b1 nodes:
33     sum = x[loc_b1] - g.nn[loc_b1] * x_r2     // north
34                     - g.ee[loc_b1] * x_r1up1  // east
35                     - g.ss[loc_b1] * x_r2vm1  // south
36                     - g.ww[loc_b1] * x_r1;    // west
37     loc_new = loc + ldd * (ty << 1) + (tx << 1) + 1;
38     y[loc_new] = sum;
39
40     // b2 nodes:
41     sum = x[loc_b2] - g.nn[loc_b2] * x_r1vp1  // north
42                     - g.ee[loc_b2] * x_r2     // east
43                     - g.ss[loc_b2] * x_r1     // south
44                     - g.ww[loc_b2] * x_r2um1; // west
45     loc_new = loc + ldd * ((ty << 1) + 1) + (tx << 1);
46     y[loc_new] = sum;
47
48     // r1 nodes:
49     loc_new = loc + ldd * (ty << 1) + (tx << 1);
50     y[loc_new] = x_r1;
51
52     // r2 nodes:
53     loc_new = loc + ldd * ((ty << 1) + 1) + (tx << 1) + 1;
54     y[loc_new] = x_r2;
55 }
```

19.7.5 The final level

The previous discussion was not complete. In our CUDA RRB-solver at some point we do not make any more new grids. We stop the process when the number of nodes becomes smaller than 16×16 or 32×32, the dimensions of the so-called *compute-block*. All previous kernels are combined into 1 big kernel for the final level, see Listing 19.13. 1 SM is used to complete all nodes in the final level, the kernel exploits the global memory cache. Notice the synchronization points which are extremely important; without them the complete solver does not function properly.

Listing 19.13: CUDA kernel solv::kernel_solvfinal().

```
1  template <class T>
2  __global__ void kernel_solvfinal(T *x, const Grid g)
3  {
4      int ld = g.nx;
5
6      int loc;
7      int st = 1;
8      int tw = 2;
9      int u, v;
10
11     // Part I) solving Lx = x
12     for (int nt = DIM_COMPUTE_BLOCK / 2; nt >= 1; nt >>= 1)
13     {
14         // Phase 2: for all red nodes
15         // This part corresponds with solv::kernel_solv2()
16         if (tx < nt) {
17             u = (st - 1) + tw * tx;
18             v = (st - 1) + tw * ty;
19         } else {
20             u = (tw - 1) + tw * (tx - nt);
21             v = (tw - 1) + tw * ty;
22         }
23         loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
24                  + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
25
26         if (ty < nt && tx < nt << 1) {
27             x[loc] -= g.ss[loc + st * ld] * x[loc + st * ld] +
28                       g.ww[loc + st     ] * x[loc + st     ] +
29                       g.nn[loc - st * ld] * x[loc - st * ld] +
30                       g.ee[loc - st     ] * x[loc - st     ];
31         }
32         __syncthreads();
33
34         // phase 1: for r2 nodes: forward substitution of r1 into r2 nodes
35         // This part corresponds with solv::kernel_solv1()
36         u = (tw - 1) + tw * tx;
37         v = (tw - 1) + tw * ty;
38         loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
39                  + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
40
41         if (ty < nt && tx < nt) {
42             x[loc] -= g.ss[loc + st * ld + st] * x[loc + st * ld + st] +
43                       g.ww[loc - st * ld + st] * x[loc - st * ld + st] +
44                       g.ee[loc + st * ld - st] * x[loc + st * ld - st] +
45                       g.nn[loc - st * ld - st] * x[loc - st * ld - st];
```

```
46          }
47          __syncthreads();
48
49          st = tw;
50          tw *= 2;
51      }
52
53
54      // Part II) solving Dx = x
55      // first half
56      loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + ty)
57                + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + tx;
58      T sum = x[loc] * g.cc[loc];
59      x[loc] = sum;
60      __syncthreads();
61
62      // second half
63      loc += ld * (DIM_COMPUTE_BLOCK / 2);
64      sum = x[loc] * g.cc[loc];
65      x[loc] = sum;
66      __syncthreads();
67
68
69      // Part III) solving L^Tx = x
70      st = DIM_COMPUTE_BLOCK / 2;
71      tw = DIM_COMPUTE_BLOCK;
72
73      for (int nt = 1; nt <= DIM_COMPUTE_BLOCK / 2; nt <<= 1)
74      {
75
76          // for r1 nodes: backward substitution of r2 into r1 nodes
77          if (tx < nt) {
78              u = (st - 1) + tw * tx;
79              v = (st - 1) + tw * ty;
80          } else {
81              u = (tw - 1) + tw * (tx - nt);
82              v = (tw - 1) + tw * ty;
83          }
84          loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
85                    + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
86
87          if (ty < nt && tx < nt) {
88              x[loc] -= g.nn[loc] * x[loc + st * ld + st] +
89                        g.ee[loc] * x[loc - st * ld + st] +
90                        g.ss[loc] * x[loc - st * ld - st] +
91                        g.ww[loc] * x[loc + st * ld - st];
92          }
93          __syncthreads();
94
95          // for all black nodes
96          if (tx < nt) {
97              u = (st - 1) + tw * tx;
98              v = (tw - 1) + tw * ty;
99          } else {
100             u = (tw - 1) + tw * (tx - nt);
101             v = (st - 1) + tw * ty;
102         }
```

```
103         loc = ld * (BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + v)
104                   + BORDER_WIDTH2 + DIM_COMPUTE_BLOCK + u;
105
106         if (ty < nt && tx < nt << 1) {
107             x[loc] -= g.nn[loc], x[loc + st * ld] +
108                       g.ee[loc], x[loc + st     ] +
109                       g.ss[loc], x[loc - st * ld] +
110                       g.ww[loc], x[loc - st     ];
111         }
112         __syncthreads();
113
114         tw = st;
115         st /= 2;
116     }
117 }
```

19.8 Computing $q = S_1p$

Every CG-iteration the matrix-vector product $q = S_1p$ has to be computed. The matrix S_1 is the first Schur complement, given by

$$S_1 := D_r - S_{rb}D_b^{-1}S_{br}.$$

The multiplication $q = S_1p$ is done in two steps, namely:

1. $y = D_b^{-1}S_{br}p$,
2. $q = D_rp - S_{rb}y$.

Since only the red nodes are involved in the computations, the black nodes can be used to store intermediate results. So it is possible to compute $q = D_b^{-1}S_{br}x$ followed by $q = D_rp - S_{rb}q$ and hence an extra vector y is no longer needed which saves memory. This trick is used in both the current `C++` version as well as in the `CUDA` version of the RRB-solver.

In the `C++` version a memory efficient implementation is used: only three stencils are used, namely StC, StW and StS. In the current `CUDA` version all five stencils are used, so less memory efficient. However, also for the `CUDA` solver it is possible to write kernels which use only the stencils StC, StW and StS. The reason why all five stencils are used in the current `CUDA` implementation is that most kernels were designed on an older GPU, namely the GTX 285 which has compute capability 1.3. We have seen in Section 16.3.2 that for older architectures it matters whether we fetch data with a shift. In the very beginning of our `CUDA` programming we chose performance over memory efficiency, and, accordingly, the kernels were written such that most memory reads are coalesced. In the next sections we describe the implementations carefully.

19.8.1 Step 1 in `C++`

The first step is to compute $q = D_b^{-1}S_{br}p$. Consider Figure 19.29. The figure shows how we are going to compute the intermediate result $q = D_b^{-1}S_{br}p$. The intermediate results are stored in the black points. On the right in Figure 19.29 we have zoomed in on the highlighted 5-point stencil (with a b_1-node as center node; for the b_2-nodes the same 5-point stencil is used). The surrounding red nodes correspond with elements in the matrix S_{br}, the center black node corresponds with an element in the matrix D_b^{-1}.

Now note that it holds that `cn[i,j] = cs[i,j+1]` and `ce[i,j] = cw[i+1,j]` so that indeed only three stencils are needed. We can express all north-dependencies in terms of south-dependencies and all east-dependencies in terms of west-dependencies. In `C++` the following code can be used:

```
// b1 nodes:
for (int i = 2; i <= Nx1; i += 2) {
    for (int j = 1; j <= Nx2; j += 2) {
        q[i][j] = cc[i][j] * (cs[i  ][j+1] * p[i  ][j+1] +  // north
                              cw[i+1][j  ] * p[i+1][j  ] +  // east
                              cs[i  ][j  ] * p[i  ][j-1] +  // south
                              cw[i  ][j  ] * p[i-1][j  ]);  // west
    }
}
```

```
// b2 nodes:
for (int i = 1; i <= Nx1; i += 2) {
    for (int j = 2; j <= Nx2; j += 2) {
        q[i][j] = cc[i][j] * (cs[i  ][j+1] * p[i  ][j+1] +  // north
                              cw[i+1][j  ] * p[i+1][j  ] +  // east
                              cs[i  ][j  ] * p[i  ][j-1] +  // south
                              cw[i  ][j  ] * p[i-1][j  ]);  // west
    }
}
```

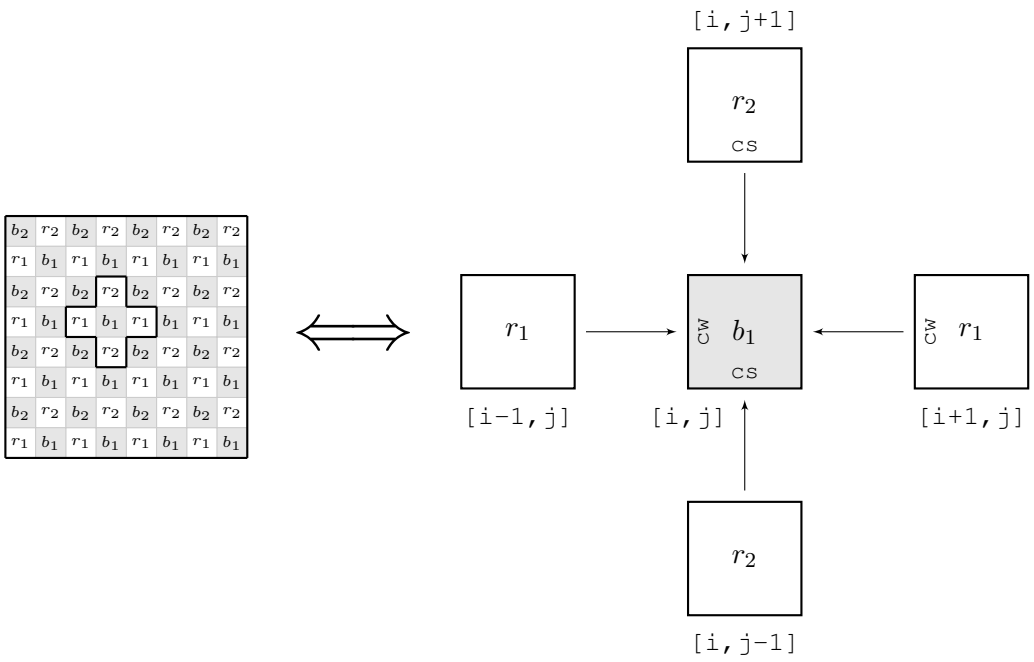


Figure 19.29: The 5-point stencil for Step 1 of the matrix-vector product $q = S_1 p$. This is the part where $q = D_b^{-1} S_{br} p$ is computed (the intermediate q result is stored in all first level black nodes). Note that only the center (cc), the west (cw) and south (cs) stencils are used.

19.8.2 Step 2 in C++

The second step is to compute $q = D_r p - S_{rb} q$. For this step the 5-point stencil of Figure 19.30 is used (the figure shows the 5-point stencil for r_1-nodes; for r_2-nodes the same 5-point stencil is used). In C++ the point-wise computation of $q = D_r p - S_{rb} q$ can be done with the following code:

```
// r1 nodes:
for (int i = 1; i <= Nx1; i += 2) {
    for (int j = 1; j <= Nx2; j += 2) {
        q[i][j] = cc[i  ][j  ] * p[i  ][j  ] -  // center
                  cs[i  ][j+1] * q[i  ][j+1] -  // north
                  cw[i+1][j  ] * q[i+1][j  ] -  // east
                  cs[i  ][j  ] * q[i  ][j-1] -  // south
                  cw[i  ][j  ] * q[i-1][j  ];   // west
    }
}

// r2 nodes:
for (int i = 2; i <= Nx1; i += 2) {
    for (int j = 2; j <= Nx2; j += 2) {
        q[i][j] = cc[i  ][j  ] * p[i  ][j  ] -  // center
                  cs[i  ][j+1] * q[i  ][j+1] -  // north
                  cw[i+1][j  ] * q[i+1][j  ] -  // east
                  cs[i  ][j  ] * q[i  ][j-1] -  // south
                  cw[i  ][j  ] * q[i-1][j  ];   // west
    }
}
```

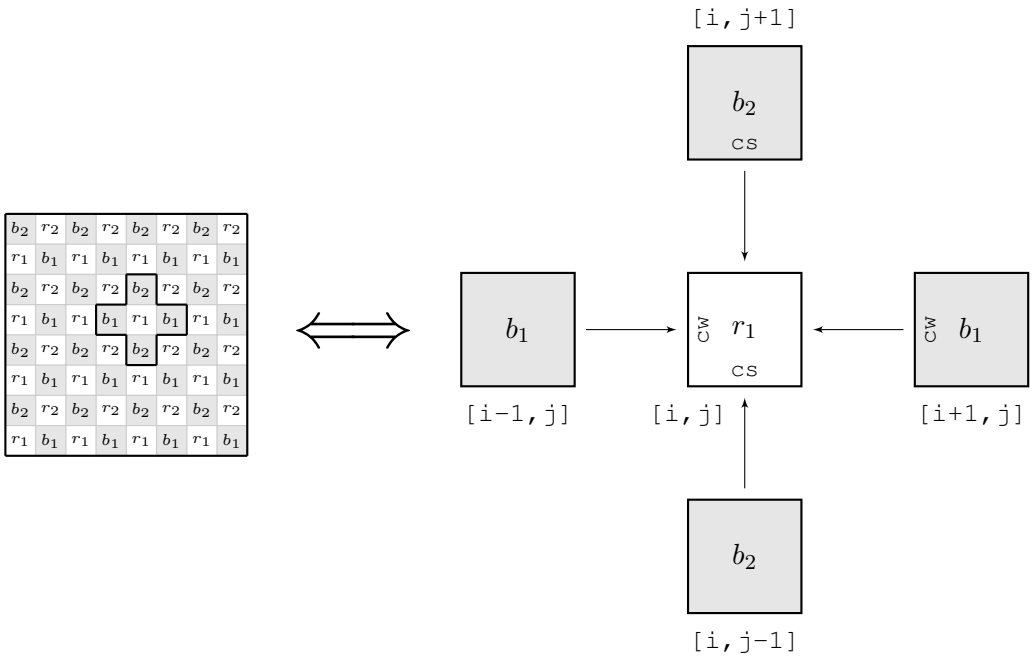


Figure 19.30: The 5-point stencil for Step 2 of the matrix-vector product $q = S_1 p$. This is the part where $q = D_r p - S_{rb} q$ is computed (all first level red nodes). Note that only the center (cc), the west (cw) and south (cs) stencils are used.

19.8.3 Towards an efficient CUDA implementation

In CUDA the matrix-vector product $q = S_1 p$ is implemented slightly differently than what the C++ code snippets suggest in the previous sections. The reason to do this differently is that we can obtain a faster implementation if we compute things differently. The idea is to shift some of the computations of Step 2 into Step 1 so that 1) overall less global memory reads are needed, 2) global memory latency can be hidden much better. Let us first discuss how a CUDA implementation would look like *without* any modifications.

Similar to two of the solver-kernels (Phase 2 and Phase 3) in CUDA the 5-point stencils for b_1- and b_2-nodes can be combined into one larger stencil. For Step 1 the extended stencil is shown in Figure 19.31.

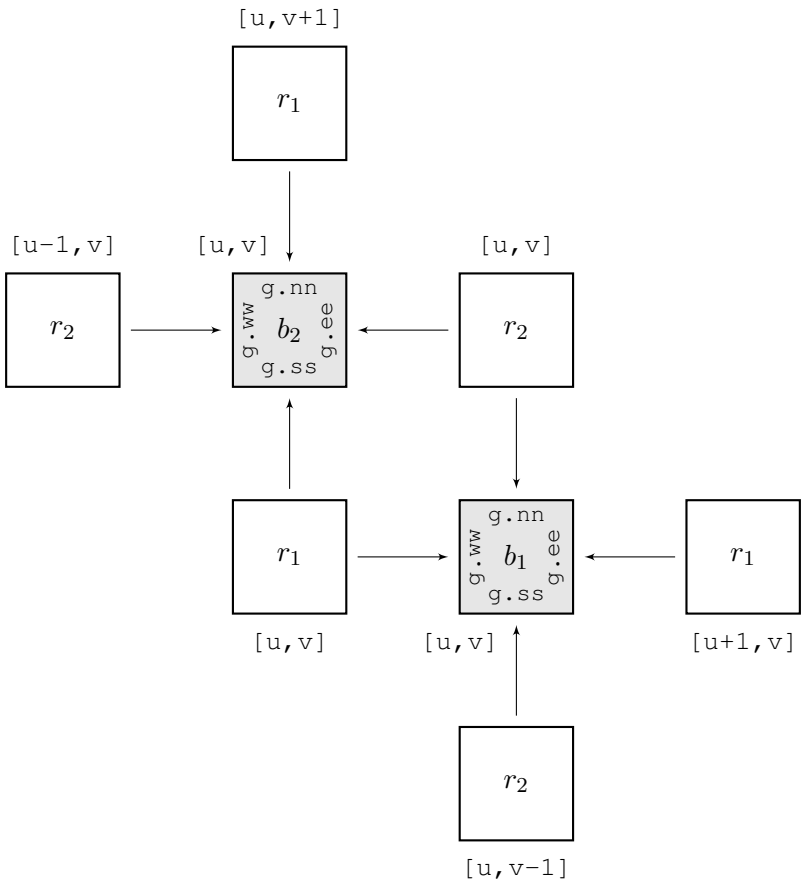


Figure 19.31: Two 5-point stencils combined. For each thread [u,v] are unique (coalesced) locations in the global memory.

From this figure and the C++ implementation code snippet for Step 1 we can easily count the number of reads and writes from and to the global memory. Each thread computes two temporary results q: one for a b_1-node and one for a b_2-node, hence the number of global memory writes is 2. Next, we need g.cc, g.nn, g.ee, g.ss and g.ww for each of the nodes, hence 10 reads from the global memory. Further, we need also 6 different values in vector p (2 of them are common to b_1 and b_2, see the figure). So the total number of global memory

reads that are really required is $10 + 6 = 16$.

For Step 2 a similar stencil can be drawn, see Figure 19.32. From this figure and the C++ implementation code snippet for Step 2 we can easily count the number of reads and writes from and to the global memory. We find: 2 writes to the global memory. Further, we need 10 stencil values, 2 different values in vector q and 6 different values in vector p, hence a total of 18 global memory reads.

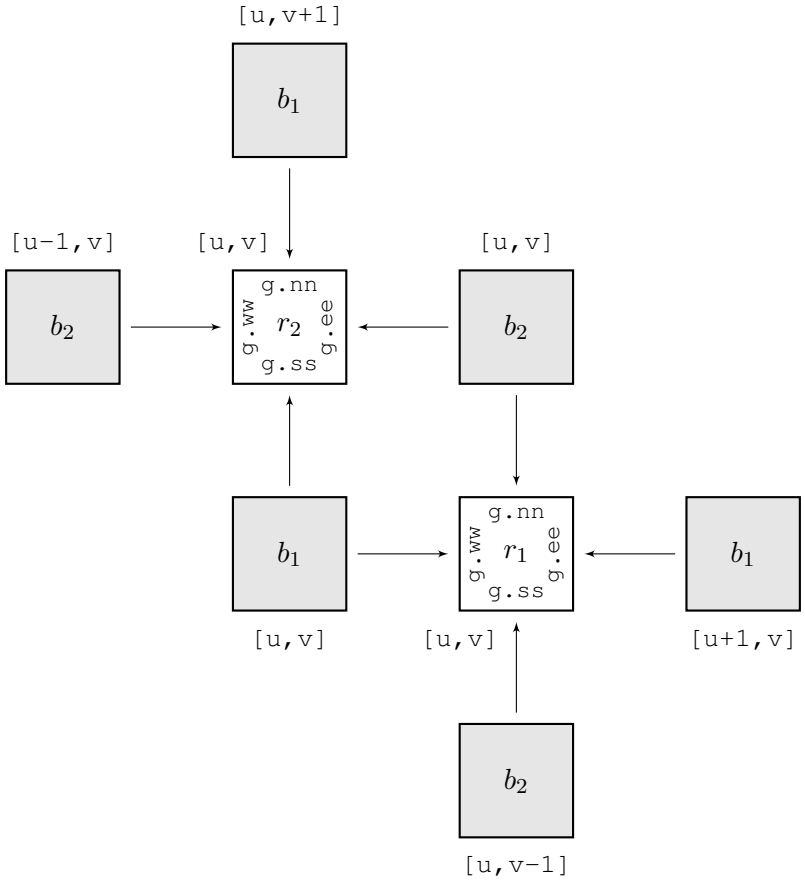


Figure 19.32: Two 5-point stencils combined. For each thread `[u,v]` are unique (coalesced) locations in the global memory.

Thus, on total, if we implement the matrix-vector product in CUDA using the unmodified versions of Step 1 and Step 2 it would cost $2 + 2 = 4$ global memory writes and $16 + 18 = 34$ global memory reads per thread.

In the next sections we discuss the modified versions of Step 1 and Step 2 and we see how with minor modifications a more efficient CUDA implementation can be achieved.

19.8.4 Step 1 in CUDA

By looking at the two extended stencils carefully, we see that each thread first computes intermediate results q in a b_1- and b_2-node and then uses these values plus two other values computed by neighbour threads to compute the final values in the r_1- and r_2-node. The key

observation is now that once a thread has computed the b_1- and b_2-values it can immediately start computing its own p values in the r_1- and r_2-nodes! We write "start computing" because, unfortunately, for a single thread it is not possible to compute the final values for the r_1-and r_2-nodes as it needs the intermediate results computed by 2 other threads. However, the thread can already make a start with its own computed b_1- and b_2-values: for the r_1-node the thread can already compute the north and west contributions that occur in Step 2, and for the r_2-node the thread can already compute the south and east contributions that occur in Step 2. At this point it is best to present the first CUDA kernel already, see Listing 19.14. Remark: y corresponds with vector q and x with vector p.

Listing 19.14: CUDA kernel matv::kernel_matv1().

```
1  template <class T>
2  __global__ void kernel_matv1(T *y, const T *x, const Grid g)
3  {
4      int ld = g.nx;
5
6      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7      int v_r1 = BORDER_WIDTH + by * By + ty;
8      int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
9      int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
12     int v_b1 = BORDER_WIDTH + by * By + ty;
13     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
14     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
15
16     int loc_r1 = ld * v_r1 + u_r1;
17     int loc_r2 = ld * v_r2 + u_r2;
18     int loc_b1 = ld * v_b1 + u_b1;
19     int loc_b2 = ld * v_b2 + u_b2;
20
21     const T x_r1    = __FETCH_XX(u_r1,   v_r1,   ld);
22     const T x_r1up1 = __FETCH_XX(u_r1+1, v_r1,   ld);
23     const T x_r1vp1 = __FETCH_XX(u_r1,   v_r1+1, ld);
24     const T x_r2    = __FETCH_XX(u_r2,   v_r2,   ld);
25     const T x_r2vm1 = __FETCH_XX(u_r2,   v_r2-1, ld);
26     const T x_r2um1 = __FETCH_XX(u_r2-1, v_r2,   ld);
27
28     const T nn_r1   = g.nn[loc_r1];
29     const T ee_r1   = g.ee[loc_r1];
30     const T ss_r2   = g.ss[loc_r2];
31     const T ww_r2   = g.ww[loc_r2];
32
33     // b1:
34     T sum_b1 = g.cc[loc_b1] * (ss_r2        * x_r2 +     // north
35                                g.ee[loc_b1] * x_r1up1 +  // east
36                                g.ss[loc_b1] * x_r2vm1 +  // south
37                                ee_r1        * x_r1);     // west
38
39     // b2:
40     T sum_b2 = g.cc[loc_b2] * (g.nn[loc_b2] * x_r1vp1 +  // north
41                                ww_r2        * x_r2 +     // east
42                                nn_r1        * x_r1 +     // south
43                                g.ww[loc_b2] * x_r2um1);  // west
44
```

```
45      y[loc_b1] = sum_b1;
46      y[loc_b2] = sum_b2;
47
48      // r1 (partial):
49      T sum_r1 = g.cc[loc_r1] * x_r1 -
50                 nn_r1        * sum_b2 -
51                 ee_r1        * sum_b1;
52
53
54      // r2 (partial):
55      T sum_r2 = g.cc[loc_r2] * x_r2 -
56                 ss_r2        * sum_b1 -
57                 ww_r2        * sum_b2;
58
59      y[loc_r1] = sum_r1;
60      y[loc_r2] = sum_r2;
61  }
```

In lines 21-28 we recognize how data is fetched through textures. The stencils that are needed multiple times are stored in registers, see lines 28-31. These stencils are used for both the computation of the intermediate q-values in the b_1- and b_2-nodes as well as for the partial computation of the final p values in the r_1- and r_2-nodes. Note that the intermediate q-values still have to be written to the global memory as other threads need them later on. Also note that for the extra computations, see lines 49-51 and 55-57, apart from `g.cc[loc_r1]` and `g.cc[loc_r2]` no new data is required. This means that the computations can be done extremely rapidly! The latency of fetching the 2 `g.cc` values is fully hidden under the computations.

This time we count a total of 4 global memory writes (2 intermediate q-results and 2 partial p-values) and 18 global memory reads. Both numbers are higher than in the original version; however, of course, in the new version of Step 2 the numbers will be lower, so that on overall the number of reads and writes is less.

19.8.5 Step 2 in CUDA

Half of the computations of Step 2 are now performed in Step 1, so that the remaining part of Step 2 takes only a few computations and a few data fetches, see Listing 19.15. Remark: `y` corresponds with vector q and `x` with vector p.

Listing 19.15: CUDA kernel matv::kernel_matv2().

```
1  template <class T>
2  __global__ void kernel_matv2(T *y, const T *x, const Grid g)
3  {
4      int ld = g.nx;
5
6      int u_r1 = BORDER_WIDTH + bx * Bx + tx;
7      int v_r1 = BORDER_WIDTH + by * By + ty;
8      int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
9      int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;
10
11     int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
12     int v_b1 = BORDER_WIDTH + by * By + ty;
13     int u_b2 = BORDER_WIDTH + bx * Bx + tx;
14     int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;
```

```
15
16      int loc_r1 = ld * v_r1 + u_r1;
17      int loc_r2 = ld * v_r2 + u_r2;
18
19      const T y_b2vm1 = __FETCH_YY(u_b2,   v_b2-1, ld);
20      const T y_b1um1 = __FETCH_YY(u_b1-1, v_b1,   ld);
21      const T y_b1vp1 = __FETCH_YY(u_b1,   v_b1+1, ld);
22      const T y_b2up1 = __FETCH_YY(u_b2+1, v_b2,   ld);
23
24      // r1 (correction):
25      T sum_r1 = y[loc_r1] - g.ss[loc_r1] * y_b2vm1  // south
26                           - g.ww[loc_r1] * y_b1um1; // west
27
28      // r2 (correction):
29      T sum_r2 = y[loc_r2] - g.nn[loc_r2] * y_b1vp1  // north
30                           - g.ee[loc_r2] * y_b2up1; // east
31
32      y[loc_r1] = sum_r1;
33      y[loc_r2] = sum_r2;
34  }
```

The corrections to the r_1- and r_2-nodes are done in lines 25-26 and 29-30. For the r_1-node the south and west contributions were missing and for the r_2-node the north and east contributions were missing. We count 2 global memory writes and 10 global memory fetches.

Hence in the modified version we find on total $4 + 2 = 6$ global memory reads and $18 + 10 = 28$ global memory reads. This means a reduction of 6 global memory reads at the cost of 2 extra global memory writes. However, as we have mentioned earlier, this is not the only improvement: also global memory latency can be hidden much better, so that on overall we gain a good 15% reduction in computing time.

19.9 Dot products

In this section we discuss how inner products, also called dot products, are computed in our CUDA RRB-solver. As emphasized earlier, the CG-algorithm operates on the first level red nodes only, so that for computing dot products also only the first level red nodes are required. In Section 17.2 we have discussed in detail how we can compute a (big) sum

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \ldots + x_n$$

and hence also dot products, i.e.,

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i$$

as fast as possible in CUDA. We have seen that the way to go is the work- and cost-efficient parallel sum reduction algorithm. This algorithm is used in the CUBLAS library routines `cublasSdot()` and `cublasDdot()` (single resp. double precision). In the CUBLAS library routines it is assumed that the vectors x and y are stored as linear (1D) arrays. This introduces already a problem for us: because of the $r_1/r_2/b_1/b_2$-storage format, the r_1- and r_2-nodes do not lie next to each other in the global memory; in between we have black nodes, see Figure 19.33.

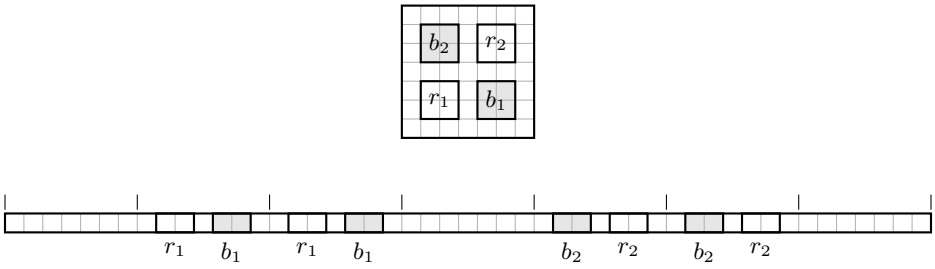


Figure 19.33: The $r_1/r_2/b_1/b_2$-storage format and how it is actually stored in the device's global memory as a linear (1D) array.

Hence we cannot use the CUBLAS library routines, or at least, not right away. At this point one may note that by switching b_1 and r_2 we can make sure that the r_1- and r_2-nodes will become almost next to each other in the global memory. We write "almost" because in between we keep the "gaps" (padded zero's) with one and two times $BORDER\_WIDTH$-width, so that by applying a CUBLASdot product routine to the vectors x and y with the suggested modified $r_1/r_2/b_1/b_2$-storage format results in some useless computations for the gaps (multiplication of padded zero's). However, for grids large enough (which is the case for realistic problems) the amount of overhead becomes insignificant (say 1 percent or so). The "switching" approach has thus good potential.

We also wrote "not right away". With that we target the following. The CUBLASdot product routine cannot be used right away because of the $r_1/r_2/b_1/b_2$-storage format, but suppose we have succeeded to do the first sum-reduction level ourselves. If we ensure that the intermediate results are written to a linear array `odata` (which is actually very logical), from

that point on we can apply the CUBLASroutine to do the remaining sum-reduction levels and come up with the final answer.

But wait, why should we strive for using the CUBLAS-routine anyways? The most important argument would be the conviction that the CUBLAS-routine will always be faster than code that we come up with ourselves. A second argument may be that by using library routines throughout the code, the solver may become faster for free each time a new version of the CUBLASlibrary is released by NVIDIA. However, for our CUDA RRB-solver this second argument would make no sense as all other routines in our CUDA RRB-solver are already custom build.

Regarding the first argument: our custom dot product implementation outperforms the CUBLASroutine cublasSdot() for most array sizes, even when not fully optimized. In the next sections we explain how our implementation works.

19.9.1 A two-step approach

The central idea is to compute the dot product $\langle x, y \rangle$ in two steps, namely:

1. Do a mass reduction on the GPU, store the intermediate results in a linear array odata, and copy this array back from the device to the host;
2. Use Kahan summation on the CPU to add the elements in odata.

Let us first explain what Kahan-summation is, and then how we make the array odata on the GPU.

19.9.2 Kahan summation

The Kahan summation algorithm greatly reduces the numerical error that occurs when many floating point numbers are added together in finite precision. At this point we should note that the parallel sum-reduction algorithm is, although it is slightly less accurate, generally a better approach to reduce the numerical error, because it keeps the summation work-efficient, see Section 17.2. However, if a parallel method cannot be used, the Kahan summation algorithm is a good alternative on sequential machines.

The central idea behind the Kahan summation algorithm is the introduction of a seperate running compensation, i.e., a variable to accumulate small errors. Is is beyond the scope of this report to give an accuracy analysis or other mathematical details. For us it is enough to know that the Kahan summation algorithm makes it possible to sum n numbers with an error that only depends on the floating-point precision [10].

In C++ the Kahan summation algorithm can be implemented as given in Listing 19.16. A C++ template is used to make the code suitable for both single and double-precision numbers. size contains the length of the linear array odata. Let us illustrate the algorithm with the following example which comes from Wikipedia[1]. Although actual computers use binary arithmetic, the example works with 6-digit decimal precision to illustrate the principle (this way you can use a 10-digit hand calculator along).

Suppose we have so far $sum = 10000.0$ and we have to add 3.14159 and 2.71828 (do you recognize these numbers?). The correct result would be: $10000.0 + 3.14149 + 2.71828 = 10005.85987$ which rounds to 10005.9. In case we were to naively add the three numbers

[1]See: http://en.wikipedia.org/wiki/Kahan_summation_algorithm.

we would get $10000.0 + 3.14159 = 10003.14159$ which rounds to 10003.1 and then $10003.1 + 2.71828 = 10005.81828$ which rounds to 10005.8. This is clearly not correct.

Listing 19.16: C++ Kahan summation algorithm.

```
1 template <class T>
2 T Kahan(T* odata, unsigned int size)
3 {
4     T sum = 0;
5     T y = 0;
6     T t = 0;
7     T c = 0;
8
9     for (unsigned int i = 0; i < size; i++) {
10        y = odata[i] - c;
11        t = sum + y;
12        c = (t - sum) - y;
13        sum = t;
14    }
15
16    return sum;
17 }
```

Let us now go step by step through the Kahan summation algorithm. Assume $c = 0$ initially.

`y = odata[i] - c;`	$= 3.14159 - 0$	`c` is 0 initially
	$= 3.14159$	
`t = sum + y;`	$= 10000.0 + 3.14159$	
	$= 10003.14159$	without rounding
	$= 10003.1$	many digits lost after rounding!
`c = (t - sum) - y;`	$= (10003.1 - 10000.0) - 3.14159$	
	$= 3.10000 - 3.14159$	
	$= -0.0415900$	
`sum = t;`	$= 10003.1$	
`y = odata[i] - c;`	$= 2.71828 - -0.0415900$	see above for new `c`
	$= 2.75987$	
`t = sum + y;`	$= 10003.1 + 2.75987$	
	$= 10005.85987$	without rounding
	$= 10005.9$	after rounding
`c = (t - sum) - y;`	$= (10005.9 - 10003.1) - 2.75987$	
	$= 2.80000 - 2.75987$	
	$= 0.040130$	
`sum = t;`	$= 10005.9$	correct result!

We see how `c` keeps track of the digits that were not assimulated into `sum`, and each time a new `y` is computed these "lost" digits are brought into account. We see how introduction of an extra variable, the accumulator `c`, leads to a method that is able to compute correct results within the floating-point precision. Clearly, the higher accuracy does not come for free: an additional 3 flops and some memory operations are needed. However, for small arrays `odata`, say less than 10k elements, we observed that the computing times are so short (just a few μs) that the extra work does not botter us.

19.9.3 Mass reduction phase on the GPU

In the previous section we have seen how Kahan summation can accurately compute the sum of the elements of a linear array. It was also mentioned that for arrays small enough, that is, $\mathcal{O}(10^4)$ elements, the computation times are in the order of micro seconds. In the first step of our CUDA dot product implementation we thus target to generate a linear array odata that has about $\mathcal{O}(10^4)$ elements. For realistic problems the grid typically consists of $\mathcal{O}(10^6)$ elements, so that the GPU has to reduce the number of elements that has to be summed with a factor $\mathcal{O}(10^2)$.

This reduction of a factor $\mathcal{O}(10^2)$ is obtained as follows. In Figure 19.34 it is shown how threads are organized to compute the dot product $\langle x, y\rangle$. Vectors x and y are stored in the $r_1/r_2/b_1/b_2$-format. The red nodes are used only, i.e., the r_1- and r_2-nodes. In the figure is is assumed that the compute block consists of 16×16 elements, but for a compute block with dimensions 32×32 there are no fundamental differences.

Each compute block is divided in sub-blocks by introduction of the block-factor called DOTP_BF. This number DOTP_BF says in how many sub-blocks the compute block is divided equally. For example, if DOTP_BF = 4 each 16×16 compute-block is divided in 4 sub-blocks with dimenions 16×4, if DOTP_BF = 1 there is only 1 sub-block as big as the compute block itself: 16×16 elements.

We write "sub-blocks" rather than "thread-blocks" because they are not exactly the same, namely: for each compute-block only *one* of the sub-blocks will also be a thread-block, i.e., the one that is on the bottom of the compute block.

Each thread in a thread-block sums up DOTP_BF different products $x_i y_i$, namely the ones that lie with stride DOTP_BF * ld in the global memory (check for yourself). For one thread $(tx = 0, ty = 0)$ the different values are indicated by the small black squares. The reason why we let each thread do more work, is to get a (more) cost-efficient algorithm and maximal throughput, see Section 17.2.2. We see that by the threads organized in this way we get indeed a reduction of a factor $\mathcal{O}(10^2)$, because each compute-block delivers 1 value to the output array odata. So, for compute-blocks with size 16×16 the number of elements is reduced by a factor $16^2 = 256$ and for compute-blocks of size 32×32 the number of elements is even reduced by a factor $32^2 = 1024$.

In Listing 19.17 the code for the kernel dotp::kernel_dotp1 is presented. In line 19 we see how precisely enough shared memory, sm, is allocated. As there are DIM_COMPUTE_BLOCK * (DIM_COMPUTE_BLOCK / DOTP_BF) threads alive per thread-block we also need this amount of shared memory. In lines 23-28 we see how each threads computes products $x_i y_i$ and adds them to a running total sum. Note that since only a few elements are added which all have about the same magnitude it is prettly unlikely that large roundoff errors occur. In lines 30-31 we see how the intermediate result sum is written to the shared memory sm and afterwards the threads are synchronized (at the level of thread-blocks) to make sure that all data is really stored in sm. In lines 33-53 the values in the shared memory sm are summed together according to a parallel sum-reduction strategy, see Section 17.2.3. Finally, in lines 56-57 we see how the thread with thread-indices $tx = 0, ty = 0$ writes the result to the global memory.

We found that choosing DOTP_BF = DIM_COMPUTE_BLOCK / 2 yields maximal throughput (about 148.31 GB/s in single-precision on a GTX 580 which is the device-to-device bandwidth). Faster than this is simply impossible as the kernel is bandwidth bound (just like CUBLAS's cublasSdot).

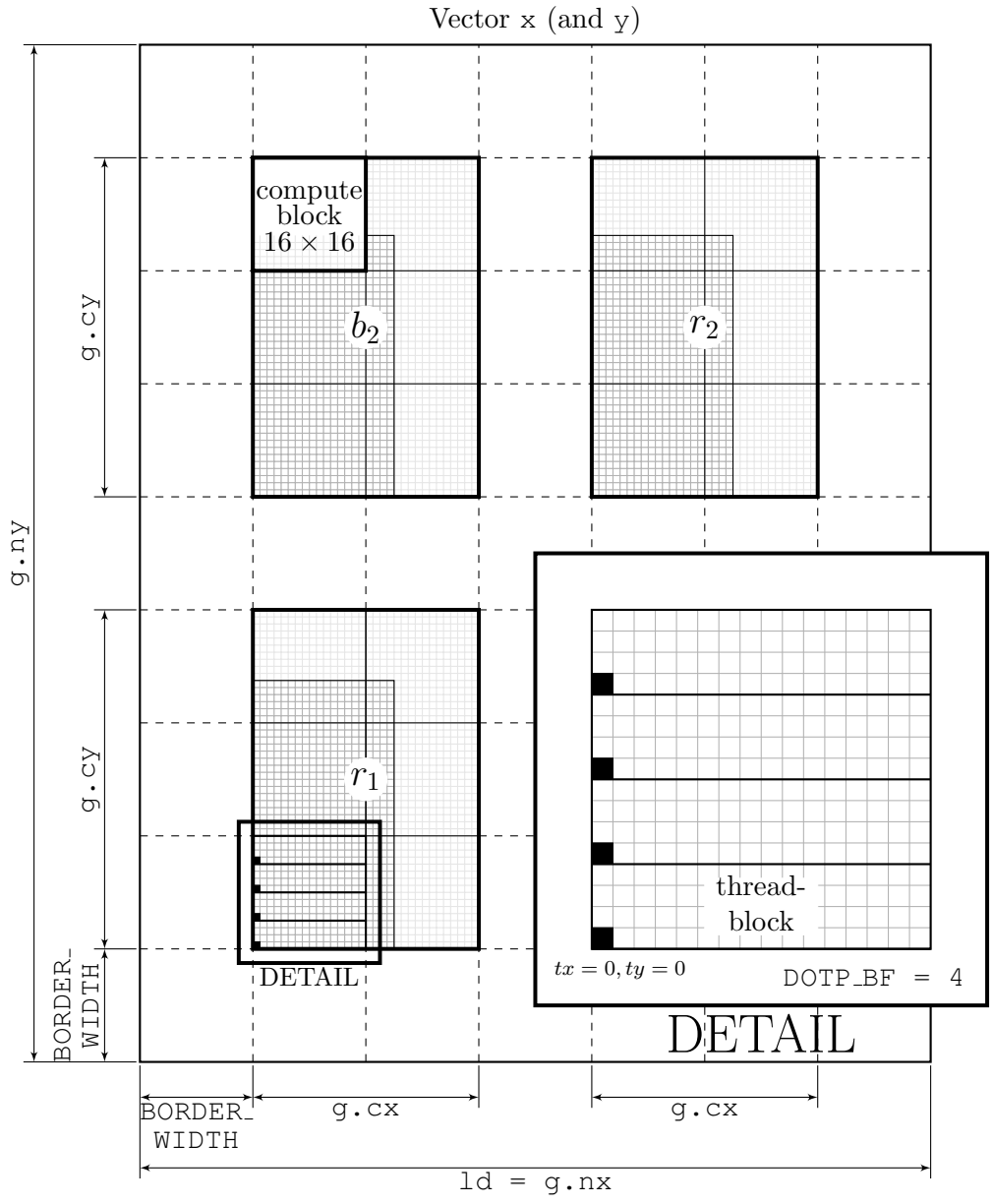


Figure 19.34: Thread organization for dot products.

Listing 19.17: CUDA kernel dotp::kernel_dotp1().

```
1  template <class T>
2  __global__ void kernel_dotp1(T *odata, const T *y, const T *x,
3      const Grid g)
4  {
5      int cx = g.cx;
6      int cy = g.cy;
7      int ld = g.nx;
8
9      int u_r1 = BORDER_WIDTH + bx * DIM_COMPUTE_BLOCK + tx;
10     int v_r1 = BORDER_WIDTH + by * DIM_COMPUTE_BLOCK + ty;
11     int u_r2 = BORDER_WIDTH2 + cx + bx * DIM_COMPUTE_BLOCK + tx;
12     int v_r2 = BORDER_WIDTH2 + cy + by * DIM_COMPUTE_BLOCK + ty;
13
14     int loc_r1 = ld * v_r1 + u_r1;
15     int loc_r2 = ld * v_r2 + u_r2;
16
17     int tid = Bx * ty + tx;
18
19     __shared__ T sm[DIM_COMPUTE_BLOCK * (DIM_COMPUTE_BLOCK / DOTP_BF)];
20
21     T sum = 0;
22
23     for (int k = 0; k < DOTP_BF; ++k) {
24         sum += y[loc_r1] * x[loc_r1];
25         sum += y[loc_r2] * x[loc_r2];
26         loc_r1 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
27         loc_r2 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
28     }
29
30     sm[tid] = sum;
31     __syncthreads();
32
33     for (int k = (DIM_COMPUTE_BLOCK / 2) *
34                  (DIM_COMPUTE_BLOCK / DOTP_BF); k > 16; k >>= 1) {
35         if (tid < k) {
36             sm[tid] += sm[tid + k];
37         __syncthreads();
38         }
39     }
40
41     if (tid < 16)
42         sm[tid] += sm[tid + 16];
43     __syncthreads();
44     if (tid < 8)
45         sm[tid] += sm[tid + 8];
46     __syncthreads();
47     if (tid < 4)
48         sm[tid] += sm[tid + 4];
49     __syncthreads();
50     if (tid < 2)
51         sm[tid] += sm[tid + 2];
52     __syncthreads();
53     if (tid < 1)
54         sm[tid] += sm[tid + 1];
55     __syncthreads();
56     if (tid == 0)
```

```
57         odata[by * gridDim.x + bx] = sm[tid];
58 }
```

19.10 AXPYs

Vector updates (AXPY) is a very easy to implement routine. The following code speaks for itself.

Listing 19.18: CUDA kernel axpy::kernel_axpy().

```
1 template <class T>
2 __global__ void kernel_axpy(T *y, const T *x,
3     const T a, const T b, const Grid g)
4 {
5     int cgx = g.cx;
6     int cgy = g.cy;
7     int ld  = g.nx;
8
9     int v_r1 = BORDER_WIDTH + by * By + ty;
10    int u_r1 = BORDER_WIDTH + bx * Bx + tx;
11    int v_r2 = BORDER_WIDTH2 + cgy + by * By + ty;
12    int u_r2 = BORDER_WIDTH2 + cgx + bx * Bx + tx;
13
14    int loc_r1 = ld * v_r1 + u_r1;
15    int loc_r2 = ld * v_r2 + u_r2;
16
17    y[loc_r1] = a * y[loc_r1] + b * x[loc_r1];
18    y[loc_r2] = a * y[loc_r2] + b * x[loc_r2];
19 }
```

Chapter 20

The CUDA IPDIAG-solver

In this chapter we discuss in detail how the IPDIAG-solver is implemented in CUDA. The mathematics and central idea behind the IP preconditioning are already discussed in Section 10.2.3.

20.1 Outline

The IPDIAG-solver is a CG-type solver with Incomplete Poisson (IP) as preconditioner combined with diagonal scaling (DIAG). Actually diagonal scaling is form of preconditioning, see Section 10.1.1. Diagonal scaling is added to ensure that the solver becomes stable for all test problems; without diagonal scaling the approximate solution may diverge at some point in time for particular test problems, e.g., Klopman's harbour.

The IPDIAG-solver is implemented in such a way that the user can choose beforehand what solver he or she wants to use: plain CG, CG + diagonal scaling, CG + IP preconditioning, or CG + diagonal scaling + IP preconditioning. Also, the user can decide whether to start the iterative process with the zero vector ($x = 0$) rather than the solution x from the previous time step. Both futures are achieved by means of preprocessor directives (`#define`) as this is the fastest alternative (unused code is just not compiled).

In Algorithm 8 an outline of the complete algorithm is given as it is implemented in CUDA. Note that the `if`-statements with capitals (e.g., `USEIPPREC`) are catched in preprocessor directives and hence are not real `if`-statements. Verify for yourself that when `USEDIAGSCAL = 0` and `USEIPPREC = 0` we get back the CG-algorithm, see Algorithm 3 (although slightly rewritten).

20.1.1 Input and output

The input consists at least of the discretization matrix S, the right-hand side (RHS) b (stored in r), an initial guess x (solution from the previous time frame), a tolerance *psitol*, and a maximum bound for the number of CG-iterations *maxiter*.

If the IP preconditioner is used (`USEIPPREC = 1`) the algorithm works with an extra input: matrix M^{-1}, the preconditioning matrix, see lines 15 and 43, accompanied with an extra vector z, the so-called preconditioned residual, see lines 16, 25, 43 and 47.

If moreover diagonal scaling is used (`USEDIAGSCAL = 1`) the algorithm works with another extra input: diagonal matrix P, see lines 5, 11, and 53.

The matrices S, M^{-1} and P come in the form of stencils. For both S and M^{-1} three stencils are needed: a center (StC), a west (StW) and a south (StS) stencil. As matrix P is a diagonal matrix only 1 stencil is required.

The only output is the solution vector x which thus serves as next initial guess in case we have set `USEZEROVECTOR = 0`.

Input: S, (M^{-1}), (P), r, x, $psitol$, $maxiter$
Output: x

```
1  if USEZEROVECTOR then
2  |  x = 0;
3  end
4  if USEDIAGSCAL then
5  |  x = P^T x;
6  |  y = S̃x;                          // SpMV: version 2
7  else
8  |  y = Sx;                           // SpMV: version 1
9  end
10 if USEDIAGSCAL then
11 |  r = P^{-1} r;
12 end
13 r = r − y;
14 if USEIPPREC then
15 |  z = M^{-1} r;                     // SpMV: version 1
16 |  ρ_new = ⟨r, z⟩;
17 else
18 |  ρ_new = ⟨r, r⟩;
19 end
20 stop = (ρ_new + 1) · (psitol)^2;
```

Algorithm 8: The IPDIAG-algorithm (see also next page).

20.1.2 SpMVs: two flavours

In lines 6, 8, 15, 34, 36 and 43 we find SpMVs (sparse matrix-vector products). As indicated there are two versions. The reason for two different implementations is that in case diagonal scaling is used the SpMV becomes (slightly) cheaper: if n is the total number of nodes ($n = N_x \times N_y$), n multiplications can be saved. This is explained in Section 10.1.1. The matrix $\widetilde{S}$ is computed by

$$\widetilde{S} := P^{-1} S P^{-T}.$$

and as a consequence the entries on the main diagonal of $\widetilde{S}$ (i.e., the center stencil) are all 1. In Section 20.2.6 we discuss the SpMVs and their implementation in detail.

```
while (ρ_new > stop & iter < maxiter) do
    iter++;
    if iter = 1 then
        if USEIPPREC then
            p = z;
        else
            p = r;
        end
    else
        β = ρ_new / ρ_old;
        p = r + βp;                          // AXPY
    end
    if USEDIAGSCAL then
        q = S̃p;                    // SpMV: version 2
    else
        q = Sp;                    // SpMV: version 1
    end
    σ = p^T q;                          // dot product
    α = ρ_new / σ;
    r = r − αq;                              // AXPY
    x = x + αp;                              // AXPY
    if USEIPPREC then
        z = M^{-1} r;              // SpMV: version 1
    end
    ρ_old = ρ_new;
    if USEIPPREC then
        ρ_new = ⟨r, z⟩;                 // dot product
    else
        ρ_new = ⟨r, r⟩;                 // dot product
    end
end
if USEDIAGSCAL then
    x = P^{-T} x;
end
```

Algorithm 8: The IPDIAG-algorithm (continued).

20.1.3 Termination criterium

The termination criterium for the IPDIAG-solver depends on whether or not IP preconditioning is used. In case of a plain CG or CG + diagonal scaling, the termination criterium is given by: stop the iterative process when

$$\rho_{\text{new}} = \langle r_i, r_i \rangle_2 = \|r_i\|_2^2 \leq (\|r_0\|_2^2 + 1) \cdot (psitol)^2$$

(or when the maximal number of iterations is exceeded). In that case the ordinary residual r_i is thus used for iteration i. In case of CG + IP preconditioning or the true IPDIAG-solver

the termination criterium is given by: stop the iterative process when

$$\rho_{\text{new}} = \langle r_i, z_i \rangle_2 = \|r_i\|^2_{M^{-1}} \leq (\|r_0\|^2_{M^{-1}} + 1) \cdot (psitol)^2$$

(or when the maximal number of iterations is exceeded), hence the criterium that is used in all PCG solvers in the `lin_wacu` software. The termination criterium is found in lines 20 and 21.

20.2 Implementation

20.2.1 General comments

The current implementation of the `CUDA` IPDIAG-solver consists mostly of custom-built kernels. For computation of dot products the `CUBLAS` library routines `cublasSdot()` (single-precision) and `cublasDdot()` (double-precision) are used. However, it may be worth to write similar own code, see Sections 17.2 and 19.9 (faster). Throughout the source code use is made of the symbol `REAL` which can either be `float` or `double`, and the kernels are written using template functions. In this way the IPDIAG-solver can handle both single-precision and double-precision numbers.

Although at the moment the preconditioner is constructed only once for an entire simulation, we have made the IPDIAG-solver "future-proof". That is, all time intensive computations are done on the GPU rather than on the CPU, so that in the future the IPDIAG-solver can also be used for time varying depth-profiles in case which multiple times per second the preconditioner has to be constructed.

In the `lin_wacu` software the IPDIAG-solver is called the "CudaSimpleSolver", the reason being obvious: this `CUDA` solver is very basic, especially compared to the `CUDA` RRB-solver. Accordingly, the IPDIAG-solver is saved in the files `CudaSimpleSolver.cpp` with corresponding header `CudaSimpleSolver.h` and various other (header-)files.

20.2.2 Memory requirements

The total amount of memory required is easy to compute. Suppose the number of nodes in the x-direction is N_x and the number of nodes in the y-direction is N_y. The total number of nodes is then $n = N_x \cdot N_y$. In Table 20.1 all (significant) data objects are listed.

Depending on what solver is chosen, all or fewer linear arrays are needed. In case of plain CG we only need the stencils for S and the vectors r, x, y, p and q, hence 8 linear arrays of size n. In case of CG + diagonal scaling we need additionaly the diagonal matrix P. However, as the center stencil of S only contains ones we do not need the center stencil, hence 8 linear arrays of size n are sufficient. In case of the IPDIAG-solver we need all of them: a total of 13 linear arrays. Also note that in a most memory efficient implementation the vector y is actually not needed.

20.2.3 Constructing the preconditioner(s)

Actually there are two preconditioners: the preconditioner matrix M^{-1} and the diagonal matrix P. Let us start with the matrix P.

Variable	Description	Storage format
*dcc	matrix S center stencil	linear array of size n
*dss	matrix S south stencil	linear array of size n
*dww	matrix S west stencil	linear array of size n
*kcc	matrix M^{-1} center stencil	linear array of size n
*kcc	matrix M^{-1} south stencil	linear array of size n
*kcc	matrix M^{-1} west stencil	linear array of size n
*dpp	matrix $P = \sqrt{D}$	linear array of size n
*dz	vector z	linear array of size n
*dr	vector r	linear array of size n
*dx	vector x	linear array of size n
*dy	vector y	linear array of size n
*dp	vector p	linear array of size n
*dq	vector q	linear array of size n

Table 20.1: Memory requirements of the CUDA IPDIAG-solver.

Diagonal matrix P

In Section 10.1.1 it was explained that P is computed at the hand of the diagonal of S. Actually, we have $D = PP^T$, where D is the diagonal of matrix S and hence P, which is thus also diagonal, is found by

$$P = \text{diag}(\sqrt{d_1}, \sqrt{d_2}, \ldots, \sqrt{d_n}),$$

where the d_i's are the main diagonal elements of D (the only nonzeros). The diagonal of S, i.e., matrix D is stored in the center stencil StC. Therefore, accordingly, in CUDA the matrix P can be stored in 1 stencil and the following CUDA kernel can be used:

Listing 20.1: CUDA kernel makeDiagonal().

```
1 template <class T>
2 __global__ void makeDiagonal(T *stP, const T *stC, const int n)
3 {
4     int tid;
5
6     tid = bx * Bx + tx;
7     if (tid < n)
8         stP[tid] = sqrt(stC[tid]);
9 }
```

Herein is n the total number of nodes, i.e., n = Nx1 * Nx2, and tid points to a unique location in the arrays. In this way each thread computes one value in P's stencil StP at the hand of the corresponding value of the center stencil of S which is stored in StC.

For a grid of 2048×2048 nodes the kernel reaches a useful throughput of 162.7 GB/s on a GeForce GTX 580; the kernel is bandwidth bound.

IP preconditioner M^{-1}

In Section 10.2.3 it was explained how the IP preconditioner M^{-1} is computed at the hand of the matrix S. The procedure was illustrated with the 2D Poisson problem. Recall that

$$M^{-1} = KK^T, \quad K = I - LD^{-1},$$

where I is the identity matrix, L the strictly lower part of S, and D the diagonal of S. It was shown that the matrix M^{-1} constructed in this way is not given by a 5-point stencil but by a 7-point stencil. Next it was shown that, in case of the 2D Poisson problem, approximating M^{-1} by $\widetilde{M}^{-1}$, where $\widetilde{M}^{-1}$ is found by leaving out the fill-in that occured, does lead to an acceptable approximation. The matrix $\widetilde{M}^{-1}$ is thus given by a 5-point stencil, so that the preconditioning step, $M^{-1}z = r$ (thus actually $\widetilde{M}^{-1}z = r$), comes down to computation of an SpMV just like for the product $q = Sp$ is needed and hence the same CUDA kernel can be used!

The procedure was illustrated at the hand of the 2D Poisson example with N nodes in the x- and y-direction. In that case the matrix M^{-1} is given by

$$\begin{aligned}\text{row}_i(M^{-1}) &= (M^{-1}_{i-N}, M^{-1}_{i-1}, M^{-1}_i, M^{-1}_{i+1}, M^{-1}_{i+N}),\\ &= (\frac{1}{4}, \frac{1}{4}, \frac{9}{8}, \frac{1}{4}, \frac{1}{4})\end{aligned}$$

or, equivalently, M^{-1} is given by the stencil

$$M^{-1} = \begin{bmatrix} & 1/4 & \\ 1/4 & 9/8 & 1/4 \\ & 1/4 & \end{bmatrix}.$$

Let us now see what changes if we apply the method to our SPD matrix $S \in \mathbb{R}^{N_xN_y \times N_xN_y}$ on a grid with N_x nodes in the x- and N_y nodes in the y-direction. Note that by symmetry, we only have to consider $\text{row}_i(S) = (S_{i-N_x}, S_{i-1}, S_i, *, *)$, i.e., the south, west and center stencil are sufficient; those three describe matrix S completely. It can be computed, just like in Section 10.2.3 is done for the 2D Poisson problem, that if S is given by

$$\begin{aligned}\text{row}_i(S) &= (S_{i-N_x}, S_{i-1}, S_i, *, *),\\ &= (s_i, w_i, c_i, *, *),\end{aligned}$$

then the matrix M^{-1} (thus the approximation) is given by

$$\begin{aligned}\text{row}_i(M^{-1}) &= (M^{-1}_{i-N_x}, M^{-1}_{i-1}, M^{-1}_i, *, *),\\ &= (\frac{s_i}{c_i}, \frac{w_i}{c_i}, 1+(\frac{w_i}{c_i})^2+(\frac{s_i}{c_i})^2, *, *).\end{aligned}$$

Note that this is only correct for most rows of matrix M^{-1}; for the first N_x rows of matrix M^{-1}, and especially the very first row, the stencil is slightly different, i.e., fewer terms occur. In Listing 20.2 corresponding CUDA code is presented. Herein are `Nx1, Nx2` the number of nodes in x- and y-direction, respectively (N_x and N_y). The stencil of S is stored by `StSC, StSS` and `StSW`, the stencil of the preconditioning matrix M^{-1} will be saved in `StMC, StMS` and `StMW`.

In lines 16, 18 and 20 we see `if`-statements that make sure that the first `Nx1` (N_x) rows are handled differently. Each thread computes one value in `StMC`, one value in `StMS` and one value in `StMW`, hence each thread does somewhat more work which is good for bandwidth. We also see how the center stencil of M^{-1}, `StMC`, is computed in terms of the west, `StMW`, and south, `StMS`, stencil of M^{-1}.

For a grid of 2048×2048 nodes the kernel reaches a useful throughput of 151.1 GB/s on a GeForce GTX 580; the kernel is bandwidth bound.

Listing 20.2: CUDA kernel makeIPPrec().

```
template <class T>
__global__ void makeIPPrec(T *StMC,       T *StMS,       T *StMW,
                           const T *StSC, const T *StSS, const T *StSW,
                           const int Nx1, const int Nx2)
{
    int tid = bx * Bx + tx;

    T valSC, valMW, valMS, valMC;

    if (tid < Nx1 * Nx2)
    {
        valSC = 1.0 / stSC[tid];
        valMW = StSW[tid] * valSC;
        valMS = StSS[tid] * valSC;

        if (tid == 0) // first row of M^{-1}
            valMC = 1.0;
        else if (tid > 0 && tid < Nx1) // first Nx1 rows except row 1
            valMC = 1.0 + valMW * valMW;
        else               // all other rows
            valMC = 1.0 + valMW * valMW + valMS * valMS;

        StMW[tid] = valMW;
        StMS[tid] = valMS;
        StMC[tid] = valMC;
    }
}
```

20.2.4 Updating the matrix S in case of diagonal scaling

In case of diagonal scaling, the matrix S is overwritten by the matrix $\widetilde{S} := P^{-1}SP^{-T}$. For this matrix $\widetilde{S}$ the entries on the main diagonal are all 1. Note that, in case diagonal scaling is enabled (`USEDIAGSCAL = 0`), the matrix S (thus actually $\widetilde{S}$) is fully described by two stencils only: a new west and south stencil; the center stencil is no longer needed as its entries are all 1.

To compute the new matrix S we need matrix P. In Listing 20.3 suitable CUDA code is presented. Each thread computes three outputs: the updated center value `StC` which is always 1, thus this part of the code can be left out, the updated west value `stW` and the updated south value `StS`.

Listing 20.3: CUDA kernel updateMatrix().

```
template <class T>
__global__ void updateMatrix(T *StC, T *StS, T *StW,
                             const T *StP,
                             const int Nx1, const int Nx2)
{
    int tid = bx * Bx + tx;

    if (tid < Nx1 * Nx2)
    {
        StC[tid] = 1.0; // center (not needed explicitly)
        if (tid > 0)
```

```
12              StW[tid] /= (StP[tid - 1  ] * StP[tid]); // west
13          if (tid > Nx1 - 1)
14              StS[tid] /= (StP[tid - Nx1] * StP[tid]); // south
15      }
16  }
```

For a grid of 2048×2048 nodes the kernel reaches a useful throughput of 197.7 GB/s on a GeForce GTX 580; the kernel is bandwidth bound.

20.2.5 The operations $x = P^T x$ and $x = P^{-T} x$

Since matrix P is a diagonal matrix, the operations $x = P^T x$ and $x = P^{-T} x$ come down to element-wise multiply and element-wise division, respectively. Listings 20.4 and 20.5 do the job.

Listing 20.4: CUDA kernel elementWiseMul().

```
1 template <class T>
2 __global__ void elementWiseMul(T *x, const T *p, const int n)
3 {
4     int tid = bx * Bx + tx;
5     if (tid < n)
6         x[tid] *= p[tid];
7 }
```

Listing 20.5: CUDA kernel elementWiseDiv().

```
1 template <class T>
2 __global__ void elementWiseDiv(T *x, const T *p, const int n)
3 {
4     int tid = bx * Bx + tx;
5     if (tid < n)
6         x[tid] /= p[tid];
7 }
```

For a grid of 2048×2048 nodes the kernels reach a useful throughput of 173.7 GB/s and 176.8 GB/s, respectively, on a GeForce GTX 580; the kernels are bandwidth bound.

20.2.6 SpMVs: two flavours

The sparse matrix-vector product (SpMV) is discussed in great detail in Section . In that section hints are given for an optimal CUDA implementation, depending on the GPU's architecture. On Fermi GPUs it is most proficient in for both memory and performance to use three stencils only: StC, StW and StS, rather than all 5. Memory speaks for itself and the slightly better performance has to do with the fact that L1 cache of the device is used in that case. On older GPUs some performance can be gained if 5 stencils are used because of more coalesced memory transactions. It is also shown that vector x in the computation $y = Sx$ should be fetched through textures (which are cached), to get a good boost in performance. In case of the Fermi GPUs textures do not make a difference as vector x is already cached in L1 without them and with the textures the L1 cache is just replaced by the texture cache. However on older GPUs 45% performance is gained with textures as older GPUs do not have L1 global memory cache.

SpMV kernel version 1

Version 1 of the SpMV is used for: 1) the preconditioning step $M^{-1}z = r$ and 2) if diagonal scaling is not enabled, for computing $q = Sp$. The corresponding code is listed below.

Listing 20.6: CUDA kernel SpMVv1().

```
1  template <class T>
2  __global__ void SpMVv1(T *y,
3                         const T *stC,
4                         const T *stS,
5                         const T *stW,
6                         const T *x,
7                         const int Nx1,
8                         const int Nx2)
9  {
10     int tid = bx * Bx + tx;
11
12     T sum = 0;
13
14     if (tid < Nx1 * Nx2)
15     {
16         sum = stC[tid] * tex1Dfetch(texRefx, tid);                  // center
17
18         if (tid + Nx1 < Nx1 * Nx2)
19             sum += stS[tid + Nx1] * tex1Dfetch(texRefx, tid + Nx1); // north
20         if (tid + 1 < Nx1 * Nx2)
21             sum += stW[tid + 1]   * tex1Dfetch(texRefx, tid + 1);   // east
22         if (tid - Nx1 >= 0)
23             sum += stS[tid]       * tex1Dfetch(texRefx, tid - Nx1); // south
24         if (tid - 1 >= 0)
25             sum += stW[tid]       * tex1Dfetch(texRefx, tid - 1);   // west
26
27         y[tid] = sum;
28     }
29 }
```

For a grid of 2048×2048 nodes the kernel reaches a useful throughput of 253.4 GB/s on a GeForce GTX 580; the kernel is bandwidth bound.

SpMV kernel version 2

Version 2 of the SpMV is used for the matrix-vector product $q = Sp$ in case diagonal scaling is enabled, thus for a matrix S which has a main diagonal consisting of 1s only. The corresponding code is listed below.

Listing 20.7: CUDA kernel SpMVv2().

```
1  template <class T>
2  __global__ void SpMVv2(T *y,
3                         const T *stC, // note: not used (because all 1's)
4                         const T *stS,
5                         const T *stW,
6                         const T *x,
7                         const int Nx1,
8                         const int Nx2)
9  {
```

```
10      int tid = bx * Bx + tx;
11
12      T sum = 0;
13
14      if (tid < Nx1 * Nx2)
15      {
16          sum = tex1Dfetch(texRefx, tid);                              // center
17
18          if (tid + Nx1 < Nx1 * Nx2)
19              sum += stS[tid + Nx1] * tex1Dfetch(texRefx, tid + Nx1); // north
20          if (tid + 1 < Nx1 * Nx2)
21              sum += stW[tid + 1]   * tex1Dfetch(texRefx, tid + 1);   // east
22          if (tid - Nx1 >= 0)
23              sum += stS[tid]       * tex1Dfetch(texRefx, tid - Nx1); // south
24          if (tid - 1 >= 0)
25              sum += stW[tid]       * tex1Dfetch(texRefx, tid - 1);   // west
26
27          y[tid] = sum;
28      }
29  }
```

Notice that the only difference with Listing 20.6 is that the value `Stc[tid]` is no longer loaded from the global memory.

For a grid of 2048×2048 nodes the kernel reaches a useful throughput of 281.7 GB/s on a GeForce GTX 580; the kernel is bandwidth bound.

20.2.7 AXPYs and dot products

The dot products are computed with the CUBLAS library routine `cublasSdot` in case of `floats` or `cublasDdot` in case of `doubles`. The routine achieves a useful throughput of 165.6 GB/s. The vector-updates (AXPYs) are computed with code similar to CUBLAS's routine `cublasSsaxpy`. Our own kernel is just as fast as CUBLAS's routine but it is a more flexible; our AXPY is actually: $y := \alpha x + \beta y$ (AXPBY). The CUDA code is given in Listing 20.8. The kernel reaches a useful throughput of 177.5 GB/s. Both the dot products and the AXPY kernel are bandwidth bound.

Listing 20.8: CUDA kernel AXPY().

```
1  template <class T>
2  __global__ void AXPY(T *y,
3                       const T *x,
4                       const T alpha,
5                       const T beta,
6                       const int n)
7  {
8      int tid = bx * Bx + tx;
9      if (tid < n)
10         y[tid] = alpha * x[tid] + beta * y[tid];
11 }
```

20.2.8 Overlapping and concurrent kernels

Consider Algorithm 8. In lines 1-9 operations are done with vector x which is already on the device. From line 10 and further the operations start on "fresh" vector r coming from the host (the right-hand side b in $S\psi = b$). As the vectors x and r do not rely on each other in these first lines, the idea is to process them concurrently using streams. And communication (vector r has to be copied from the host to the device) can be overlapped with computation. From experiments we know that transferring the data takes a couple of ms, so it would be nice to do as much computations simultanesouly to get some reduction in overall execution time. In Figure 20.1 we have visualized this idea.

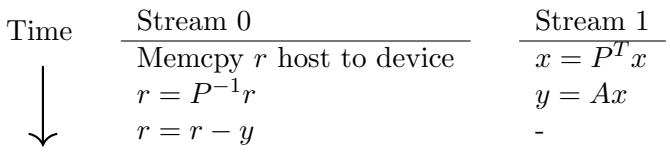


Figure 20.1: Overlapping and concurrent kernels.

To get an idea, in our CUDA IPDIAG-solver this is done as follows:

```
cudaStream_t stream1;
cudaStream_t stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// copy vector b to device (to r)
cudaMemcpy2DAsync(m_dr, m_pitch, pB->getElements() + m_srcOffset,
    pB->columns() * m_elementSize, m_Nx1 * m_elementSize,
    m_Nx2, cudaMemcpyHostToDevice, stream1);

#if USEZEROVECTOR
    misc::memset_kernel<REAL> <<< m_numBlocks, m_numThreads, 0, stream2 >>>
                                  (m_dx, m_n, 0);
#endif

#if USEDIAGSCAL
    // x = P^Tx
    cg::elementWiseMul<REAL>  <<< m_numBlocks, m_numThreads, 0, stream2 >>>
                                  (m_dx, m_dpp, m_n);

    // y = A * x
    cudaBindTexture(NULL, texRefx, m_dx);
    cg::SpMVv2<REAL> <<< m_numBlocks, m_numThreads, 0, stream2 >>>
                          (m_dy, m_dcc, m_dss, m_dww, m_dx, m_Nx1, m_Nx2);
#else
    cudaBindTexture(NULL, texRefx, m_dx);
    cg::SpMVv1<REAL> <<< m_numBlocks, m_numThreads, 0, stream2 >>>
                          (m_dy, m_dcc, m_dss, m_dww, m_dx, m_Nx1, m_Nx2);
#endif

#if USEDIAGSCAL
    // r = P^{-1} * r = P^{-T} * r
```

```
    cg::elementWiseDiv<REAL> <<< m_numBlocks, m_numThreads, 0, stream1 >>>
                                (m_dr, m_dpp, m_n);
#endif

    // r = r - y
    cg::AXPY<REAL> <<< m_numBlocks, m_numThreads, 0, stream1 >>>
                      (m_dr, m_dy, -1, 1, m_n);
```

N.B. Later on, in lines 40 and 41 in Algorithm 8 we see two vector-updates that are completely independent. Also these two can be put in different streams.

Part V

TESTS AND RESULTS

Chapter 21

Testing method

In the sections we shall discuss how we are going to test our new CUDA solvers and compare it with the existing solvers. What basically matters is how fast are our CUDA solvers compared to the current C++ RRB-solver in terms of execution time as the C++ RRB-solver is the fastest of all C++ solvers available. For reliable measurements we have to be consistent and need a performance/timing plan so that there is no room for doubt or confusion.

21.1 Measures and terminology

To see how well our CUDA solvers really perform we have to introduce some extra measures and additional terminology.

21.1.1 Frame time

The simulator is a real-time simulator which basically means that multiple times per second the wave field has to be computed entirely from scratch (possibly using the previous wave field as starting solution). Typically the frame rate is 20 fps, i.e. the program computes 20 times per second a new wave field. The frame time (or time frame) is the period of time that corresponds to the frame rate, i.e., when the frame rate is 20, then the frame time is $1/20 = 0.050\,s = 50\,ms$.

Every time frame many computations are performed. Most of the computations are gathered in the routine `WavesComputer::compute()`. First a new right-hand side b is computed externally, and then in the routine `WavesComputer::compute()` in consecutive order incoming waves are computed, a system $S\psi = b$ is solved, time derivates are computed and finally the time integration is performed using the Leapfrog-method (and lots of minor computations). For a complete overview see Table 25.3.

For the simulator to fulfill the real-time requirement obviously all computations must be performed in less time than the frame time. Therefore we keep track of the total execution time.

21.1.2 Total time

The total (execution) time is the time taken by the computer to perform all computations described above (incoming waves, solver, time derivatives, Leapfrog). Hence for the simulator to allow real-time simulation we require: total time $<$ frame time.

21.1.3 Solver time

The solver (execution) time is the time taken by a solver to solve the system $S\psi = b$ once in a particular time frame. The solver time is thus a fraction of the total time. If the solver time is almost equal to the total time, we say that the solver part is a bottleneck.

21.1.4 Additional time

The additional (execution) time is the time taken by all compututations different from those that needed for solving the system $S\psi = b$. The additional time is thus formed by the computations for incomping waves, time derivatives, Leapfrog, etcetera. We have

$$\text{Total time} = \text{solver time} + \text{additional time}.$$

21.1.5 Speed up

We define speed up as follows. The speed up is a relative measure between two arbitrary solvers, independent from the implemented method and platform that is used (CPU, GPU, etc.). So just the performance counts. We say that solver Y has a speed up of a factor n over solver X if

$$\frac{\text{time solver } X \text{ to solve system}}{\text{time solver } Y \text{ to solve system}} = n.$$

So, for example, if solver X takes 30 ms to solve the system $S\psi = b$ and solver Y takes 10 ms we say that solver Y has a speed up factor of $30/10 = 3$ over solver X. The term speed up is thus used very freely; if solver X were a sequential `C++` RRB-solver (CPU) and solver Y a highly parallel `CUDA` IPDIAG-solver (GPU), we also would just say that solver Y is 3 times faster than solver X. This is a bit tricky and may lead to dishonest comparisons; therefore, we also introduce the term solver speed up.

21.1.6 Solver speed up

The solver speed up is the increase in performance when the sequential (`C++`, CPU) version of a solver is compared to its parallel (`CUDA`, GPU) version. The underlying method is thus kept the same, e.g., the RRB-solver, Nop-solver (CG with diagonal scaling), etc., is used for both the sequential and parallel version.

21.1.7 Total speed up

Like we distinguish between total time and solver time, we also have a total speed up. The total speed up thus relates to the increase in performance taken over a complete time frame. We say that solver y yields a total speed up of a factor n if

$$\frac{\text{total time with solver } X}{\text{total time with solver } Y} = n.$$

For example, suppose that with solver X the system $S\psi = b$ is solved in 30 ms and that the additional computations take 10 ms (incoming waves, time derivates, Leapfrog, etc.). So the total time is 30 + 10 = 40 ms (thus real-time). Next suppose that with solver Y the system $S\psi = b$ is now solved in 10 ms. Of course the additional computations still take 10 ms. The total time is reduced to 10 + 10 = 20 ms, and hence the total speed up is a factor $40/20 =$

2. So, although solver Y is 3 times as fast, because of the additional computations, the total speed up is only a factor 2 (Amdahl's law).

21.1.8 Useful throughput

Throughput has already been discussed in detail in Section 16.1.2. We pointed out that we make a distinction between throughput (= effective bandwidth) and useful throughput. The difference between the terms is that throughput is the total amount of data that passes the device caused by global memory read en global memory write operations. Memory operations are performed on chunks of data; typical are 32-, 64- and 128-byte operations. So, if we use single-precision numbers (`floats` = 4 bytes) and a 32-byte operations is used, 8 numbers are read or written simultaneously. The fraction of the data that is really used to compute things yields the useful throughput, e.g., only the odd numbers are used. The useful throughput is thus a fraction of the throughput, and this is the number that really matters. Therefore, we always look at the useful throughput rather than just the throughput.

21.2 Performance/timing plan

In this section we explain exactly how we ran our experiments. We went through two phases:

1. The solvers evaluated in a special `poisson` testing environment;
2. The solvers plugged-in in the `lin_wacu` software.

21.2.1 Special `poisson` testing environment

The `poisson` testing environment is a custom-built environment to test MARIN's solvers in greater detail. The environment consists a `poisson.cu`-file in which a 2D Poisson problem $Ax = b$ is set-up and solved once with a solver specified by the user. In the header-file `defines.h` of the `CUDA` solver the user has to switch to the output modus for output to the screen or file. This is done by setting various preprocessor directives `#define` to 1 instead of 0 (default). Below we give an outline of the `poisson.cu`-file.

Listing 21.1: Test file `poisson.cu`.

```
1  int main(int argc, char **argv)
2  {
3      // declaration of variables
4      ...
5
6      const REAL tol = 1e-5;              // default tolerance
7      const char *solvername =          ; // default solver is specified
8      Solver* pSolver;
9
10     Array2D<REAL> matC(COLUMNS, ROWS);  // center stencil
11     Array2D<REAL> matW(COLUMNS, ROWS);  // west stencil
12     Array2D<REAL> matS(COLUMNS, ROWS);  // south stencil
13     ...
14
15     // fill matC, matW, matS arrays
16     ...
17
```

```
18      // System to solve: Ax = b
19      // We generate a target y and compute a corresponding b via b = A*y
20      Array2D<REAL> y(COLUMNS, ROWS);
21      Array2D<REAL> b(COLUMNS, ROWS);
22      Array2D<REAL> x(COLUMNS, ROWS);
23
24      // Create target solution y
25      ...
26
27      // Compute corresponding right-hand side (RHS) b on CPU
28      ...
29
30      // Set an initial guess x
31      x = 0.0;                              // zero vector as initial guess
32
33      // setup the solver and make preconditioner
34      START_CPU_TIMER(time, 1);
35      pSolver = SolverFactory::create(solvername, 100, 0, &matC, &matS, &matW,
36              Nx1, Nx2, tol);
37      STOP__CPU_TIMER(time, 1);
38      ...
39
40      // solve the system Ax = b
41      START_CPU_TIMER(time, 1);
42      pSolver -> solve(&b, &x);
43      STOP__CPU_TIMER(time, 1);
44      ...
45
46      delete pSolver; // delete solver and memory
47
48      return EXIT_SUCCESS;
49  }
```

The symbols `COLUMNS` and `ROWS` relate to the overall dimensions of the arrays. Recall that we have some "ghost" layers around the relevant data. Therefore, we have `COLUMNS = Nx1 + 3` and `ROWS = Nx2 + 3`. At line 16 code is inserted that fills `matC`, `matW` and `matS` according to the 2D Poisson stencil

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix},$$

i.e., the arrays `matC` consists of 4's only, and `matW` and `matS` consist of -1's and some zeros. At line 24 code is inserted that constructs the target solution `y`. At line 28 code is inserted that computes the corresponding right-hand `b` via $b = Ay$ for this `y`. In line 31 the initial guess `x` is set to zero, i.e. $x \equiv 0$. In lines 34-37 the solver is constructed, that is, the constructor of the `C++` class is invoked. During construction the preconditioner is build as well; for example, in case of the RRB-solver, the matrix $M = LDL^T$ is constructed. The preconditioner step is quite expensive but luckely it has to performed only once[1]. The time taken for construction of the solver and the preconditioner is measured with a CPU timer

[1]This is only true for a constant depth profile. The current `lin_wacu` works with constant depth profiles. In case the depth profile changes in time, the software must be rewritten in a form such that every time frame a new preconditioning matrix is computed. More details on this see:

(lines 34 and 37). In lines 41-43 the system $Ax = b$ is solved once with initial guess `x = 0.0`. The final solution, corresponding to a tolerance defined above (`tol = 1e-5`), is also stored in `x`. The approximate result is compared with the target solution `y`. The time taken to solve the system $Ax = b$ once is measured with a CPU timer (lines 41 and 43).

The `poisson` testing environment is especially designed to gather additional information on the `CUDA` solvers such as:

- Time needed to construct the preconditioner;
- Number of iterations needed to solve the 2D Poisson problem;
- Residual charts;
- Memory usage for larger problems (larger than the test problems, e.g., 3000 $\times$ 2000 nodes);
- speed up factor for these larger problems;
- Useful throughput, solver efficiency.

The `poisson` testing environment was actually used to build the `CUDA` solvers. During implementation of the `CUDA` RRB-solver we constantly checked whether the output of the `CUDA` RRB-solver corresponded one-to-one to the existing `C++` RRB-solver. However, using the plain 2D Poisson problem would not have worked as many coefficients are the same, therefore, during implementation, we temporarily added noise to the 4's and -1's to get all different numbers.

The environment was also build in a way such that after testing, the new solver can immediately be plugged-in in the `linwace` software. The solver in the `lin_wacu` is called in the exact same way as in the testing environment.

21.2.2 Plugging-in in the `lin_wacu` software

After extensive testing the new `CUDA` solvers in the `poisson` testing environment and taking care of all bugs, the `CUDA` solver is ready for the real job. The source code is copied to the `lin_wacu` folder and the `SolverFactory.cpp`-file and `Makefile` must be updated so that the `lin_wacu` code "knows" that the `CUDA` solvers exist.

The testing and timing with the `lin_wacu` code is done in various ways:

1. `perl` macros;
2. Profiling with `Valgrind`;
3. Visual check by actually plotting the waves to the screen.

Note that to get accurate timing results all output to the screen or files must be switched off. If one wants to have additional information the `poisson` testing environment can be used or one may run the code 1 time in the output (debug) modus to get the information but the timing results are ignored, and thereafter 1 time in the performance modus to get the correct timing but no output is generated.

perl macros

With the macros `testserieel.pl` and `testparallel1.pl` we can run a bunch of experiments. Therefore we select the solvers that we want to study and we list the test problems that we are interested in. The solvers are gathered after `my @solvers` and the test problems after `my @data`. The first listed test problem is treated first using the first listed solver, then the second listed solver, etc. When all solvers are applied to the same test problem, the next listed test problem is treated. This continues as many times as there are test problems. The simulations are started by entering

```
> perl testserieel.pl
```

or

```
> perl testparallel1.pl
```

The difference between `testserieel.pl` and `testparallel1.pl` is the way in which the rest of the code (all code except the solve part) is handled; in case of `testserieel.pl` all other computations are done sequentially on 1 core on the CPU, in case of `testparallel1.pl` all other computations are done in parallel on as many cores as the CPU has. Depending on which solver is used the solve part is done on the CPU (1 core only) or on the GPU (using all its cores (SPs)). The results are written to the screen and the file `lin_wacu.out`.

The following lines are an example of the output that we can expect.

```
Data                sol      br   ho   nodes   total  solver (pct)      Fext
plymouth-1500000    cudarrb 1250 1200  1500000 289.06  18.17 ( 6%)   7.36809
plymouth-1500000    rrb     1250 1200  1500000 767.90 491.41 (64%)   7.36810
portpresto-1500000  cudarrb 1200 1250  1500000 214.89  19.72 ( 9%)   3.40904
portpresto-1500000  rrb     1200 1250  1500000 657.07 461.95 (70%)   3.40904
ijssel-1500000      cudarrb 1500 1000  1500000 213.36  17.83 ( 8%)   4.10701
ijssel-1500000      rrb     1500 1000  1500000 770.92 533.71 (69%)   4.10701
```

We see that we have let the software run three different test problems using two solvers, the `CUDA` RRB-solver, and the `C++` RRB-solver. The 1st column (`Data`) lists the test problems, the 2nd column (`sol`) the solver, the 3th column (`br`) is the number of nodes in the x-direction, the 4th column (`ho`) the number of nodes in the y-direction, the 5th column (`total`) the total (execution) time, the 6th column (`solver`) the solver (execution) time, the 7th column (`pct`) the fraction: solver time / total time, and the 8th column (`Fext`) is a number to check whether the problem is correctly solved. If for different solvers the numbers are significantly different, one (or both) of the solvers failed to solve the test problem correctly.

In the above sample output we see already a) that the `CUDA` RRB-solver is much faster than the `C++` RRB-solver, and b) the test problems are (most likely) correctly solved (as both `Fext`-numbers are equal).

The total time consists of the time taken by all computations apart from the solve part $S\psi = b$ (thus incoming waves, time derivatives, Leapfrog, etcetera). The solver time is the time to solve the system $S\psi = b$.

Profiling with `Valgrind`

To get insight in which computations are expensive and which are not, the `lin_wacu` software can be run using a profiler like `Valgrind`. The generated `.callgrind`-files can thereafter be studied using `kcachegrind`. `Valgrind` can only be used to profile CPU tasks; unfortunately, GPU-tasks cannot be profiled. When the program contains both CPU and GPU tasks, like the `lin_wacu` code in combination with our new `CUDA` RRB-solver, the overall timing results are no longer correct. However, as the CPU tasks are timed correctly, the results are still valuable as we can sort out new CPU bottlenecks in the code.

In Figure 21.1 we have included a sample chart as it is displayed in `kcachegrind`. The chart that summarizes the profiling results for the Plymouth 1M nodes problem. The solver that is used in the experiment is the `C++` RRB-solver. The results are grouped by class.

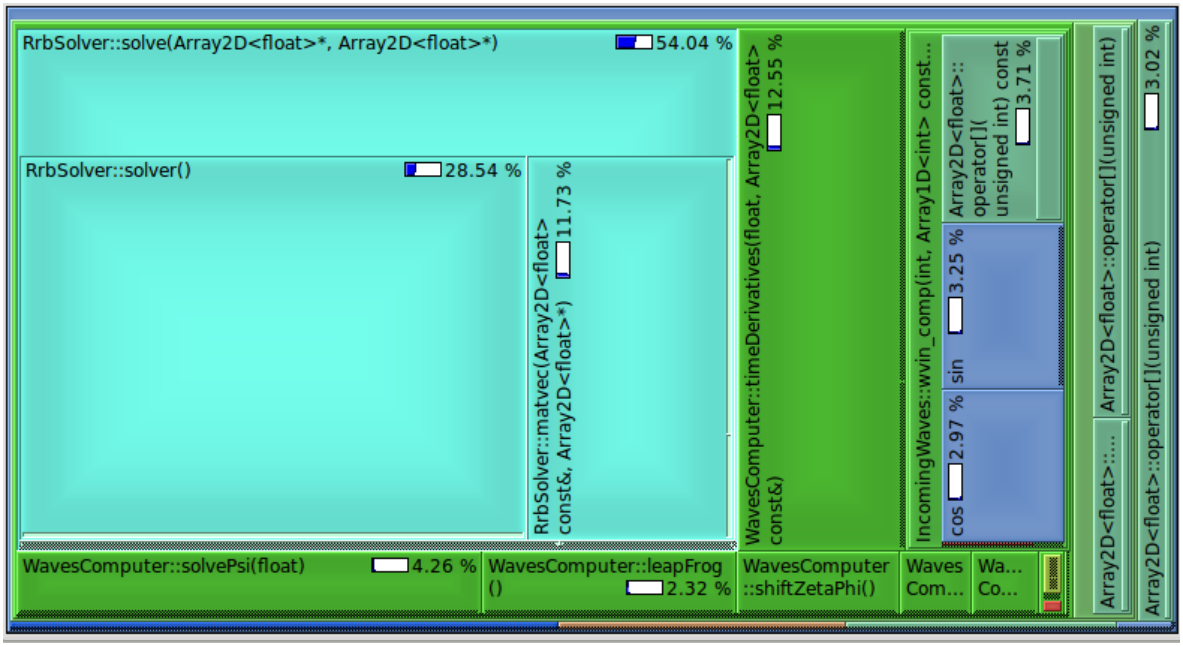


Figure 21.1: Sample output of `kcachegrind`. Results for the Plymouth 1M nodes test problem using the `C++` RRB-solver.

The chart must be interpreted as follows. The class `RRB-solver` (light-blue) takes 54.04% of the total time. Obviously, for this test problem the RRB-solver is the current bottleneck. The class routine `RrbSolver::solver()` solves the system $Mz = r$ for z, and it is the most expensive part of the RRB-solver; namely $(28.54/54.04) \times 100\% = 52.8\%$ of the solver time is consumed by that step. And this is as much as 28.54% of the total time.

The class `WavesComputer` (green) is the next most expensive class. Within this class we see that especially the class routines `timeDerivatives` is expensive (12.55% of total time). This implies that if the solver becomes much faster than computing the time derivatives becomes the new bottleneck.

We see that using the `Valgrind` profiler, we can get insight in what the bottlenecks are in our program. In Section 25.2.3 and Section 25.3 we elaborate upon the current and new bottlenecks in the `lin_wacu` software.

Visual check

To get a conclusive answer whether the simulation is performed correctly, and thus whether the solver is robust, we just do a visual check by letting the program display the wave field on the screen.

21.3 Profiling of the `CUDA` solvers

GPU code cannot be profiled with CPU profilers like `Valgrind`. To study the performance and efficiency of, and profile our `CUDA` solvers we proceed as follows:

1. Use the built-in performance monitor from the `CUDA` RRB-solver;
2. Use the NVIDIA profiler.

The `CUDA` IPDIAG-solver does not contain a performance monitor yet and can therefore only be studied with the NVIDIA profiler.

21.3.1 Built-in performance monitor

If enabled the `CUDA` RRB-solver writes output tothe following `.txt`-files:

1. `profiler_output.txt`: contains the optimal tiling strategy (number of threads in x- and y-direction) and the achieved useful write and read throughput in GB/s per kernel;
2. `runinfo_RRB.txt`: contains memory usage in MB, number of CG-iterations, norms (start residual) and the solver time in seconds.

21.3.2 NVIDIA profiler

We have mentioned the NVIDIA profiler already in Section 16.2.3. The profiler is a valuable tool for gaining insight in the efficiency and bottlenecks in our `CUDA` code. With the latest version in particular (the one that comes with CUDA 4.0) almost everything can be measured: kernel execution times, occupancy, number of registers, throughput, cache usage, etcetera. Typically this profiler can be found in: `../cuda/computeprof/bin`, and in Linux you can run it via `./computeprof`.

Chapter 22

Results — 2D Poisson test problem

In this chapter we discuss the results for the 2D Poisson test problem, see Section 3.1.

22.1 Specification of the problem

What was missing in Section 3.1 was a specification of the function $f = f(x, y)$ in system (3.1.1), that is, the system

$$\begin{aligned} -\Delta u &= f(x,y) \quad \text{on} \quad \Omega = (0,1) \times (0,1), \\ u(x,y) &= 0 \qquad\quad \text{on} \quad \partial\Omega. \end{aligned}$$

However, it is easier to pick a target solution $u = u(x, y)$ and compute the Laplacian Δu to find f, or, realizing that we are going to use the computer, we can use the corresponding discretization matrix A and compute $f(x_i, y_j) = Au(x_i, y_j)$ with $x_i = ih$, $y_j = jh$.

As we use the test problem primarily to get our new CUDA solvers working and see how they perform, we can pick u almost arbitrarily. In accordance with the test problem in [4] we have chosen to pick

$$u(x, y) = x(x - 1)y(y - 1)\,\mathrm{e}^{xy}.$$

Note that indeed $u = 0$ on $\partial\Omega$. For this u we readily compute

$$\frac{\partial^2 u}{\partial x^2} = \big((x^2 - x)y^4 + (-x^2 + 5x - 2)y^3 + (4 - 4x)y^2 - 2y\big)\,\mathrm{e}^{xy}$$

and by symmetry of u we can just interchange x and y above to find $\partial^2 u/\partial y^2$. Hence for f we consider

$$\begin{aligned} f(x,y) = \; -\big\{&\big((x^2 - x)y^4 + (-x^2 + 5x - 2)y^3 + (4 - 4x)y^2 - 2y\big) + \\ &\big((y^2 - y)x^4 + (-y^2 + 5y - 2)x^3 + (4 - 4y)x^2 - 2x\big)\big\}\,\mathrm{e}^{xy}. \end{aligned}$$

The solution u is a cone-shaped surface in space, see Figure 22.1.

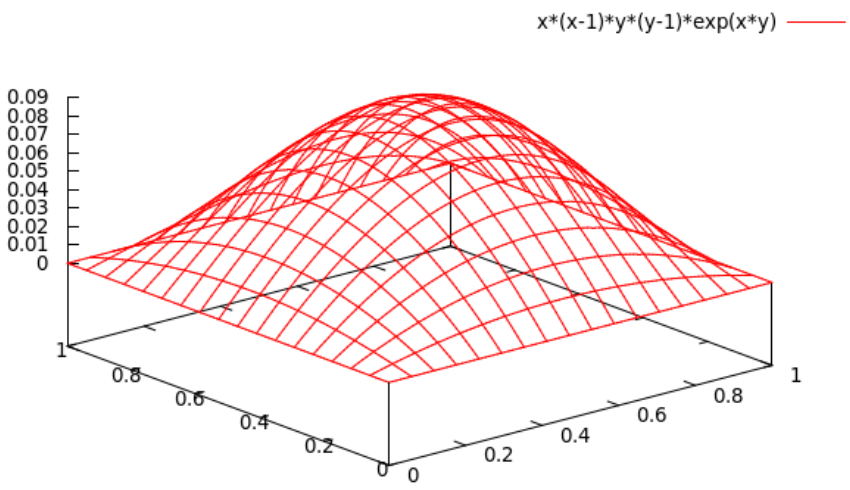


Figure 22.1: Target solution $u(x,y) = x(x-1)y(y-1)\,\mathrm{e}^{xy}$ approximated on a grid with $(N+1)^2$ equidistant grid cells. Plot generated with `gnuplot`.

22.2 Problem related results

22.2.1 Number of CG-iterations

Let us see how many CG-iterations are required to solve the 2D Poisson problem. The number of required CG-iterations depends on variables as:

- what preconditioner is used;
- the accuracy with which we want to solve the problem;
- the size of the problem.

Using MARIN's PCG solvers database and our new `CUDA` solvers we have evaluated the number of CG-iterations for the following PCG solvers:

Name	Preconditioner	Remarks
None	no preconditioner	
Cg4	Modified Incomplete Cholesky	uses Eisenstat's implementation
DIAG	Diagonal scaling	Also known as the "NopSolver"
IP	Incomplete Poisson	
IPDIAG	Incomplete Poisson + diagonal scaling	
RRB	Repeated Red-Black	

Further, we have chosen *psitol* =1e-5 for the accuracy. This means that the iterative process is stopped whenever the residual is $10^{10}\times$ smaller than the initial residual (see termination criterium, Section 18.1). For the problem size we have taken N = 16, 32, 64, 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, where N is the number of unknowns in both the x- and y-direction. The results are gathered in Figure 22.2.

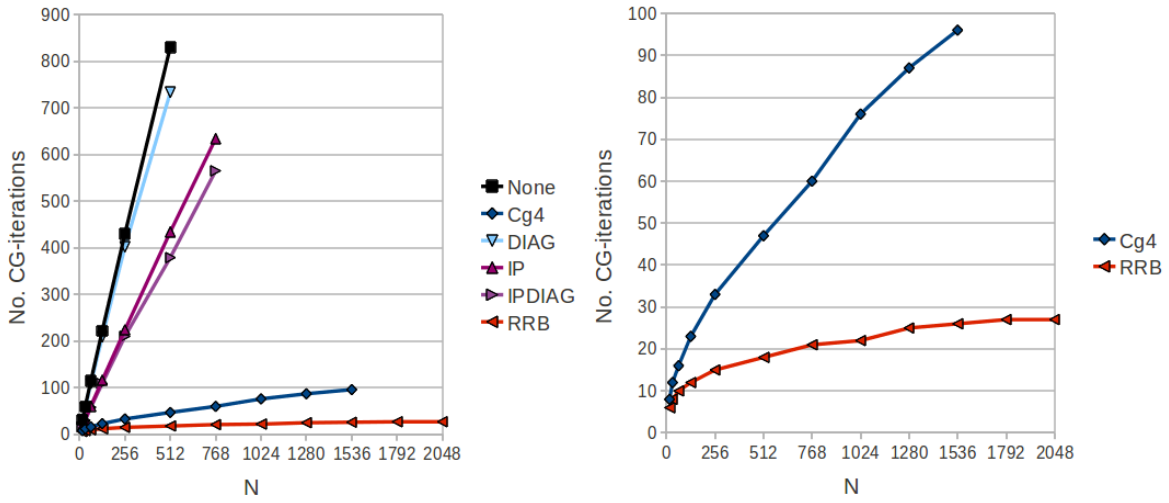


Figure 22.2: Required number of CG-iterations for the 2D Poisson problem for various grids and various PCG solvers. For the accuracy is taken *psitol* = 1e-5. *On the left:* All six solvers in one plot. *On the right:* Among the six evaluated, the Cg4 and RRB preconditioner are the most proficient preconditioners for the 2D Poisson problem.

The left part of Figure 22.2 shows that for the 2D Poisson problem simple preconditioners such as diagonal scaling (DIAG) and Incomplete Poisson (IP) are not very suitable. On contrast Modified Incomplete Cholesky (MIC) and Repeated Red-Black (RRB) do a very good job. On the right in Figure 22.2 we see the good scaling properties of the RRB-solver: the required number of CG-iterations grows very slowly with increasing grid size. As the VBM is a Poisson-type problem the RRB-solver should also perform well for the realistic test problems.

22.2.2 Convergence behaviour of the RRB-solver

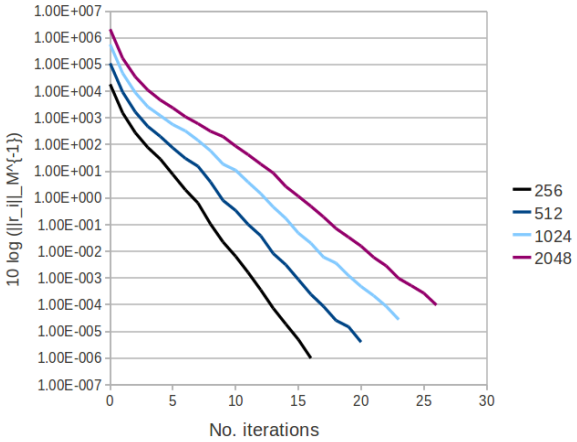


Figure 22.3: The convergence behaviour of the RRB-solver.

22.3 CUDA RRB-solver related results

22.3.1 Solver speed up

The most important result is how much faster is our CUDA implementation compared to the C++ implementation of the RRB-solver. Of course this strongly depends on what GPU and what CPU we use (how modern is it?), and on how efficient the solvers are implemented. Let us repeat the properties of the CUDA and C++ versions first.

The C++ implementation of the RRB-solver is fully sequential: only 1 core of the CPU is used. Considered that only 1 core is used the implementation is very efficient thanks to the usage of pointers for the solve part and the usage of a superior storage format: the class Array2D. The RRB-solver greatly benefits from the large amount of available (L3, L2, L1) cache. Note: the phase in which the preconditioner is constructed does not use pointers.

The CUDA implementation of the RRB-solver is fully parallel: most of the time all available cores (SPs) of the GPU are used. Thanks to the $r_1/r_2/b_1/b_2$ storage format most of the time maximal throughput is achieved. The coalesced memory transfers compensate for the little amount of cache.

We believe that — in some sense — both solvers are "equally efficiently" implemented. Further, to get an honest comparison we are going to compare a state-of-the-art CPU (Xeon W3520) with a state-of-the-art GPU (GeForce GTX 580). System II: GTX 580 houses this hardware.

In Table 22.1 we have gathered the timing results for the 2D Poisson problem with sizes 256, 512, 768, 1024, 1280, 1536, 1792 and 2048.

	Constructor (ms)			Solve part (ms)		
N	C++	CUDA	speed up	C++	CUDA	speed up
256	7.6	2.1	3.62×	7.7	2.1	3.67×
512	52.3	4.3	12.16×	81.5	5.3	15.38×
768	152.1	7.2	21.13×	261.7	10.8	24.23×
1024	347.4	11.3	30.74×	528.9	19.1	27.69×
1280	550.1	15.6	35.26×	965.4	32.0	30.17×
1536	834.5	20.3	41.11×	1425.7	43.5	32.77×
1792	1184.3	26.6	44.52×	1972.5	58.1	33.95×
2048	1625.5	34.0	47.81×	2767.1	76.3	36.27×

Table 22.1: Timing results for the 2D Poisson problem for various sizes and *psitol* = 1e-5. The C++ version ran on 1 core of a Xeon W3520 processor and the CUDA version ran on all cores of a GeForce GTX 580, see System II: GTX 580.

The left part of Table 22.1 corresponds with the constructing phase of the RRB-solver. This is the part in which memory is allocated and in which the preconditioning matrix $M = LDL^T$ is constructed. The right part of Table 22.1 corresponds with solving the system $S\psi = b$ once. For completeness we have also plotted the speed up numbers in a chart, see Figure 22.4.

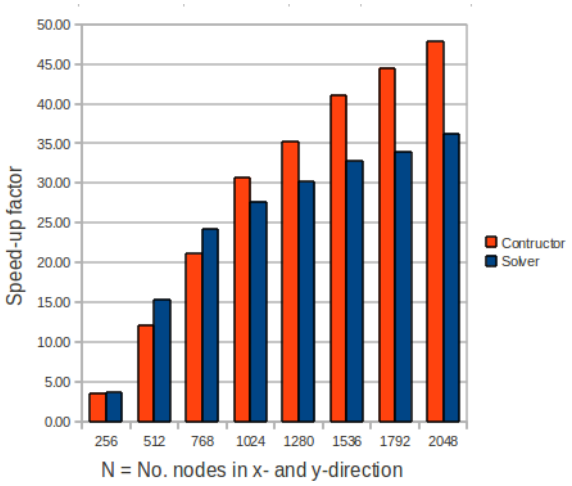


Figure 22.4: speed up.

We observe the following. The larger the problem size the larger the speed up number. This is not surprising as for smaller grids the CUDA solver suffers from overhead (idle threads, communication host-device), and, moreover, for larger grids the amount of CPU cache is no longer sufficient. Further, the speed up number for the constructing part is larger than for the solve part. This has to do with the fact that making the preconditioner in the C++ version is done without pointers, whereas the constructing part and the solver part of CUDA solver are equally efficiently implemented (both use the $r_1/r_2/b_1/b_2$ storage format).

22.3.2 Useful throughput

To see how well our CUDA RRB-solver has been implemented we look at how much useful throughput is achieved. In Table 22.2 we have listed the useful throughput per kernel in case of a 2048×2048 nodes test problem. The results are obtained using the built-in performance monitor and must be interpreted as follows.

The useful throughput consists of read and write throughput. The read throughput is computed via

$$\text{Useful read throughput (GB/s)} = \frac{\#\{\text{bytes read per thread}\} \cdot \#\{\text{threads required}\}}{10^9 \cdot \text{kernel time in s}}.$$

The number of threads required depends on the level, each thread basically handles 1 node. For most kernels the first $r_1/r_2/b_1/b_2$ level is used to compute the useful throughput.

Let us give an example: in case of a 2048×2048 grid the first $r_1/r_2/b_1/b_2$ grid consists of 4 parts (r_1, r_2, b_1 and b_2) of 1024×1024 nodes each. Further, for the kernel `kernel_prec2` 49 read operations are required per thread/node, see Listing 19.2. Hence in case of single-precision numbers (`floats` = 4 byte) and a kernel execution time of 1310 μs (see table), we get:

$$\text{Useful read throughput } \texttt{kernel\_prec2} = \frac{49 \cdot 4 \cdot (1024 \cdot 1024)}{10^9 \cdot 1310 \cdot 10^{-6}} = 156.89 \text{ GB/s}.$$

Likewise the write throughput is computed via

$$\text{Useful write throughput (GB/s)} = \frac{\#\{\text{bytes written per thread}\} \cdot \#\{\text{threads required}\}}{10^9 \cdot \text{kernel time in s}}.$$

Kernel	Opt x	Opt y	Time	Read (GB/s)	Write (GB/s)	Total (GB/s)
kernel_prec1	32	4	81	129.52	25.90	155.43
kernel_prec2	32	2	1310	156.89	57.63	214.53
kernel_prec3	32	2	183	166.28	51.60	217.88
kernel_prec4	32	4	1031	81.35	81.35	162.70
kernel_prec5	32	8	102	82.57	82.57	165.15
kernel_subs1	32	4	433	213.09	38.74	251.83
kernel_subs2	32	4	407	185.29	41.18	226.47
kernel_solv1	32	4	233	180.28	36.06	216.34
kernel_solv2	32	4	93	180.52	22.56	203.08
kernel_solv3	32	4	116	144.62	36.15	180.77
kernel_solv4	32	8	180	210.08	23.34	233.42
kernel_solv5	32	8	41	101.21	50.60	151.81
kernel_solv6	32	8	22	95.06	47.53	142.60
kernel_matv1	32	4	460	163.99	36.44	200.43
kernel_matv2	32	4	251	100.11	66.74	166.84
kernel_axpy	32	8	154	109.05	54.52	163.57
kernel_dotp	32	2	113	148.31	0.00	148.31
kernel_from1to4	32	4	210	79.81	79.81	159.62
kernel_from4to1	32	4	213	78.81	78.81	157.61
kernel_divR2in4	32	32	661	57.13	57.13	114.27
kernel_cmp4toR2	32	32	561	67.28	67.28	134.57

Table 22.2: Timing (in μs) and useful throughput per kernel of the CUDA RRB-solver on System II: GTX 580. The device-to-device bandwidth of the GTX is 141.35 GB/s according to NVIDIA's code example bandwidthTest. Thanks to the $r1/r2/b1/b2$ storage format the kernels attain optimal throughput (at the first level). Moreover, thanks to texture cache and L1 cache most kernels run at speeds (much) higher than the device theoretically allows (141.35 GB/s). Remark: the five last listed kernels have not been fully optimized.

Some of the kernels are invoked multiple times on different grids, because of the repeated $r_1/r_2/b_1/b_2$ levels. For example, the kernel kernel_prec2, which helps constructing the preconditioning matrix $M = LDL^T$, is launched as many times as there are grid levels (for a 2048×2048 grid there are 6 $r_1/r_2/b_1/b_2$ levels). For the coarsest grids (fewest nodes) overhead becomes more and more significant. The listed useful throughput is for the finest grid (highest level) on which the kernel runs.

Furthermore, "opt x" and "opt y" refer to the optimal tiling strategy per kernel. The optimal tiling is computed by the CUDA RRB-solver itself prior to all other computations. The solver ran a series of tests to determine the optimal settings.

For completeness we have also plotted the results in a chart, see Figure 22.5. From the table and the figure we observe that most kernels run on speeds higher than the device theoretically allows (for the GTX 580 this is 141.35 GB/s). This has to do with the fact that cache is used throughout the computations: many kernels use textures which are cached and the device also has some L1 cache.

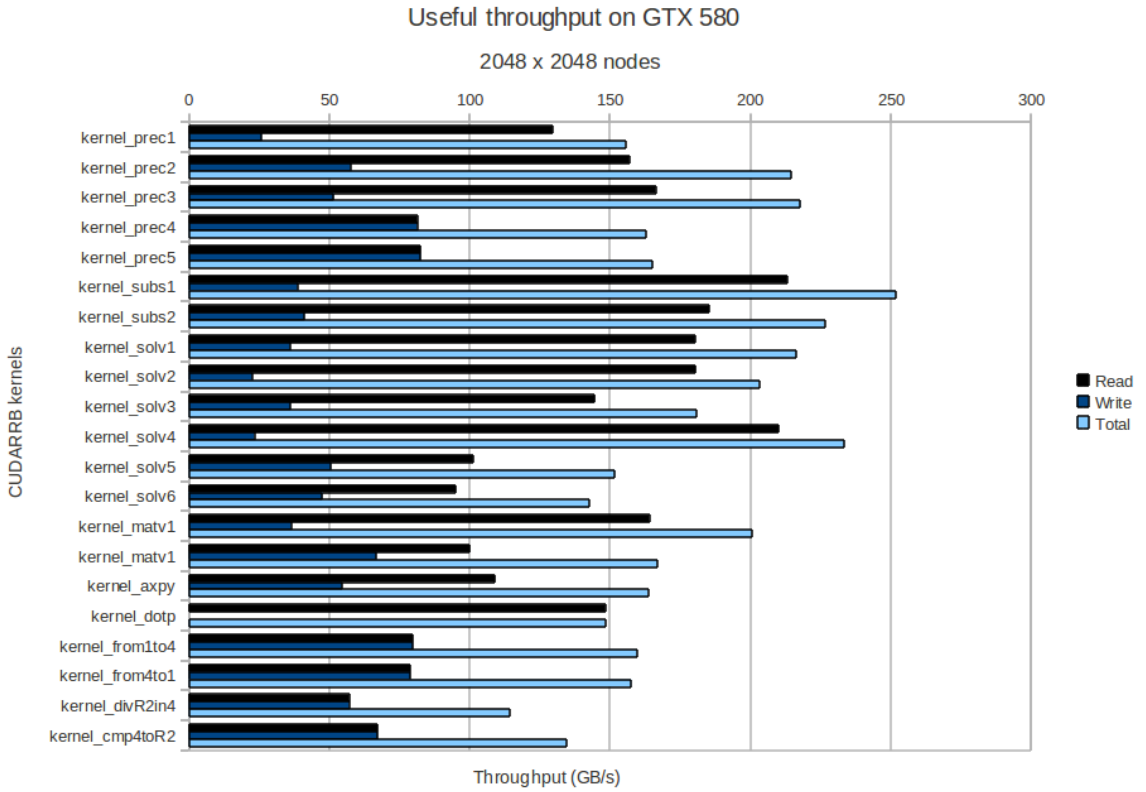


Figure 22.5: Useful throughput per kernel of the CUDA RRB-solver on System II; GTX 580.

22.3.3 Solver profile

It is interesting to see how the time is distributed over the kernels of the CUDA RRB-solver. By doing so one can see, for example, if there is one major time consuming part (bottleneck) in the code. How time is distibuted depends on the problem size and the number of iterations. Some of the parts are only performed once, e.g., memory transfers between the host and the device, whereas other parts are performed every iteration, e.g., solving the system $Mz = r$ for z. In Table 22.3 we have listed the actions that are performed by the CUDA RRB-solver as well as the corresponding execution times (in μs) for a system of 2048×2048 nodes.

At first instance, for the number of CG-iterations we have taken the actual required number of CG-iterations for the 2048×2048 nodes Poisson problem, which is 26 iterations, see Table 22.3 and corresponding pie chart in Figure 22.6. However, actually, we can substitute any number of iterations; if the underlying problem were, for example, VBM rather than 2D Poisson (on a 2048×2048 nodes grid), the number of iterations would be different, say 20, but the timing results would not be different. Hence the timing results can be used to construct the time profile (the pie chart) for *any* problem on a grid consisting of 2048×2048 nodes.

We see that for this many iterations (26), the actual computations take the most time: solving $Mz = r$ for z (38%), matrix-vector product $q = S_1 p$ (26%), vector-updates (16%) and dot products (10%). The memory transfers for this many iterations become relatively cheap (5%). However, if the system were solved in fewer CG-iterations, say 5, the memory transfers would become much more expensive: each about 15%.

		time in μs	
Memcpy HtoD	vector r	2982	
From 1 to 4	vector r	225	
Forward substitution	vector r	463	
Matrix-vector	$y = S_1 x$	732	
Vector-update	$r = r - y$	159	
Solver	$Mz = r$	1085	
Dot product	$\rho_{\text{new}} = \langle r, z \rangle$	139	
While-loop			
Vector-update	$p = z + \beta p$	160	$(26\times)$
Matrix-vector	$q = S_1 p$	732	$(26\times)$
Dot product	$\sigma = \langle p, q \rangle$	139	$(26\times)$
Vector-update	$x = x + \alpha p$	159	$(26\times)$
Vector-update	$r = r - \alpha q$	159	$(26\times)$
Solver	$Mz = r$	1085	$(26\times)$
Dot product	$\rho_{\text{new}} = \langle r, z \rangle$	139	$(26\times)$
End while-loop			
Backward substitution	vector x	424	
From 4 to 1	vector x	216	
Memcpy DtoH	vector x	3848	

Table 22.3: Typical distribution of time for the CUDA RRB-solver for 1 time step on System II: GTX 580 for a grid of 2048×2048 nodes.

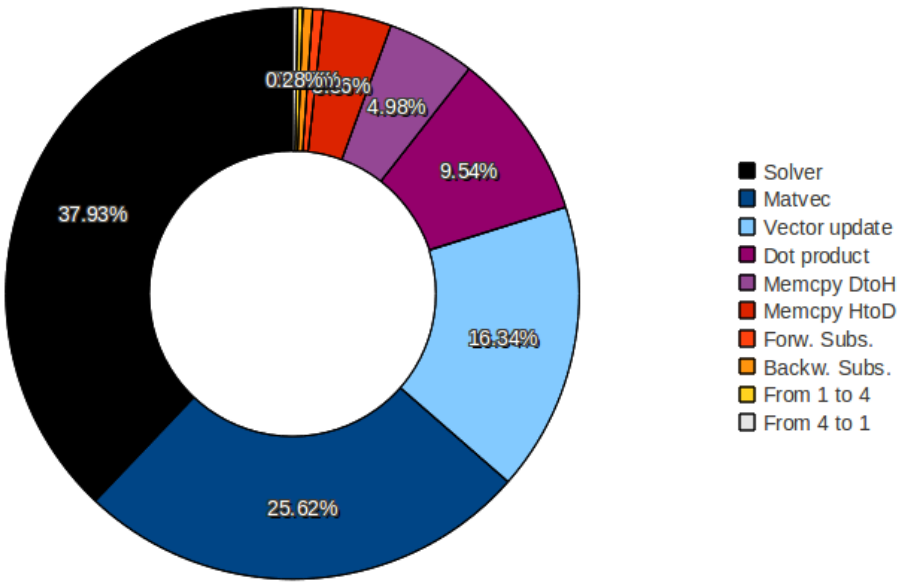


Figure 22.6: Typical distribution of time for the CUDA RRB-solver for 1 time step on System II: GTX 580 for a grid of 2048×2048 nodes. The plot is based on the assumption that 26 iterations were needed to solve the system $S\psi = b$. The fractions are CPU time fractions.

22.3.4 Amount of overhead / idle threads

Recall that the useful throughput of the kernels is being measured for the finest grids (many nodes). Most kernels operate on the red nodes of the first level. However, the kernels that are needed to solve the preconditioner step $Mz = r$ for z operate on *all* levels.

Of course the useful throughput drops with increasing grid level; as the grid becomes coarser (fewer nodes), the amount of overhead increases, and at some point there will be idle threads. To see the effect of the coarsening process during this solve step, we have used the NIVIDA profiler to profile the preconditioner step $Mz = r$ of our CUDA RRB-solver. We have done this for a problem with 2048×2048 nodes. The results are gathered in Table 22.4.

		solv::kernel_solv[x]							Total	
Level	$N_x = N_y$	1	2	3	4	5	6	final	time level	Perc.
1	1024	242.4							242.4	22.2%
2	512	54.2	102.0			45.0	25.8		227.1	20.8%
3	256	19.3	31.1			14.8	9.7		74.9	6.9%
4	128	7.0	14.2			7.0	3.0		31.2	2.9%
5	64	4.0	5.0			3.0	4.0		16.0	1.5%
6	32	4.0	6.0	4.0	4.0	3.0	3.0	30.0	54.0	4.9%
5	64			3.0	6.0				9.0	0.8%
4	128			12.4	6.0				18.4	1.7%
3	256			36.5	19.0				55.5	5.1%
2	512			125.1	53.0				178.1	16.3%
1	1024				185.3				185.3	17.0%
	Time kernel	330.9	158.3	181.0	273.3	72.9	45.5	30.0	1092.0	100.0%
	Percentage	30.3%	14.5%	16.6%	25.0%	6.7%	4.2%	2.8%	100.0%	

Table 22.4: Time distribution for solving $Mz = r$. Time in μs. In case of a grid of 2048×2048 nodes and a compute-block of 32×32 threads, the number of grid levels is 6.

The number of computations reduces with a factor 4 each time we go a level up (next coarser level). Let us see if this is the case in our CUDA RRB-solver. Consider for example `solv::kernel_solv2`. In Table 22.4 we see that on level 2 it takes 102.0 μs and on level 3 it takes 31.1 μs, and thus only a reduction of a factor $(102.0/31.1) \approx 3$. Even worse, from level 3 to level 4 we observe a reduction of only a factor $(31.1/14.2) \approx 2$ rather than a factor 4. For the other kernels we observe the same behaviour. The reason that we do not get a reduction of a factor 4 is the introduction of overhead.

However, this is not so bad as it seems. Let us explain why. We know that the useful throughput on the first levels is very high. In Section 22.3.2 we reported rates of 150-230 GB/s, which are even much higher than the global memory bandwidth of the GTX 580 (about 150 GB/s). We explained that this is possible thanks to the usage of cache. From Table 22.4 we see that the first two levels take 22.2 + 20.8 + 16.3 + 17.0 = 76.3% of the total compututions time. On the first two levels the useful throughput is optimal: 150-230 GB/s, hence the computations on the first levels cannot be performed much faster anymore. This means that 76.3% of a total of 1092 μs = 833 μs is always there. So, regardless how fast the higher grid levels can be processed, we can solve the system $Mz = r$ at best $(1092/833) \approx 1.2\times$ faster.

Chapter 23

Results — realistic test problems

23.1 Number of CG-iterations

In Table 23.1 we have gathered the average number of CG-iterations for all realistic test problems when using the RRB-solver and the IPDIAG-solver. The reason why we have also listed the results for the IPDIAG-solver is that, in contrast to the 2D Poisson problem, the IPDIAG-solver turns out to be also a good candidate for solving the realistic test problems IJssel, Plymouth and Port Presto.

	IJssel		Plymouth		Port Presto	
	RRB	IPDIAG	RRB	IPDIAG	RRB	IPDIAG
100k	5.814	12.595	5.804	11.906	5.924	18.713
200k	5.832	11.839	5.892	11.919	5.962	27.288
500k	5.836	12.591	5.964	19.866	5.986	33.299
1M	5.859	12.359	5.976	21.498	6.362	37.202
1.5M	5.766	12.569	5.984	20.330	6.921	37.915

Table 23.1: Average number of CG-iterations over 1000 time frames.

The table is generated using the `lin_wacu` software which ran a simulation of 1000 time frames. We see that in terms of convergence the RRB-solver is superior to the IPDIAG-solver; however, the larger number of CG-iterations required by the IPDIAG-solver is still manageable thanks to fast computing times per iteration, see Section 23.2.1.

The table only provides information on the average number of iterations but it does not give us any information on how the number of iterations changes in time. In Figure 23.1 it is shown for the RRB-solver how the number of CG-iterations required to solve the 1.5M nodes problems grows in time as the field gets filled with more and more waves. We see that, although the initial residual and RHS keeps growing, the number of required CG-iterations settles at 6 (IJssel and Plymouth) or 7 (Port Presto) for the rest of the simulation. In Figure 23.2 the same is done but this time for the IPDIAG-solver. We notice different behaviour: in the beginning the number of CG-iterations grow very rapidly to a maximum but after 200 time frames the number of CG-iterations also settles to quite constant values (30 for IJssel, 20 for Plymouth and 36 for Port Presto).

All the other test problems (the smaller ones) show the same behaviour: in the beginning just a few CG-iterations are required since there are hardly any waves, and after some time

the number of CG-iterations settles at a certain number.

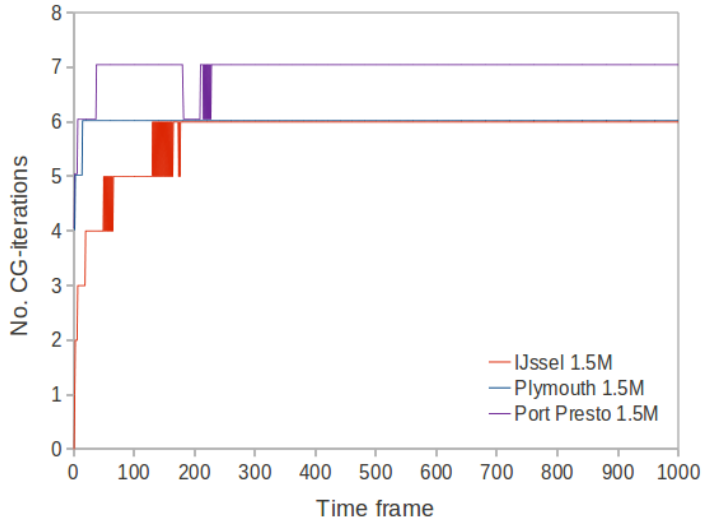


Figure 23.1: Number of CG-iterations for the first 1000 time frames of the 1.5M nodes problems when using the RRB-solver.

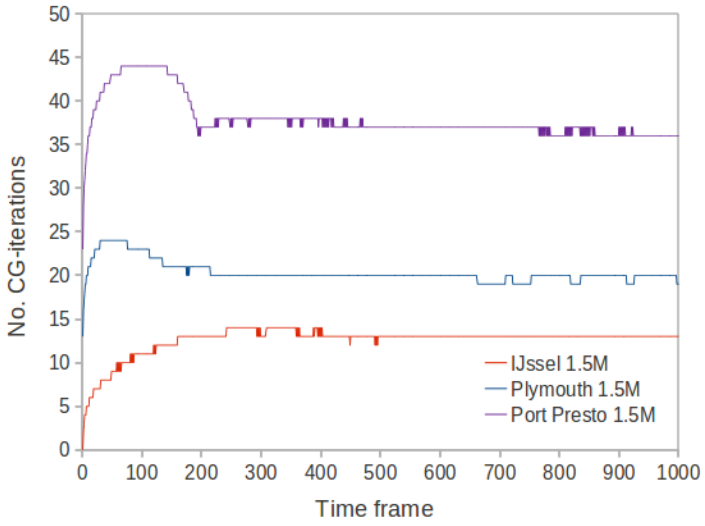


Figure 23.2: Number of CG-iterations for the first 1000 time frames of the 1.5M nodes test problems when using the IPDIAG-solver.

23.2 Timing

23.2.1 Solver time

More interesting than the number of CG-iterations are, for us, the corresponding computing times. In Tables 23.2 to 23.4 we have gathered all results. We have run the simulations on both System I and System II. On each system we timed both the `C++` and `CUDA` version of

the RRB-solver, as well as the CUDA IPDIAG-solver (we do not have a C++ implementation for this solver).

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	49.229	4.310	6.518	47.321	2.148	2.193
200k	94.343	5.902	8.163	83.782	2.832	3.296
500k	191.210	8.863	12.895	130.414	4.852	6.312
1M	390.995	13.420	20.918	266.269	7.749	11.001
1.5M	534.363	17.567	29.874	347.331	10.709	15.711

Table 23.2: Average solver time (in ms) over 1000 time frames for the IJssel test problem.

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	15.988	4.234	6.298	11.450	1.894	2.110
200k	41.022	5.472	7.850	57.661	2.628	3.072
500k	141.007	8.704	18.372	71.493	4.728	8.941
1M	332.706	13.992	30.463	178.297	7.865	16.915
1.5M	490.332	18.523	40.824	298.446	10.618	23.159

Table 23.3: Average solver time (in ms) over 1000 time frames for the Plymouth test problem.

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	14.168	4.240	8.846	10.264	1.886	2.842
200k	40.132	5.524	5.962	22.345	2.684	5.969
500k	127.789	8.648	28.138	64.601	4.732	13.832
1M	272.533	14.007	48.667	148.022	7.964	27.288
1.5M	462.946	20.010	71.370	219.051	11.619	39.737

Table 23.4: Average solver time (in ms) over 1000 time frames for the Port Presto test problem.

We observe that, although the required number of CG-iterations for the IPDIAG-solver is much larger than for the RRB-solver, the corresponding computing times are much closer. In particular for the IJssel test problem, which requires not too many iterations, the CUDA IPDIAG-solver can almost match the performance of the CUDA RRB-solver. For the Plymouth and Port Presto the CUDA RRB-solver is really superior, thanks to the fast convergence.

23.2.2 Additional time

The additional time is the time needed to do all computations apart from the solve step ($S\psi = b$) within a complete frame: incoming waves, time derivatives, Leapfrog, etcetera. With the `perl` macro `testserieel.pl`, see Section 21.2.2 we can easily measure both the solver time and total time. We run all test problems with the C++ and CUDA RRB-solver

and the CUDA IPDIAG-solver on System I and System II. The raw data can be found in Appendix C.

By substracting the solver time from the total time we find the additional time. However, to get more accurate results we take the average over the 3 additional times per test problem. The results are gathered in Table 23.5.

	#nodes	System I: GTX 285 (C2D E6850)	System II: GTX 580 (Xeon W3520)
IJssel	100k	13.60	9.32
	200k	26.24	17.90
	500k	92.14	42.42
	1M	126.91	99.46
	1.5M	195.80	131.08
Plymouth	100k	25.10	21.45
	200k	42.62	34.83
	500k	65.08	70.18
	1M	176.70	134.88
	1.5M	274.58	208.50
Port Presto	100k	13.33	9.11
	200k	25.90	17.49
	500k	63.10	41.78
	1M	131.39	89.62
	1.5M	195.31	131.01

Table 23.5: Average additional time (in ms).

We see that the IJssel and the Port Presto test problem require about equal additional time. The Plymouth test problem requires apparently some extra computions. These extra computations come from incoming waves with a Jonswap spectrum.

23.2.3 Total time

Although the total time is directly measured by using `testserieel.pl` it is slightly more accurate to use the average times from earlier tables. The total time is found by the sum of the average solver time and average additional time. By doing so we get Tables 23.6 to 23.8.

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	62.83	17.91	20.12	56.64	11.47	11.51
200k	120.58	32.14	34.40	101.68	20.73	21.20
500k	283.35	101.00	105.04	172.83	47.27	48.73
1M	517.91	140.33	147.83	365.73	107.21	110.46
1.5M	730.16	213.37	225.67	478.41	141.79	146.79

Table 23.6: Average total time (in ms) over 1000 time frames for the IJssel test problem.

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	41.09	29.33	31.40	32.90	23.34	23.56
200k	83.64	48.09	50.47	92.49	37.46	37.90
500k	206.09	73.78	83.45	141.67	74.91	79.12
1M	509.41	190.69	207.16	313.18	142.75	151.80
1.5M	764.91	293.10	315.40	506.95	219.12	231.66

Table 23.7: Average total time (in ms) over 1000 time frames for the Plymouth test problem.

	System I: GTX 285			System II: GTX 580		
	C++ RRB	CUDA RRB	CUDA IPDIAG	C++ RRB	CUDA RRB	CUDA IPDIAG
100k	27.50	17.57	22.18	19.37	11.00	11.95
200k	66.03	31.42	31.86	39.84	20.17	23.46
500k	190.89	71.75	91.24	106.38	46.51	55.61
1M	403.92	145.40	180.06	237.64	97.58	116.91
1.5M	658.26	215.32	266.68	350.06	142.63	170.75

Table 23.8: Average total time (in ms) over 1000 time frames for the Port Presto test problem.

We immediately see that, although the CUDA solvers are much faster than the C++ RRB-solver, the total times are much closer to each other. The see how much closer let us compute the corresponding speed up numbers.

23.3 Speed up numbers

With the tables from the previous sections we can compute speed up numbers. Although one can do that for all test problems, in the end we are only really interested for the largest problems. Therefore, we shall only compute the corresponding speed up numbers for the 1.5M nodes problems.

The results are shown in Figures 23.3 and 23.4. Figure 23.3 shows the (solver) speed up numbers, whereas Figure 23.4 shows the total speed up numbers for all three 1.5M nodes test problems.

We observe the following. The CUDA RRB-solver is about 20-30× as fast as its C++ counterpart, and the CUDA IPDIAG-solver is about 5-20× as fast as the C++ RRB-solver. How much faster strongly depends on the underlying test problem. Although the CUDA yield great speed up factors, the total speed up is disappointing. Almost regardless of what CUDA solver is used we get a speed up of only 2-3×.

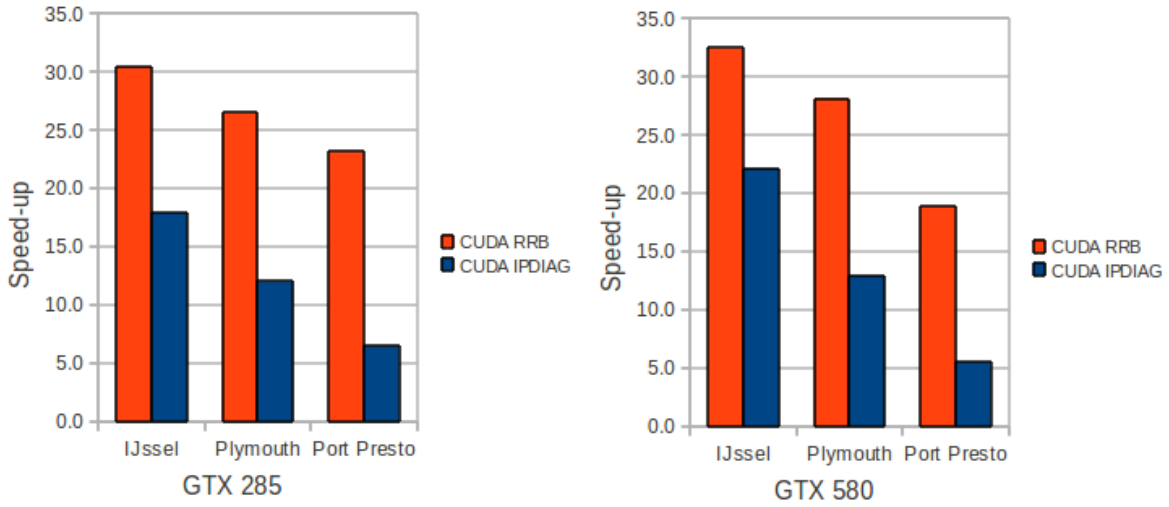


Figure 23.3: speed up numbers for the 1.5M nodes test problems. *On the left:* The speed up of the CUDA RRB-solver and CUDA IPDIAG-solver compared to the C++ RRB-solver on System I: GTX 285 versus 1 core of an Intel C2D E6850 @ 3.0 GHz. *On the right:* The same speed up numbers but then for System II: GTX 580 versus 1 core of an Intel Xeon W3520 @ 2.67 GHz.

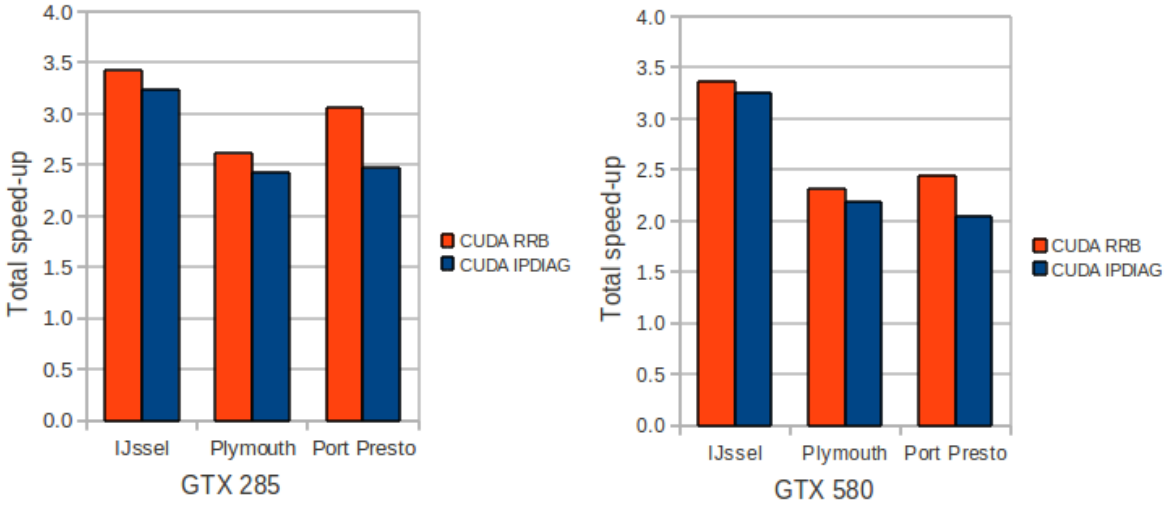


Figure 23.4: Total speed up numbers for the 1.5M nodes test problems. *On the left:* The total speed up of the CUDA RRB-solver and CUDA IPDIAG-solver compared to the C++ RRB-solver on System I: GTX 285 versus 1 core of an Intel C2D E6850 @ 3.0 GHz. *On the right:* The same total speed up numbers but then for System II: GTX 580 versus 1 core of an Intel Xeon W3520 @ 2.67 GHz.

Chapter 24

Screenshots from a simulation

In this chapter we just present some nice screenshots resulting from a simulation in the `lin_wacu` software. The simulation that we have run to get the pictures is the Plymouth 100k test problem from Section 3.3.2 for 3000 time frames. A time step of $\Delta t = 0.05\,s$ is used, so the entire simulation would take $150\,s$ in real-time. The domain is $2000\,m \times 1250\,m$. The corresponding grid a mesh size of $5\,m$ and thus consists of 400×250 nodes. In Figure 24.1 the path is indicated that the ship follows.

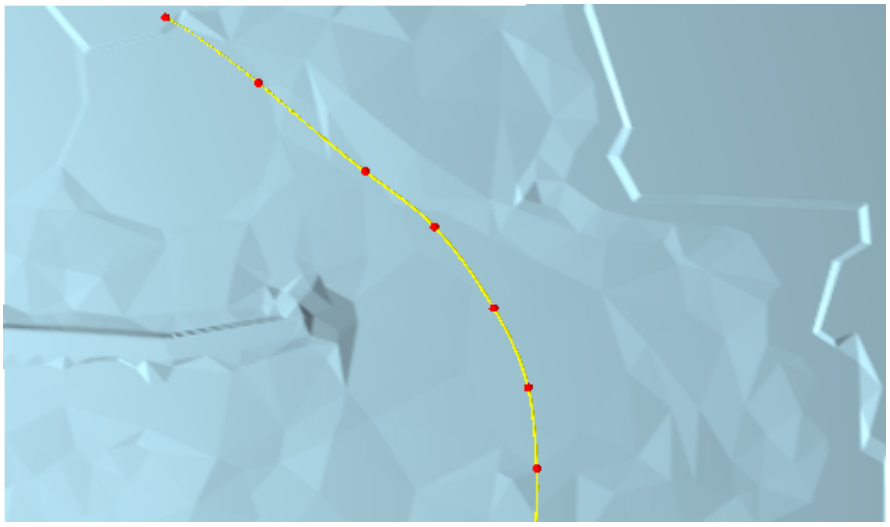


Figure 24.1: The path that the ship follows in the Plymouth 100k test problem (top view).

In the next set of figures we show how the wave field changes in time. We have taken screenshots for time frames 500, 1000, 1500, 2000, 2500 and 3000.

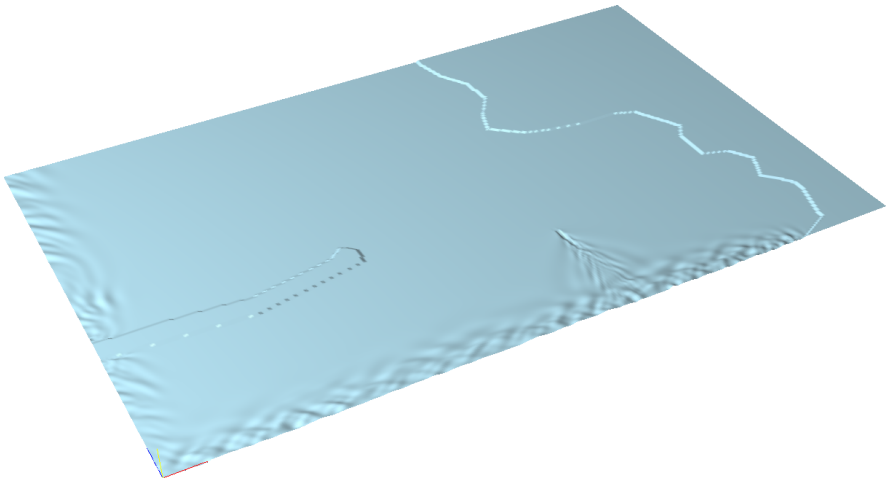


Figure 24.2: Plymouth 100k for time frame 500 (3D view).

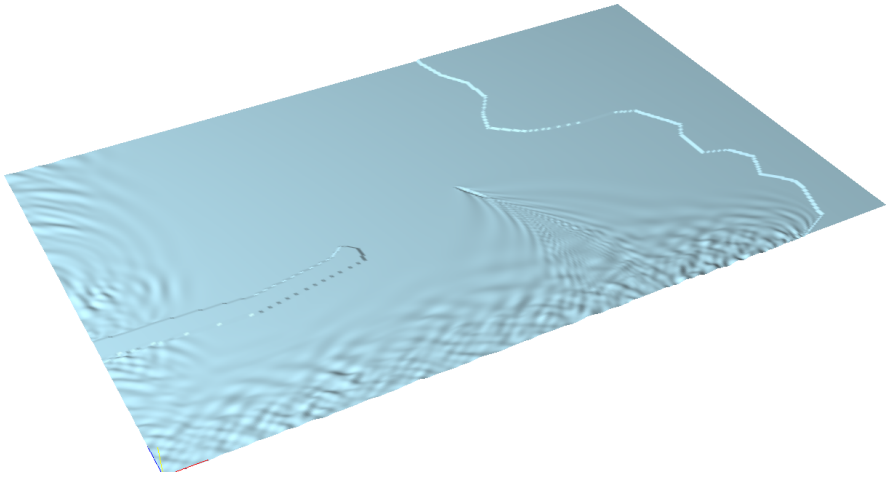


Figure 24.3: Plymouth 100k for time frame 1000 (3D view).

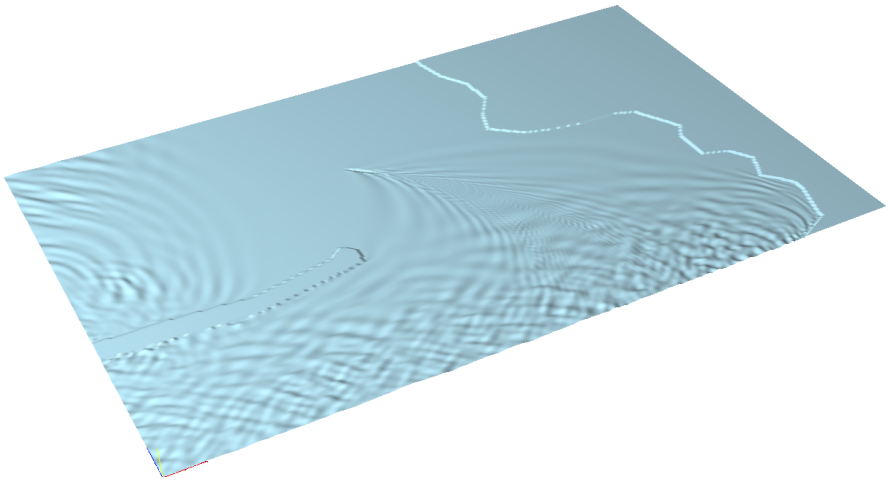


Figure 24.4: Plymouth 100k for time frame 1500 (3D view).

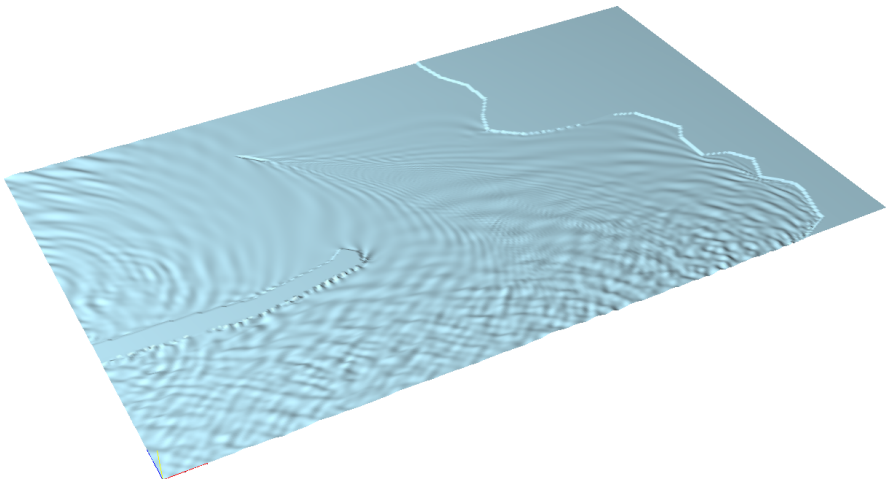


Figure 24.5: Plymouth 100k for time frame 2000 (3D view).

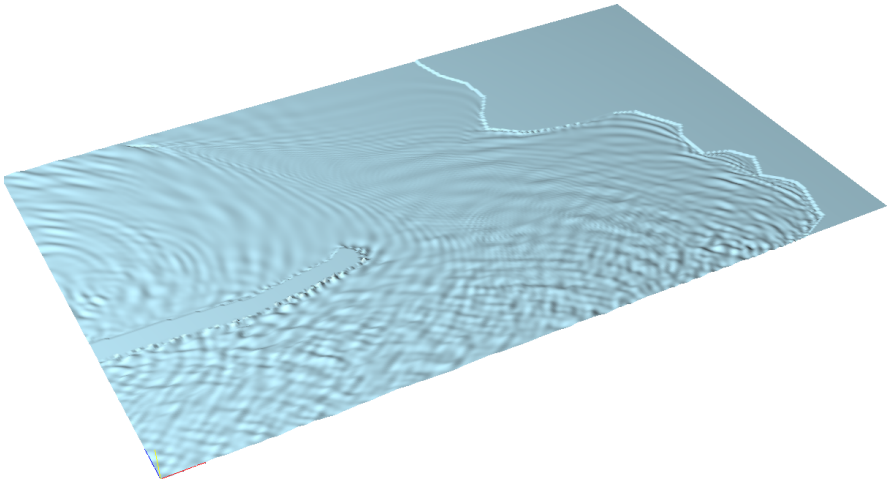


Figure 24.6: Plymouth 100k for time frame 2500 (3D view).

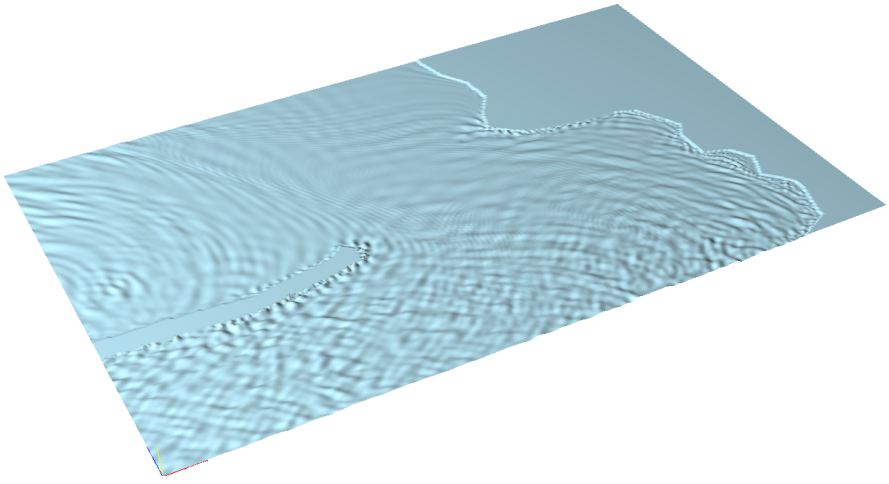
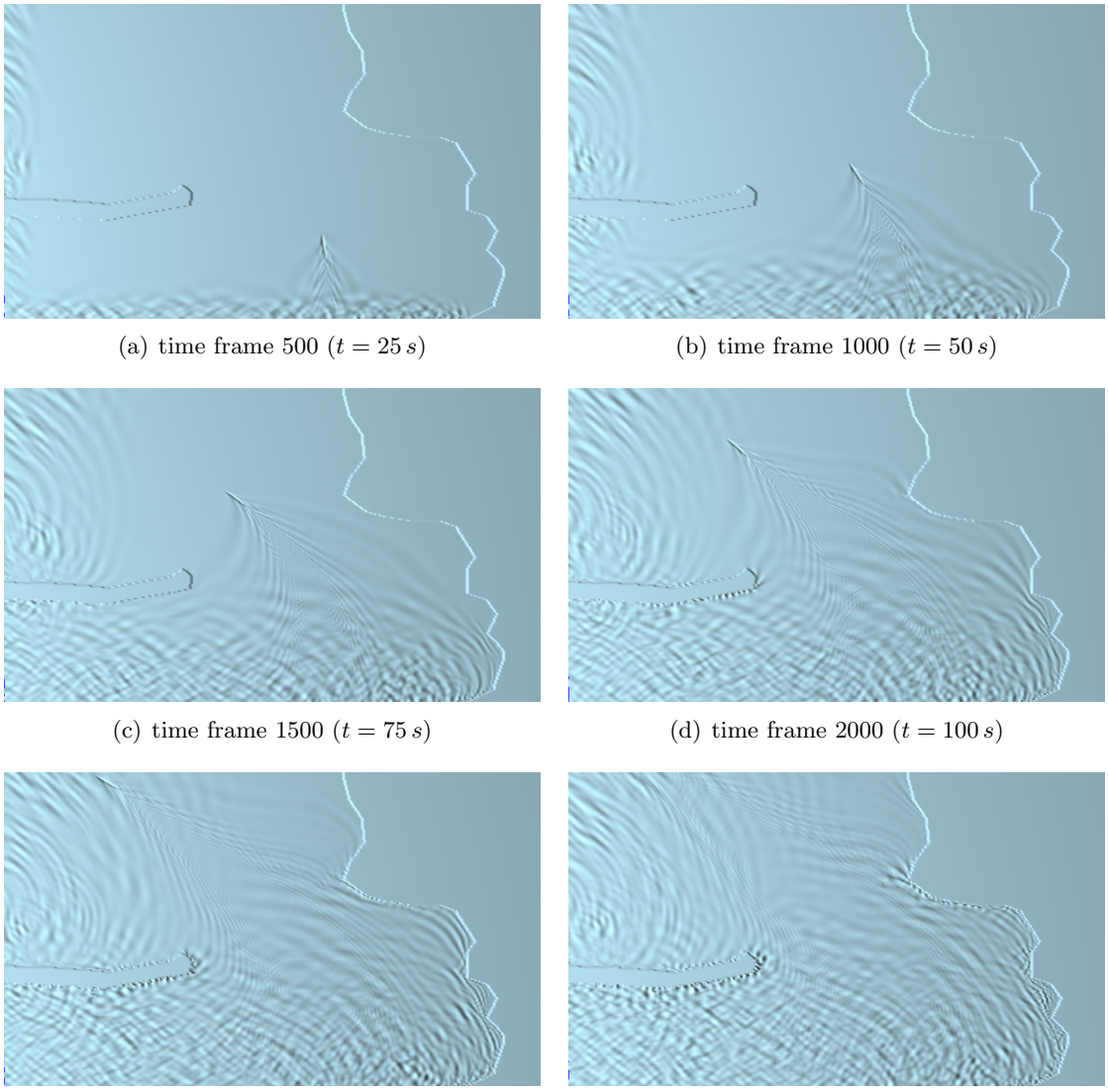


Figure 24.7: Plymouth 100k for time frame 3000 (3D view).

For clarity we have gathered all impressions in one figure and this time the waves are observed from above, see Figure 24.8.



(a) time frame 500 ($t = 25\,s$)

(b) time frame 1000 ($t = 50\,s$)

(c) time frame 1500 ($t = 75\,s$)

(d) time frame 2000 ($t = 100\,s$)

(e) time frame 2500 ($t = 125\,s$)

(f) time frame 3000 ($t = 150\,s$)

Figure 24.8: Wave pattern of the realistic Plymouth 100k nodes test problem for time frames 500, 1000, 1500, 200, 2500 and 3000 (top view).

In the figures it is seen how the ship navigates along its path and how waves come in from the west and south boundary (incoming waves). The wave pattern seems realistic, as far as we can judge that.

Chapter 25

Further analysis and discussion

25.1 Parallel host code with OpenMP

In Section 23.3 we saw how the code around the solver leads to poor speed up results. To remedy this the code around the solver, i.e., code that remains on the CPU, should also be processed in parallel. This is already possible; the code around the solver has been parallelized with OpenMP. By using the perl macro testparallel1.pl rather than testserieel.pl the host code is processed in parallel. Let us find out how much we gain when we run the host code not on 1 core but on 4 cores on a Xeon W3520 processor.

We have run the test problems again for 1000 time frames with the CUDA RRB-solver on System II, but this time with OpenMP "enabled". The timing results are gathered in Table 25.2.

	#nodes	total time (ms)	solver time (ms)	additional (ms)
IJssel	100k	5.70	2.12	3.58
	200k	8.96	2.85	6.11
	500k	18.86	4.76	14.10
	1M	37.44	7.63	29.81
	1.5M	52.21	10.62	41.59
Plymouth	100k	13.13	1.84	11.29
	200k	18.66	2.60	16.60
	500k	35.41	4.70	30.71
	1M	67.21	7.83	59.38
	1.5M	100.44	10.49	89.95
Port Presto	100k	5.18	1.81	3.37
	200k	8.67	2.62	6.05
	500k	18.32	4.72	13.60
	1M	38.14	7.90	30.24
	1.5M	54.24	11.05	43.19

Table 25.1: Timing results in case the host code is parallelized over 4 cores of a Xeon W3520 processor with OpenMP.

If we now compare the additional times from Table 23.5 with those in Table 25.2 we can make the following table.

	#nodes	additional time (ms)		
		on 1 core	on 4 cores	host speed up
IJssel	100k	9.32	3.58	2.6×
	200k	17.90	6.11	2.9×
	500k	42.42	14.10	3.0×
	1M	99.46	29.81	3.3×
	1.5M	131.08	41.59	3.2×
Plymouth	100k	21.45	11.29	1.9×
	200k	34.83	16.60	2.1×
	500k	70.18	30.71	2.3×
	1M	134.88	59.38	2.3×
	1.5M	208.50	89.95	2.3×
Port Presto	100k	9.11	3.37	2.7×
	200k	17.49	6.05	2.9×
	500k	41.78	13.60	3.1×
	1M	89.62	30.24	3.0×
	1.5M	131.01	43.19	3.0×

Table 25.2: speed up factors for the host code in case the host code runs sequentially on 1 core or parallely on 4 cores of a Xeon W3520 processor with `OpenMP`.

We see that running the host code parallely pays off. In case of no incoming waves (IJssel, Port Presto) we observe a speed up of 3×, and in case of incoming Jonswap waves we observe a speed up of a factor 2. A speed up of 3× for 4 cores is a good result. Let us now see how the total speed up charts from Section 23.3 change because of this reduction in additional time. The new speed up numbers are given in Table 25.1

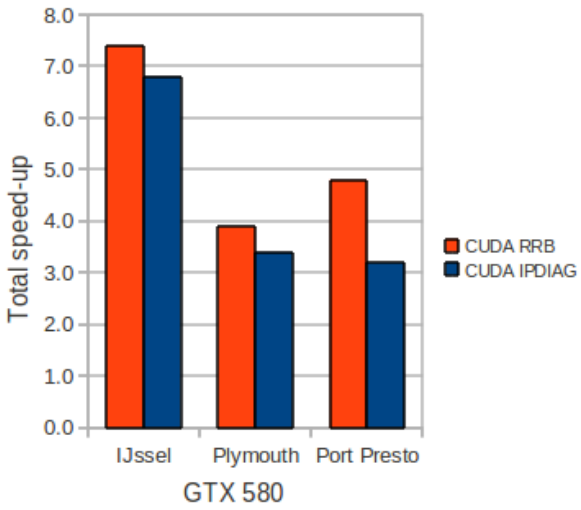


Figure 25.1: Total speed up numbers for the 1.5M nodes test problems for the `CUDA` RRB-solver and `CUDA` IPDIAG-solver compared to the `C++` RRB-solver on System II: GTX 580. The host code ran on 4 cores of the Xeon W3520 of System II.

We see that if we run the code around the solver parallely then we get a speed up around

3-7× depending on the particular test problem if we use the CUDA RRB-solver or the CUDA IPDIAG-solver instead of the C++ RRB-solver.

25.2 Profile of the lin_wacu code

To get better insight in how much the overall effect is of speeding-up particular parts of the code, we are going to look at how time is distributed among the computations in the lin_wacu software. To do so we have to use a test problem in which all possible computations are actually performed. The Plymouth test problem is a good candidate as this test problem contains incoming waves.

To get the profile of the lin_wacu software we may use the Valgrind profiler, see Section 21.2.2. However, we have to be careful with that as timing measurements may be incorrect. For example, CUDA code is not being measured correctly with the Valgrind profiler. Therefore, we have chosen to do all timing results by measuring the wall-clock time, see Section 16.2.1, rather than using the Valgrind profiler. However, the Valgrind profiler still can be used to gain additional information such as the time profile of a specific subroutine.

25.2.1 Overview of computations

Within the C++ class wavesComputer the wave field is computed. The following subroutines in the routine WavesComputer::compute() are called in the indication order.

Routine:	Function:	Remarks:
shiftZetaPhi()	Shifts $\mathbf{q} = [\zeta, \varphi]^T$ (t_{n+1}, t_n, t_{n-1})	
m_pIncomingWaves->compute()	Computes incoming waves	Class object
waveBreaking()	–	
boundaryZetaPhi()	–	
solvePsi()	–	
adjustPsi()	–	
m_pSolver->solve()	Solves the system $S\psi = b$	Class object
shiftPsi()	Shifts ψ (t_{n+1}, t_n, t_{n-1})	
boundaryPsi()	–	
timeDerivatives()	Computes $\frac{\partial \zeta}{\partial t}$ and $\frac{\partial \varphi}{\partial t}$	
weaklyReflective()	–	
smoothZetaPhi()	–	
leapFrog()	Computes $\mathbf{q}^{n+1}$ with $\mathbf{q}^n$ and $\mathbf{q}^{n-1}$	$\mathbf{q} = [\zeta, \varphi]^T$
m_pLPFil->filter(m_zeta)	–	Class object
m_pLpFil->filter(m_phi)	–	Class object
"maxwaves"	–	Inlined code

Table 25.3: List of subroutines that are called within the routine WavesComputer::compute(). A "–" means that we do not know the function of the specific routine yet.

25.2.2 Wall-clock timing results

With wall-clock timing we find the following profiles for the Plymouth 1.5M test nodes problems when the C++ and CUDA RRB-solver are applied (CPU: C2D E6850 @ 3GHz, GPU: GeForce GTX 580), see Table 25.4. Notice that the machine that is used is neither System I or System II. The machine that is used here is System I but with the GTX 285 replaced by the GTX 580, because System II was already dismantled before we could to these measurements.

		100k	200k	500k	1M	1.5M
shiftZetaPhi()		3.2	6.4	16.1	32.0	48.8
m_pIncomingWaves->compute()		12.2	17.3	29.4	43.6	52.4
waveBreaking()		0	0	0	0	0
boundaryZetaPhi()		0	0	0	0.1	0.1
solvePsi()		1.7	3.0	6.6	16.4	20.1
adjustPsi()		0.3	0.6	1.3	3.2	4.3
m_pSolver->solve()	CUDA	1.8	2.6	4.7	7.8	10.6
	(C++)	(14.3)	(29.9)	(112.3)	(266.3)	(379.5)
shiftPsi()		0.4	0.8	1.3	4.3	6.5
boundaryPsi()		0	0	0	0	0
timeDerivatives()		6.4	11.7	27.2	63.0	85.8
weaklyReflective()		0.1	0.2	0.2	0.4	0.4
smoothZetaPhi()		0	0	0	0	0
leapFrog()		2.2	4.2	9.9	22.2	30.1
m_pLPFil->filter(m zeta)		0	0	0	0	0
m_pLpFil->filter(m phi)		0	0	0	0	0
"maxwaves"		0.2	0.3	1.1	2.0	2.9
Total	CUDA	28.5	47.1	97.8	195.0	262.0
	(C++)	(41.0)	(74.4)	(205.4)	(453.5)	(630.9)

Table 25.4: Time profile of the routine WavesComputer::compute().

25.2.3 Time profile charts

In Figures 25.2 and 25.3 we have plotted the profile of the most time-consuming routines. The routines which take no or a little time are combined in "rest". In Figure 25.2 the old situation is shown, that is, the situation in which $S\psi = b$ is solved with the C++ RRB-solver.

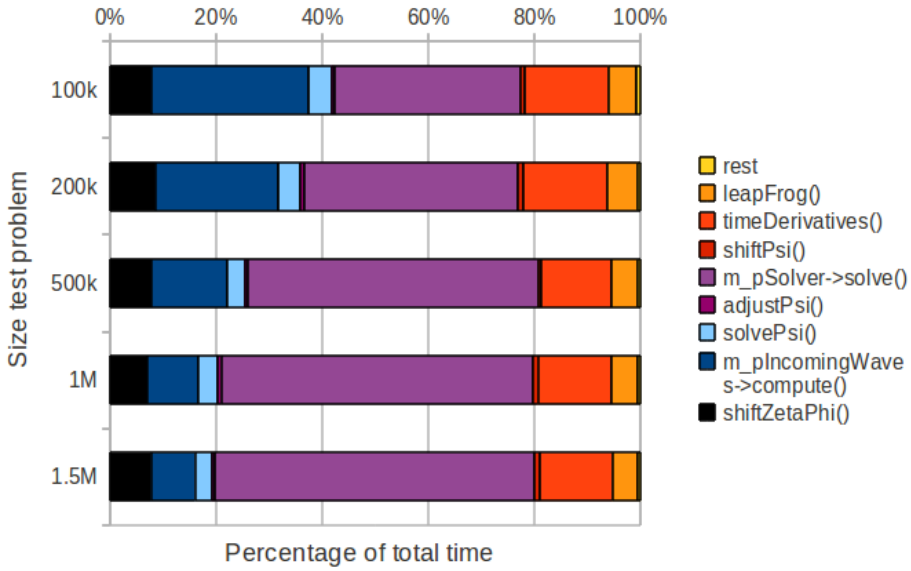


Figure 25.2: Time profile for `WavesComputer::compute()` for 5 test problems using the C++ RRB-solver (CPU, sequential, 1 core).

In the next figure we see the great impact of the CUDA RRB-solver on the time profile. We see that the CUDA RRB-solver is now among the cheapest routines.

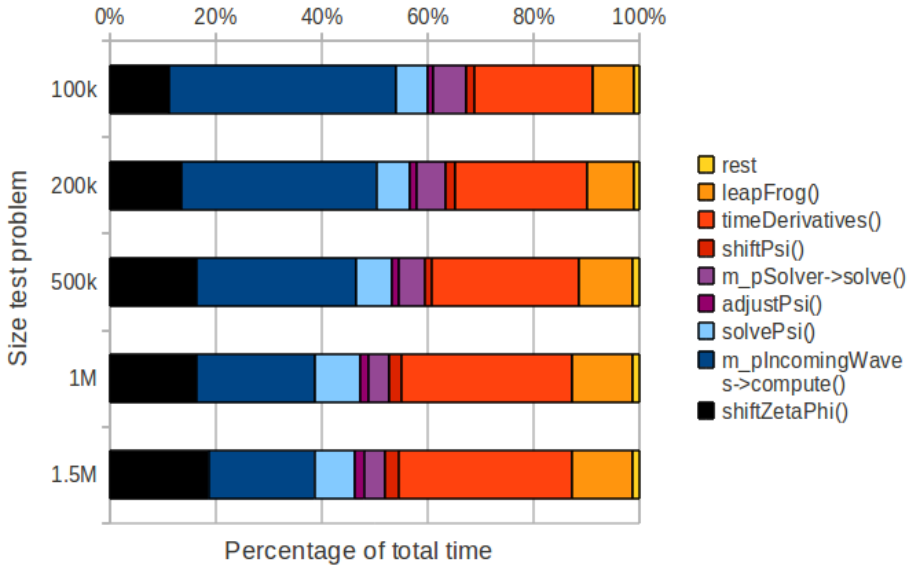


Figure 25.3: Time profile for `WavesComputer::compute()` for 5 test problems using new CUDA RRB-solver (GPU, parallel, all available SPs/cores).

Further we observe that the time profile changes with increasing problem size: incomping waves becomes relatively cheaper and time derivatives becomes more expensive in terms of computation time. For the two largest problems (1M and 1.5M) the profiles are more or less the same which may indicate the existence of some kind of "equilibrium".

25.3 New bottlenecks in the code

In the previous section it was shown that the solver is no longer the bottleneck in the `lin_wacu` code. From the time profile chart, see Figure 25.3, we clearly see what the new bottlenecks are. In indicating order the following routines are the most important new bottlenecks:

1. `timeDerivatives();`
2. `m_pIncomingWaves->compute();`
3. `shiftZetaPhi();`
4. `leapFrog()`.

Part VI

CONCLUSIONS, RECOMMENDATIONS AND FUTURE WORK

Chapter 26

Conclusions

From the results and accompanying analysis we can draw many conclusions. Let us start with what is most important for MARIN regarding the Interactive Waves project, and finish with less important conclusions.

The new CUDA RRB-solver is 30× faster

Previously, the C++ RRB-solver had to be used, which was the fastest solver available in the lin_wacu software. The C++ RRB-solver is able to solve the system $S\psi = b$ within 50 ms for domains no bigger than 100k - 200k nodes, depending on the specific underlying test problem.

From now on, MARIN can use the new CUDA RRB-solver in the Interactive Waves project. The CUDA RRB-solver does exactly the same as its C++ counterpart, but it is much faster. From simulations it is seen that the new solver can solve systems that have more than 1.5 million nodes within 50 ms with ease. Actually, the provided realistic test problems turned out somewhat too small: the CUDA RRB-solver solved them all in no more than 12 ms. If a 2048×2048 test problem were to take 10 iterations, which seems reasonable, than the CUDA RRB-solver would be able to solve such a big problem also within 50 ms[1].

In general it holds that the larger the problem the larger the speed up factor of the CUDA RRB-solver compared to the C++ RRB-solver. Depending on the specific problem and its size we find a speed up up to a factor 30. In other words, with the new CUDA RRB-solver we are able to simulate 30× larger domains, if, and this is important, the rest of the lin_wacu code would be fast enough.

The CUDA RRB-solver allows depth profiles that change in time

The constructor phase of the CUDA also takes only a few ms. In the constructor phase the device memory is allocated, the grid levels are computed and the preconditioner is constructed. All this takes only 34 ms for a grid of 2048×2048 nodes, see Table 22.1. Therefore, for somewhat smaller grids, the preconditioner can be constructed every time frame if we want to. This makes it possible to handle bottom profiles that change in time due to water currents.

[1] In Table 22.1 it is seen that a 2048×2048 nodes problem takes about 76 ms. In Figure 22.2 it is seen that this takes about 27 CG-iterations. So, if 10 CG-iterations were needed for a realistic problem of this size, which is reasonable guess as the 1.5M test problems only need 6 or 7 CG-iterations, then we would expect a solver time of: $(10/27) \cdot 76 = 28$ ms).

The solver is no longer the bottleneck in the `lin_wacu` software

Unfortunately, we do not get speed ups of a factor 30× for the complete `lin_wacu` code. This has to do with the other computations that are involved, such as computing incoming waves, time derivatives, etcetera. Amdahl's law applies here. It turns out the rest of the code becomes the time consuming part in the new situation when the `CUDA` RRB-solver is plugged-in in the software. Overall we get a speed up of the code of only a factor 2-3×, depending on the specific test problem and when the host code is run sequentially. If `OpenMP` for the additional computations (incoming waves, time derivatives, etcetera) is enabled and the work are divided among the 4 physical cores of a quadcore CPU we find speed ups of a factor 3-7×.

In Figure 25.3 we have shown the time profile of the `lin_wacu` code. We have found that the `CUDA` RRB-solver belongs in the new situation to the least time consuming parts of the code. The chart shows that in the new situation the most time consuming routines of the code are now (percentages are for the 1.5M problem):

1. `timeDerivatives()` (around 30%);
2. `m_pIncomingWaves->compute()` (around 20%);
3. `shiftZetaPhi()` (around 20%);
4. `leapFrog()` (around 10%).

Available device memory is not a problem

In Section 19.5.4 the total memory needed for the `CUDA` RRB-solver has been listed for different sizes of test problems. The largest problem that we have run was on a grid of 2048×2048 nodes. The memory required for that problem is 460 MB. A GeForce GTX 580 has at least 1 GB of global memory, thus plenty enough. Other NVIDIA GPUs, such as the Tesla series, even have up to 5 GB of global memory. Moreover, from the results it follows that the total computation time becomes earlier a problem than memory: the `lin_wacu` code would reach much earlier the bound of 50 ms (20 fps) than that the device's global memory would become fully allocated. Hence storage is not a problem[2].

A much faster implementation of the `CUDA` RRB-solver does not exist

In Section 22.3.2 we reported throughput rates of 150-250 GB/s for the kernels on the finest levels. These rates are already much higher than the global memory bandwidth (about 150 GB/s) thanks to cache benefits. In Section 22.3.4 we have shown that although the coarser levels introduce overhead, they do not degrade the performance of the `CUDA` RRB-solver significantly (at most a factor 1.2). Hence, our first version of the `CUDA` RRB-solver already adresses almost the full resources of the device. If one finds manners to increase the L1 and texture cache hit ratios or manners to reduce global memory read and write transactions (e.g., by rewriting the RRB-solver routines), the useful throughput may get higher, but not much higher: perhaps a factor 1.5× at best.

[2] However, we have to make a side note here. If only one GPU is used for both computations and rendering, maybe the amount of available memory for particular devices is not sufficient. This can also occur when other programs allocate and preserve device memory simultaneously. However, a better GPU (more global memory) or a second dedicated device (like in our test System II) fix this problem.

The CUDA IPDIAG-solver is surprisingly good for realistic problems

In our experiments we found that the CUDA IPDIAG-solver is not a good preconditioner for the 2D Poisson problem, see Section 22.2.1. The number of required CG-iterations grows rapidly with the size of the problem, and becomes soon way too big. Compared to plain CG only a reduction of a factor 2 in CG-iterations is observed. However, for realistic test problems the CUDA IPDIAG-solver performs quite well. This has to do with the fact that our system matrix S is much more diagonally dominant than the system matrix corresponding to the 2D Poisson problem.

In Section 2.5 it is explained why the matrix S is diagonally dominant. It is also mentioned that for small mesh spacings h the diagonal dominance is not very strong. However, if there are many so-called "dry nodes", that is nodes that correspond to land rather than water, the matrix S becomes much stronger diagonally dominant. This follows from the fact that for dry nodes the center diagonal element of S is set to 1 and the outer diagonals are set to 0. In one of our test problems this is observed clearly: the Gelderse IJssel. We found that the average number of CG-iterations for the CUDA IPDIAG-solver is 12 regardless of the problem size, and thus does not grow with the problem size, see Table 23.1. This is only possible when the matrix S is strongly diagonally dominant. From Figure 3.3 we can see that the Gelderse IJssel is a small river. The computational domain is always a rectangle and thus there is much land compared to water, i.e., many dry nodes.

Although the number of CG-iterations of the CUDA IPDIAG-solver is 2-6$\times$ larger than the number of CG-iterations of the RRB-solver, the computation times are much closer. The reason is simple: an iteration of the CUDA IPDIAG-solver costs less than an iteration of the CUDA RRB-solver. The CUDA IPDIAG-solver is a PCG solver that can be parallelized easily because of its simplicity; the preconditioner step $Mz = r$ is done explicitly: $z = M^{-1}r$ and since M^{-1} (the IP preconditioner) is also a pentadiagonal matrix, the preconditioner step is nothing more than a sparse matrix-vector multiplication (SpMV). This also means that the CUDA IPDIAG-solver is easy to divide over multiple PCs and GPUs, see [1]. So, if in the future a multi-GPU solver is used, the CUDA IPDIAG-solver is a good candidate.

Chapter 27

Recommendations and future research

We would like to recommend MARIN and in particular the MSG group the following regarding the Interactive Waves project. The recommendations also indicate topics for future research.

Optimize the rest of the `lin_wacu` software

The solver is no longer the bottleneck in the `lin_wacu` software. From our experiments it is seen that four `C++` subroutines in the routine `WavesComputer::compute()` consume the most time when the new `CUDA` RRB-solver is plugged-in in the software. Because of these bottlenecks we only observe a total speed up of a factor 2-3 when the `CUDA` RRB-solver is used (Amdahl's law). To get a much better total speed up we have to make the code around the solver much faster. This should be on top of the "to do list". Actually, before starting or while optimizing and parallelizing the rest of the code, we also advise to review the code and find out whether the computations can be fundamentally differently or more efficiently from an algorithm point of view.

Document the `lin_wacu` code and appoint somebody to become an expert

Although the source code is commented, it is unsufficient to really understand what is going on. Our advice is to document the `lin_wacu` code. Furthermore, at the moment MARIN relies heavily on the expertise of Gert Klopman; he is the only one that fully understands the `lin_wacu` software and the only one who can make fundamental changes. It is a good idea to have an expert internally at MARIN who also fully understands the Variational Boussinesq model (VBM) and `lin_wacu` code.

Allow depth profiles that change in time

At the moment the code is such that the preconditioner is constructed once for a fixed system matrix S. However, in some cases one desires a bottom profile that changes accross time, e.g, because sand of the bottom is moved by the water current. In these cases the system matrix S also changes in time (some or all coefficients change). Hence the preconditioner has to be constructed multiple times. It is not really necessary to do this every time frame but say once

in a second (20 time frames). At the moment the `lin_wacu` code cannot handle this, but with minor modifications this can be achieved.

Research how to handle much bigger problems

For a simulator to fulfill the real-time requirement all computations have to be performed within 50 ms. But do not forget: the data also must be visualized, which also takes time. From our experiments it is found that the solver can solve systems of about 2048×2048 nodes in real-time. However, with a mesh spacing of $5\,m$ this corresponds to a domain of about $10\,km \times 10\,km$, which is still not very big. Although computing power increases with the years, it seems there is no way that a single PC with a single GPU will be able compute much larger domains in real-time. Therefore, we recommend to study how data and computations can be divided over a cluster of PCs, one PC with multiple CPUs/GPUs, etcetera.

Another option is to look at domains which can be approximated by a constant and flat bottom profile (like open sea). In this case the system matrix S becomes a pentadiagonal matrix similar to the 2D Poisson's matrix. For this kind of problems a specific class solvers can be used: Poisson-solvers. As those type of solvers use FFTs, we can expect a speed up of a factor 100 compared to a PCG-solver, and hence domains up to $100\,km \times 100\,km$ may become computable in real-time.

Appendix A

List of symbols

Symbol	Unit	Name
ζ	m	water level
h	m	water depth
ϕ	$\frac{m^2}{s}$	velocity potential
φ	$\frac{m^2}{s}$	surface velocity potential
ρ	$\frac{kg}{m^3}$	mass density
$\mathcal{D}, \mathcal{M}, \mathcal{N}$	–	model parameters
t	s	time
x, y	m	horizontal coordinates
z	m	vertical coordinate
$\mathbf{U} = (U, V)^T$	$\frac{m}{s}$	current
g	$\frac{m}{s^2}$	gravity
p	$Pa = \frac{kg}{ms^2}$	pressure
P_s	$\frac{m^2}{s^2}$	"pressure puls"
N_x, N_y	–	number of grid points in x- resp. y-direction
L_x, L_y	m	length of computational domain in x- resp. y-direction
$\Delta x, \Delta y$	m	mesh size in x- resp. y-direction
h	m	characteristic mesh size
S	–	system matrix
S_1	–	first Schur complement

Appendix B

List of abbreviations

Abbreviation	Full name or meaning
AXPY	vector-update: $y := ax + y$
BIM	Basic Iterative method
BLAS	Basic Linear Algebra Subprograms
CG	Conjugate Gradient(s)
Cg4	CG with MIC as preconditioner
CPU	central processing unit
CSR	compressed sparse row
CUDA	Compute Unified Device Architecture
DIAG/DIA	diagonal (scaling)
DDR	double data rate
FFT	Fast Fourier Transform
GB	gigabyte
GPU	graphics processing unit
GS	Gauss-Seidel
HF	high frequencey
IC	Incomplete Cholesky
ILP	instruction level parallelism
IP	Incomplete Poisson
IPDEF	CG with IP as preconditioner and deflation
IPDIAG	CG with IP as preconditioner and diagonal scaling
IU	instruction unit
JAC	Jacobi
kB	kilobyte
LF	low frequency
MARIN	Maritime Research Institute Netherlands
MB	megabyte

(continued on next page)

Abbreviation	Full name or meaning
MG	Multigrid
MGCG	CG with MG as precontioner
MIC	Modified Incomplete Cholesky
MV	matrix-vector
Nop	CG with diagonal scaling
OpenCL	Open Computing Language
OpenMP	Open Multi Processing
OSG	OpenSceneGraph
PC	personal computer
PCG	preconditioned CG
RHS	right-hand side
RIC	Relaxed Incomplete Cholesky
RICCG	CG with RIC as preconditioner
RICDEF	RICCG with deflation
RRB	Repeated Red-Black
RRB-k	RRB with k levels
SDK	Software Development Kit
SIMD	single-input multiple-data
SIMT	single-input mutliple-threads
SM	streaming multiprocessor
SOR	Successive Overrelaxation
SP	streaming processor
SpAI	Sparse Approximate Inverse
SPD	symmetric positive definite
SPSD	symmetric positive semi-definite
SpMV	sparse matrix-vector
SSOR	Symmetric SOR
TG	Two-grid
VBM	Variational Boussinesq model

Appendix C

Raw data

C.1 Timing results System I: GTX 285 — all test problems

Data	sol	br	ho	nodes	total	solver	(pct)	Fext
plymouth-100000	cudarrb	400	250	100000	29.19	4.17	(14%)	7.10787
plymouth-100000	rrb	400	250	100000	41.13	15.98	(39%)	7.10787
plymouth-100000	cudasimple	400	250	100000	31.21	6.09	(20%)	7.10787
portpresto-100000	cudarrb	250	400	100000	17.53	4.20	(24%)	2.49029
portpresto-100000	rrb	250	400	100000	27.51	14.21	(52%)	2.49029
portpresto-100000	cudasimple	250	400	100000	21.93	8.57	(39%)	2.49029
ijssel-100000	cudarrb	500	200	100000	17.90	4.26	(24%)	3.35114
ijssel-100000	rrb	500	200	100000	62.09	48.61	(78%)	3.35114
ijssel-100000	cudasimple	500	200	100000	19.98	6.30	(32%)	3.35114
plymouth-200000	cudarrb	500	400	200000	47.97	5.42	(11%)	5.69041
plymouth-200000	rrb	500	400	200000	82.99	40.38	(49%)	5.69041
plymouth-200000	cudasimple	500	400	200000	50.56	7.87	(16%)	5.69040
portpresto-200000	cudarrb	500	400	200000	31.33	5.46	(17%)	3.18055
portpresto-200000	rrb	500	400	200000	65.99	40.12	(61%)	3.18054
portpresto-200000	cudasimple	500	400	200000	41.38	15.43	(37%)	3.18054
ijssel-200000	cudarrb	800	250	200000	32.04	5.93	(19%)	3.90261
ijssel-200000	rrb	800	250	200000	121.83	95.23	(78%)	3.90261
ijssel-200000	cudasimple	800	250	200000	34.43	8.42	(24%)	3.90261
plymouth-500000	cudarrb	800	625	500000	99.97	8.49	(8%)	7.35204
plymouth-500000	rrb	800	625	500000	231.41	140.10	(61%)	7.35206
plymouth-500000	cudasimple	800	625	500000	114.45	20.81	(18%)	7.35205
portpresto-500000	cudarrb	800	625	500000	71.54	8.49	(12%)	3.08224
portpresto-500000	rrb	800	625	500000	191.74	128.88	(67%)	3.08224
portpresto-500000	cudasimple	800	625	500000	95.72	32.32	(34%)	3.08224
ijssel-500000	cudarrb	1000	500	500000	72.69	8.69	(12%)	3.36611
ijssel-500000	rrb	1000	500	500000	258.42	191.16	(74%)	3.36611
ijssel-500000	cudasimple	1000	500	500000	78.35	14.37	(18%)	3.36611
plymouth-1000000	cudarrb	1250	800	1000000	188.17	13.74	(7%)	6.91291
plymouth-1000000	rrb	1250	800	1000000	510.27	331.76	(65%)	6.91292
plymouth-1000000	cudasimple	1250	800	1000000	211.28	34.13	(16%)	6.91292
portpresto-1000000	cudarrb	1000	1000	1000000	144.91	13.74	(9%)	3.39525
portpresto-1000000	rrb	1000	1000	1000000	403.92	272.63	(67%)	3.39525
portpresto-1000000	cudasimple	1000	1000	1000000	189.15	57.43	(30%)	3.39525
ijssel-1000000	cudarrb	1600	625	1000000	139.73	13.17	(9%)	3.92232
ijssel-1000000	rrb	1600	625	1000000	518.80	391.50	(75%)	3.92232
ijssel-1000000	cudasimple	1600	625	1000000	151.09	24.22	(16%)	3.92232
plymouth-1500000	cudarrb	1250	1200	1500000	289.06	18.17	(6%)	7.36809
plymouth-1500000	rrb	1250	1200	1500000	767.90	491.41	(64%)	7.36810

plymouth-1500000	cudasimple	1250	1200	1500000	323.15	46.79	(14%)	7.36812
portpresto-1500000	cudarrb	1200	1250	1500000	214.89	19.72	(9%)	3.40904
portpresto-1500000	rrb	1200	1250	1500000	657.07	461.95	(70%)	3.40904
portpresto-1500000	cudasimple	1200	1250	1500000	283.59	87.95	(31%)	3.40904
ijssel-1500000	cudarrb	1500	1000	1500000	213.36	17.83	(8%)	4.10701
ijssel-1500000	rrb	1500	1000	1500000	730.92	533.71	(73%)	4.10701
ijssel-1500000	cudasimple	1500	1000	1500000	228.31	33.64	(15%)	4.10701

C.2 Timing results System II: GTX 580 — all test problems

Data	sol	br	ho	nodes	total	solver	(pct)	Fext
plymouth-100000	cudarrb	400	250	100000	23.33	1.85	(8%)	7.10787
plymouth-100000	rrb	400	250	100000	33.01	11.53	(35%)	7.10787
plymouth-100000	cudasimple	400	250	100000	23.41	2.01	(9%)	7.10787
portpresto-100000	cudarrb	250	400	100000	11.03	1.82	(17%)	2.49029
portpresto-100000	rrb	250	400	100000	19.05	10.06	(53%)	2.49029
portpresto-100000	cudasimple	250	400	100000	11.89	2.75	(23%)	2.49029
ijssel-100000	cudarrb	500	200	100000	11.46	2.07	(18%)	3.35114
ijssel-100000	rrb	500	200	100000	56.84	47.68	(84%)	3.35114
ijssel-100000	cudasimple	500	200	100000	11.51	2.10	(18%)	3.35114
plymouth-200000	cudarrb	500	400	200000	37.49	2.61	(7%)	5.69041
plymouth-200000	rrb	500	400	200000	58.71	23.78	(41%)	5.69041
plymouth-200000	cudasimple	500	400	200000	37.68	2.99	(8%)	5.69041
portpresto-200000	cudarrb	500	400	200000	20.16	2.64	(13%)	3.18055
portpresto-200000	rrb	500	400	200000	40.27	22.81	(57%)	3.18054
portpresto-200000	cudasimple	500	400	200000	23.35	5.85	(25%)	3.18055
ijssel-200000	cudarrb	800	250	200000	20.55	2.77	(13%)	3.90261
ijssel-200000	rrb	800	250	200000	100.57	82.34	(82%)	3.90261
ijssel-200000	cudasimple	800	250	200000	20.85	3.17	(15%)	3.90261
plymouth-500000	cudarrb	800	625	500000	74.28	4.64	(6%)	7.35206
plymouth-500000	rrb	800	625	500000	141.49	71.48	(51%)	7.35206
plymouth-500000	cudasimple	800	625	500000	79.64	8.76	(11%)	7.35206
portpresto-500000	cudarrb	800	625	500000	46.21	4.70	(10%)	3.08224
portpresto-500000	rrb	800	625	500000	106.54	64.91	(61%)	3.08224
portpresto-500000	cudasimple	800	625	500000	55.82	13.63	(24%)	3.08224
ijssel-500000	cudarrb	1000	500	500000	47.08	4.78	(10%)	3.36612
ijssel-500000	rrb	1000	500	500000	171.48	129.30	(75%)	3.36611
ijssel-500000	cudasimple	1000	500	500000	48.89	6.11	(12%)	3.36612
plymouth-1000000	cudarrb	1250	800	1000000	139.32	7.76	(6%)	6.91292
plymouth-1000000	rrb	1250	800	1000000	312.77	177.39	(57%)	6.91292
plymouth-1000000	cudasimple	1250	800	1000000	154.34	16.64	(11%)	6.91292
portpresto-1000000	cudarrb	1000	1000	1000000	97.23	7.93	(8%)	3.39525
portpresto-1000000	rrb	1000	1000	1000000	237.22	147.72	(62%)	3.39525
portpresto-1000000	cudasimple	1000	1000	1000000	117.07	27.02	(23%)	3.39525
ijssel-1000000	cudarrb	1600	625	1000000	105.72	7.66	(7%)	3.92232
ijssel-1000000	rrb	1600	625	1000000	366.64	265.91	(73%)	3.92232
ijssel-1000000	cudasimple	1600	625	1000000	110.31	10.72	(10%)	3.92232
plymouth-1500000	cudarrb	1250	1200	1500000	214.29	10.50	(5%)	7.36809
plymouth-1500000	rrb	1250	1200	1500000	508.38	298.36	(59%)	7.36810
plymouth-1500000	cudasimple	1250	1200	1500000	234.60	22.92	(10%)	7.36809
portpresto-1500000	cudarrb	1200	1250	1500000	141.35	11.46	(8%)	3.40904
portpresto-1500000	rrb	1200	1250	1500000	349.73	218.75	(63%)	3.40904
portpresto-1500000	cudasimple	1200	1250	1500000	171.60	39.45	(23%)	3.40903
ijssel-1500000	cudarrb	1500	1000	1500000	142.66	10.65	(7%)	4.10701
ijssel-1500000	rrb	1500	1000	1500000	478.63	348.50	(73%)	4.10701
ijssel-1500000	cudasimple	1500	1000	1500000	146.61	15.52	(11%)	4.10701

Bibliography

[1] Ament, M., Knittel, G., Weiskopf, D. Straßer, W., *A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform*, Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010), pp. 583–592, 2010.

[2] Arnold, D.N., *A Concise Introduction to Numerical Analysis*, Lecture Notes, University of Minnesota, 2001.

[3] Bell, N., Garland, M., *Efficient Sparse Matrix-Vector Multiplication on CUDA*, Technical Report NVR-2008-004, NVIDIA, December 2008, http://www.nvidia.com/object/nvidia_research_pub_001.html.

[4] Brand, C.W., *An Incomplete-factorization Preconditioning using Repeated Red-Black Ordering*, Numerische Mathematic, pp. 433–454, July 1992.

[5] Fujimoto, N., *Faster Matrix-Vector Multiplication on GeForce 8800GTX*, Proceedings of IEEE Interational Parallel and Distributed Processing Symposium (IPDPS), 22:1-8, 2008.

[6] Golub, G.H., Van Loan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, third edition, 1996.

[7] Gupta, R., *Implementation of the Deflated Preconditioned Conjugate Gradient Method for Bubbly Flow on the Graphical Processing Unit (GPU)*, MSc Thesis Computer Engineering, Delft University of Technology, Delft, August 2010.

[8] Harris, M., *Optimizing Parallel Reduction in CUDA*, CUDA Webinar 2, http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[9] Hestenes, M.R., Stiefel, E., *Methods of Conjugate Gradients for Solving Linear Systems.* J. Research Nat. Bur. Standards, 49:409436 (1953), 1952.

[10] Higham, N.J., *The Accuracy of Floating Point Summation*, SIAM Journal on Scientific Computing 14 (4): 783-799, 1993, http://www.maths.Manchester.ac.uk/~nareports/narep198.pdf.

[11] Karimi, K., Dickson, N.G., Hamze, F., *A Performance Comparison of CUDA and OpenCL*, D-Wave Systems Inc., British Columbia, Canada, http://arxiv.org/pdf/1005.2581.

[12] Kirk, D.B., Hwu, W.W., *Programming Massively Parallel Processors*, Morgan Kaufmann Publishers, 2010.

[13] Klopman, G., *Variational Boussinesq Modelling of Surface Gravity Waves over Bathymetry*, Phd Thesis, University of Twenty, Twente, May 2010.

[14] Meurant, G., *Computer Solution of Large Linear Systems*, Studies in Mathematics and its Applications 28, Elsevier, Amsterdam, 1999.

[15] Meurant, G., *The Lanczos and Conjugate Gradient Algorithms*, SIAM, August 2006.

[16] Murthy, G.S., *Optimal Loop Unrolling for GPGPU Programs*, Msc Thesis, Ohio State University, 2009, http://etd.ohiolink.edu/send-pdf.cgi/Sreenivasa%20Murthy%20Giridhar.pdf?osu1253131903.

[17] NVIDIA team, *NVIDIA CUDA C Programming Guide v4.0*, NVIDIA Corporation, June, 2011, http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[18] NVIDIA team, *NVIDIA CUDA Best Practices Guide v4.0*, NVIDIA Corporation, March, 2011, http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

[19] Ploeg, A. van der, *Preconditioning for Sparse Matrices with Applications*, PhD Thesis Mathematics and Physics, University of Groningen, Groningen, February 1994.

[20] Saad, Y. *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics (SIAM), 2nd edition, Philadelphia, 2003.

[21] Sanders, J., Kandrot, E., *CUDA by Example — An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, July, 2010.

[22] Shewcuck, J.R., *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*, August 1994.

[23] Tang, J.M., *Two-Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems*, PhD Thesis Applied Mathematics, Delft University of Technology, Delft, 2008.

[24] Volkov, V., *Better performance at lower occupancy*, GPU Technology Conference (GTC 2010), 2010, www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[25] Volkov, V., *Use registers and multiple outputs per thread on GPU*, International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10), 2010, http://www.cs.berkeley.edu/~volkov/volkov10-PMAA.pdf.

[26] Vuik, C., Lahaye, D.J.P., *Scientific Computing (wi4201)*, Lecture notes, Delft University of Technology, Delft, 2010, http://ta.twi.tudelft.nl/nw/users/vuik/wi4201/wi4201_notes.pdf.

[27] Wesseling, P., *An Introduction to Multigrid Methods*, Corrected Reprint, Philadelphia: R.T. Edwards, Inc., 2004.

[28] Wout, E. van 't, *Improving the Linear Solver used in the Interactive Wave Model of a Real-time Ship Simulator*, MSc Thesis Applied Mathematics, Delft University of Technology, Delft, August 2009.