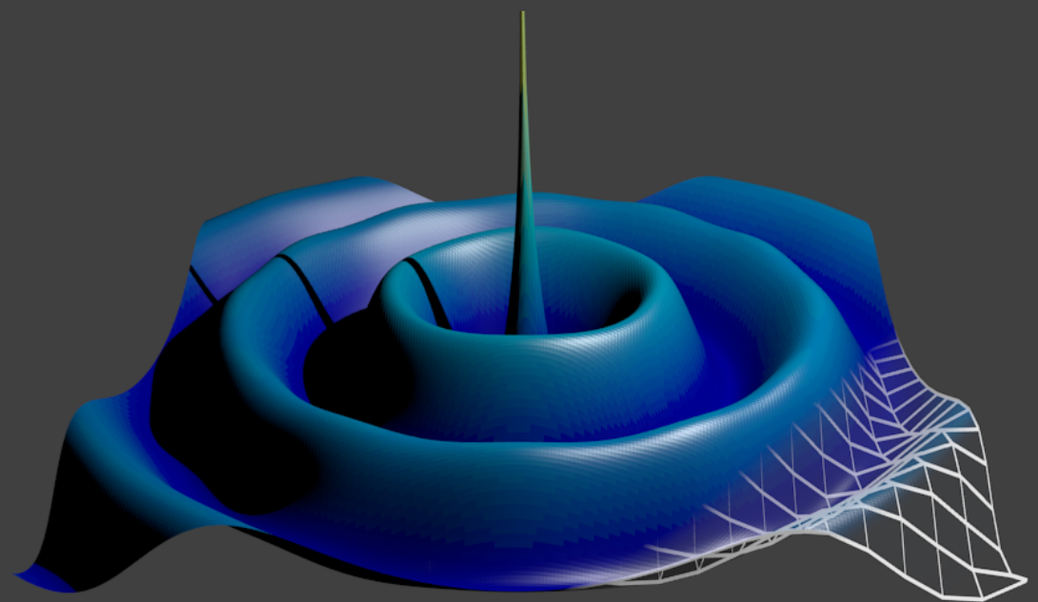
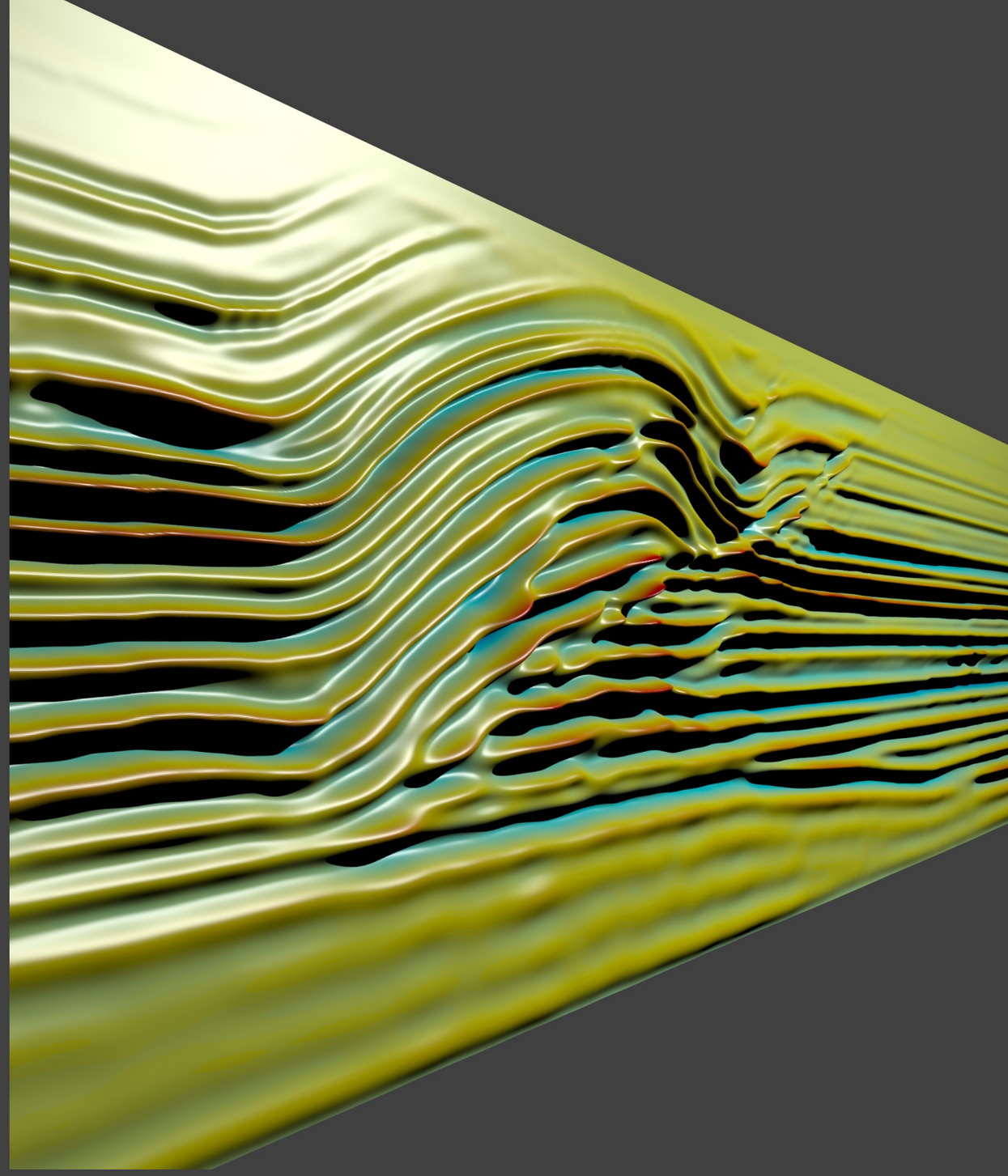


ISBN 978-94-6186-427-7

Reduction of computing time for seismic applications based on the Helmholtz equation by Graphics Processing Units



Reduction of computing time for seismic applications based on the Helmholtz equation by Graphics Processing Units

Hans Peter Knibbe

REDUCTION OF COMPUTING TIME FOR SEISMIC APPLICATIONS BASED ON THE HELMHOLTZ EQUATION BY GRAPHICS PROCESSING UNITS

REDUCTION OF COMPUTING TIME FOR SEISMIC APPLICATIONS BASED ON THE HELMHOLTZ EQUATION BY GRAPHICS PROCESSING UNITS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 3 maart 2015 om 10:00 uur

door

Hans Peter KNIBBE

MSc Computer Science, Technische Universiteit Delft
geboren te Reims, Frankrijk.

Dit proefschrift is goedgekeurd door de promotors:

Prof. dr. ir. Cornelis W. Oosterlee en Prof. dr. ir. Kees Vuik

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. Cornelis W. Oosterlee	Technische Universiteit Delft, promotor
Prof. dr. ir. Kees Vuik	Technische Universiteit Delft, promotor
Prof. dr. S. F. Portegies Zwart	Universiteit Leiden
Prof. dr. K. J. Batenburg	Universiteit Leiden
Prof. dr. W. A. Mulder	Technische Universiteit Delft
Prof. dr. ir. H. J. Sips	Technische Universiteit Delft
Prof. dr. ir. H. X. Lin	Technische Universiteit Delft

Keywords: Helmholtz, Shifted Laplace Preconditioner, Multigrid, GPU, CUDA, Seismic Migration, Acceleration, Least-Squares Migration, VCRS.

ISBN 978-94-6186-427-7

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

CONTENTS

1

INTRODUCTION

The oil and gas industry makes use of computational intensive algorithms such as reverse-time migration and full waveform inversion to provide an image of the subsurface. The image is obtained by sending wave energy into the subsurface and recording the signal required for a seismic wave to reflect back to the surface from the interfaces with different physical properties. A seismic wave is usually generated by shots at known frequencies, placed close to the surface on land or to the water surface in the sea. Returning waves are usually recorded in time by hydrophones in the marine environment or by geophones during land acquisition. The goal of seismic imaging is to transform the seismograms to a spatial image of the subsurface.

Seismic imaging algorithms include Kirchoff migration, reverse time migration (RTM), least-squares migration (LSM) and full waveform inversion (FWI). Migration methods assume that the velocity model is given, whereas FWI updates the initial velocity model to match the recorded signal. In this work, we only focus on migration, however, some findings can also be applied to FWI.

Migration algorithms produce an image of the subsurface given seismic data measured at the surface. In particular, pre-stack depth migration produces the depth locations of reflectors by mapping seismic data from the time domain to the depth domain, assuming that a sufficiently accurate velocity model is available. The classic imaging principle [1, 2] is based on the correlation of the forward propagated wavefield from a source and a backward propagated wavefield from the receivers. To get an approximation of the reflector amplitudes, the correlation is divided by the square of the forward wavefield [3, 4]. For true-amplitude or amplitude-preserving migration, there are a number of publications based on the formulation of migration as an inverse problem in the least-squares sense [5–9].

The two main approaches for seismic imaging include the ray-based and wave-equation-based approaches. The first approach is based on a high-frequency approximation or one-way propagation approximation, for instance, Kirchhoff integral method, see e.g. [10], [11], [12], [13], [14]. This approach is efficient, however it fails in complex geological settings. The wave-equation-based approach consists of differential, extrapolation

or continuation methods. The most well known method is the wavefield continuation method with finite-difference discretization. It images all arrivals and the whole subsurface. Because of the last, this set of methods is computationally costly. For an overview see e.g, [15]. In this thesis we consider the wave-equation-based approach.

Modeling is a major component of migration and inversion algorithms. Traditionally, seismic data is modeled in the time domain because of the implementation simplicity as well as the memory demands. However, modeling in the frequency domain offers such advantages as parallelization over frequencies or reuse of earlier results if an iterative solver is employed for computing the wavefields, for example, during LSM or FWI.

Algorithms for seismic imaging can be divided into two main groups: those formulated in the time domain and those that perform in the frequency domain. Combinations and transitions from one group to another are also possible by using the Fourier transformation. An overview of numerical methods for both cases is given in [16]. Discretization with finite differences of the time-domain formulation leads to an explicit time-marching scheme, where the numerical solution is updated every time step. The discretization of the frequency-domain formulation gives a linear system of equations that needs to be solved for each frequency. The wave equation in the frequency-domain, or Helmholtz equation, is difficult to solve because the resulting discretized system is indefinite.

The demand for better resolution of the subsurface image increases the bandwidth of the seismic data and leads to larger computational problems. Therefore, considering an efficient implementation on parallel architectures is of major importance. Currently, the many-core computers are dominant for large-scale intensive computations. The general purpose graphics cards (GPUs), field-programmable gate arrays (FPGAs) are mainly used as accelerators to speedup the computations. Both hardware configurations have been shown to accelerate certain problems. GPUs are used more widely because of open source compilers and shorter development time. Recently, Intel released the Intel Many Integrated Core Architecture (MIC), which is a coprocessor compute architecture based on the Xeon Phi processors. The MIC with 61 cores and 8 GB of memory is positioned between a many-core CPU and a GPU. The main advantage of MIC is that an existing code can be easily run without completely rewriting it. However, to achieve optimal performance for parallel computations on a MIC, one also needs to adjust the code. Unfortunately, there is no optimal "fit for all purposes" hardware solution. Therefore, each problem needs to be considered separately.

1.1. SCOPE AND OUTLINE OF THE THESIS

In this thesis we combine research from three different areas: (1) numerical analysis, (2) computational science and (3) geophysics. By using an enhanced Helmholtz solver accelerated on GPUs we focus on an efficient implementation of migration in the frequency domain and least-squares migration.

The numerical core of the thesis is a Helmholtz solver in three dimensions preconditioned by a shifted Laplace multigrid method. We focus on developing an efficient solver using three different strategies. Firstly, the preconditioner is enhanced by using the matrix-dependent prolongation and multi-colored Gauss-Seidel smoother. Sec-

ondly, we introduce a new sparse matrix storage format that not only reduces the memory usage but also speeds up the matrix-vector computations. Thirdly, the Helmholtz solver is accelerated by using GPUs.

The idea of using GPUs is to search for the most efficient way of speeding up our computations. Therefore, we consider two approaches of using GPUs: as a replacement and as an accelerator. The main difference between these approaches is where the matrix is stored: in the GPU or CPU memory. The implementation is challenging, since the CPU and GPU have different properties, therefore, we develop a common code concept and a task system to target an efficient parallel implementation.

Here, we focus on geophysical applications such as migration. The depth migration in the frequency domain uses an enhanced and accelerated Helmholtz solver and is compared to the reverse time migration in the time domain. For an efficient implementation of the least-squares migration we introduce a frequency decimation method.

The outline of the thesis is as follows.

In Chapter 2, a Helmholtz equation in two dimensions discretized by a second-order finite difference scheme is considered. Krylov subspace methods such as Bi-CGSTAB and IDR(s) have been chosen as solvers. Since the convergence of the Krylov subspace solvers deteriorates with increasing wave number, a shifted Laplacian multigrid preconditioner is used to improve the convergence. The implementation of the preconditioned solver on a CPU (Central Processing Unit) is compared to an implementation on a GPU (Graphics Processing Units or graphics card) using CUDA (Compute Unified Device Architecture).

Chapter 3 is focusing on an iterative solver for the three-dimensional Helmholtz equation on multi-GPU using CUDA. The Helmholtz equation discretized by a second-order finite difference scheme is solved with Bi-CGSTAB preconditioned by a shifted Laplace multigrid method with matrix-dependent prolongation. Two multi-GPU approaches are considered: data parallelism and split of the algorithm. Their implementations on multi-GPU architecture are compared to a multi-threaded CPU and single GPU implementation.

In Chapter 4 we investigate whether migration in the frequency domain can compete with a time-domain implementation when both are performed on a parallel architecture. In the time domain we consider 3-D reverse time migration with the constant-density acoustic wave equation. For the frequency domain, we use an iterative Helmholtz Krylov subspace solver based on a shifted Laplace multigrid preconditioner with matrix-dependent prolongation. As a parallel architecture, we consider a commodity hardware cluster that consists of multi-core CPUs, each of them connected to two GPUs. Here, GPUs are used as accelerators and not as an independent compute node. The parallel implementation over shots and frequencies of the 3-D migration in the frequency domain is compared to a time-domain implementation. We optimized the throughput of the latter with dynamic load balancing, asynchronous I/O and compression of snapshots.

In Chapter 5 an efficient least-squares migration (LSM) algorithm is presented using several enhancements. Firstly, a frequency decimation approach is introduced that makes use of redundant information present in the data. Secondly, a new matrix storage format VCRS (Very Compressed Row Storage) is presented. It does not only reduce

the size of the stored matrix by a certain factor but also increases the efficiency of the matrix-vector computations. The effect of lossless and lossy compression is investigated. Thirdly, we accelerate the LSM computational engine by graphics cards (GPUs). The GPU is used as an accelerator, where the data is partially transferred to a GPU to execute a set of operations, or as a replacement, where the complete data is stored in the GPU memory. We demonstrate that using GPU as a replacement leads to higher speedups and allows us to solve larger problem sizes. In Chapter 6 we draw conclusions and give some remarks and suggestions for future work.

In Appendix A, specifications of the Little Green Machine (LGM) are given, which has been used for the most computations and performance comparisons.

In Appendix B we show an example of the common code on CPU and GPU.

Appendix C presents coefficients for the matrix-dependent prolongation matrix in three-dimensions used in multigrid.

1.2. HELMHOLTZ EQUATION

The wave propagation in an acoustic medium can be described in time domain or in frequency domain. We start with the description in frequency domain, because this is the main focus of the thesis.

The Helmholtz equation meaning the wave equation in the frequency domain reads

$$-\Delta\phi - (1 - i\alpha)k^2\phi = g, \tag{1.1}$$

where $\phi = \phi(x, y, z, \omega)$ is the wave pressure field as a function of a spatially-dependent frequency, $k = k(x, y, z, \omega)$ is the wavenumber and $g = g(x, y, z, \omega)$ is the source term. The coefficient $0 \leq \alpha \ll 1$ represents a damping parameter that indicates the fraction of damping in the medium. The corresponding differential operator has the following form:

$$\mathscr{A} = -\Delta - (1 - \alpha i)k^2, \tag{1.2}$$

where Δ denotes the Laplace operator. In our work we consider a rectangular domain $\Omega = [0, X] \times [0, Y] \times [0, Z]$.

In many real world applications the physical domain is unbounded, and artificial reflections should be avoided. In the frequency domain, non-reflecting boundary conditions can be used:

- First order radiation boundary condition (described in e.g. Clayton et al. [17], Engquist et al. [18])

$$\left(-\frac{\partial}{\partial\eta} - ik\right)\phi = 0, \tag{1.3}$$

  where η is the outward unit normal component to the boundary. The disadvantage of this boundary condition is that it is not accurate for inclined outgoing waves.

- Second order radiation boundary condition (described in Engquist et al. [18])

$$\mathscr{B}\phi|_{edge} := -\frac{3}{2}k^2\phi - ik\sum_{j=1,j\neq i}^{2}\left(\pm\frac{\partial\phi}{\partial x_j}\right) - \frac{1}{2}\frac{\partial^2\phi}{\partial x_i} = 0, \tag{1.4}$$

$$\mathscr{B}\phi|_{corner} := -2ik\phi + \sum_{i=1}^{2}\left(\pm\frac{\partial\phi}{\partial x_i}\right) = 0, \tag{1.5}$$

  where x_i is a coordinate parallel to the edge for $\mathscr{B}\phi|_{edge}$. The $\pm$ sign is determined such that for outgoing waves the non-reflecting conditions are satisfied.

We consider here the first order radiation boundary condition 1.3.

The wave equation in time domain reads

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \Delta u = f, \tag{1.6}$$

where $u(x,y,z,t)$ denotes the pressure wavefield as a function of time, $c(x,y,z)$ is the velocity in the subsurface and $f(x,y,z,t)$ describes the source. The Laplace operator is denoted by Δ. Position (x,y,z) belongs to a computational domain Ω. Dirichlet or Neumann boundary conditions can be used at reflecting boundaries. If reflections from the boundaries should be suppressed, then layers of absorbing boundary conditions can be used, see e.g. [19].

If the wavelet or signature of the source term g is given in the time domain, its frequency spectrum is obtained by a Fast Fourier Transform (FFT). Given the seismic data, the Nyquist theorem dictates the frequency sampling and the maximum frequency. In practice, the maximum frequency in the data is lower than the Nyquist maximum frequency and is defined by the wavelet. Given the range of frequencies defined by Nyquist's theorem and the data, the Helmholtz equation is solved for each frequency and the wavefield is sampled at the receiver positions, producing a seismogram in the frequency domain. The wavelet and an inverse FFT are applied to obtain the synthetic seismogram in the time domain.

The discretization of the Helmholtz equation in space depends on the number of points per wavelength. The general rule of thumb is to discretize with at least 10 points per wavelength [20]. In that case, the error behaves as $(kh)^2$, which is inversely proportional to the square of the number of points per wavelength. For high wave numbers the discretization results in a very large sparse system of linear equations which can not be solved by direct methods on current computers within reasonable time. To avoid the pollution effect, $kh = 0.625$ has been chosen constant, as described by [21].

The resulting discretized linear system is symmetric but indefinite, non-Hermitian and ill-conditioned which brings difficulties when solving with basic iterative methods. The convergence of the Krylov subspace methods deteriorates with higher wave numbers, so the need for preconditioning becomes obvious. In this thesis we consider Bi-CGSTAB (see Van der Vorst [22]) and IDR(s) (see Sonneveld, van Gijzen [23]) as Krylov solvers.

There have been many attempts to find a suitable preconditioner for the Helmholtz equation, see, for example, Gozani et al. [24], Kerchroud et al. [25]. Recently the class of

shifted Laplacian preconditioners evolved, see Laird and Giles [26], Turkel [27], Erlangga et al. [28], Erlangga [21]. The authors in [29] showed that the number of iterations of the Helmholtz solver does not depend on the problem size for a given frequency, but the number of iterations increases with frequency. The authors in [30] and [31] presented an improved version that requires fewer iterations but still requires more iterations at higher frequencies.

1.2.1. MATRIX STORAGE FORMATS

As already known, an iterative solver for the wave equation in the frequency domain requires more memory than an explicit solver in the time domain, especially, for a shifted Laplace multigrid preconditioner based on matrix-dependent prolongation. Then, the prolongation and coarse grid-correction matrices need to be stored in memory. Due to the discretization with finite differences, the three-dimensional Helmholtz equation on the finest multigrid level has a structured symmetric matrix with non-zero elements on 7 diagonals. However, on the coarser levels due to the Galerkin coarse-grid approach, the coarse-grid correction matrix has non-zero elements on 27 diagonals. The prolongation matrix is rectangular and has in general 27 non-zero matrix entries in each column. Therefore, the matrix storage format for our purposes needs to reduce memory usage and speed up the matrix-vector operations. Also, it should be suitable to do the calculations on GPUs.

There are a number of common storage formats used for sparse matrices, see e.g. [32], [33] for an overview. Most of them employ the same basic technique: store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix. For instance, one of the most popular formats is the CSR (Compressed Sparse Row) format for storage of sparse matrices, e. g. [34], [33]. Each of the matrix storage formats have their advantages and disadvantages considering a specific problem. However, all of them are too general to be efficient for the Helmholtz solver, especially when using GPUs. In Chapter 5 we propose a new matrix format suited for our purpose.

1.3. ACCELERATION WITH GPUS

High-performance computer architectures are developing quickly by having more and faster cores in the CPUs (Central Processing Units) or GPUs (Graphics Processing Units). Recently, a new generation of GPUs appeared, offering tera-FLOPs performance on a single card.

The GPU was originally designed to accelerate the manipulation of images in a frame buffer that was mapped to an output display. GPUs were used as a part of a so-called graphics pipeline, meaning that the graphics data was sent through a sequence of stages that were implemented as a combination of CPU software and GPU hardware. A very important step for the GPU evolution was made by IBM with the release of the so-called Professional Graphics Controller (PGA). They used a separate on-board processor for graphics computations. On the software side, OpenGL (Open Graphics Library) intro-

duced by Silicon Graphics Inc. played a major role in the GPU development. It was the most widely used, platform independent, cross-language application programming interface for the rendering of 2D and 3D graphics.

1.3.1. HISTORY OF GPU DEVELOPMENT

The evolution of modern graphics processors begins by the mid 1990's with the introduction of the first GPU supporting 3D graphics in a PC. Thanks to the accessibility of the hardware to the consumer markets and the development of the game industry, graphics cards began to be widely used. Actually, the term *GPU* was first introduced as part of Nvidia's marketing for the GeForce 256 in 1999. The company defined it as "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second." At that time, the design of a graphics card began to move from a pipeline to data parallelism by adding more parallel pipelines and eventually more cores to the GPU.

The next step in the evolution of GPU hardware was the introduction of the programmable pipeline on the GPU. In 2001, Nvidia released the GeForce 3 which gave programmers the ability to program parts of the pipeline with so-called shaders. These shader programs were small kernels, written in assembly-like shader languages. It opened up a way towards fully programmable graphics cards, also for non graphics purposes. Next, high level GPU programming languages such as Brook and Sh were being introduced, which formed a trend towards GPU programmability. On the hardware side, higher precision, multiple rendering buffers, increased GPU memory and texture accesses were being introduced.

The release of the GeForce 8 series or Tesla micro-architecture based GPUs by Nvidia in 2006 marked the next step in the GPU evolution: exposing the GPU as massively parallel processors. Moreover, this architecture was the first to have a fully programmable unified processor that handled all the graphics computations. To address these general purpose features from the software side, Nvidia introduced a new programming language called CUDA (Compute Unified Device Architecture) for Nvidia's GPUs, that gave rise to a more generalized computing device so-called GPGPU: general purpose GPU. Also ATI and Microsoft released similar languages for their own graphics cards.

The trend towards more general, programmable GPU cores continues with the introduction of the Fermi architecture by Nvidia in 2010. The Fermi GPU was the first GPU designed for GPGPU computing, bringing features such as true HW cache hierarchy, ECC (Error-correcting code), unified memory address space, concurrent kernel execution, increased double precision performance. GPGPUs began to compete with CPUs in terms of computational efficiency, however, in other aspects such as power consumption, GPUs were lacking behind. Nvidia started to tackle the power problem by introducing the Kepler architecture in 2012 which focused on power efficiency. It was followed by the Maxwell microarchitecture which was released in 2014 with completely redesigned processors targeted to reduce the power consumption.

The general specifications and features of a Nvidia graphics card are given by the compute capability. The compute capability version consists of a major and a minor version number. Devices with the same major revision number are of the same core

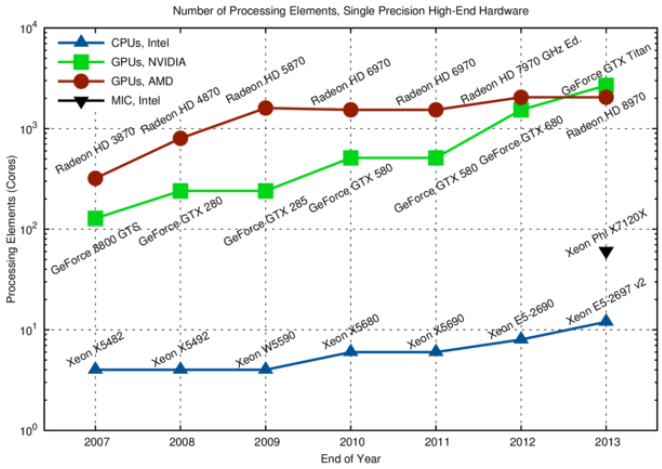


Figure 1.1: Comparison of processing elements or cores for high-end hardware using single precision arithmetics.

architecture. The major revision number is 5 for devices based on the Maxwell architecture, 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture. The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features. For example, the native double-precision floating-point support appears in devices of compute capability 2.x and higher.

Apart from Nvidia, another big player on the GPU market is AMD that acquired ATI in 2006. Their strategy is the development of processors that integrate general processing abilities with graphics processing functions within a single chip. The state-of-art architecture of AMD is Kaveri.

Currently, Nvidia and AMD have a similar gaming market share for GPUs (see [35]), however Nvidia is leading in high performance computing (HPC) accelerators.

Recently, Intel tries to compete for an HPC accelerator market share with its coprocessors based on the MIC (Many Integrated Cores) architecture. It combines many Intel CPU processors on a single chip. Xeon Phi coprocessors based on MIC architecture provide up to 61 cores, 244 threads, and 1.2 teraFLOPS of performance, and they come in a variety of configurations to address diverse hardware configurations, software, workload, performance, and efficiency requirements (www.intel.com). The biggest selling point is the code portability: one can simply recompile and run the already parallelized code on Xeon Phi. There is a debate on which is better from a performance point of view: a GPU or a Xeon Phi. This is a difficult question to answer, since not only computation performance needs to be considered, but also time/cost required to port existing applications to the platform, training programmers in order to get the maximum out of the new systems, cooling costs, floor space, etc.

Figure 1.1 shows the development of GPUs and CPUs over the years, see [36]. Three largest players in the HPC area are presented: Intel (blue and black curve), AMD (red

curve) and Nvidia (green curve). The comparison considers high-end hardware available by the end of the respective calendar year. Here, only CPUs for dual socket machines are considered to compare against the Intel MIC platform (Xeon Phi). At the beginning, AMD graphics cards had more cores than GPUs from Nvidia, however, with introduction of the Kepler microarchitecture, the number of cores became similar for both. Obviously, there is a gap between GPUs and CPUs that has grown to 2 orders of magnitude over time. The Xeon Phi can be considered as an attempt to close this gap.

The increase in the number of cores requires the development of scalable numerical algorithms. These methods have shown their applicability on traditional hardware such as multi-core CPUs, see e.g. [37]. However, the most common type of cluster hardware consists of a multi-core CPU connected to one or more GPUs. In general, a GPU has a relatively small memory compared to the CPU. Heterogeneous computational resources bring additional programming challenges, since several programming interfaces need to be used to accelerate numerical algorithms.

The GPU and CPU architectures have their own advantages and disadvantages. CPUs are optimized for sequential performance and good at instruction level parallelism, pipelining, etc. With a powerful hierarchical cache, and scheduling mechanism, the sequential performance is very good. In contrast, GPUs are designed for high instruction throughput with a much weaker cache or scheduling ability. In GPU programming, users have to spend more time to ensure good scheduling, load balancing and memory access, which can be done automatically on a CPU. As a result, GPU kernels are always simple and computationally intensive. The performance comparison of a CPU versus GPU is shown in Figure 1.2 (see [36]). One can clearly see a five- to fifteen-fold margin when comparing high-end CPUs with high-end GPUs. The introduction of Xeon CPUs based on the Sandy Bridge architecture in 2009 slightly reduced the gap, yet there is still a factor of five when comparing a single GPU with a dual-socket system. Note that the fact that the theoretical peak is harder to reach on the GPU than on the CPU is not reflected in this graph. However, even when taking this into account the GPU is still significantly faster than a CPU. The Xeon Phi falls behind in terms of single precision arithmetics, yet this is not a major concern as the architecture aims at providing high performance using double precision arithmetics.

1.3.2. CUDA

The CUDA platform was the earliest widely adopted programming model for GPU computing. CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. A recent Apple initiative within the Khronos Group is called OpenCL (Open Computing Language) which is an open standard and can be used to program CPUs, GPUs and other devices from different vendors (see [38]). It has been shown that converting a CUDA program to an OpenCL program involves minimal modifications, see Karimi et al. [39]. According to Du, Luszczek and Dongarra [40], at the beginning of this work CUDA was more efficient on the GPU than OpenCL. OpenCL solutions are supported by Intel, AMD, Nvidia, and ARM.

A CUDA program calls parallel kernels. A kernel executes in parallel across a set of

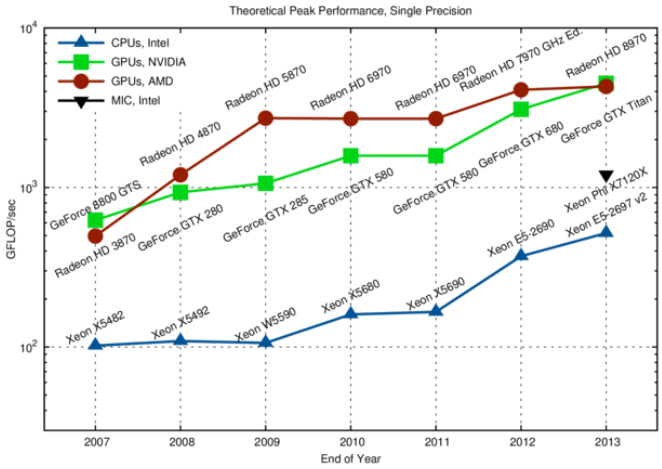


Figure 1.2: Comparison of theoretical peak GFLOP/sec in single precision. Higher is better.

parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.

According to [41], a thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

The hierarchy of threads in CUDA maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

1.3.3. ACCELERATOR OR REPLACEMENT?

A GPU can be used as replacement for the CPU, or as an accelerator. In the first case, the data lives in GPU memory to avoid memory transfers between CPU and GPU memory. The advantage of the seismic migration algorithm with frequency domain solver is that it does not require large amounts of disk space to store the snapshots. However, a disadvantage is the memory usage of the solver. As GPUs have generally much less memory available than CPUs, this impacts the size of the problem significantly.

In the second case, the GPU is considered as an accelerator, which means that the problem is solved on the CPU while off-loading some computational intensive parts of the algorithm to the GPU. Here, the data is transferred to and from the GPU for each new task. While the simplicity of the time domain algorithm makes it easy to use GPUs of modest size to accelerate the computations, it is not trivial to use GPUs as accelerators for the Helmholtz solver. By using the GPU as an accelerator, the Helmholtz matrices are distributed across two GPUs. The vectors would "live" on the CPU and are transferred when needed to the relevant GPU to execute matrix-vector multiplications.

Ideally, GPUs would be used as a replacement but the limited memory makes this difficult for large numerical problems. There seem to be a trend where CPUs and GPUs are merging so that the same memory can be accessed equally fast from the GPU or the CPU. In that case the question "accelerator or replacement?" would become irrelevant as one can alternate between both hardware without taking into account the data location.

NVIDIA recently announced that Tesla is now compatible with ARM-based chips, the kind of low-powered processors that run smartphones and tablets, which are increasingly taking business away from Intel's x86 architecture. When combining ARM with GPU, this HPC solution provides power efficiency, system configurability, and a large, open ecosystem. CUDA 6.5 takes the next step, enabling CUDA on 64-bit ARM platforms. The heritage of ARM64 is in low-power, scale-out data centers and microservers, while GPUs are built for ultra-fast compute performance. GPUs bring to the table high-throughput, power-efficient compute performance, a large HPC ecosystem, and hundreds of CUDA-accelerated applications. For HPC applications, ARM64 CPUs can offload the heavy computational tasks to GPUs. CUDA and GPUs make ARM64 competitive in HPC from the start.

1.4. MIGRATION IN TIME AND FREQUENCY DOMAIN

For our purpose of comparing migration in the time and frequency domain in this thesis, we focus on the classical imaging condition [13]

$$I(\mathbf{x}) = \sum_{shots} \sum_{t} W_s(\mathbf{x}, t) W_r(\mathbf{x}, t), \tag{1.7}$$

in time domain or

$$I(\mathbf{x}) = \sum_{shots} \sum_{k} W_s^*(\mathbf{x}, k) W_r(\mathbf{x}, k), \tag{1.8}$$

in the frequency domain. Here, I denotes the image, W_s is the wavefield propagated from the source and W_r from the receivers, respectively; t denotes time and k denotes

the frequency. The star indicates the complex conjugate. Basically, the idea of migration in time domain is to calculate first of all the forward wavefield by injecting the source wavelet. Secondly, compute the wavefield backward in time by injecting the recorded signal at the receiver locations. And finally, cross-correlate the forward and backward wavefields at given timesteps. Note, that in the frequency domain the cross-correlation becomes a simple multiplication of the forward and backward wavefields.

Migration of seismic data in 2D is commonly carried out in the frequency domain by using a direct solver. The *LU*-decomposition of the matrix arising from the discretization of the wave equation is computed once with a direct method for each frequency and stored in memory. The result can be used to compute all wavefields for all shots and also for back-propagated receiver wavefields, which correspond to the reverse-time wavefields in the time domain ([42], [43]). This method is an order of magnitude faster than the time-domain implementation when many shots must be processed. In three dimensions, however, the storage requirement for the direct solver exceeds the capacity of the available disk space. Therefore, migration of seismic data in 3D is commonly carried out in the time domain.

The classic reverse-time migration algorithms in the time domain are known to be computationally and I/O intensive [44, 45] because the forward and time-reversed wavefields have to be computed and stored. If the correlation between these fields is carried out during the time-reversed computation of the receiver data, only snapshots of the forward wavefield have to be stored.

There are two main strategies to reduce the overhead of storing snapshots. Reconstruction of the forward wavefield by marching backward in time using the last two wavefields is difficult, if not impossible, in the presence of absorbing boundary conditions. The author in [46] proposed to circumvent this problem by only storing boundary values of the snapshots or by using random instead of absorbing boundaries. For the latter, the energy of the wavefield entering the boundary is scattered and does not stack during the imaging condition. With checkpointing [47, 48], the forward wavefield is only stored intermittently at pairs of subsequent time steps. During the reverse-time computations and correlation, the missing time steps are recomputed by forward time stepping from stored snapshots over a relatively short time interval. These methods represent a trade-off between storage and computational time.

A second strategy is based on reducing the time needed to write the snapshots to disk, for instance, by asynchronous I/O [49] and wavefield compression. For the last, standard libraries with Fourier transformation or wavelet compression can be used [45].

Migration in the frequency domain is historically less mature because of the necessity to solve a sparse indefinite linear system of equations for each frequency, which arises from the discretization of the Helmholtz equation, whereas in the time domain, the discretization of the wave equation in space and time leads to an explicit time marching scheme. An important advantage of migration in the frequency domain is that the cross-correlation needed for the imaging condition becomes a simple multiplication. As a result, wavefields do not have to be stored. Parallelization over frequencies is natural. If a direct solver is used to compute the solution of the sparse matrix, typically a nested-dissection *LU*-decomposition is applied [50]. When many shots need to be treated, the frequency-domain solver in two dimensions can be more efficient than a

time-domain time-stepping methods [51, 52], because the LU-decomposition can be reused for each shot as well as for each 'reverse-time' computation. Also, lower frequencies can be treated on coarser meshes.

In three dimensions, however, frequency-domain migration is considered to be less efficient than its time-domain counterpart. One of the reasons is the inability to construct an efficient direct solver for problems of several millions of unknowns [53]. The authors of [54, 55] proposed a direct solver based on nested-dissection that compresses intermediate dense submatrices by hierarchical matrices.

An iterative solver is an obvious alternative, for instance, the one with a preconditioner that uses a multigrid method to solve the same wave equation but with very strong damping, see e.g. [21, 56, 57], also described in Chapter 2. This method, however, needs a number of iterations that increases with frequency, causing the approach to be less efficient than a time-domain method on CPUs. Note that the iterative method requires a call to the solver for each shot and each 'reverse-time' computation, so the advantage of reusing a LU-decomposition is lost. This approach was parallelized by [37]. In Chapter 3 we use GPUs to speed up the computations.

1.5. LEAST-SQUARES MIGRATION

An alternative to the depth migration is least-squares migration (LSM). LSM was introduced as a bridge between full-wave form inversion and migration. Like migration, LSM does not attempt to retrieve the background velocity model, however, like full waveform inversion the modeled data are fit to the observations. Least-squares migration [58] has been shown to have the following advantages: (1) it can reduce migration artifacts from a limited recording aperture and/or coarse source and receiver sampling; (2) it can balance the amplitudes of the reflectors; and (3) it can improve the resolution of the migration images. However, least-squares migration is usually considered expensive.

Significant work has been done in the field of least-squares migration. The authors in [58] use a Kirchhoff operator to perform least-squares migration of incomplete surface seismic and Vertical Seismic Profile (VSP) data. They find that the migration results are more focused and suffer less from the so-called acquisition footprint compared with the standard migration. The authors in [59] use least-squares migration for further Amplitude Versus reflection Angle (AVA) inversion. To attenuate artifacts, the authors in [60] use a Kirchhoff time migration operator with dip-oriented filtering. While in [61] least squares migration is used to attenuate low frequency reverse time migration artifacts. The authors in [62] and [63] show that it improves images of reservoirs located under complex overburden and that the imaging by least-squares inversion provides more reliable results than conventional imaging by migration.

Originally, Kirchhoff operators have been proposed for the modeling and migration in LSM (see e.g. [64], [58]). Recently, in the least-squares migration algorithm, wave-equation based operators were used in the time domain (see e.g. [65], [66]) and in the frequency domain (see e.g. [9], [67], [68]). The major advantage of a frequency domain computational engine is that each frequency can be treated independently in parallel.

The LSM method is posed as a linear inverse problem of finding the reflectivity r that minimizes the difference between the recorded data d and the modeled wavefield $\mathscr{A}r$ in

a least-squares sense

$$J = \frac{1}{2}\|\mathscr{A}r - d\|^2 + \frac{1}{2}\lambda R(m). \tag{1.9}$$

Here, $\mathscr{A}$ denotes so-called the de-migration operator, that contains modeling at source locations and migration. This operator can represent a Kirchhoff operator or a wave-equation based operator. In addition, R is a regularization term with damping λ needed to stabilize the solution. The objective function J is minimized using the method of conjugate gradients. For the damping term $\lambda = 0$, one can show that the solution converges to the minimum norm solution.

The LSM method is costly because for each source and receiver, we need to compute the modeling and migration stage of the data. In the time domain, it is the reverse time migration method and in the frequency domain, it is multiplication of the source and receiver modeled wavefields for each frequency.

2

GPU IMPLEMENTATION OF A HELMHOLTZ KRYLOV SOLVER PRECONDITIONED BY A SHIFTED LAPLACE MULTIGRID METHOD

Abstract

A Helmholtz equation in two dimensions discretized by a second-order finite difference scheme is considered. Krylov subspace methods such as Bi-CGSTAB and IDR(s) have been chosen as solvers. Since the convergence of the Krylov solvers deteriorates with increasing wave number, a shifted Laplacian multigrid preconditioner is used to improve the convergence. The implementation of the preconditioned solver on a CPU (Central Processing Unit) is compared to an implementation on a GPU (Graphics Processing Units or graphics card) using CUDA (Compute Unified Device Architecture). The results show that preconditioned Bi-CGSTAB on the GPU as well as preconditioned IDR(s) on the GPU are about 30 times faster than on the CPU for the same stopping criterion.

2.1. INTRODUCTION

In this chapter we focus on iterative solvers for the Helmholtz equation in two dimensions on the GPU using CUDA. The Helmholtz equation represents the time-harmonic wave propagation in the frequency domain and has applications in many fields of science and technology, e.g. in aeronautics, marine technology, geophysics, and optical

Parts of this chapter have been published in H. Knibbe, C. W. Oosterlee, and C. Vuik. Journal of Computational and Applied Mathematics, 236:281– 293, 2011, [69].

problems. In particular we consider the Helmholtz equation discretized by a second-order finite difference scheme. The size of the discretization grid depends on the wave number, which means, the higher the wave number the more grid points are required. For instance to get accurate results with a 7-point discretization scheme in three dimensions, at least 10 grid points per wave length have to be used, see Erlangga [29]. For high wave numbers the discretization results in a very large sparse system of linear equations which can not be solved by direct methods on current computers within reasonable time. The linear system is symmetric but indefinite, non-Hermitian and ill-conditioned which brings difficulties when solving with basic iterative methods. The convergence of the Krylov subspace methods deteriorates with increasing wave number, so the need for preconditioning becomes obvious. In this chapter we consider Bi-CGSTAB (see Van der Vorst [22]) and IDR(s) (see Sonneveld, van Gijzen [23]) as the Krylov subspace solvers.

There have been many attempts to find a suitable preconditioner for the Helmholtz equation, see, for example, Gozani et al. [24], Kerchroud et al. [25]. Recently the class of shifted Laplacian preconditioners evolved, see Laird and Giles [26], Turkel [27], Erlangga et al. [28], Erlangga [21]. In this work, we focus on a shifted Laplace multigrid preconditioner introduced in Erlangga, Vuik and Oosterlee [28], Erlangga [70], to improve the convergence of the Krylov subspace methods.

The purpose of this work is to compare the implementations of the Krylov subspace solver preconditioned by the shifted Laplace multigrid method in two dimensions on a CPU and a GPU. The interest is triggered by the fact that some applications on GPUs are 50-200 times faster compared with a CPU implementation (see e.g. Lee at al [71], Nvidia [41]). However there are no recordings of a Helmholtz solver on a GPU which we present in this chapter. There are two main issues: the first one is the efficient implementation of the solver on the GPU and the second one is the behavior of the numerical methods in single precision[1]. Nevertheless, even on a modern graphics card with double precision units (for example, Tesla 20 series or Fermi), single precision calculations are still at least two times faster. The first issue can be resolved by knowing the further details of a GPU and CUDA. The second issue can be addressed by using mixed precision algorithms, see e.g. Baboulin et al. [72].

The chapter is organized as follows. In Section 2 we describe the Helmholtz equation and its discretization. Also the components of the solver are described, including Krylov subspace methods such as Bi-CGSTAB and IDR(s) and the shifted Laplace multigrid method. The specific aspects of the GPU implementation for each method are considered in detail in Section 3 and optimizations for the GPU are suggested. In Section 4 two model problems are defined: with constant and variable wave numbers. We solve those problems with Krylov subspace methods preconditioned by the shifted Laplacian on a single CPU and a single GPU and compare the performance. Finally Section 5 contains conclusions and an outlook of this chapter.

[1] Note that in 2010 the single precision was state-of-the-art.

2.2. PROBLEM DESCRIPTION

The two-dimensional Helmholtz equation 1.1 for a wave problem in a heterogeneous medium is considered.

2.2.1. DISCRETIZATION

The domain Ω is discretized by an equidistant grid Ω_h with the grid size h

$$\Omega_h := \{(ih, jh) \mid i, j = 1, \dots, N\}.$$

For simplicity we set the same grid sizes in $x-$ and $y-$directions. After discretization of equation 1.1 on Ω_h using central finite differences we get the following linear system of equations:

$$A\phi = g, \quad A \in \mathbb{C}^{N\times N}, \quad \phi, g \in \mathbb{C}^N. \tag{2.1}$$

The matrix A is based on the following stencil for inner points $x \in \Omega_h/\partial\Omega_h$:

$$A_h = \frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4-(kh)^2(1-\alpha i) & -1 \\ & -1 & \end{bmatrix}. \tag{2.2}$$

The Dirichlet boundary conditions do not change the matrix elements at boundaries and the matrix will be structured and sparse.

The first-order radiation boundary condition 1.3 is discretized using a one-sided scheme, for example on the right boundary at x_{N+1} the solution can be expressed as

$$\phi_{N+1,j} = \frac{\phi_{N,j}}{1+ikh_x}.$$

The matrix stencils at the boundaries change accordingly.

2.2.2. KRYLOV SUBSPACE METHODS

The discretized matrix A in 2.1 is complex-valued, symmetric, non-Hermitian, i.e. $A^* \neq A$. Moreover, for sufficiently large wave numbers k, the matrix A is indefinite, that means there are eigenvalues of A with a positive real part and eigenvalues with a negative real part. Furthermore, the matrix A is ill-conditioned. The mentioned properties of the matrix are the reason that the classical iterative methods (such as Jacobi, Gauss-Seidel, etc...) simply diverge. However, we may still be able to use them e.g. as smoothers for a multigrid method.

BI-CGSTAB

One of the candidates to solve the discretized Helmholtz equation 2.1 is the Bi-CGSTAB method (see Van der Vorst [22], Saad [33]). The advantage of this method is that it is easily parallelizable. However, even if the Bi-CGSTAB method converges for small wave

numbers k, the convergence is too slow and it strongly depends on the grid size, see Erlangga [29]. The original system of linear equations 2.1 can be replaced by an equivalent preconditioned system:

$$AM^{-1}u = g, \qquad M^{-1}u = \phi, \tag{2.3}$$

where the inverse systems of the form $M\phi = u$ are easy to solve. The matrix AM^{-1} is well-conditioned, so that the convergence of Bi-CGSTAB (and any other Krylov subspace methods) is improved.

As the preconditioner for Bi-CGSTAB we consider the shifted Laplace preconditioner introduced by Erlangga, Vuik and Oosterlee, see [28], [70], [21], which is based on the following operator:

$$\mathscr{M}_{(\beta_1,\beta_2)} = -\Delta - (\beta_1 - i\beta_2)k^2, \qquad \beta_1, \beta_2 \in \mathbb{R}, \tag{2.4}$$

with the same boundary conditions as $\mathscr{A}$ in 1.2. The system 2.1 is then preconditioned by

$$M_{(\beta_1,\beta_2)} = -\Delta_h - (\beta_1 - i\beta_2)k^2 I, \qquad \beta_1, \beta_2 \in \mathbb{R}, \tag{2.5}$$

where Δ_h is the discretized Laplace operator, I is the identity matrix and β_1, β_2 can be chosen optimally. Depending on β_1 and β_2, the spectral properties of the matrix AM^{-1} change. In Erlangga, Oosterlee, Vuik [21] Fourier analysis shows that $M_{(\beta_1,\beta_2)}$ as given in 2.5 with $\beta_1 = 1$ and $0.4 \leq \beta_2 \leq 1$ gives rise to favorable properties that leads to considerably improved convergence of Krylov subspace methods (e.g. Bi-CGSTAB), see also van Gijzen, Erlangga, Vuik [73].

IDR(s)

An alternative to Bi-CGSTAB to solve large non-symmetric linear systems of equations 2.1 is the IDR(s) method, which was proposed by Sonneveld, van Gijzen [23]. IDR(s) belongs to the family of Krylov subspace methods and it is based on the Induced Dimension Reduction (IDR) method introduced by Sonneveld, see e.g. Wesseling and Sonneveld [74]. IDR(s) is a memory-efficient method to solve large non-symmetric systems of linear equations.

We are currently using the IDR(s) variant described in van Gijzen and Sonneveld [73]. This method imposes a bi-orthogonalization condition on the iteration vectors, which results in a method with fewer vector operations than the original IDR(s) algorithm. It has been shown in van Gijzen and Sonneveld [73] that this IDR(s) and the original IDR(s) yield the same residual in exact arithmetics. However the intermediate results and numerical properties are different. The IDR(s) with bi-orthogonalization converges slightly faster than the original IDR(s).

Another advantage of the IDR(s) algorithm with bi-orthogonalization is that it is more accurate than the original IDR(s) for large values of s. For $s = 1$ the IDR(1) will be equivalent to Bi-CGSTAB. Usually s is chosen smaller than 10. In our experiments we set $s = 4$, since this choice is a good compromise between storage and performance, see Umetani, MacLachlan, Oosterlee [75].

The preconditioned IDR(s) method can be found in van Gijzen and Sonneveld [73] and is given in Algorithm 1.

Require: $A \in \mathbb{C}^{N\times N}$; $\mathbf{x}, \mathbf{b} \in \mathbb{C}^N$; $Q \in \mathbb{C}^{N\times s}$; $\epsilon \in (0,1)$;
Ensure : $\mathbf{x}_n$ such that $\|\mathbf{b} - A\mathbf{x}_n\| \leq \epsilon$

Calculate $\mathbf{r} = \mathbf{b} - A\mathbf{x}$;
$\mathbf{g}_i = \mathbf{u}_i = 0,\ i = 1,\ldots,s$; $\bar{M} = I$; $\omega = 1$;
while $\|r\| > \epsilon$ **do**
  $\mathbf{f} = Q^H\mathbf{r}$, $\mathbf{f} = (\phi_1,\ldots,\phi_s)$;
  for *k=1:s* **do**
    Solve $\mathbf{c}$ from $\bar{M}\mathbf{c} = \mathbf{f}, \mathbf{c} = (\gamma_1,\ldots,\gamma_s)^T$;
    $\mathbf{v} = \mathbf{r} - \sum_{i=k}^{s} \gamma_i \mathbf{g}_i$;
    $\mathbf{v} = M^{-1}\mathbf{v}$;
    $u_k = \omega\mathbf{v} + \sum_{i=k}^{s} \gamma_i \mathbf{u}_i$;
    $\mathbf{g}_k = A\mathbf{u}_k$;
    for *i=1:k-1* **do**
      $\alpha = \frac{\mathbf{q}_i^H \mathbf{g}_k}{\mu_{i,i}}$;
      $\mathbf{g}_k = \mathbf{g}_k - \alpha\mathbf{g}_i$;
      $\mathbf{u}_k = \mathbf{u}_k - \alpha\mathbf{u}_i$;
    end
    $\mu_{i,k} = \mathbf{q}_i^H \mathbf{g}_k,\ i = k,\ldots,s,\ \bar{M}_{i,k} = \mu_{i,k}$;
    $\beta = \frac{\phi_k}{\mu_{k,k}}$;
    $\mathbf{r} = \mathbf{r} - \beta\mathbf{g}_k$;
    $\mathbf{x} = \mathbf{x} + \beta\mathbf{u}_k$;
    if $k + 1 \leq s$ **then**
      $\phi_i = 0, i = 1,\ldots,k$;
      $\phi_i = \phi_i - \beta\mu_{i,k}, i = k+1,\ldots,s$;
      $\mathbf{f} = (\phi_1,\ldots,\phi_s)^T$;
    end
  end
  $\mathbf{v} = M^{-1}\mathbf{r}$;
  $\mathbf{t} = A\mathbf{v}$;
  $\omega = (\mathbf{t}^H\mathbf{r})/(\mathbf{t}^H\mathbf{t})$;
  $\mathbf{r} = \mathbf{r} - \omega\mathbf{t}$;
  $\mathbf{x} = \mathbf{x} + \omega\mathbf{v}$;
end

Algorithm 1: Preconditioned IDR(s)

2.2.3. SHIFTED LAPLACE MULTIGRID PRECONDITIONER

When $A\phi = g$ in 2.1 is solved using the standard multigrid method, then severe conditions on the smoother and the coarse grid correction must be met. For the smoother the conditions are:

- k^2 must be smaller than the smallest eigenvalue of the Laplacian;
- The coarsest level must be fine enough to keep the representation of the smooth vectors.

Furthermore, the standard multigrid method may not converge in case k^2 is close to an eigenvalue of M. This issue can be resolved by using subspace correction techniques within multigrid (see Elman et al. [76]).

Because of the above reasons, we do not use multigrid as a solver. Instead, we choose a complex-valued generalization of the matrix-dependent multigrid method by de Zeeuw [77] as a preconditioner, as shown in 2.3. It provides an h-independent convergence factor in the preconditioner, as shown in Erlangga, Oosterlee, Vuik [21].

In the coarse grid correction phase, the Galerkin method is used in order to get coarse grid matrices:

$$M_H = RM_hP, \tag{2.6}$$

where M_H and M_h are matrices on the coarse and fine grids, respectively, P is prolongation and R is restriction. The prolongation P is based on the matrix-dependent prolongation, described in de Zeeuw [77] for real-valued matrices. Since the matrix M_h is a complex symmetric matrix, the prolongation is adapted for this case, see Erlangga [29]. This prolongation is also valid at the boundaries.

The restriction R is chosen as full weighting restriction and not as adjoint of the prolongation. This choice of the transfer operators and Galerkin coarse grid discretization brings a robust convergence for several complex-valued Helmholtz problems, see Erlangga, Oosterlee, Vuik [21].

Classical iterative methods in general do not converge for the Helmholtz equation, but we can apply them as smoothers for the multigrid method. We consider a parallel version of the Gauss-Seidel method as the smoother, the so-called multi-colored Gauss-Seidel smoother. In the 2D case we use 4 colors, where the neighbors of a grid point do not have the same color.

2.3. IMPLEMENTATION ON GPU

2.3.1. CUDA

As described in the previous section, we solve the discretized Helmholtz equation 2.1 with Bi-CGSTAB and IDR(s) preconditioned by the shifted Laplace multigrid method, where the multi-color Gauss-Seidel method is used as a smoother. Those algorithms are parallelizable and therefore can be implemented on the GPU architecture.

For our GPU computations we use the CUDA library (version 3.1) developed by NVIDIA. CUDA supports C++, Java and Fortran languages with some extensions. In this work we

use C++. CUDA offers various libraries out of the box such as CUFFT for Fast Fourier Transforms and CUBLAS for Basic Linear Algebra Subprograms.

2.3.2. VECTOR AND MATRIX OPERATIONS ON GPU

The preconditioned Bi-CGSTAB and IDR(s) algorithms consist of 4 components: the preconditioner and 3 operations on complex numbers: dot (or inner) product, matrix-vector multiplication and vector addition. In this section we compare those operations on the GPU with a CPU version. The preconditioner is considered in Section 2.3.3.

Let us first consider the dot product. On the GPU we are using the dot product from CUBLAS library, that follows the IEEE 757 standard. The dot product on the CPU needs more investigation, since there is no correct open source version of the dot product BLAS subroutine, to the author's knowledge. The simplest algorithm, given by

$$(u, v) = \sum_{i=1}^{N} \bar{u}_i v_i, \tag{2.7}$$

is not accurate for large N. The loss of accuracy becomes visible especially in single precision if we add a very small number to a very large one.

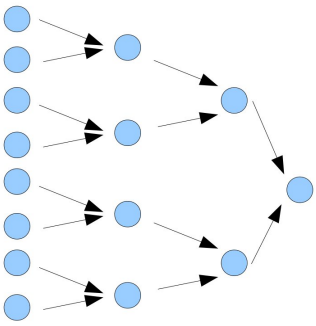


Figure 2.1: Original recursive dot product algorithm.

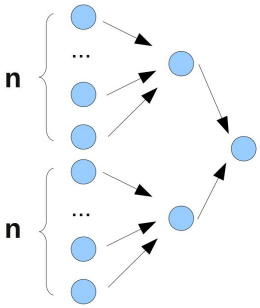


Figure 2.2: Modified recursive dot product algorithm, where $n = 1000$.

To increase the precision we developed a recursive algorithm as shown in Figure 2.1. The main idea is to add numbers that have approximately the same order of magnitude. If we assume that two consecutive numbers have indeed the same order of magnitude, then summing them will be done with optimal accuracy. Recursively, two sums should also have the same magnitude and the steps can be applied again. However this operation has a significant impact on the performance as it does not take advantage of the CPU cache. To solve this problem, the recursive algorithm is not applied to the finest level, instead we add 1000 floating point numbers according to 2.7, see Figure 2.2. Our experiments show that adding batches of 1000 single precision floating numbers is fast and accurate. Our test machine has 512 Kb of CPU memory cache, but we do not see any performance improvement beyond 1000 numbers, so in our case this number is a good compromise between speed and accuracy. The computational time of this version

of the dot product is almost the same as of the inaccurate initial algorithm. The results are presented in Table 2.1. Note that memory transfers CPU-GPU are not included in these timings.

n	CPU Time (ms)	GPU Time (ms)	Time CPU/time GPU
1,000	0.04	0.04	1.3
10,000	0.23	0.03	6
100,000	2.35	0.06	59
1,000,000	23.92	0.32	75
10,000,000	231.48	2.9	80

Table 2.1: Single precision dot product for different sizes on CPU and GPU, data transfers are excluded (status 2010).

For the vector addition we use a CUBLAS function for complex vectors, that follows the IEEE 757 standard. The comparisons between CPU and GPU performance are given in Table 2.2. Note that on the CPU a vector addition is about 4 times faster than the dot product operation. The dot product requires 3 times more floating point operations (flops) than addition:

$$(\mathbf{a}+i\mathbf{b})+(\mathbf{c}+i\mathbf{d})=\mathbf{a}+\mathbf{c}+i(\mathbf{b}+\mathbf{d}),$$
$$(\mathbf{a}+i\mathbf{b})\cdot(\mathbf{c}+i\mathbf{d})=\mathbf{a}\cdot\mathbf{c}-\mathbf{b}\cdot\mathbf{d}+i(\mathbf{b}\cdot\mathbf{c}+\mathbf{a}\cdot\mathbf{d})$$

Moreover the recursive algorithm for the dot product has a non-linear memory access pattern, which can result in cache misses and impacts performance. Having batches of 1000 consecutive numbers, as described above, minimizes the non-linearity so that memory cache misses are kept to a minimum.

On our processor (see the detailed description in Section 2.4.1) the assembly instructions for addition and multiplication (fmull and fadd) have the same timings, see [78].

n	CPU Time (ms)	GPU Time (ms)	Time CPU/time GPU
1,000	0.01	0.03	$\frac{1}{3}$
10,000	0.08	0.01	8
100,000	0.58	0.09	6
1,000,000	6.18	0.42	12
10,000,000	58.06	4.41	13

Table 2.2: Single precision vector additions for different sizes on CPU and GPU, data transfers are excluded (status 2010).

CUBLAS provides a full matrix-vector multiplication which in our case is not useful since our matrices are very large and sparse. For this reason we opted for a compressed row storage (CRS) scheme in this chapter and implemented the complex sparse matrix-vector multiplication on the GPU. The comparisons for matrix-vector multiplication on a single CPU and a GPU are given in Table 2.3.

n	CPU Time (ms)	GPU Time (ms)	Time CPU/time GPU
1,000	0.38	0.13	3
10,000	3.82	0.28	14
100,000	38.95	1.91	20
1,000,000	390.07	18.27	21

Table 2.3: Single precision matrix vector multiplication for different sizes on CPU and GPU, data transfers are excluded

In Tables 2.1, 2.2 and 2.3, it can be clearly seen that the speedup increases with growing size. If the size of the problem is small, the so-called overhead time (allocations, data transfer, etc) becomes significant compared to the computational time. The best performance on the GPU is achieved by full occupancy of the processors, see Nvidia [79].

2.3.3. MULTIGRID METHOD ON GPU

The algorithm for the multigrid preconditioner is split into two phases: generation of transfer operators and coarse-grid matrices (setup phase) and the actual multigrid solver.

The transfer operators will remain unchanged during the program execution and the speed of the setup phase is not crucial. Operations like sparse matrix multiplications are performed in that phase. The setup phase is done on the CPU, taking advantage of the double precision.

The setup phase is executed only once at the beginning. Furthermore, it has some sequential elements, for example a coarse grid matrix can be constructed only knowing the matrix and transfer operators on the finer level. The first phase is implemented in double precision on the CPU and is later converted to single precision and the matrices are transferred to the GPU. The second phase consists mainly of the same three operations as in the Bi-CGSTAB algorithm: dot product, vector addition and matrix-vector multiplication, including a smoother: damped Jacobi, multi-colored Gauss-Seidel (4 colors), damped multi-colored Gauss-Seidel iteration (4 colors, parallel version of SOR, see Golub and Van Loan [80]). Note that we chose a parallelizable smoother which can be implemented on the GPU. The second phase is implemented on the GPU.

As a solver on the coarsest grid we have chosen to perform a couple of smoothing iterations instead of an exact solver. That allows us to keep the computations even on the coarsest level on the GPU and save time for transferring data between GPU and CPU.

The timings for the multigrid method on a CPU and GPU without CPU-GPU data transfers are presented in the Table 2.4. It is easy to see that the speedup again increases with increasing problem size. The reason for this is that for problems with smaller size the so-called overhead part (e.g. array allocation) is significant compared to the actual computations (execution of arithmetic operations). Again, a GPU gives better performance in case of full occupancy of the processors (see Nvidia [79]).

N	k	Time CPU (s)	Time GPU (s)	Time CPU/time GPU
64	40	0.008	0.0074	1.15
128	80	0.033	0.009	3.48
512	320	0.53	0.03	17.56
1024	640	2.13	0.08	26.48

Table 2.4: One F-cycle of the multigrid method on CPU and GPU for a 2D Helmholtz problem with various k, $kh = 0.625$.

2.3.4. ITERATIVE REFINEMENT

In 2010, double precision arithmetic units was not mainstream for GPGPU hardware. To improve the precision, the Iterative Refinement algorithm (IR or Mixed Precision Iterative Improvement as referred by Golub, Van Loan [80]) can be used where double precision arithmetic is executed on the CPU, and single precision on the GPU, see Algorithm 2. This technique has been already applied to GMRES methods and direct solvers (see Baboulin et al. [72]) and to Karzcmarz's and other iterative methods (see Elble et al. [81]).

Double Precision: b, x, r, ϵ
Single Precision : $\hat{A}, \hat{r}, \hat{e}$

while $\|r\| > \epsilon$ **do**
  $r = b - Ax$;
  Convert r in double precision to $\hat{r}$ in single precision ;
  Convert A in double precision to $\hat{A}$ in single precision ;
  Solve $\hat{A}\hat{e} = \hat{r}$ on GPU ;
  $x = x + \hat{e}$
end

Algorithm 2: Iterative refinement

The measured time and number of iterations on a single GPU and single CPU are given in Table 2.5. The stopping criterion ϵ for the outer loop is set to 10^{-6}. In this experiment, the tolerance of the inner solver (Bi-CGSTAB) is set to 10^{-3}. The results show that IR requires approximately 2 times more iterations in total, however as Bi-CGSTAB on the GPU is much faster than on the CPU, the overall performance of IR is better. This experiment proves that a GPU can successfully be used as an accelerator even with single precision.

2.3.5. GPU OPTIMIZATIONS

A GPU contains many cores, thus even without optimization it is easy to obtain relatively good performance compared to a CPU. Nvidia's programming guide [79] helps to achieve optimal performance when a number of optimizations are employed, that are

Method	Total #iter	Total time (s)	Accuracy
Bi-CGSTAB (1 CPU)	6047	370.6	9.7e-7
IR with Bi-CGSTAB (GPU)	11000	27.1	9.6e-7

Table 2.5: Convergence of iterative refinement with Bi-CGSTAB (double precision) for $k = 40$, (256×256), $kh < 0.625$. The stopping criterion for the preconditioner on CPU is $\|r\|/\|r_0\| < 10^{-6}$ and on GPU is $\|r\|/\|r_0\| < 10^{-3}$.

listed below:

- **Memory transfer**
  Our implementation minimizes memory transfers between CPU and GPU. Once a vector has been copied to the GPU, it remains in the GPU memory during the solver life time.

- **Memory coalescing**
  Accessing the memory in a coalesced way means that consecutive threads access consecutive memory addresses. This access pattern is not trivial to implement. Maximum memory access performance is achieved when threads within a warp access the same memory block for our hardware with compute capability 1.1[2]. By the construction of the algorithms on a GPU, especially for the sparse matrix-vector multiplication, this has been taken into account.

- **Texture memory**
  Texture memory optimization is an easy way to improve performance significantly, because the texture memory is cached. Read-only memory can be bound to a texture. In our case the matrix A does not change during iterative solution and therefore can be put into the texture memory.

- **Constant memory**
  Constant memory is as fast as shared memory but is read-only during kernel execution. In our case constant memory is used to store small amounts of data that will be read many times, but do not require to be stored in registers (such as array lengths, matrix dimensions, etc.).

- **Registers**
  Registers are the fastest memory on a GPU but their amount is very limited. For example, in our graphics card (NVIDIA GeForce 9800 GTX/9800 GTX+) the number of registers used in a block can not exceed 8192, otherwise the global memory will be used to offload registers, which results in performance degradation. The CUDA compiler has an option to display the number of registers used during execution. To maximize occupancy the maximum number of registers is set at 16. The CUDA occupancy calculator is a useful tool to compute the number of registers. During compilation, the CUDA compiler displays the number of registers used per kernel so it can be checked that this number is indeed not higher than 16.

[2] The compute capability indicates certain features and specifications for a given GPU. The higher the number, the more features are available by using the GPU.

2.4. NUMERICAL EXPERIMENTS

For the experiments the following 2D model problems have been selected.

MP1 : Constant wave number

For $0 < \alpha << 1$ and $k = \text{const}$, find $\phi \in \mathbb{C}^{N\times N}$

$$-\Delta\phi(x,y) - (1-\alpha i)k^2\phi(x,y) = \delta\left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right), \tag{2.8}$$

$(x,y) \in \Omega = [0,1]\times[0,1]$, with the first-order boundary conditions 1.3. In our experiments in this section we assume that $\alpha = 0$ which is the most difficult case. The grid sizes for different k satisfy the condition $kh = 0.625$, where $h = \frac{1}{N-1}$.

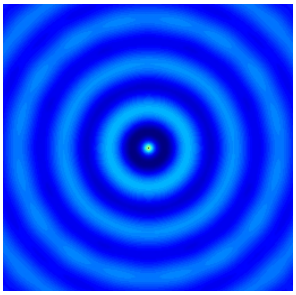


Figure 2.3: The solution of the model problem **MP1** 2.8 for $k = 40$.

MP2 : Wedge problem

This model problem represents a layered heterogeneous problem taken from Plessix and Mulder [82]. For $\alpha \in \mathbb{R}$ find $\phi \in \mathbb{C}^{N\times N}$

$$-\Delta\phi(x,y) - (1-\alpha i)k(x,y)^2\phi(x,y) = \delta(x-500)(y), \tag{2.9}$$

$(x,y) \in \Omega = [0,1000]\times[0,1000]$, with the first order boundary conditions 1.3. The coefficient $k(x,y)$ is given by $k(x,y) = 2\pi f l/c(x,y)$ where $c(x,y)$ is presented in Figure 2.4. The grid size satisfies the condition $\max_x(k(x))h = 0.625$, where $h = \frac{1}{N-1}$. The solution for the model problem 2.9 for $f = 30$ Hz is given in Figure 2.5.

2.4.1. HARDWARE AND SOFTWARE SPECIFICATIONS

Before we start to describe the convergence and timing results, it is necessary to detail the hardware specifications. The experiments have been run on an AMD Phenom$^{(tm)}$ 9850 Quad-Core Processor, 2.5 GHz with 8 GB memory. Further we refer either to a single CPU or to a Quad-Core (4 CPUs). The compiler on the CPU is gcc 4.3.3. The graphics card is an NVIDIA GeForce 9800 GTX/9800 GTX+, compute capability 1.1, 128 cores, 512 MB global memory, clock rate 1.67 GHz. The code on the GPU has been compiled with NVIDIA CUDA 3.1[1].

[1] On different hardware we observed no significant differences in execution times between CUDA 3.1 and CUDA 3.2 for our software.

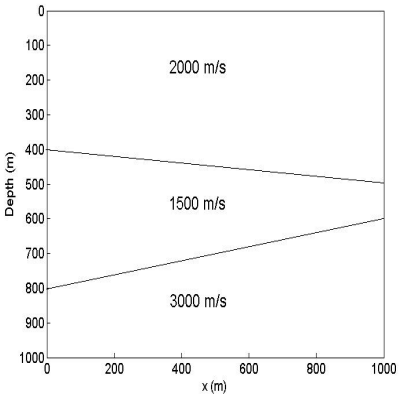


Figure 2.4: The velocity profile of the wedge problem **MP2**.

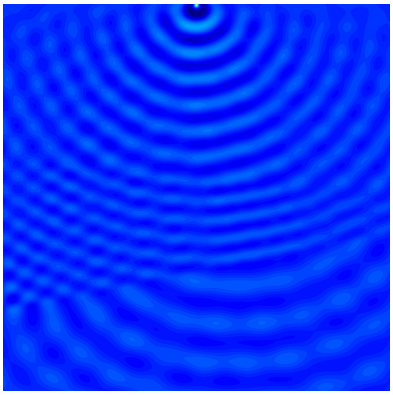


Figure 2.5: Real part of solution of the wedge problem **MP2**, $f = 30$ Hz.

2.4.2. BI-CGSTAB AND IDR(s)

We first consider the timings characteristics of the Bi-CGSTAB and IDR(s) methods described in Section 2.2.2 and compare their performance on a single-threaded CPU against a GPU implementation. In both cases single precision has been used, see Table 2.6. We have chosen the number of iterations equal to 100, because even for relatively small wave numbers like $k = 40$ Bi-CGSTAB and IDR(s) do not converge. To achieve convergence a preconditioner should be used, see Section 2.4.3. For IDR(s) we use $s = 4$ normally distributed random vectors, that are orthogonalized by the Gram-Schmidt orthogonalization technique.

N	Bi-CGSTAB			IDR(s)		
	t_{CPU}(s)	t_{GPU}(s)	Speedup	t_{CPU}(s)	t_{GPU}(s)	Speedup
64	0.5	0.07	7.8	1.3	0.36	3.7
128	2.3	0.1	21.1	5.3	0.47	11.3
256	8.9	0.2	35.8	23.4	0.89	26.1
512	33.0	0.9	35.3	124.5	2.6	47.8
1024	130.4	3.2	40.6	363.2	9.5	38.3

Table 2.6: Timing for 100 iterations of Bi-CGSTAB and IDR(s), $s = 4$, for $k = 40$ and different grid sizes.

As it is shown in Table 2.6, the speedups for Bi-CGSTAB and IDR(s) on GPU are comparable. It means that IDR(s) is parallelizable and suitable for the GPU in a similar way as Bi-CGSTAB is. Note that the timings for IDR(s) on the CPU ($s = 4$) are approximately three times slower than the timings for Bi-CGSTAB on the CPU. The reason for this is that the IDR(4) algorithm has 5 SpMVs (Sparse Matrix-Vector-Products) and dot-products per iteration and Bi-CGSTAB has only 2 of them per iteration, which gives a factor 2.5. More-

over, in the current implementation the preconditioned IDR(s) algorithm is applied (see Algorithm 1 or Sonneveld, van Gijzen [23]), where as the "preconditioner" the identity matrix is used. With additional optimizations of the current implementation, the factor 2.5 between Bi-CGSTAB and IDR(s) ($s = 4$) on the CPU can be achieved.

2.4.3. PRECONDITIONED KRYLOV SUBSPACE METHODS

BI-CGSTAB PRECONDITIONED BY SHIFTED LAPLACE MULTIGRID

Since neither Bi-CGSTAB nor IDR(s) converges as a stand-alone solver not even for low wave numbers, a preconditioner is required to improve the convergence properties. As the preconditioner we apply the shifted Laplace multigrid method described in Section 2.4.3. Parameter β_1 is set to 1. The idea is to choose a combination of the parameter β_2 and the relaxation parameter ω for the multigrid smoother that the number of iterations of the Krylov subspace methods will be reduced. Several smoothers are considered: damped Jacobi (ω-Jacobi), multi-colored Gauss-Seidel and damped multi-colored Gauss-Seidel (ω-Gauss-Seidel). Two-dimensional convergence results, in dependence of parameter β_2, for problem size $N = 1024$ are given in Figure 2.6. The results have been computed using Bi-CGSTAB in double precision on CPU with the preconditioner on a GPU. It can be clearly seen that using the damped multi-colored Gauss-Seidel iteration ($\omega = 0.9$) as a smoother and $\beta_2 = 0.6$ in the shifted Laplace multigrid preconditioner gives the minimum number of iterations of Bi-CGSTAB. In our further experiments those parameters are applied.

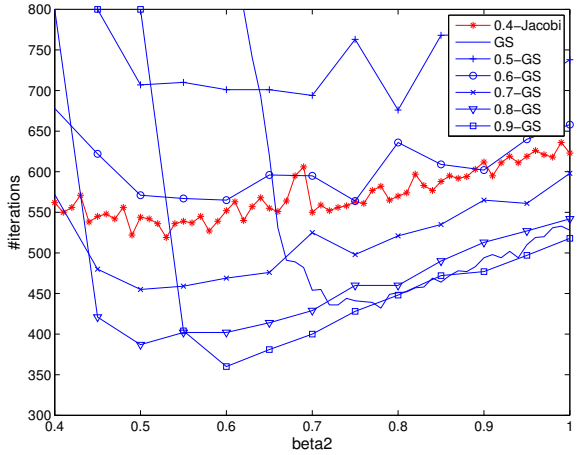


Figure 2.6: Comparison of number of iterations of Bi-CGSTAB preconditioned by shifted Laplace multigrid preconditioner with various β_2 and different smoothers: damped Jacobi ($\omega = 0.4$), Gauss-Seidel and damped Gauss-Seidel iteration ($\omega = 0.5, 0.6, 0.7, 0.8, 0.9$).

By solving the **MP1** 2.8 with Bi-CGSTAB preconditioned by one F-cycle of the shifted Laplace multigrid method, the following results have been achieved, see Table 2.7. The parameters are $\beta_1 = 1$, $\beta_2 = 0.6$. As a smoother the damped multi-colored Gauss-Seidel

iteration with $\omega = 0.9$ has been applied (4 colors). The grid sizes for different k satisfy the condition $kh = 0.625$. The implementation on the CPU is in double precision whereas on the GPU it is in single precision. Note that the number of iterations on the CPU and GPU is comparable. A stopping criterion of $\|r\|/\|r_0\| = 10^{-3}$ allows Bi-CGSTAB to converge on CPU and GPU so that performance comparisons can be made.

N	k	Bi-CGSTAB(CPU) +MG(CPU)		Bi-CGSTAB(CPU) +MG(GPU)			Bi-CGSTAB(GPU) +MG(GPU)		
		#iter	time(s)	#iter	time(s)	Speedup	#iter	time(s)	Speedup
64	40	12	0.3	12	0.2	1.5	12	0.2	1.3
128	80	21	1.9	21	0.9	2.1	21	0.5	3.8
256	160	40	15.1	40	4.4	3.4	40	1.5	10.03
512	320	77	115.1	78	28.6	4.0	72	5.3	21.7
1024	640	151	895.1	160	218.8	4.1	157	31.1	28.8

Table 2.7: Timing comparisons of Bi-CGSTAB preconditioned by one F-cycle of the shifted Laplace multigrid method on a single CPU and a single GPU for **MP1** 2.8, $kh = 0.625$. The combination of methods on a single CPU (double precision) and a single GPU (single precision) is also presented.

In Table 2.7 we compare the preconditioned solver on a CPU and a GPU as well as the solver on the CPU preconditioned by the multigrid method implemented on the GPU. The idea behind this last implementation is to use the GPU as an accelerator for the preconditioner while we can maintain accuracy by the double precision solver on the CPU. However in this case the data transfer (residual and solution) between CPU and GPU reduces the benefits of the GPU implementation. If all data stays on the GPU as for Bi-CGSTAB preconditioned by multigrid on GPU, we find a much better speedup.

We also solved the wedge problem **MP2** 2.9 with Bi-CGSTAB on the CPU preconditioned by the shifted Laplace multigrid method on the GPU. The parameters in the preconditioner are $\beta_1 = 1$ and $\beta_2 = 0.6$. The relaxation parameter for the multicolored Gauss-Seidel iteration is $\omega = 0.9$. The results are shown in Table 2.8. As stopping criterion $\|r\|/\|r_0\| = 10^{-3}$ is used.

N	Bi-CGSTAB(CPU) +MG(CPU)		Bi-CGSTAB(CPU) +MG(GPU)			Bi-CGSTAB(GPU) +MG(GPU)		
	#iter	time (s)	#iter	time (s)	Speedup	#iter	time (s)	Speedup
1024	27	155	38	57	2.7	26	5.6	27.7

Table 2.8: Convergence of Bi-CGSTAB preconditioned by one F-cycle of the shifted Laplace multigrid method on a single CPU and a single GPU for **MP2** 2.9, $\max(k)h = 0.625$. Note that here Bi-CGSTAB on the CPU is in double precision.

IDR(s) PRECONDITIONED BY SHIFTED LAPLACE MULTIGRID

By solving **MP1** 2.8 with IDR(s) preconditioned by one F-cycle of the shifted Laplace multigrid method, the following results have been achieved, see Table 2.9. We use $s = 4$

normally distributed random vectors, which are orthogonalized by the Gram-Schmidt orthogonalization technique. As the smoother a multi-colored Gauss-Seidel iteration has been applied. In order to obtain an optimal number of iterations for IDR(s), we have used the same procedure to find optimal $\beta_2 = 0.65$ and $\omega = 0.9$, as described in Section 2.4.3. The grid sizes for different k satisfy the condition $kh = 0.625$. The implementation on the CPU is in double precision and on the GPU it is in single precision. As stopping criterion again $\|r\|/\|r_0\| = 10^{-3}$ is used.

N	k	IDR(s)(CPU) +MG(CPU)		IDR(s)(CPU) +MG(GPU)			IDR(s)(GPU) +MG(GPU)		
		#iter	time (s)	#iter	time (s)	Speedup	#iter	time (s)	Speedup
64	40	6	0.36	6	0.32	1.1	6	0.27	1.3
128	80	10	2.3	10	1.04	2.2	10	0.7	3.7
256	160	17	15.8	17	4.65	3.3	18	1.7	9.1
512	320	33	126.7	34	33.1	3.8	33	6.1	20.6
1024	640	69	1061.8	68	252.2	4.2	73	37.3	28.5

Table 2.9: Timing comparisons of IDR(s) preconditioned by one F-cycle of the shifted Laplace multigrid method on a single CPU and a single GPU for **MP1** 2.8, $kh = 0.625$. The combination of methods on a single CPU (double precision) and a single GPU (single precision) is also presented.

The convergence curves of IDR(s) compared with Bi-CGSTAB are shown in Figure 2.7, where the horizontal axis represents the number of iterations. In this case we use Krylov subspace methods in double precision on a CPU with the preconditioner on a GPU. One iteration of Bi-CGSTAB contains 2 SpMVs (short for Sparse Matrix-Vector-Multiplication), whereas one iteration of IDR(4) contains 5 SpMVs. From Figure 2.7 it is easy to see that the total number of SpMVs for Bi-CGSTAB and IDR(4) is the same. The fact that the IDR(s) method converges in fewer iterations, but has more SpMVs, implies that the total performance of IDR(4) and Bi-CGSTAB is approximately the same.

We solve the wedge problem **MP2** 2.9 with the IDR(s) on the CPU preconditioned by the shifted Laplace multigrid method on the GPU. The parameters in the preconditioner are $\beta_1 = 1$ and $\beta_2 = 0.65$. The relaxation parameter for the multicolored Gauss-Seidel iteration is $\omega = 0.9$. The same stopping criterion $\|r\|/\|r_0\| = 10^{-3}$ is used. The results are shown in Table 2.10.

N	IDR(s)(CPU) +MG(CPU)		IDR(s)(CPU) +MG(GPU)			IDR(s)(GPU) +MG(GPU)		
	#iter	time (s)	#iter	time (s)	Speedup	#iter	time (s)	Speedup
1024	11	162.9	12	43.9	3.7	12	6.3	25.7

Table 2.10: Convergence of IDR(s) preconditioned by one F-cycle of the shifted Laplace multigrid method on a single CPU and a single GPU for **MP2** 2.9, $\max(k)h = 0.625$. Note that here the IDR on the CPU is in double precision.

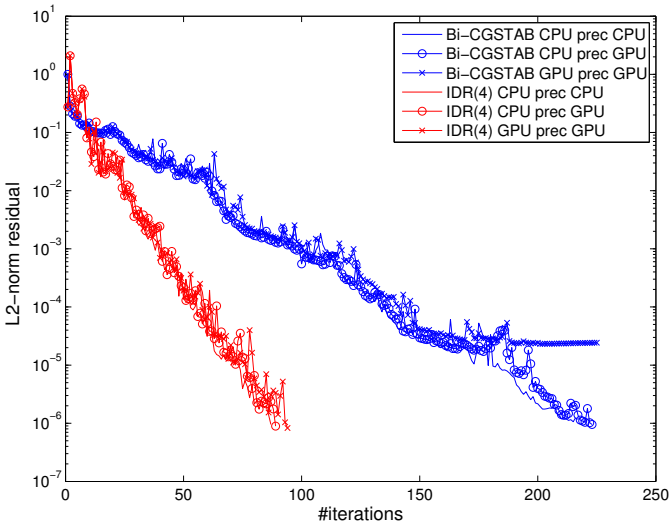


Figure 2.7: Convergence curves of IDR(4) and Bi-CGSTAB for the model problem **MP1** 2.8 for $k = 320$.

2.5. CONCLUSIONS

In this chapter we have presented a GPU implementation of Krylov subspace solvers preconditioned by a shifted Laplace multigrid preconditioner for the two-dimensional Helmholtz equation. On the CPU, double precision was used whereas on the GPU computations were in single precision. We have seen that Bi-CGSTAB and IDR(s) are parallelizable on a GPU and have similar speedups of about 40 compared to a single-threaded CPU implementation.

It has been shown that a matrix-dependent multigrid can be implemented efficiently on a GPU where a speedup of 20 can be achieved for large problems. As the smoother we have considered parallelizable methods such as weighted Jacobi (ω-Jacobi), multi-colored Gauss-Seidel and damped multi-colored Gauss-Seidel iteration (ω-GS). Parameter $\beta_2 = 0.6$ in the preconditioner is optimal for damped multi-colored Gauss-Seidel smoother with $\omega = 0.9$. With those parameters, the number of iterations is optimal for Bi-CGSTAB.

For IDR(s) the optimal parameters were $\beta_2 = 0.65$ and $\omega = 0.9$. One iteration of preconditioned IDR(s) is more intensive than one iteration of preconditioned Bi-CGSTAB, however IDR(s) needs fewer iterations so it does not affect the total computation time.

To increase the precision of a solver, iterative refinement has been considered. We have shown that iterative refinement with Bi-CGSTAB on a GPU is about 4 times faster than Bi-CGSTAB on a CPU for the same stopping criterion. The same result has been achieved for IDR(s). Moreover, combinations of Krylov subspace solvers on the CPU and GPU and the shifted Laplace multigrid preconditioner on the CPU and GPU are considered. A GPU Krylov subspace solver with a GPU preconditioner give the best speedup. For example for the problem size $n = 1024 \times 1024$ Bi-CGSTAB on the GPU with the GPU

preconditioner as well as IDR(s) on the GPU with the GPU preconditioner are about 30 times faster than the analogous solvers on the CPU.

3

3D Helmholtz Krylov Solver Preconditioned by a Shifted Laplace Multigrid Method on Multi-GPUs

Abstract

We focus on an iterative solver for the three-dimensional Helmholtz equation on a multi-GPU architecture using CUDA (Compute Unified Device Architecture). The Helmholtz equation discretized by a second-order finite difference scheme is solved with Bi-CGSTAB preconditioned by a shifted Laplace multigrid method. Two multi-GPU approaches are considered: data parallelism and split of the algorithm. Their implementations on the multi-GPU architecture are compared to a multi-threaded CPU and single GPU implementation. The results show that the data parallel implementation is suffering from communication between GPUs and CPU, but is still a number of times faster compared to many-cores. The split of the algorithm across GPUs requires less communication and delivers speedups comparable to a single GPU implementation.

3.1. Introduction

As it has been shown in the previous chapter the implementation of numerical solvers for 2-D indefinite Helmholtz problems with spatially dependent wavenumbers, such as

Parts of this chapter have been published in H. Knibbe, C. Vuik, and C. W. Oosterlee. In A. Cangiani, R. L. Davidchack, E. Georgoulis, A. N. Gorban, J. Levesley, and M. V. Tretyakov, editors, in Proceedings of ENUMATH 2011, the 9th European Conference on Numerical Mathematics and Advanced Applications, Leicester, September 2011, pages 653–661. Springer-Verlag Berlin Heidelberg, 2013, [83].

Bi-CGSTAB and IDR(s) preconditioned by a shifted Laplace multigrid method on a GPU is more than 25 times faster than on a single CPU. Comparison of a single GPU to a single CPU is important but it is not representative for problems of realistic size. By realistic problem sizes we mean three-dimensional problems which lead after discretization to linear systems of equations with more than one million unknowns. Such problems arise when modeling a wavefield in geophysics.

Problems of realistic size are too large to fit in the memory of one GPU, even with the latest NVIDIA Fermi graphics card (see [41]) in 2012. One solution is to use multiple GPUs. A widely used architecture in 2012 consists of a multi-core machine connected to one or at most two GPUs. Moreover, in most of the cases those GPUs have different characteristics and memory size. A setup with four or more identical GPUs is rather uncommon, but it would be ideal from a memory point of view. It implies that the maximum memory would be four times or more compared to a single GPU. However GPUs are connected to a PCI bus and in some cases two GPUs share the same PCI bus, which creates data transfer limitation. To summarize, using multi-GPUs increases the total memory size but data transfer problems appear.

The aim of this chapter is to consider different multi-GPU approaches and understand how data transfer affects the performance of a Krylov subspace solver with the shifted Laplace multigrid preconditioner for the three-dimensional Helmholtz equation.

3.2. HELMHOLTZ EQUATION AND SOLVER

The Helmholtz equation in three dimensions 1.1 in a heterogeneous medium is considered. Discretizing equation 1.1 using the 7-point central finite difference scheme gives the following linear system of equations: $A\phi = g$, $A \in \mathbb{C}^{N\times N}$, $\phi, g \in \mathbb{C}^N$, where $N = n_x n_y n_z$ is a product of the number of discretization points in the $x-$, $y-$ and $z-$directions. Note that the closer the damping parameter α is set to zero, the more difficult it is to solve the Helmholtz equation. We focus on the original Helmholtz equation with $\alpha = 0$ here.

As the solver for the discretized Helmholtz equation we have chosen the Bi-CGSTAB method preconditioned by the shifted Laplace multigrid method with matrix-dependent transfer operations and a Gauss-Seidel smoother, (see [21]). It has been shown in the previous chapter that this solver is parallelizable on CPUs as well as on a single GPU and provides good speed-up on parallel architectures. The prolongation in this work is based on the three dimensional matrix-dependent prolongation for real-valued matrices described in [84]. This prolongation is also valid at the boundaries. The restriction is chosen as full weighting restriction. As the smoother the multi-colored Gauss-Seidel method has been used. In particular, for 3D problems the smoother uses 8 colors, so that the color of a given point will be different from its neighbours.

Since our goal is to speed up the Helmholtz solver with the help of GPUs, we still would like to keep the double precision convergence rate of the Krylov subspace method. Therefore Bi-CGSTAB is implemented in double precision. For the preconditioner, single precision is sufficient for the CPU as well as the GPU.

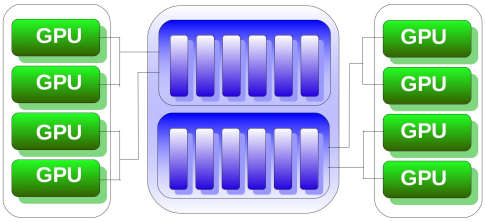


Figure 3.1: NVIDIA machine with 12 Westmere CPUs and 8 Fermi GPUs, where two GPUs share a PCI bus connected to a socket.

3.3. MULTI-GPU IMPLEMENTATION

For our numerical experiments in 2012 NVIDIA [41] provided a Westmere based 12-cores machine connected to 8 GPUs Tesla 2050 as shown on Figure 3.1. The 12-core machine has 48 GB of RAM. Each socket has 6 CPU cores Intel(R) Xeon(R) CPU X5670 @ 2.93GHz and is connected through 2 PCI-buses to 4 graphics cards. Note that two GPUs are sharing one PCI-bus connected to a socket. Each GPU consist of 448 cores with clock rate 1.5 GHz and has 3 GB of memory.

In the experiments CUDA version 3.2[1] is used. All experiments on the CPU are done using a multi-threaded CPU implementation (pthreads [85]).

In general, GPU memory is much more limited than CPU memory so we chose a multi-GPU approach to be able to solve larger problems. The implementation on a single GPU of major components of the solver such as vector operations, matrix-vector-multiplication or the smoother has been described in Chapter 2. In this section we focus on the multi-GPU implementation.

There are two ways to perform computations on a multi-GPU machine: push different CUDA contexts to different GPUs (see [41]) or create multiple threads on the CPU, where each thread communicates with one GPU. For our purposes we have chosen the second option, since it is easier to understand and implement.

Multiple open source libraries for multi-threading have been considered and tested. For our implementation of numerical methods on a GPU the main requirement for multi-threading was that a created thread stays alive to do further processing. It is crucial for performance that a thread remains alive as a GPU context is attached to it. Pthreads has been chosen as we have total control of the threads during the program execution.

There are several approaches to deal with multi-GPU hardware:

1. *Domain-Decomposition approach*, where the original continuous or discrete problem is decomposed into parts which are executed on different GPUs and the overlapping information (halos) is exchanged by data transfer. This approach can however have difficulties with convergence for higher frequencies (see [86]).

[1] During the work on this paper, the newer version of CUDA 4.0 has been released. It was not possible to have the newer version installed on all systems for our experiments. That is why for consistency and comparability of experiments, we use the previous version.

2. *Data-parallel approach*, where all matrix-vector and vector-vector operations are split between multiple GPUs. The advantage of this approach is that it is relatively easy to implement. However, matrix-vector multiplication requires exchange of the data between different GPUs, that can lead to significant data transfer times if the computational part is small. The convergence of the solver is not affected.

3. *Split of the algorithm*, where different parts of the algorithm are executed on different devices. For instance, the solver is executed on one GPU and the preconditioner on another one. In this way the communication between GPUs will be minimized. However this approach is specific to each algorithm.

Note that the data-parallel approach can be seen as a method splitting the data across multi-GPUs, whereas the split of the algorithm can be seen as a method splitting the tasks across multiple devices. In this chapter we investigate the data-parallel approach and the split of the algorithms and make a comparison between multi-core and multi-GPUs. We leave out the domain decomposition approach because the convergence of the Helmholtz solver is not guaranteed. The data parallel approach is more intuitive and is described in detail in Section 3.4.

3.3.1. SPLIT OF THE ALGORITHM

The split can be unique for every algorithm. The main idea of this approach is to limit communication between GPUs but still be able to compute large problems.

One way to apply this approach to Bi-CGSTAB preconditioned by shifted Laplace multigrid method is to execute Bi-CGSTAB on one GPU and the multigrid preconditioner on another one. In this case the communication only between the Krylov subspace solver and preconditioner is required but not for intermediate results.

The second way to apply a split of the algorithm to our solver is to execute the Bi-CGSTAB and the finest level of the shifted Laplace multigrid preconditioner across all available GPUs using the data parallel approach. The coarser levels of multigrid method are executed on only one GPU due to small memory requirements. Since the LU-decomposition is used to compute an exact solution on the coarsest level, we use the CPU for that.

3.3.2. ISSUES

Implementation on multi-GPUs requires careful consideration of possibilities and optimization options. The issues we encountered during our work are listed below:

- Multi-threading implementation, where the life of a thread should be as long as the application. This is crucial for the multi-threading way of implementation on a multi-GPU architecture. Note that in case of pushing contexts this is not an issue.
- Because of limited GPU memory size, large problems need multiple GPUs.
- Efficient memory reusage to avoid allocation/deallocation. Due to memory limitations the memory should be reused as much as possible, especially in the multigrid method. In our work we create a pool of vectors on the GPU and reuse them during the whole solution time.

- Limit communications CPU→GPU and GPU→CPU.
- The use of texture memory on multi-GPU architectures is complicated as each GPU needs its own texture reference.
- Coalescing is difficult since each matrix row has a different number of elements.

3.4. NUMERICAL RESULTS ON MULTI-GPU

3.4.1. VECTOR- AND SPARSE MATRIX-VECTOR OPERATIONS

Vector operations such as addition, dot product are trivial to implement on multi-GPU machines. Vectors are split across multiple GPUs, so that each GPU gets a part of the vector. In case of vector addition, the parts of a vector remain on the GPU or can be send to a CPU and be assembled in a result vector of original size. The speedup for vector addition on 8-GPUs compared to a multi-threaded implementation (12 CPUs) is about 40 times for single and double precision arithmetic. For the dot product, each GPU sends its own sub-dot product to a CPU, where they will be summed into the final result. The speedup for the dot product is about 8 for single precision and 5 for double precision arithmetic. The speedups for vector addition and dot product on multi-GPU machines are smaller compared to the single GPU because of the communication between the CPU and multiple GPUs.

The matrix here is stored in a CRS matrix format (Compressed Row Storage, see e.g. [87]) and is split in a row-wise fashion. In this case a part of the matrix rows is transferred to each GPU as well as the whole vector. After matrix-vector multiplication parts of the result are transferred to the CPU where they are assembled into the final resulting vector. The timings for the matrix-vector multiplication are given in Table 3.1.

Table 3.1: Matrix-vector-multiplication in single (SP) and double (DP) precision.

Size	Speedup (SP) 1 GPU against 12-cores	Speedup (SP) 8 GPU against 12-cores	Speedup (DP) 1 GPU against 12-cores	Speedup (DP) 8 GPUs against 12-cores
100,000	54.5	6.81	30.75	5.15
1 Mln	88.5	12.95	30.94	5.97
20 Mln	78.87	12.13	32.63	6.47

3.4.2. BI-CGSTAB AND GAUSS-SEIDEL ON MULTI-GPU

Since the Bi-CGSTAB algorithm is a collection of vector additions, dot products and matrix-vector multiplications described in the previous section, the multi-GPU version of the Bi-CGSTAB is straight forward. In Table 3.2 the timings of Bi-CGSTAB on a many-core CPU, single GPU and multi-GPU are presented. The stopping criterion is 10^{-5}. It is

easy to see that the speedup on multi-GPUs is smaller than on a single GPU due to the data transfer between CPU and GPU. Note that for the largest problem in Table 3.2 it is not possible to compute on a single GPU because there is not enough memory available. However it is possible to compute this problem on multi-GPUs and the computation on the multi-GPU machine is still many times faster than on a 12-core Westmere CPU.

Table 3.2: Speedups for Bi-CGSTAB in single (SP) and double (DP) precision.

Size	Speedup (SP) 1 GPU against 12-cores	Speedup (SP) 8 GPUs against 12-cores	Speedup (DP) 1 GPU against 12-cores	Speedup (DP) 8 GPUs against 12-cores
100,000	12.72	1.27	9.59	1.43
1 Mln	32.67	7.58	15.84	5.11
15 Mln	45.37	15.23	19.71	8.48

As mentioned above, the shifted Laplace multigrid preconditioner consists of a coarse grid correction based on the Galerkin method with matrix-dependent prolongation and of a multi-color Gauss-Seidel smoother. The implementation of the coarse grid correction on multi-GPU architectures is straight forward, since the main ingredient of the coarse grid correction is the matrix-vector multiplication. The coarse grid matrices are constructed on a CPU and then transferred to the GPUs. The matrix-vector multiplication on the multi-GPU is described in Section 3.4.1.

The multi-color Gauss-Seidel smoother on the multi-GPU requires adaptation of the algorithm. We use the 8-colored Gauss-Seidel iteration, since problem 1.1 is given in three dimensions and computations at each discretization point should be done independently of the neighbours to allow parallelism. For the multi-GPU implementation the rows of the matrix for one color will be split between multi-GPUs. Basically, the colors are computed sequentially, but within a color the data parallelism is applied across the multi-GPUs. The timing comparisons for the 8-colored Gauss-Seidel implementation on different architectures are given in Table 3.3.

Table 3.3: Speedups for colored Gauss-Seidel method on different architectures in single precision.

Size	Speedup (SP) 1 GPU against 12-cores	Speedup (SP) 8 GPUs against 12-cores
5 Mln	16.5	5.2
30 Mln	89.1	6.1

Table 3.4: Timings for Bi-CGSTAB preconditioned by the shifted Laplace multigrid.

	Bi-CGSTAB (DP)	Preconditioner (SP)	Total	Speedup
12-cores	94 s	690 s	784 s	1
1 GPU	13 s	47 s	60 s	13.1
8 GPUs	83 s	86 s	169 s	4.6
2 GPUs+split	12 s	38 s	50 s	15.5

3.5. NUMERICAL EXPERIMENTS FOR THE WEDGE PROBLEM

This model problem represents a layered heterogeneous problem taken from [21]. Find $\phi \in \mathbb{C}^{n\times n\times n}$

$$-\Delta\phi(x,y,z) - k(x,y,z)^2\phi(x,y,z) = \delta\left((x-500)(y-500)z\right), \tag{3.1}$$

$(x,y,z) \in \Omega = [0,0,0] \times [1000,1000,1000]$, with the first-order boundary conditions. The coefficient $k(x,y,z)$ is given by $k(x,y,z) = 2\pi f l/c(x,y,z)$ where $c(x,y,z)$ is presented in Figure 3.2. The grid size satisfies the condition $\max_x(k(x,y,z))h = 0.625$, where $h = \frac{1}{n-1}$. Table 3.4 shows timings for Bi-CGSTAB preconditioned by the shifted Laplace multigrid method on problem 3.1 with 43 millions unknowns. The single GPU implementation is about 13 times faster than a multi-threaded CPU implementation. The data-parallel approach shows that on multi-GPUs the communication between GPUs and CPUs takes a significant amount of the computational time, leading to smaller speedup than on a single GPU. However, using the split of the algorithm, where Bi-CGSTAB is computed on one GPU and the preconditioner on the other one, increases the speedup to 15.5 times. Figure 3.3. shows the real part of the solution for 30 Hz.

3.6. CONCLUSIONS

In this chapter we presented a multi-GPU implementation of the Bi-CGSTAB solver preconditioned by a shifted Laplace multigrid method for a three-dimentional Helmholtz equation. To keep the double precision convergence the Bi-CGSTAB method is implemented on the GPU in double precision and the preconditioner in single precision. We have compared the multi-GPU implementation to a single-GPU and a multi-threaded CPU implementation on a realistic problem size. Two multi-GPU approaches have been considered: a data parallel approach and a split of the algorithm. For the data parallel approach, we were able to solve larger problems than on one GPU and got a better performance than by the multi-threaded CPU implementation. However due to the communication between GPUs and the CPU the resulting speedups have been considerably smaller compared to the single-GPU implementation. To minimize the communication but still be able to solve large problems we have introduced the split of the algorithm technique. In this case the speedup on multi-GPUs is similar to the single GPU compared to the multi-core implementation.

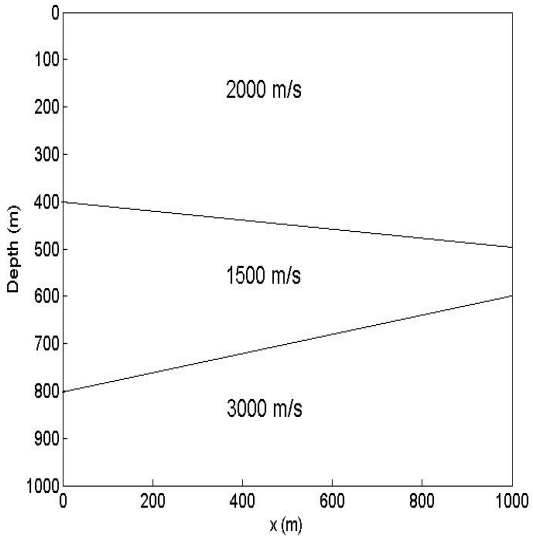


Figure 3.2: The velocity profile of the wedge problem.

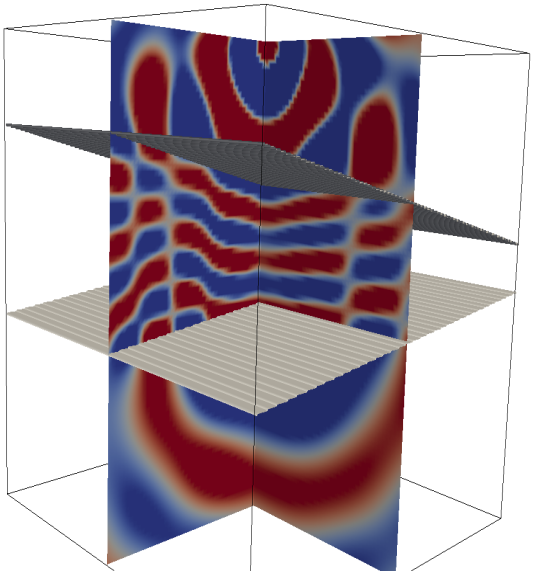


Figure 3.3: Real part of the solution, $f = 30$ Hz.

4

CLOSING THE PERFORMANCE GAP BETWEEN AN ITERATIVE FREQUENCY-DOMAIN SOLVER AND AN EXPLICIT TIME-DOMAIN SCHEME FOR 3-D MIGRATION ON PARALLEL ARCHITECTURES

Abstract

3-D reverse time migration with the constant-density acoustic wave equation requires an efficient numerical scheme for the computation of wavefields. An explicit finite-difference scheme in the time domain is the common choice. However, it requires a significant amount of disk space for the imaging condition. The frequency-domain approach simplifies the correlation of the source and receiver wavefields, but requires the solution of a large sparse linear system of equations. For the latter, we use an iterative Helmholtz Krylov subspace solver based on a shifted Laplace multigrid preconditioner with matrix-dependent prolongation. The question is whether migration in the frequency domain can compete with a time-domain implementation when both are performed on a parallel architecture. Both methods are naturally parallel over shots, but the frequency-domain method is also parallel over frequencies. If we have a sufficiently large number of compute nodes, we can compute the result for each frequency in parallel and the required time is dominated by the number of iterations for the highest frequency. As a parallel architecture, we consider

Parts of this chapter have been published in H. Knibbe, W. A. Mulder, C. W. Oosterlee, and C. Vuik. GEOPHYSICS, 79(2), pp S47–S61, 2014, doi: 10.1190/geo2013-0214.1, [88]

a commodity hardware cluster that consists of multi-core CPUs, each of them connected to two GPUs. Here, GPUs are used as accelerators and not as independent compute nodes. The parallel implementation of the 3-D migration in the frequency domain is compared to a time-domain implementation. We optimized the throughput of the latter with dynamic load balancing, asynchronous I/O and compression of snapshots. Since the frequency-domain solver uses matrix-dependent prolongation, the coarse grid operators requires more storage than available on GPUs for problems of realistic size. Due to data transfer, there is no significant speedup using GPU-accelerators. Therefore, we consider an implementation on CPUs only. Nevertheless, with the parallelization over shots and frequencies, this approach can compete with the time-domain implementation on multiple GPUs.

4 INTRODUCTION

The oil and gas industry makes use of computational intensive algorithms such as reverse-time migration and full waveform inversion to provide an image of the subsurface. The demand for better resolution increases the bandwidth of the seismic data and leads to larger computational problems. At the same time, high-performance computer architectures are developing quickly by having more and faster cores in the CPUs (Central Processing Units) or GPUs (Graphics Processing Units). The increase in the number of cores requires the development of scalable algorithms.

The finite-difference solution of the constant-density acoustic wave equation has become the common tool for reverse-time migration, usually discretized by high-order finite differences in space and second-order differencing in time. The discretization leads to a fully explicit method. Higher-order finite differences reduce problem size compared to low-order finite differences because they require fewer grid points per wavelength [89] if the underlying model is sufficiently smooth, which is usually the case in reverse-time migration. Explicit methods based on finite-differences exhibit natural parallelism since the computation of one point in space for a given time step is independent of its neighboring points. They can be easily parallelized with OpenMP on shared-memory architectures and on GPUs [90]. We refer to the paper by [91] for an overview.

Migration of seismic data is commonly carried out in the time domain. The classic reverse-time migration algorithms in the time domain are known to be computationally and I/O intensive [44, 45] because the forward and time-reversed wavefields have to be computed and stored. If the correlation between these fields is carried out during the time-reversed computation of the receiver data, only snapshots of the forward wavefield have to be stored.

It is possible to reduce the time needed to write the snapshots to disk, for example by using asynchronous I/O [49] and wavefield compression. Standard libraries for Fourier transformation or wavelet compression can be used [45]. However, this approach may have difficulties to preserve the frequency content of the image and may introduce compression artifacts.

Migration in the frequency domain is historically less mature because of the necessity to solve a sparse indefinite linear system of equations for each frequency, which arises from the discretization of the Helmholtz equation, whereas in the time domain

the discretization of the wave equation in space and time leads to an explicit time marching scheme. An important advantage of migration in the frequency domain however is that the cross-correlation needed for the imaging condition becomes a simple multiplication. As a result, no wavefields have to be stored. Parallelization over frequencies is natural. When a direct solver is used to compute the solution of the sparse matrix, typically a nested-dissection LU-decomposition is applied [50]. When many shots need to be treated, the frequency-domain solver in two dimensions can be more efficient than a time-domain time-stepping method [51, 52], because the LU-decomposition can be reused for each shot as well as for each 'reverse-time' computation. Also, lower frequencies can be treated on coarser meshes.

In three dimensions, however, frequency-domain migration is considered to be less efficient than its time-domain counterpart. One of the reasons is the inability to construct an efficient direct solver for problems of several millions of unknowns [53]. The authors in [54, 55] proposed a direct solver based on nested-dissection that compresses intermediate dense submatrices by hierarchical matrices.

An iterative solver is an obvious alternative, for instance, the one with a preconditioner that uses a multigrid method to solve the same Helmholtz equation but with very strong damping [21, 56, 57, 69]. This method, however, needs a number of iterations that increases with frequency, causing the approach to be less efficient than a time-domain method. Note that the iterative method requires a call to the solver for each shot and each 'reverse-time' computation, so the advantage of reusing an LU-decomposition is lost. This approach was parallelized by [37]. In Chapter 3 we have used GPUs to speed up the computations.

However, with the development of iterative methods on the one hand and hardware accelerators on the other hand, we have to reconsider the performance of migration in the frequency domain. As a parallel architecture we consider a commodity hardware cluster that consists of multi-core CPUs, each of them connected to two GPUs. In general, a GPU has a relatively small amount of memory compared to the CPU.

A GPU can be used in two different ways: as an independent compute node replacing the CPU or as an accelerator. In the first case, the algorithm is split to solve a number of independent sub-problems that are then transferred to the GPU and computed separately. To achieve the best performance, the data is kept on the GPU when possible. We have exploited this way of using a GPU for the Helmholtz equation earlier in Chapters 2 and 3.

In the second case, the GPU is considered as an accelerator, which means that the problem is solved on the CPU while off-loading the computational intensive parts of the algorithm to the GPU. Here, the data is transferred to and from the GPU for each new task. In this chapter we focus on the second approach.

The aim of this chapter is to demonstrate that migration in the frequency domain, based on a Krylov subspace solver preconditioned by a shifted-Laplace multigrid preconditioner on CPUs, can compete with reverse-time migration in the time domain on commodity parallel hardware, a multi-core CPU connected to two GPUs.

We will make a comparison in terms of computational time, parallelization and scalability aspects. We use a finite-difference discretization of the constant-density acoustic wave equation for computing the wavefields. Here, we solve the 3-D wave equation

in the frequency domain with the iterative Helmholtz solver described in Chapter 3. This solver reduces the number of iterations by a complex-valued generalization of the matrix-dependent multigrid method. The price paid for improved convergence is that the implementation is no longer matrix-free. The matrix-dependent prolongation requires the storage of the coarse-grid operators. As a result, the use of a GPU as an independent compute node becomes less attractive for realistic problem sizes. Applying GPUs as accelerators involves substantial data transfer, requiring a significant amount of time and reducing the speedup as compared to parallel computations on CPUs. For that reason, we will here only consider a CPU implementation for the Helmholtz equation. For the migration in the time domain, this problem disappears because of the explicit time stepping and we can exploit GPUs as accelerators. Complexity estimates show that both approaches scale in the same way with grid size but that does not give an indication of the actual performance on a problem of realistic size. We therefore made a comparison of our actual implementations for the frequency and time domain.

We will describe seismic modeling and migration in the time and in the frequency domain. The advantages of the frequency-domain solver are explained and demonstrated. We review the parallel strategies for both time and frequency domain and describe implementation details on multi-cores and on GPUs. Finally we compare the parallel performance on two 3-D examples.

4.1. CHOICE OF METHOD

The choice of the numerical scheme is motivated by complexity analysis. Consider a 3-D problem of size $N = n^3$, with n_s shots, n_t time steps, n_f frequencies, n_{it} iterations. The number of shots is usually $n_s \approx n^2$, the number of time steps $n_t \approx n$ (see equation 4.5), the number of frequencies is $n_f \approx n$ at most, and the number of iterations for the iterative frequency-domain method is $n_{it} \approx n_f$ (see equation 4.8).

The complexity of time-domain modeling is $n_s n_t O(n^3)$ [43, 92]. The direct solver in the frequency domain has a complexity of $O(n^7 + n_s n^3)$ for a single frequency when using a standard LU-decomposition. This can be reduced to $O(n^6 + n_s n^3)$ with nested dissection [50]. The authors in [54, 55] suggested to use a low-rank approximation of the dense matrices arising from the nested dissection. In that case, the complexity of the method lies between $O(n^3(\log n + n_s))$ and $O((n^4 + n_s n^3)\log n)$, depending on the problem. The authors in [37, 56] considered the complexity of the shifted Laplace solver preconditioned by a multigrid method and obtained $O(n_s n_f n_{it} n^3)$ for n_f frequencies.

Considering the parallel aspects over shots and frequencies, Table 4.1 captures the complexity of the algorithms mentioned above. Factors of $O(\log n)$ have been ignored. It is readily seen that the time domain is the most efficient method in the sequential case. However, if the implementation is parallel over shots and frequencies, the time-domain and iterative frequency-domain methods appear to be the most attractive methods in terms of turn-around time, assuming sufficient resources for parallel computing are available and their cost is not included. This leaves questions about actual performance unanswered, since the constants in the complexity estimates are absent. To get an indication of the actual performance of the two algorithms, both were implemented and tested on a problem of realistic size.

Table 4.1: Complexity of various methods, split between setup and application cost. The required amount of storage is given in column 4. Next, a factor that can be dealt with by trivial parallelization is listed. The last column shows the scaling of compute time when the trivial parallelization is applied, assuming n^3 grid points, $n_s \approx n^2$ shots, $n_t \approx n$ time steps, $n_f \approx n$ frequencies and $n_{it} \approx n$ iterations.

Method	Setup	Apply	Storage	Parallel	Overall
time domain	–	$n_s n_t n^3$	$n_s n^3$	n_s	$(0+n^6)/n^2 \approx n^4$
LU, nested dissection	$n_f n^6$	$n_s n_f n^4$	$n_f n^4$	n_f	$(n^7+n^7)/n \approx n^6$
LU, low rank	$n_f n^4$	$n_s n_f n^3$	$n_f n^3$	n_f	$(n^5+n^6)/n \approx n^5$
iterative	$n_f n_s n^3$	$n_s n_f n_{it} n^3$	$n_s n_f n^3$	$n_s n_f$	$(n^6+n^7)/n^3 \approx n^4$

4.2. MODELING

Modeling is a major component of migration and inversion algorithms. Traditionally, seismic data are modeled in the time domain because of the simplicity of implementation as well as the memory demands. However, modeling in the frequency domain offers such advantages as parallelization over frequencies and reuse of earlier results if an iterative solver is employed for computing the wavefields, for example, during least squares migration or full waveform inversion. We will compare modeling in the time domain to that in the frequency domain.

4.2.1. MODELING IN THE TIME DOMAIN

Modeling in the time domain requires the solution of the wave equation 1.6. Discretization of the three-dimensional wave equation with second-order finite differences in space and time leads to an explicit time marching scheme of the form

$$\begin{aligned} u^{n+1}_{i,j,k} &= 2u^n_{i,j,k} - u^{n-1}_{i,j,k} + \Delta t^2 c^2_{i,j,k} \\ &\left(\frac{u^n_{i-1,j,k} - 2u^n_{i,j,k} + u^n_{i+1,j,k}}{h_x^2} + \frac{u^n_{i,j-1,k} - 2u^n_{i,j,k} + u^n_{i,j+1,k}}{h_y^2} \right. \\ &\left. + \frac{u^n_{i,j,k-1} - 2u^n_{i,j,k} + u^n_{i,j,k+1}}{h_z^2} + s^n_{i,j,k} \right), \end{aligned} \tag{4.1}$$

where the superscript $n+1$ denotes a new time level that is computed using the solutions at the two previous time steps, n and $n-1$. A higher-order discretization of the second derivative in space in one direction is obtained by

$$\left. \frac{\partial^{(2)} u}{\partial x^{(2)}} \right|_i \simeq -\frac{1}{h_x^2} \left(b_0^M u_i + \sum_{k=1}^{M} b_k^M (u_{i-k} + u_{i+k}) \right), \tag{4.2}$$

where $2M$ denotes the order of the spatial discretization and the coefficients are

$$b_0^M = \sum_{m=1}^{M} \frac{2}{m^2}, \tag{4.3}$$

$$b_k^M = (-1)^k \sum_{m=k}^{M} \frac{2}{m^2} \frac{(m!)^2}{(m+k)!(m-k)!}, \quad k = 1, \ldots, M. \tag{4.4}$$

The authors in [93] describe a higher-order discretization of the second derivative in time.

To ensure stability of the time marching scheme above, the time step has to satisfy the stability constraint

$$\Delta t \leq \mathrm{CFL} \frac{d}{c_{\max}}, \tag{4.5}$$

with the maximum velocity $c_{\max}$, the diameter

$$d = \frac{1}{\sqrt{1/h_x^2 + 1/h_y^2 + 1/h_z^2}} \tag{4.6}$$

and the constant $\mathrm{CFL} = 2/\sqrt{a}$, with

$$a = \sum_{k=1}^{M/2} 4^k \Big/ \left(k^2 \begin{pmatrix} 2k-1 \\ k-1 \end{pmatrix} \right). \tag{4.7}$$

For details see [94].

In order to simulate an infinite domain and avoid reflections from the boundaries, sponge absorbing boundary conditions have been implemented [19].

4.2.2. MODELING IN THE FREQUENCY DOMAIN

For wave propagation in the frequency domain, we consider the Helmholtz equation 1.1 in a 3-D heterogeneous medium. Equation 1.1 was solved with a Krylov subspace method preconditioned by a shifted-Laplace preconditioner [21, 69, 83]. We have described the method in great detail in Chapter 2.

If the wavelet or signature of the source term g is given in the time domain, its frequency dependence is readily obtained by a Fast Fourier Transform (FFT). Given the seismic data, the Nyquist theorem dictates the frequency sampling and the maximum frequency. In practice, however, the maximum frequency in the data is lower and is defined by the wavelet. Given the range of frequencies defined by Nyquist's theorem and the data, the Helmholtz equation 1.1 is solved for each frequency and the wavefield is sampled at the receiver positions, producing a seismogram in the frequency domain. Finally, the wavelet and an inverse FFT are applied to obtain the synthetic seismogram in the time domain.

The discretization of equation 1.1 in space depends on the number of points per wavelength. The general rule of thumb is to discretize with at least 10 points per wavelength [20]. In that case, the error behaves as $(kh)^2$, which is inversely proportional to the

square of the number of points per wavelength. To avoid the pollution effect, $kh = 0.625$ has been chosen constant, as described by [21]. The authors in [29] showed that the number of iterations of the Helmholtz solver does not depend on the problem size for a given frequency, but the number of iterations increases with frequency. The authors in [30] and [31] presented an improved version that requires fewer iterations but still requires more iterations at higher frequencies. Therefore, the computational time for modeling in the frequency domain mainly depends on the highest frequency used and on the total number of frequencies. To reduce computational time, the computations for each frequency can be parallelized over several compute nodes. Then, the question arises: how many compute nodes do we need to minimize the computational time?

It is obvious that for a parallel implementation over an unlimited number of compute resources, the computational time is at least equal to the time needed to solve the Helmholtz equation 1.1 for the highest frequency.

Since the problem size is the same for each frequency and the iterative method has a fixed number of matrix-vector and vector-vector operations, it is easy to see that the time per iteration is the same for each frequency. In principle, the lower frequencies can be calculated on coarser meshes [92]. However, using a Krylov subspace solver preconditioned with a shifted-Laplace multigrid method, the number of iterations is already quite low. Therefore, the additional complexity due to interpolation between different grid sizes does not pay off. The number of iterations per frequency f_i, $i \in \mathbb{N}$, can be expressed as

$$n_i \approx \gamma f_i, \tag{4.8}$$

where $\gamma = n_N / f_N$, f_N denotes the maximum frequency and n_N the number of iterations for f_N. The frequencies are given by $f_i = i\Delta f$ where Δf is the frequency sampling interval. The total number of iterations for computing all frequencies is given by

$$\sum_{i=1}^{N} n_i \approx \gamma \sum_{i=1}^{N} i\Delta f = \gamma \Delta f \frac{N(N+1)}{2} = n_N \frac{N+1}{2}. \tag{4.9}$$

In other words, the least computational time can be achieved by using a number of compute nodes equal to half the number of frequencies. As an example, let us consider a problem with maximum frequency $f_{\max} \simeq 30\,\text{Hz}$ and $\Delta f \simeq 1/6\,\text{Hz}$. Then, 180 frequencies need to be computed, which would require 90 compute nodes. Here, we assume that the problem size corresponding to the maximum frequency fits into the memory of a single compute node.

We can adopt a different point of view by fixing the number of compute nodes and then determining the minimum workload of each node in terms of the number of iterations. Let us denote by M the number of available compute nodes. Then, using equation 4.9, the minimum time per node is equal to

$$T_{\min} = \frac{t_N(N+1)}{2M}, \tag{4.10}$$

where t_N is the compute time needed for the highest frequency.

4.3. MIGRATION

Migration algorithms produce an image of the subsurface given seismic data measured at the surface. In particular, pre-stack depth migration produces the depth locations of reflectors by mapping seismic data from the time domain to the depth domain, assuming a sufficiently accurate velocity model is available. The classic imaging principle [1, 2] is based on the correlation of the forward propagated wavefield from a source and a backward propagated wavefield from the receivers. To get an approximation of the reflector amplitudes, the correlation is divided by the square of the forward wavefield [3, 4]. For true-amplitude or amplitude-preserving migration, there are a number of publications based on the formulation of migration as an inverse problem in the least-squares sense [5–9]. For our purpose of comparing migration in the time and frequency domain, we focus on the classical imaging condition [13]

$$I(\mathbf{x}) = \sum_{shots} \sum_{t} W_s(\mathbf{x}, t) W_r(\mathbf{x}, t), \tag{4.11}$$

in time domain, or

$$I(\mathbf{x}) = \sum_{shots} \sum_{\omega} W_s^*(\mathbf{x}, \omega) W_r(\mathbf{x}, \omega), \tag{4.12}$$

in the frequency domain. Here, I denotes the image, W_s is the wavefield propagated from the source and W_r from the receivers, respectively; t denotes time and ω denotes the frequency. The star indicates the complex conjugate.

4.3.1. BORN APPROXIMATION

The seismic data that needs to be migrated should not contain multiple reflections from the interfaces if imaging artifacts are to be avoided. Often, the Born approximation is used for modeling without multiples. Migration can be viewed as one step of an iterative procedure that attempts to minimize the difference between observed and modeled data subject to the Born approximation of the constant-density acoustic wave equation [5, 6, 8, 43].

The wave equation 1.6 in matrix form is given by

$$Au = f, \tag{4.13}$$

with wave operator $A = m\partial_{tt,h} - \Delta_h$ and model parameter $m = 1/c^2$. The last can be split into $m = m_0 + m_1$, where m_0 ideally does not produce reflections in the bandwidth of the seismic data. The wavefield can be split accordingly into $u = u_0 + u_1$ into a reference and a scattering wavefield, respectively. The reference wavefield u_0 describes the propagation of a wave in a smooth medium without any hard interfaces. The scattering wavefield u_1 represents a wavefield in a medium which is the difference between the actual and reference medium. Wave propagation in the reference medium is then described by $A_0 u_0 = f$ with $A_0 = m_0\partial_{tt,h} - \Delta_h$ in the time domain. What remains is $A_0 u_0 + A_1 u_0 + A_1 u_1 = 0$ with $A_1 = A - A_0 = m_1\partial_{tt,h}$. In the Born approximation, the term

$A_1 u_1$ is removed, leading to the system of equations

$$A_0 u_0 = f, \tag{4.14}$$
$$A_0 u_1 = -A_1 u_0. \tag{4.15}$$

Its counterpart in frequency domain is given by

$$\begin{aligned} A &= -k^2 - \Delta_h, \\ A_0 &= -k_0^2 - \Delta_h, \\ A_1 &= A - A_0 = -k_1^2, \end{aligned} \tag{4.16}$$

where k_0 is the wave number in the reference and k_1 in the scattered medium, respectively.

With this, migration becomes a linear inverse problem of finding a scattering model $m_1(x, y, z)$ that minimizes the difference between the recorded and modeled wavefields u_1 in a least-squares sense. This assumes that the recorded data were processed in such a way that only primary reflections were preserved, since wavefield u_1 will not contain the direct wave and multiple reflections. A few iterations with a preconditioned Krylov subspace method will suffice to solve the linearized inverse problem and, moreover, just one iteration may already produce a useful result [9, 43].

To illustrate the difference between modeling with the wave equation and its Born approximation, we consider the simple velocity model shown in Figure 4.1. The background velocity (white) is 1500 m/s and the horizontal layer shown with grey color has a velocity of 2500 m/s.

We model the seismogram using modeling in the time domain, see Figure 4.2 (left). The seismogram obtained using the Born approximation is presented in Figure 4.2 (right). Time-weighting was applied to boost the amplitude of the reflections, meaning that the amplitudes were scaled with the square of time. The first event in both figures represents the reflection from the shallow interface and the second from the deeper interface. The third event in Figure 4.2 (left) is the interbed multiple that does not appear with the Born approximation.

4.3.2. MIGRATION IN THE TIME DOMAIN

We briefly summarize the algorithm for reverse-time migration (RTM) in the time domain. The method consists of three major parts: forward propagation of the wavefield from a source, backward propagation from the receivers while injecting the observed seismic data and the imaging condition.

During forward propagation in time, the snapshots of the wavefield are stored at every imaging time step, so that they can be reused later for imaging, as sketched in Figure 4.3. After that, the wavefield is propagated backwards in time using the seismic data as sources at the receiver locations. The image is built by correlation, taking the product of the forward and backward wavefields stored at the same imaging time step and summing the result over time. The imaging time step is usually the same as the sampling interval of the seismic data, which is usually larger than the time step used for modeling.

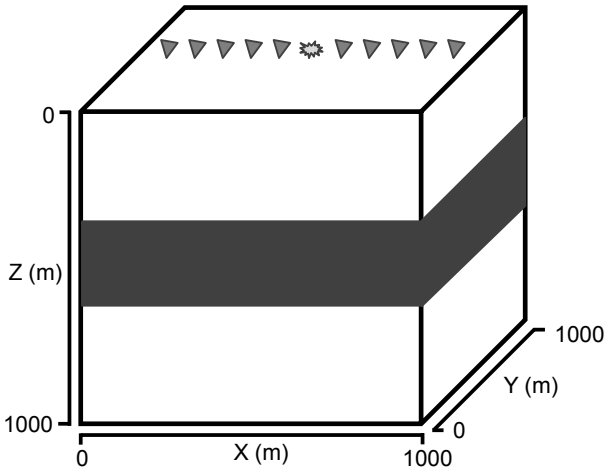


Figure 4.1: Model with a single high-velocity layer of 2500 m/s (grey) in a homogeneous background of 1500 m/s (white). The star represents the source and the triangles the receivers.

To save time and storage space, the imaging condition can be incorporated in the backward propagation. During the forward wave propagation, the wavefields are written to disk, because for problems of realistic size, random-access memory is usually too small. While backward propagating every imaging time step, the forward wavefields are read from disk and correlated with the computed backward wavefield. Even if this reduces the amount of I/O, disk access can take a significant amount of time compared to the computations.

The author in [48] used an optimized check-pointing method that only saves the wavefields at predefined checkpoints in time and recomputes the wavefields at other instances from these checkpoints. The amount of recomputation is reduced by choosing optimal checkpoints. However, the recomputation ratio may be very high when the number of checkpoints is not large enough. If the number of checkpoints is too large, the disk space demand and I/O will be high.

With the use of absorbing boundary conditions to simulate infinite domains, the recomputation of the forward wavefield may require some special techniques. One possibility is to store only the boundary values of the wavefield. However, in three dimensions the boundaries can have a substantial width and this may not be efficient. Another possibility is to use the random boundary technique [46, 95], which leads to random scattering of the wavefield at the boundary. The idea is that the forward and backward wavefields are based on different sets of random numbers and the artifacts due to scattering do not stack in the final image. In this case, the propagation effort is doubled, because first the forward wavefield is computed after which the backward propagation and the time-reversed forward wavefield are calculated simultaneously. This method has been implemented on a GPU by [44].

If the noise due to random scatterers is to be avoided, alternative techniques to re-

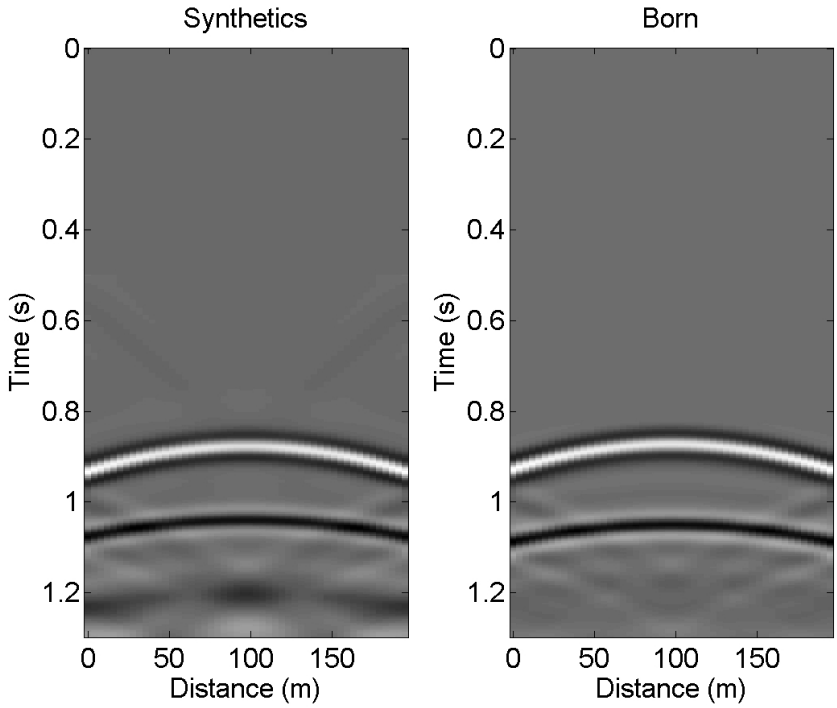


Figure 4.2: (left) Synthetic seismic data produced with regular modeling in the time domain. The direct arrival has been removed. (right) With the Born approximation in the time domain, showing that the interbed multiple around 1.2 s has disappeared.

duce the effect of I/O for the RTM algorithm can be applied. One of them is to hide the writing and reading times of the snapshots by using asynchronous I/O. However, this is only effective when the reading of the wavefields from disk is faster than the computations needed for one imaging time step. One way to achieve this is by compressing the wavefield before storing it to disk on a GPU or on a CPU, see e.g. [49].

The Fourier transform technique can offer compression. The periodicity of trigonometric functions requires special care at the boundaries. An alternative is to use functions that are compact in space and time, such as wavelets. The wavelet transform has been extensively documented, see e.g. [96, 97]. The authors in [45] use wavelets for compression of the wavefields. In that way, disk I/O is reduced and the GPU, CPU and disk I/O are balanced well. The idea is to decompose the snapshot by means of the wavelet transform into an 'average' and a 'detailed' part. The average part contains the dominant features of the data and the detailed part contains small-scale features. We are interested in keeping the average part as is and focus on the details. Before compressing the snapshots, we can choose the amount of detail we would like to keep. For the detailed part, the mean and deviation are calculated. We introduce a parameter $\hat{\lambda}$ that is multiplied by the deviation. This product defines the threshold for compression. Table 4.2 describes the effect of values of $\hat{\lambda}$ on the compression ratio, which is defined as the size of the orig-

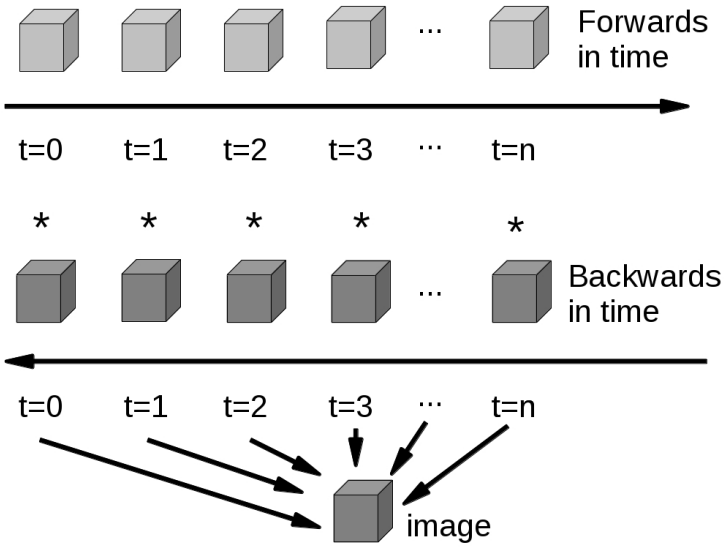


Figure 4.3: Migration in the time domain. The forward propagated wavefield from the source is stored at each imaging time step (light grey cubes). Then, the wavefield is propagated backwards in time while injecting seismic data at the receiver locations and the snapshots are stored (dark grey cubes). The imaging condition involves the summation of the product of the forward and backward wavefields.

inal data divided by the size of the compressed data, as well as the compression time. It is clear that the compression ratio depends on the parameter $\hat{\lambda}$, which means that the more data we remove, the better the compression. The second column in the table shows the L_1-norm of the absolute differences between the original and the compressed wavefields. The third column represents the relative L_2-norm, which is the usual L_2-norm of the difference scaled by the L_2-norm of the original wavefield. We observe that the more the wavefields are compressed, the more the L_1- and L_2-norms increase monotonically, due to the loss of information during compression.

Which is the optimal compression parameter? There is a trade-off between small L_1- and L_2-errors and a large compression ratio. The table shows that the required compression time hardly changes with various choices of $\hat{\lambda}$. Figure 4.4 depicts the compression ratio as a function of parameter $\hat{\lambda}$. It starts of with an exponential increase and becomes linear for $\hat{\lambda} > 1$. Therefore, we have selected $\hat{\lambda} = 1$ as our compression parameter, as it provides an acceptable balance between compression errors and required compression time.

4.3.3. MIGRATION IN THE FREQUENCY DOMAIN

Migration in the frequency domain requires the selection of a set of frequencies that avoids spatial aliasing [92]. The seismic data and the source signature are transformed

Table 4.2: Compression for wavefield snapshots for a problem of size 512^3 and about 100 MB of storage. The L_1-norm measures the difference between the original and compressed wavefield. The relative L_2-norm is the usual L_2-norm of the differences, scaled by the L_2-norm of the original snapshot. The compression ratio is defined as the size of the original data divided by the size of the compressed data.

$\hat{\lambda}$	L_1-norm	Relative L_2-norm	Compr. ratio	Compr. time (s)
0.1	1.08e-1	1.98e-3	3.88	16.16
0.5	1.14e-1	2.11e-3	4.16	15.46
1	1.52e-1	2.87e-3	4.32	15.27
1.5	2.21e-1	4.27e-3	4.42	15.07
2	3.07e-1	5.95e-3	4.51	14.92
2.5	4.06e-1	7.86e-3	4.59	14.77
3	5.22e-1	1.01e-2	4.67	14.58
3.5	6.47e-1	1.25e-2	4.75	14.53
4	7.96e-1	1.53e-2	4.83	14.37
4.5	9.44e-1	1.82e-2	4.91	14.27
5	1.09e-0	2.09e-2	4.99	14.28

to the frequency domain by an FFT. For each frequency, the Helmholtz equation is solved iteratively. The imaging condition in the frequency domain consists of a simple multiplication of the wavefields at each frequency, followed by a summation over the selected frequencies. Figure 4.5 illustrates the procedure. The forward and backward propagation are computed in parallel and there is no need to store the wavefields on disk. Basically, for each frequency the forward and backward fields are computed one after the other and are then multiplied with each other. Only two wavefields are kept in memory, whereas in time domain, all the consecutive wavefields for the forward propagation need to be stored.

4.4. IMPLEMENTATION DETAILS

As mentioned before, we consider the GPU as an accelerator in our implementation strategy and we will use the terms 'CPU' and 'compute node' when referring to a multicore CPU machine.

Presently, a common hardware configuration is a CPU connected to two GPUs that contain less memory than the CPU. We identified the parts of the algorithms that can be accelerated on a GPU and implemented them in CUDA 5.0. As already explained, the parallelization process for migration in the time domain is different from that for the frequency domain, because explicit time stepping is used in the time domain whereas the frequency domain requires solving a linear system of equations. We summarize the levels of parallelism in Table 4.3.

The highest level of parallelization for time-domain migration is over the shots. Each shot is treated independently. We assume that the migration volume for one shot is com-

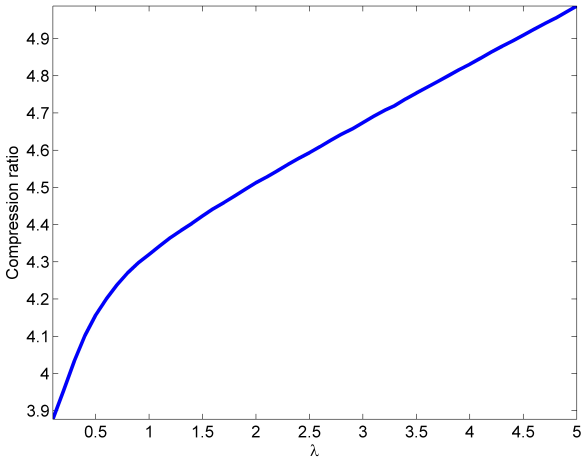


Figure 4.4: Compression ratio as a function of the parameter $\hat{\lambda}$.

	Time Domain	Frequency Domain
Level 1	Parallelization over shots	Parallelization over shots
Level 2	Domain decomposition	Parallelization over frequencies
Level 3	Overlap for computations with memory transfer, load balancing	Matrix decomposition
Level 4	Data parallelism (grid points)	Linear algebra parallelization (MVM, vector operations)

Table 4.3: Levels of parallelism for migration in time and frequency domain.

puted on one compute node connected to one or more GPUs.

The next step is to split the problem into subdomains that will fit on a GPU. We use a domain decomposition approach or grid partitioning. The idea is that at each time step, the subdomains are treated independently of each other. Once the computation is completed, the subdomains are copied back to the CPU, after which the next time step can be started. The third level of parallelization is to perform computations and data transfer simultaneously, to save time and achieve optimal load balancing. The compression algorithms and simultaneous computations of the forward and backward propagations are also part of the third level. The fourth level of parallelism for time-domain computations is data parallelism, see e.g. [90].

For migration in the frequency domain as well as in the time domain, the highest level of parallelization is over the shots. The next level of parallelism involves the frequencies. For each frequency, a linear system of equations needs to be solved. As mentioned before, the matrix size and memory requirements are the same for each frequency, but the lower frequencies require less compute time than the higher ones [21].

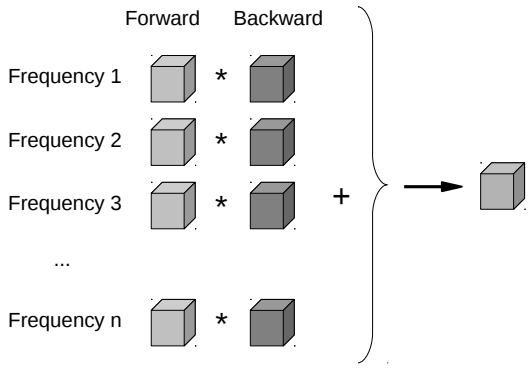


Figure 4.5: Migration in the frequency domain requires multiplication of forward and backward wavefields, followed by summation.

Here, we assume that one shot in the time domain and one shot for one frequency in the frequency domain fit in one compute node connected to one or more GPUs, respectively. The third level of parallelism includes matrix decomposition, where the matrix for the linear system of equations is decomposed into subsets of rows that fit on a single GPU, see Chapter 3. With this approach, we can deal with problems that are larger than can be handled by a single GPU. So far, the simultaneous use of 2 GPUs to accelerate off-loaded matrix-vector multiplications of a large sparse-matrix did not produce any performance improvements compared to a many-core CPU due to the data transfer. Therefore, we use only CPUs for the frequency-domain approach. Note that with an increasing number of GPUs connected to the same CPU, faster PCI buses, etc., this situation may change. The last level of parallelism for migration in frequency domain is parallelization of matrix-vector multiplications (MVMs) and vector-vector operations.

4.4.1. DOMAIN DECOMPOSITION APPROACH

The time-domain implementation on multi-GPUs is done by domain decomposition. The problem is divided into sub-domains that fit in the limited memory of a GPU. This approach can also be applied if a large problem needs to be computed on a single GPU.

For simplicity of implementation and communication between GPUs, the domain is split only in the z-direction, as Figure 4.6 shows. The overlapping areas are attached to both sub-domains. The size of a sub-domain is determined by the available memory divided by the number of discretization points in the x-and y-directions, multiplied by the byte-size of a floating-point number. Each sub-domain should fit entirely in GPU memory.

After partitioning the problem, tasks are set up, where each task represents a sub-domain. These tasks are added to a queue and handled by pthreads [85] that are dis-

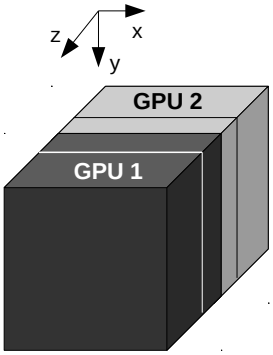


Figure 4.6: An example of domain decomposition into two sub-domains in the z-direction. The overlapping area has half the size of the discretization stencil.

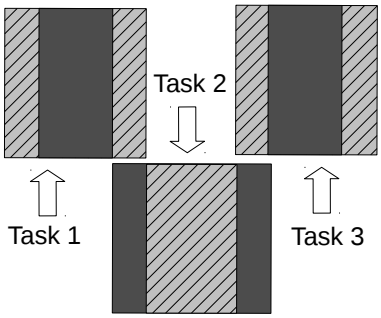


Figure 4.7: An example of task distribution into three tasks. Note that the overlapping areas have to be assigned to all neighbors.

tributed among the GPUs.

Once a sub-domain has been processed, the interior domain, i.e., the domain without the overlapping parts, is copied back to the CPU. Then, the next time step can be performed.

4.4.2. IMPLICIT LOAD BALANCING

The common approach for parallelization across multiple CPU nodes in the cluster is the so-called server-to-client approach [98]. The server is aware of the number of nodes and the amount of work involved. It equally distributes the work-load among the nodes. This approach is efficient on clusters with homogeneous nodes as all CPU nodes have the same characteristics.

In this chapter, we propose a client-to-server approach where clients request tasks to the server. GPU clusters are either heterogeneous or they have to be shared simultaneously amongst the users. For example, the cluster at our disposition, Little Green Machine, see Appendix A, has the same hardware (with the exception of one node). Similarly within one compute node, the GPUs have the same specifications, but one GPU can already be used by a user while the other one remains available. To address the issue of load balancing, we created a task system. When doing migration in time domain, an 'MPI-task' defines the work to be done for one shot. For each time step during forward modeling, one 'GPU-task' is created for each sub-domain of the domain decomposition algorithm. The philosophy behind a task system is the same over multiple compute nodes as well as over multiple GPUs within one node: process all the tasks as fast as possible until they are all processed. In this approach, the work is spread dynamically according to the speed of the computing nodes and GPUs. Depending on the level of

parallelism, the implementation of the tasks systems differs.

MPI-TASKS

The server or 'master node' creates one task per shot. Each task is added to a queue. When a client requests a task, a given task is moved from the queue to the active list. It can happen that a node will crash due to a hardware failure. In that case, the task will remain on the active list until all the other tasks have finished. Once that happens, any unfinished task will be moved back to the queue, so that another compute node can take over the uncompleted work.

Towards the end of the migration, the queue is empty while the list of active tasks is not. As there is no way of telling whether a task has crashes or just takes long time, the former is assumed. The task that is being processed for the longest period of time is submitted again but to a different node. At this point, this particular task may end up being processed by 2 nodes. As soon as the server receives the result of one of these tasks, the other task is killed. When all tasks have been processed, the master node saves the migration image to disk and stops.

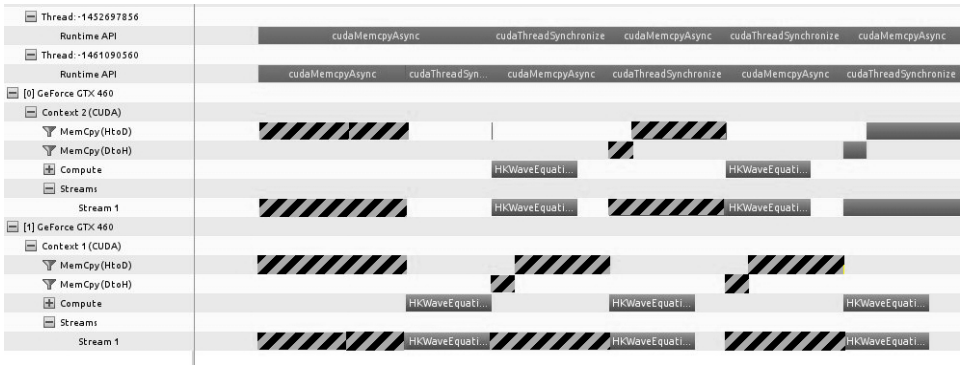


Figure 4.8: Profiling for seismic modeling in the time domain using a 16-th order discretization on 2 GPUs. The column on the left shows the tasks per GPU, such as data transfer from host to device *MemCpy(HtoD)*, from device to host *MemCpy(DtoH)* and wave propagation *Compute*. The length of a bar on the left represents the duration of a task. Dashed bars show the data transfer and dark grey bars represent the computational kernel. The dark grey bars are only overlapping in time with the dashed bars, illustrating that both GPUs are operating asynchronously.

GPU-TASKS

GPU hardware failures are less frequent than the compute node ones, therefore, there is no need to have 2 different queues for GPU-tasks. As an example, let us consider time domain modeling on a CPU node with two GPUs. A GPU-task defines a sub-domain and consists of the following workflow: transfer the sub-domain data from CPU to GPU, propagate the wavefield and, finally, copy the data back to CPU.

When the program starts, first the data have to be transferred to both GPUs. We expect that the two GPUs start transferring data and then will compete for the PCI-bus. Eventually, one GPU will be slightly faster with either the computations or with the data transfer. Then, the two GPUs will work asynchronously, meaning that while one GPU is computing, the other is transferring data. This leads to dynamic load balancing, self-regulated by the system.

We performed profiling with the CUDA profiler, as illustrated in Figure 4.8. Interestingly enough, the two GPUs are acting asynchronously almost from the start of the program execution. The column on the left in Figure 4.8 shows that we have used two GPUs GeForce GTX 460, each of which has to perform a memory copy from host to device (*MemCpy(HtoD)*), a memory copy from device to host (*MemCpy(DtoH)*) and computations of the wave propagation kernel (*Compute*). On the right side of the figure, the horizontal axis denotes time, bars represent different tasks and the length of a bar shows the duration of a task. It is easy to see that at the beginning, both GPUs simultaneously start to transfer data from the host, as the dashed bars are on top of each other. Then, GPU[1] starts computing (dark grey bar) and GPU[0] is idle. Afterwards, GPU[1] transfers the results to the CPU, and GPU[0] is computing at the same time. We have performed several profiling tests and every time we obtained the same outcome, with the GPUs running in asynchronous mode already from the start. This provides an optimal load balancing.

Order	n	CPU	1 GPU	Speedup	2 GPUs	Speedup	sub-volume size
4th	200	6	3	1.93	3	1.95	1565
	400	49	25	1.94	24	2.02	391
	800	398	202	1.97	137	2.9	97
	1200	1342	673	1.99	442	3.03	43
6th	200	8	3	2.33	3	2.32	1565
	400	61	26	2.34	25	2.48	391
	800	489	207	2.36	138	3.56	97
	1200	1709	684	2.50	468	3.65	43
8th	200	9	3	2.74	3	2.77	1565
	400	73	26	2.75	26	2.83	391
	800	591	209	2.82	137	4.31	97
	1200	2002	685	2.92	473	4.23	43
16th	200	15	3	4.34	3	4.36	1565
	400	125	29	4.30	27	4.58	391
	800	1008	221	4.55	142	7.09	97
	1200	3475	671	5.18	433	8.02	41

Table 4.4: Elapsed time comparisons (in s) for 4th-, 6th-, 8th- and 16th-order discretizations for a 3-D problem of size n^3. The speedup is computed as the ratio of CPU time to GPU time. The last column denotes the number of grid points in the z-direction of one (x, y)-slice that will fit in the memory of one GPU (1 GB).

Table 4.4 presents elapsed times (in s) for the modeling in time domain for a finite difference discretization with several discretization orders. For problems of larger size, domain decomposition is applied and the task system with load balancing is used. The

column 'sub-volume size' denotes the number of grid points in the z-direction of one (x, y)-slice that will fit in the memory of one GPU (1 GB). If the total number of grid points is higher than the sub-volume size, then domain decomposition has to be applied, since the problem will not fit in GPU memory otherwise. It is clearly seen that a problem of size 200^3 is sufficiently small to fit into 1 GByte of memory, but larger problems have to be split. Also, the last column shows that the sub-volume size does not give rise to significant changes in CPU time when the discretization order increases. A higher discretization order requires more floating point operations per discretization point and causes an increase in the compute time on the CPU. However, the GPU time hardly changes with increasing order. The reason for this behavior is the load balancing strategy. As the profiling suggests, the transfer and computational time overlap in time asynchronously. On the one hand, the wave propagation kernel takes less time than the data transfer. On the other hand, the transfer time does not change significantly for higher-order discretizations. Therefore, the increase of computational and transfer time for higher-order discretizations is hidden and the overall GPU-time stays the same.

We propose the following workflow for migration in time domain, combining the techniques mentioned above. For forward propagation, first, a main thread is created on a CPU. Its role is to launch other threads, create tasks and be responsible for the GPU-CPU and CPU-GPU transfer. Two child threads are created, one for each GPU, that perform the actual time-stepping computations on the GPUs. When one imaging time step is finished, the wavefield is copied from the GPU to the CPU. The main thread launches several child threads to keep each CPU-core busy. The role of those processes is to compress the wavefield on the fly, using the wavelet transform described above, and write the results to disk. For the backward propagation, the workflow is similar, except that the wavefields are read from memory and decompressed on the fly. Moreover, during the backward propagation the imaging condition is applied. Computations on the GPUs as well as the compression and disk I/O are all done in parallel. For the proposed workflow, I/O (including compression and decompression) in the time domain takes about 5 % of the overall computational time.

4.5. RESULTS

In this section we present some results for migration in the time and frequency domains and make comparisons in terms of performance.

4.5.1. WEDGE

The first example is a wedge model that consists of two dipping layers, depicted in Figure 4.9. The main purpose is to validate the migration results in time and frequency domains. The problem is defined on a cube of size $[0, 1000]^3$ m^3. For our experiments, we consider a series of uniform grids with increasing size n^3. The grid size satisfies the condition $h \max_{x,y,z} k(x, y, z) = 0.625$, where the grid spacing is $h = \frac{1000}{n-1}$ and the wave number $k(x, y, z)$ is given by $k(x, y, z) = 2\pi f / c(x, y, z)$ with velocity $c(x, y, z)$. The problem is discretized with 4th-order finite differences in space and 2nd-order in time for

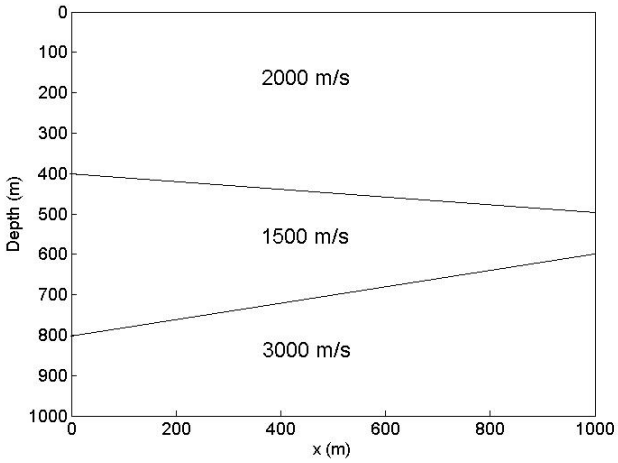


Figure 4.9: Velocity profile for the wedge model.

Table 4.5: Performance of the migration for one source in the time domain in the wedge problem. The problem size is n^3, not counting the extra points at each absorbing boundary.

n^3	Timings forward (s)	Timings backward (s)	Migration time (s)
201	837	846	1683
251	1653	1663	3317
301	2998	2990	5988
901	6649	7015	13666

migration in time domain and 2nd-order finite differences in space for migration in frequency domain. The source is a Ricker wavelet with a peak frequency of 15 Hz located at $(500, 500, 10)$. The receivers are placed at a horizontal plane on a regular grid of 50×50 m^2 at a depth of 20 m. The sampling interval for the seismic data is 4 ms and the maximum simulation time is 2 s. The imaging time step is 4 ms. The experiment was carried out on the Little Green Machine, see Appendix A.

Table 4.5 lists the timings for migration in the time domain and Table 4.6 for the frequency domain. The first column contains the size of the problem, excluding an additional 40 points at each absorbing boundary. The second and third columns show the elapsed time for the forward and backward propagation, respectively. For the experiment in the frequency domain, the timings are given for the highest frequency of 30 Hz, since the preconditioned Helmholtz solver requires the longest computational time at this frequency. From Tables 4.5 and 4.6, it is clear that migration in the frequency domain is more than 2 times faster than the migration in the time domain. Here, we assume that we have enough compute nodes for the calculation of all frequencies in parallel.

Table 4.6: Performance of migration for one source in the frequency domain for the wedge problem at highest frequency 30 Hz.

n^3	Timings forward (s)	Timings backward (s)	Migration time (s)
201	375	365	740
251	732	718	1450
301	1119	1145	2264

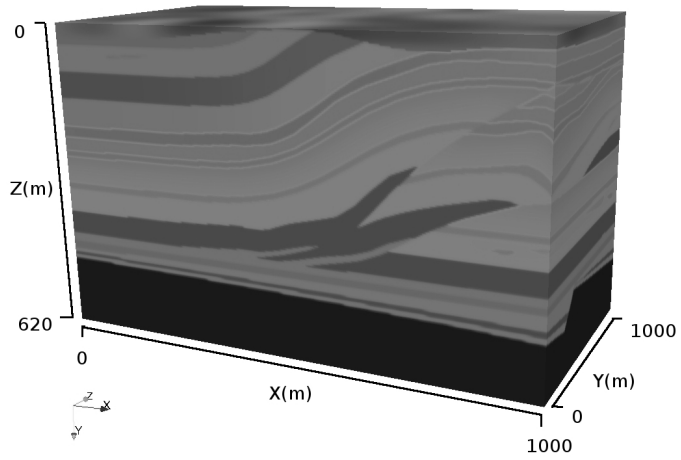


Figure 4.10: Overthrust velocity model.

4.5.2. OVERTHRUST EAGE/SEG MODEL

The SEG/EAGE Overthrust model has been introduced in [99]. It represents an acoustic constant-density medium with complex, layered structures and faults. We chose a subset of the large initial model, containing the fault features shown in Figure 4.10, and rescaled it to fit on a single compute node. The volume has a size of $1000 \times 1000 \times 620$ m^3. The problem is discretized on a grid with $301 \times 301 \times 187$ points and a spacing of 3.33 m in each coordinate direction. As described earlier, one criterion for choosing the grid spacing is the number of points per wavelength needed to accurately model the maximum frequency. Another criterion is the available memory size of the computational node. In addition, we add 40 points for each absorbing boundary in the time-domain scheme to avoid boundary reflections. The discretization for migration in time domain is 4th-order in space and 2nd-order in time and for migration in frequency domain is 2nd-order in space. A Ricker wavelet with a peak frequency of 15 Hz is chosen for the source and the maximum frequency in this experiment is 30 Hz. Note that by reducing the maximum frequency, we can increase the grid spacing. For instance, by choosing a maximum frequency of 8 Hz, the grid spacing can be chosen as 25 m in each direction. The line of

Table 4.7: Performance of migration of one source in the time domain for the Overthrust problem.

Timings forward (s)	Timings backward (s)	Migration time (s)
1156	1168	2324

Table 4.8: Performance of migration of one source in the frequency domain for the Overthrust problem at the highest frequency of 30 Hz.

Timings forward (s)	Timings backward (s)	Migration time (s)
276	294	570

sources is located at a depth of 10 m and is equally spaced with an interval of 18.367 m in the x-direction. The receivers are equally distributed in the two horizontal directions with the same spacing as the sources, at the depth of 20 m. The sampling interval for the modelled seismic data is 4 ms. The maximum simulation time is 0.5 s. For migration in the time domain, an imaging time-step of 0.0005 s was chosen, while in the frequency domain we chose a frequency interval of 2 Hz.

Images produced by reverse-time migration in the time domain and in the frequency domain are shown in Figures 4.11 and 4.12, respectively.

The timings for migration in the time domain are given in Table 4.7 and for migration in the frequency domain in Table 4.8, respectively. The first and second columns show the elapsed time for the forward and backward propagation, correspondingly. For the experiment in the frequency domain, the timings are given for a highest frequency of 30 Hz, since the preconditioned Helmholtz solver requires the longest computational time for this frequency. The timings show that migration in the frequency domain is about 4 times faster than in time domain. Here, we again assume that we have enough computational nodes for the calculation of all frequencies in parallel.

4.6. DISCUSSION

For a single source, migration in the frequency domain would be more time consuming on one computational node as all the frequencies are computed sequentially. However, if enough computational nodes are available, then migration in the frequency domain can compete with migration in the time domain as shown in Table 4.1. Our experiments in the previous section confirm that.

One might wonder what the actual timings would be in the time and in the frequency domain for a given number of computational nodes for a given problem. The wall-clock time for the time domain can be estimated as a function of n_s sources on n_c computational nodes:

$$t^{td} = T^{td}_{\max} \max\left(1, \frac{n_s}{n_c}\right), \tag{4.17}$$

where $T^{td}_{\max}$ is the computational time for one source on one computational node. For

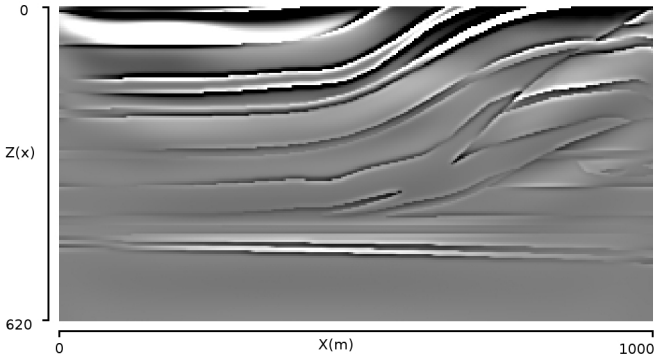


Figure 4.11: Migration in the time domain for a subset of the SEG/EAGE Overthrust problem.

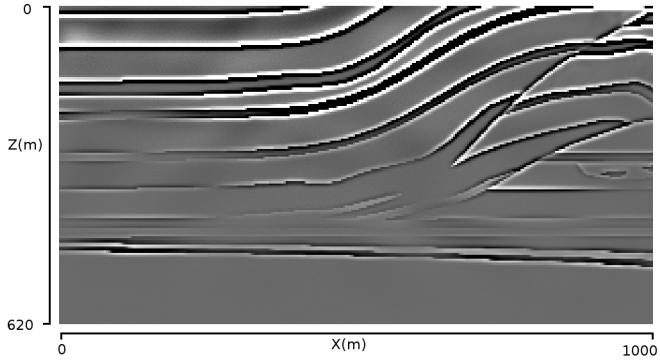


Figure 4.12: Migration in the frequency domain for the Overthrust problem.

the frequency domain it also depends on n_f frequencies:

$$t^{fd} = T^{fd}_{\max} \max\left(1, \frac{n_s n_f}{2n_c}\right), \tag{4.18}$$

where $T^{fd}_{\max}$ is the computational time needed for the maximum frequency on one computational node. Combining the number of sources and computational nodes in a new variable n_c/n_s, the time functions for the time and frequency domain are shown in Figure 4.13, given the experimental results from the Overthrust model. If the number of sources is smaller than the twice the number of compute nodes, then the time domain is faster on our hardware. Otherwise, the frequency-domain approach outperforms the time-domain method.

Moreover, from the experiments described in the previous sections, one can obtain

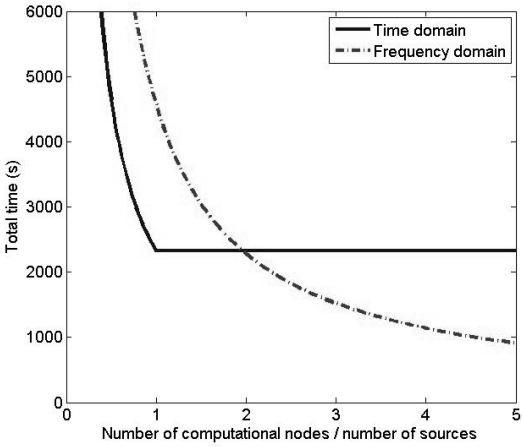


Figure 4.13: Performance of the time-domain scheme (dashed line) and the frequency-domain solver (solid line) as a function of the number of compute nodes divided by the number of sources.

the relation between memory usage and problem size. For the frequency domain, we find

$$\text{Memory(GB)} = 2 \cdot 10^{-7} \cdot N, \tag{4.19}$$

where N is the total number of grid points including absorbing boundary conditions. For the time domain, we obtain

$$\text{Memory(GB)} = 5.5 \cdot 10^{-8} \cdot N. \tag{4.20}$$

On the one hand, the frequency-domain method uses more than twice the memory compared to the time-domain scheme. On the other hand, the migration in the frequency domain does not make use of disk for storing snapshots.

Obviously, there is a trade off between the computational time, the amount of memory and disk usage when considering performance for migration in the time and frequency domain. At this point, the memory needed for the solver is the main bottleneck for solving larger problems.

4.7. CONCLUSIONS

We have considered migration in the frequency domain based on a Krylov subspace solver preconditioned by a shifted-Laplace multigrid method. Its implementation has been compared to the implementation of the reverse-time migration in the time domain in terms of performance and parallelization. The hardware configuration is a many-core CPU connected to two GPUs that contain less memory than the CPU. The implementation in the frequency domain is done by using parallel techniques on a many-core CPU

system and the implementation in the time domain is accelerated using GPUs. The parallelization strategy uses domain decomposition and dynamic load balancing.

The experiments show that migration in the frequency domain on a multi-core CPU is faster than reverse-time migration in the time domain accelerated by GPUs, given enough compute nodes to calculate all frequencies in parallel. This observation is based on our own implementation of both approaches, optimization details and the hardware we had access to. Despite such uncertainties, the methods can obviously compete. We expect to have similar results on different hardware since the GPU-CPU performance ratio is not changing dramatically.

5

ACCELERATING LEAST-SQUARES MIGRATION WITH DECIMATION, GPU AND NEW MATRIX FORMAT

Abstract

In geophysical applications, the interest in least-squares migration (LSM) as an imaging algorithm is increasing due to the demand for more accurate solutions and the development of high-performance computing. The computational engine of LSM is the numerical solution of the 3D Helmholtz equation in the frequency domain. The Helmholtz solver is Bi-CGSTAB preconditioned with the shifted Laplace matrix-dependent multigrid method. In this chapter an efficient LSM algorithm is presented using several enhancements. First of all, a frequency decimation approach is introduced that makes use of redundant information present in the data. It leads to a speedup of LSM, whereas the impact on accuracy is kept minimal. Secondly, a new matrix storage format VCRS (Very Compressed Row Storage) is presented. It not only reduces the size of the stored matrix by a certain factor but also increases the efficiency of the matrix-vector computations. The effects of lossless and lossy compression with a proper choice of the compression parameters are positive. Thirdly, we accelerate the LSM engine by graphics cards (GPUs). A GPU is used as an accelerator, where the data is partially transferred to a GPU to execute a set of operations, or as a replacement, where the complete data is stored in the GPU memory. We demonstrate that using the GPU as a replacement leads to higher speedups and allows us to solve larger problem sizes. Summarizing the effects of each improvement, the resulting speedup can be at least an order of magnitude compared to the original LSM method.

Parts of this chapter have been submitted for publication.

5.1. INTRODUCTION

In the oil and gas industry, one of the challenges is to obtain an accurate image of the subsurface to find hydrocarbons. A source, for instance an explosion, sends acoustic or elastic waves into the ground. Part of the waves is transmitted through the subsurface, another part of the waves is reflected at the interfaces between layers with different properties. Then the wave amplitude is recorded at the receiver locations, for example by geophones. The recorded signal in time forms a seismogram. The data in frequency domain can be easily obtained by the Fourier transform of the signal in time. Using the recorded data, there are several techniques, called depth migration, to map it to the depth domain, given a sufficiently accurate velocity model. The result is a reflectivity image of the subsurface. The techniques include ray based and wave equation based algorithms and can be formulated in time or in frequency domain.

An alternative to the depth migration is least-squares migration (LSM). Least-squares migration [58] has been shown to have the following advantages: (1) it can reduce migration artifacts from a limited recording aperture and/or coarse source and receiver sampling; (2) it can balance the amplitudes of the reflectors; and (3) it can improve the resolution of the migration images. However, least-squares migration is usually considered expensive, because it contains many modeling and migration steps.

Originally, ray-based Kirchhoff operators have been proposed for the modeling and migration in LSM (see e.g. [64], [58]). Recently, in least-squares migration algorithms, wave-equation based operators were used in the time domain (see e.g. [65], [66]) and in the frequency domain (see e.g. [9], [67], [68]). The major advantage of a frequency domain engine is that each frequency can be processed independently in parallel.

With the recent developments in high-performance computing, such as increased memory and processor power of CPUs (Central Processing Units) and the introduction of GPGPUs (General Purpose Graphic Processing Units), it is possible to compute larger and more complex problems and use more sophisticated numerical techniques. For example, the wave equation has been traditionally solved by an explicit time discretization scheme in the time domain requiring large amounts of disk space. In Chapter 4, we have shown that solving the wave equation in the frequency domain, i.e. the Helmholtz equation, can compete with a time domain solver given a sufficient number of parallel computational nodes with a limited usage of disk space. The Helmholtz equation is solved using iterative methods. Many authors showed the suitability of preconditioned Krylov subspace methods to solve the Helmholtz equation, see, for example, [24], [25]. Especially, the shifted Laplace preconditioners improve the convergence of the Krylov subspace methods, see [26], [27], [28], [21].

These methods have shown their applicability on traditional hardware such as a multi-core CPUs, see e.g. [37]. However, the most common type of cluster hardware consists nowadays of a multi-core CPU connected to one or two GPUs. In general, a GPU has a relatively small memory compared to the CPU.

A GPU can be used as a *replacement* for the CPU, or as an *accelerator*. In the first case, the data lives in GPU memory to avoid memory transfers between CPU and GPU memory. We have already investigated this approach for the Helmholtz equation in the frequency domain in Chapter 2 and Chapter 3. The advantage of the migration with a

frequency domain solver is that it does not require large amounts of disk space to store the snapshots. However, a disadvantage is the memory usage of the solver. As GPUs have generally much less memory available than CPUs, this impacts the size of the problem significantly.

In the second case, the GPU is considered as an accelerator, which means that the problem is solved on the CPU while off-loading some computational intensive parts of the algorithm to the GPU. Here, the data is transferred to and from the GPU for each new task. This approach has been investigated for the wave equation in the time domain in Chapter 4. While the simplicity of the time domain algorithm makes it easy to use GPUs of modest size to accelerate the computations, it is not trivial to use GPUs as accelerators for the Helmholtz solver. By using the GPU as an accelerator, the Helmholtz matrices are distributed across two GPUs. The vectors would "live" on the CPU and are transferred when needed to the relevant GPU to execute matrix-vector multiplication or Gauss-Seidel iterations. As a parallel Gauss-Seidel iteration is generally more expensive than a matrix-vector multiplication, it would still pay off to transfer the memory content back-and-forth between GPU and CPU. For a frequency domain solver, the off-loaded matrix-vector multiplication in the well-known CSR (Compressed Sparse Row) format does not result in any significant improvements compared to a many-core CPU due to the data transfer.

The goal of this chapter is to accelerate the least-squares migration algorithm in frequency domain using three different techniques. Firstly, a decimation algorithm is introduced using the redundancy of the data for different frequencies. Secondly, we introduce a *Very Compressed Row Storage* (VCRS) format and consider its effect on the accuracy and performance of the Helmholtz solver, which is our numerical engine for each source and frequency of the LSM algorithm. The third goal is to achieve an improved performance of LSM by using GPUs either as accelerators or as replacements for CPUs.

5.2. LEAST-SQUARES MIGRATION

5.2.1. DESCRIPTION

The solution for a wave problem in a heterogeneous medium is given by the Helmholtz wave equation in three dimensions 1.1, equivalent to

$$A\phi = g, \qquad A = -k^2\sigma^2 - \Delta_h \tag{5.1}$$

where $\sigma = 1/c^2$ is the *slowness* which is the inverse of the square velocity $c = c(x, y, z)$. Here, Δ_h denotes the discrete spatial Laplace operator. A first-order radiation boundary condition is applied $\left(-\frac{\partial}{\partial\eta} - ik\right)\phi = 0$, where η is the outward normal vector to the boundary (see [18]).

The slowness σ can be split into $\sigma = \sigma_0 + r\sigma_0$, where the perturbation of slowness r denotes reflectivity and σ_0 ideally does not produce reflections in the bandwidth of the seismic data. Then, the Helmholtz operator in 5.1 can be written as

$$A = -k^2\sigma_0^2 - 2k^2 r\sigma_0^2 - k^2 r^2\sigma_0^2 - \Delta_h. \tag{5.2}$$

Assuming reflectivity being very small $r << 1$, gives

$$A = -k^2\sigma_0^2 - 2k^2 r\sigma_0^2 - \Delta_h. \tag{5.3}$$

The wavefield $\phi = \phi_0 + \phi_1$ can be split accordingly into a reference and a scattering wavefield, respectively. The reference wavefield ϕ_0 describes the propagation of a wave in a smooth medium without any hard interfaces. The scattering wavefield ϕ_1 represents a wavefield in a medium which is the difference between the actual and the reference medium. Substituting the split to 5.1, gives

$$\left(-k^2\sigma_0^2 - 2k^2 r\sigma_0^2 - \Delta_h\right)(\phi_0 + \phi_1) = f. \tag{5.4}$$

Wave propagation in the reference medium is described by $A_0\phi_0 = g$ with $A_0 = -k^2\sigma_0^2 - \Delta_h$. Then, the Helmholtz equation can be written as

$$A_0\phi_0 - 2k^2 r\sigma_0^2\phi_0 + A_0\phi_1 - 2k^2 r\sigma_0^2\phi_1 = g. \tag{5.5}$$

In the Born approximation the term $2k^2 r\sigma_0^2\phi_1$ is assumed to be negligible, leading to the system of equations

$$\begin{cases} A_0\phi_0 = g, \\ A_0\phi_1 = 2\omega^2 r\sigma_0^2\phi_0, \end{cases} \tag{5.6}$$

which represents the forward modeling. Let us denote $\hat{\phi}(x_s, x_r)$ the solution of the wave equation 5.6 from the source g at the position x_s and recorded at the receiver positions x_r

$$\hat{\phi}(x_s, x_r) = R(x_r)(\phi_0(\omega, x_s) + \phi_1(\omega, x_s)). \tag{5.7}$$

Here, R can be seen as a projection operator to the receiver positions. Then, migration becomes the linear inverse problem of finding the reflectivity r that minimizes the difference between the recorded data $d(\omega, x_s, x_r)$ and the modeled wavefield $\hat{\phi}(\omega, x_s, x_r)$ dependent on the reflectivity r, in a least-squares sense

$$J(r) = \frac{1}{2}\sum_{\omega}\sum_{x_s, x_r} \|d(\omega, x_s, x_r) - \hat{\phi}(\omega, x_s, x_r)(r)\|^2. \tag{5.8}$$

Removing the first arrival from the recorded data and denoting it by d_1, the previous equation is equivalent to

$$J(r) = \frac{1}{2}\sum_{\omega}\sum_{x_s, x_r} \|d_1(\omega, x_s, x_r) - R(x_r)u_1(\omega, x_s, r)\|^2. \tag{5.9}$$

Equation 5.9 can be also written in a matrix form, as

$$J(r) = \frac{1}{2}\left(\mathbf{d} - \mathbf{RFr}\right)^H\left(\mathbf{d} - \mathbf{RFr}\right), \tag{5.10}$$

where $\mathbf{d}$ contains the recorded data without first arrival for each source and receiver pair, $\mathbf{R}$ denotes the projection matrix, $\mathbf{F}$ is the modeling operator from 5.6 and $\mathbf{r}$ contains reflectivity. By setting the gradient of the Jacobian in 5.10 to zero, we obtain the solution to the least-squares problem in a matrix form,

$$\mathbf{F}^H\mathbf{R}^H\mathbf{RFr} = \mathbf{F}^H\mathbf{R}^H\mathbf{d}. \tag{5.11}$$

Here, the operator $\mathbf{R}^H$ denotes the adjoint of the projection operator $\mathbf{R}$ and is defined as "extending the data $\mathbf{d}$ given at the receiver positions to the whole computational domain". The right-hand side is the sum over each source of its subsurface image, that is obtained by migration $\mathbf{F}^H$ of the data at the receiver position corresponding to the given source. Note that migration in the frequency domain is described in detail in Chapter 4. The left-hand side consists of a sum over the forward modeling 5.6 for a given set of reflectivity coefficients for each source, consecutively followed by the migration.

5.2.2. CG AND FREQUENCY DECIMATION

Equation 5.11 represents the normal equation that can be solved iteratively, for example, with a conjugate gradient method (CGNR, see e.g. [33]), which belongs to the family of Krylov subspace methods. For each iteration of the CGNR method a number of matrix-vector multiplications and vector operations are performed. Usually, the iteration matrix is constructed once before the start of the iteration. However, to construct the matrix in 5.11 is very costly, since it requires the number of sources times the number of frequencies of matrix-matrix multiplications. Therefore, we only compute the vector by matrix-vector operations, where the used parts of the matrix are constructed on the fly. Since we consider the problem in the frequency domain, the iteration matrix consists of Helmholtz matrices for each source and a corresponding set of frequencies:

$$\mathbf{F}^H\mathbf{R}^H\mathbf{R}\mathbf{F}\mathbf{r} =: \sum_s \sum_\omega (F^H_{s,\omega} R^H_{s,\omega} R_{s,\omega} F_{s,\omega}\mathbf{r}). \tag{5.12}$$

Next, we assume that there is a redundancy in the seismic data (both modeled and observed) with respect to the frequencies. This assumption has been suggested for migration in frequency domain in [92].

The idea is to reduce the number of frequencies in such a way that for each source several frequencies are discarded. Therefore, we benefit from the redundancy of seismic data, so that the total amount of computations is reduced. We introduce *decimation* over the frequencies and the sources by choosing subsets ω' and s', respectively, and decimation parameter δ. The decimation parameter is defined as a factor by which the original set of frequencies and sources is reduced. Note, that the decimation factor also indicates a reduction of the computational effort. The subset of frequencies and sources, which has size of ω' times s', is constructed by applying a mask consisting of zeros and ones to the original set of size ωs. The number of ones is δ times smaller than the total size of the original subset. The positions of the ones are randomly generated with a normal distribution. A similar technique has been used for random shot decimation for the full-waveform inversion in the time domain in two dimensions (see [100]). The subset of size $\omega' s'$ is changing for each iteration of the CGNR method. This way, the decimation of frequencies and sources is compensated for by the redundancy of the data.

The frequency decimation is only applied to the left-hand side of 5.11, the right-hand side that represents input data is not decimated. Therefore, the iteration matrix with the frequency decimation is given by

$$\mathbf{F}^H\mathbf{R}^H\mathbf{R}\mathbf{F}\mathbf{r} =: \delta \sum_{\omega' \times s'} (F^H_{s',\omega'} R^H_{s',\omega'} R_{s',\omega'} F_{s',\omega'}\mathbf{r}). \tag{5.13}$$

Here, the decimation parameter is used to compensate for the energy of the sum in case of reduction over frequencies and sources. Let us explain it for a simple example with decimation factor $\delta = 2$. In this case, there are two subsets of equal size: one with decimated sources and frequencies, $\Omega_{\text{decimated}} = \omega' \times s'$, and a second one with the removed sources and frequencies, $\Omega_{\text{removed}} = \omega \times s - \omega' \times s'$. The iteration matrix can then be presented as a sum of the two matrices

$$\mathbf{F}^H\mathbf{R}^H\mathbf{R}\mathbf{F}\mathbf{r} = A_{\text{decimated}} + A_{\text{removed}} := \sum_{i\in\Omega_{\text{decimated}}} (F_i^H R_i^H R_i F_i) + \sum_{i\in\Omega_{\text{removed}}} (F_i^H R_i^H R_i F_i). \tag{5.14}$$

Here, the randomness of the decimated subset is important because for a given source the randomly selected frequencies are assumed to represent the spectrum of the source. Randomly selected sources are collected without affecting the total signal energy. We can assume that

$$A_{\text{decimated}} \approx A_{\text{removed}}, \tag{5.15}$$

which leads to

$$\mathbf{F}^H\mathbf{R}^H\mathbf{R}\mathbf{F}\mathbf{r} = 2A_{\text{decimated}}. \tag{5.16}$$

Note that the matrices do not have to be assembled. In our matrix notation, each matrix is implemented as an operator.

5.2.3. HELMHOLTZ SOLVER

The computational engine of the least-squares migration is the damped Helmholtz equation in three dimensions 1.1. The closer the damping parameter α is set to zero, the more difficult it is to solve the Helmholtz equation, as shown in [21]. In this case, we choose $\alpha = 0.05$. From our experiments we have observed that this choice of α does not affect the quality of the image significantly within the LSM framework, however, it leads to faster computational times of the Helmholtz solver, see Chapter 2.

It has been shown in Chapter 2 that the preconditioned Helmholtz solver is parallelizable on CPUs as well as on a single GPU and provides an interesting speedup on parallel architectures.

5.3. MODEL PROBLEMS

Before we dive into the acceleration techniques, let us consider three model problems: one with a "close to constant" velocity field, a second one with significant velocity variation and a realistic third velocity field. These model problems will be used further for illustration and comparison purposes.

The first model problem **MP1** represents a wedge that consists of two dipping interfaces separating in the medium different constant velocities. Figure 5.1 (left) shows the velocities in **MP1**. This model problem represents a very smooth medium with two contrast interfaces. Since the velocities in both parts of the model are constant, the coefficients in the discretization and the prolongation matrices are mainly constant too.

The second model **MP2** is based on the previous model with additional smooth sinusoidal velocity oscillations in each direction, shown in Figure 5.1 (center). The velocity

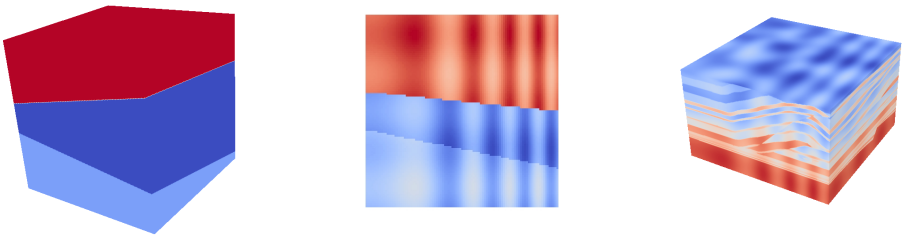


Figure 5.1: Velocity functions for the wedge model problem **MP1** (left), for the modified wedge problem **MP2** (center) and for the modified Overthrust model problem **MP3** (right).

model is heterogeneous, thus the coefficients of the discretization and prolongation matrices are not constant anymore. Since we can vary easily the problem size, this model problem is used to study the effects of compression on the convergence of the preconditioned Bi-CGSTAB method. For our experiments we use this problem in three- as well as in two-dimensions $\mathbf{MP2_{3d}}$ and $\mathbf{MP2_{2d}}$, respectively.

As the background velocity for the third model problem **MP3**, the SEG/EAGE Overthrust velocity model has been chosen, described in [99]. On top of it, additional smooth sinus oscillations are added in each direction. This model problem is close to a realistic problem. The velocity model is heterogeneous as shown on Figure 5.1 (right), so that matrix entries of the discretization and prolongation matrices exhibit many variations and are far from constant. A reason for choosing additional smooth oscillations is to simulate a smooth update in the case of the full waveform inversion algorithm. This way the robustness of the proposed scheme can be validated for this application area.

5.4. VERY COMPRESSED ROW STORAGE (VCRS) FORMAT

As already known, an iterative solver for the wave equation in frequency domain requires more memory than an explicit solver in time domain, especially for a shifted Laplace multigrid preconditioner based on matrix-dependent prolongation. Then, the prolongation and coarse grid-correction matrices need to be stored in memory. Since we are focusing on sparse matrices, in this section we suggest a new format to store the sparse matrices that reduces memory and speeds up the matrix-vector operations.

5.4.1. VCRS DESCRIPTION

First of all, let us briefly describe the well-known CSR (Compressed Sparse Row) format for storage of sparse matrices, e. g. [34], [33]. It consists of two integers and one floating point array. The non-zero elements $a_{i,j}$ of a matrix A are consecutively, row by row, stored in the floating point array `data`. The column index j of each element is stored in an integer array `cidx`. The second integer array `first` contains the location of the beginning of each row. To illustrate this storage format, let us consider a small matrix

from a one-dimensional Poisson equation, with Dirichlet boundary conditions,

$$A = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 2 \end{bmatrix}.$$

The CSR format of this matrix is given by

$$\begin{aligned} \texttt{first} &= \{0\,2\,5\,8\,10\}, \\ \texttt{cidx} &= \{0\,1 \mid 0\,1\,2 \mid 1\,2\,3 \mid 2\,3\}, \\ \texttt{data} &= \{2\,-1 \mid -1\,2\,-1 \mid -1\,2\,-1 \mid -1\,2\}. \end{aligned}$$

Note that the count starts at zero, which can however be easily adjusted to a starting index equal to 1.

To take advantage of the redundancy in the column indices of a matrix constructed by a discretization with finite differences or finite elements on structured meshes, we introduce a *new sparse storage format* inspired by the CSR format. The first array contains the column indices of the first non-zero elements of each row

$$\texttt{col\_offset} = \{0\,0\,1\,2\}.$$

The second array consists of the number of non-zero elements per row

$$\texttt{num\_row} = \{2\,3\,3\,2\}.$$

From an implementation point of view, if it is known that a row does not have more than 255 non-zero elements, then 8 bits integers can be used to reduce the storage of `num_row`. The third array is `col_data` which represents a unique set of indices per row, calculated as the column indices of the non-zero elements in the row `cidx` minus `col_offset`

$$\texttt{col\_data} = \{0\,1 \mid 0\,1\,2\}.$$

Here, the row numbers 0 and 4 have the same set of indices, `col_data` = {0 1}, and the row numbers 1 and 2 have the same set of indices as well, `col_data` = {1 2 1}. To reduce redundancy in this array, we introduce a fourth array `col_pointer` that contains an index per row, pointing at the starting positions in the `col_data`, i.e.,

$$\texttt{col\_pointer} = \{0\,2\,2\,0\}.$$

This approach is also applied to the array containing values of the non-zero elements per row, i. e., the set of values is listed uniquely,

$$\texttt{data} = \{2\,-1 \mid -1\,2\,-1 \mid -1\,2\}.$$

Therefore, also here we need an additional array of pointers per row pointing at the positions of the first non-zero value in a row in `data`, i.e.,

$$\texttt{data\_pointer} = \{0\,2\,2\,5\}.$$

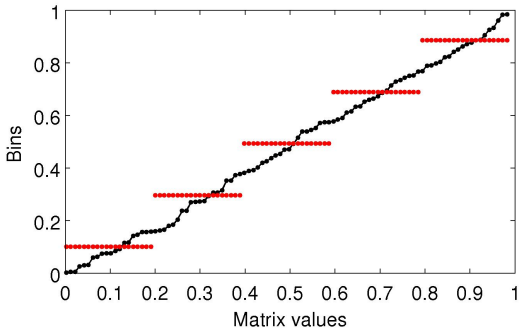


Figure 5.2: Quantization of a matrix with normal distribution of entries in the interval $[0,1]$. The number of bins is equal to 5.

For ease of notation, let us call the new format *VCRS* (Very Compressed Row Storage). At a first glance, it seems that the VCRS format is based on more arrays than the CSR format, six versus three, respectively. However, the large arrays in the CSR format are `cidx` and `data`, and they contain redundant information of repeated indices and values of the matrix. For small matrices, the overhead can be significant, however, for large matrices it can be beneficial to use the VCRS, especially on GPU hardware with limited memory.

Summarizing, the following factors contribute to the usage of the VCRS format:

1. The CSR format of a large matrix contains a large amount of redundancy, especially if the matrix arises from a finite-difference discretization;

2. The amount of redundancy of a matrix can vary depending on the accuracy and storage requirements, giving the opportunity to use a lossy compression;

3. The exact representation of matrices is not required for the preconditioner, an approximation might be sufficient for the convergence of the solver.

The lossy compression of preconditioners can be very beneficial for a GPU-implementation, as it allows to store the data on hardware with a limited amount of memory, but at the same time takes advantage of its speed compared to CPU hardware.

In this paper, we use two mechanisms to adjust the data redundancy: quantization and row classification. Note that these mechanisms can be used separately or in combination.

Quantization is a lossy compression technique that compresses a range of values to a single value, see e.g. [101]. It has well-known applications in image processing and digital signal processing. By lossy compression, as opposed to lossless compression, some information will be lost. However, we need to make sure that the effect of the data loss in lossy compression does not affect the accuracy of the solution. The simplest example of quantization is rounding a real number to the nearest integer value. A similar idea applied to the lossless compression of the column indices was described in [102]. The

quantization technique can be used to make the matrix elements in different rows similar to each other for better compression. The quantization mechanism is based on the maximum and minimum values of a matrix and on a number of so-called bins, or sample intervals. Figure 5.2 illustrates the quantization process of a matrix with values on the interval $[0,1]$. In this example the number of bins is set to 5, meaning there are 5 intervals $[\,0.2(i-1),\,0.2i\,)$, $i = 1,\ldots,5$. The matrix entries are normally distributed between 0 and 1, as shown by the black dots connected with the solid line. By applying quantization, the matrix values that fall in a bin, are assigned to be a new value equal to the bin center. Therefore, instead of the whole range of matrix entries, we only get 5 values. Obviously, the larger number of bins, the more accurate is the representation of matrix entries.

Next, we introduce *row classification* as a mechanism to define similarity of two different matrix rows. Given a sorted array of rows and a tolerance, we can easily search for two rows that are similar within a certain tolerance. The main assumption for row comparison is that the rows have the same number of non-zero elements. Let $R_i = \{a_{i1}\ a_{i2}\ \ldots\ a_{in}\}$ be the i-th row of matrix A of length n and $R_j = \{a_{j1}\ a_{j2}\ \ldots\ a_{jn}\}$ be the j-th row of A.


```
Procedure: IsRowSmaller(R_i,R_j,λ)
for each integer k = 1,...,n do
    if IsComplexValueSmaller(a_ik,a_jk,λ) then
        return true ;
    end
    if IsComplexValueSmaller(a_jk,a_ik,λ) then
        return false ;
    end
    a_jk,a_ik are equal, continue ;
end
rows R_i,R_j are equal ;
return falses ;
```

Algorithm 3: Comparison of two matrix rows

```
Procedure: IsComplexValueSmaller(x,y,λ)
if IsValueSmaller(Re(x),Re(y),λ) then
    return true ;
end
if IsValueSmaller(Re(y),Re(x),λ) then
    return false ;
end
return IsValueSmaller(Im(x),Im(y),λ);
```

Algorithm 4: Comparison of two complex numbers

```
Procedure: IsValueSmaller(x,y,λ)
return x + λ max(|x|,|y|) < y ;
```

Algorithm 5: Comparison of two floating-point numbers

The comparison of two rows is summarized in Algorithm 3. If R_i is not smaller than

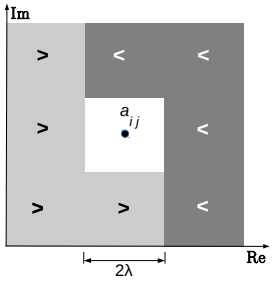


Figure 5.3: Classification of a complex number a_{ij}. The numbers falling in the white square around a_{ij} are assumed to be equal to a_{ij}. Then, a_{ij} is smaller than the numbers in the dark gray area and larger than the numbers in light gray area.

R_j and R_j is not smaller than R_i, then the rows R_i and R_j are *"equal within the given tolerance λ"*. Algorithm 4 then describes the comparison of two complex values and Algorithm 5 compares two floating-point numbers. Figure 5.3 illustrates the classification of a complex number a_{ij}. Within a distance λ the numbers are assumed to be equal to a_{ij}. Then, a_{ij} is smaller than the numbers in the dark gray area in Figure 5.3, and larger than the numbers in the light gray area.

The number of bins and tolerance have influence on

1. the compression factor $c = m/m_c$, which is ratio between the memory usage of the original matrix m and the memory usage of the compressed matrix m_c;
2. the maximum norm of the compression error $\|e\|_\infty = \max_{i,j}(|a_{ij}| - |\bar{a}_{ij}|)$, where a_{ij} are the original matrix entries and $\bar{a}_{ij}$ are entries of the compressed matrix;
3. the computational time;
4. the memory usage;
5. the speedup on modern hardware which is calculated as a ratio of the computational time of the algorithm using the original matrix and of the computational time using the compressed matrix.

Next, we consider the effect of the VCRS format for matrix-vector multiplication, on the multigrid preconditioner, and the preconditioned Bi-CGSTAB method.

5.4.2. MATRIX-VECTOR MULTIPLICATION

Two parameters, the number of bins from quantization and the tolerance λ from row classification, have an impact on the accuracy, performance and memory usage. To illustrate the effect, let us first consider model problem **MP1**. We compare the matrix-vector multiplication in VCRS format with that in the standard CSR format on a CPU. As we have mentioned, matrix-vector multiplication for the discretization matrix on the finest level is performed in a matrix-free way. Therefore, the prolongation matrix on the finest level has been chosen as the test example, since it is the largest matrix that needs to be kept in memory.

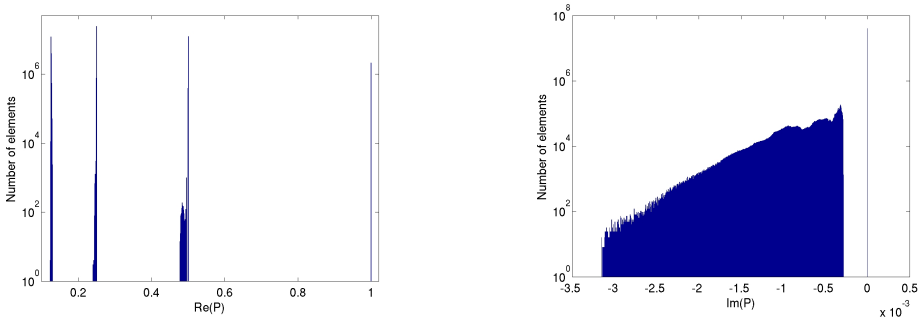


Figure 5.4: Value distribution of the prolongation matrix on the finest level for the model problem **MP3**.

The results are shown in Figure 5.5(left), where the maximum error is given in subfigure (a), the computational time in seconds in subfigure (b), the memory in GB in subfigure (c), the speedup in subfigure (d) and the compression factor in subfigure (e), respectively. As expected, the smaller the tolerance λ used, the more accurate is the representation of the compressed matrix, and the maximum error is reduced. It is also clear that the number of bins does not affect the maximum error for λ smaller than 10^{-5}. Since model problem **MP1** has two large areas with constant velocity, the entries of the prolongation matrix are mostly constant. Even if the entries are not represented accurately because of the larger tolerance or the smaller number of bins, the number of non-zero elements does not change significantly. Therefore, the computational time and thus speedup, memory usage and compression factor are very similar for all combinations of λ and number of bins.

Model problem **MP3** has significant velocity variation, so that the prolongation matrix has different coefficients in each row, as shown in Figure 5.4. Note that the quantization has been done on the real and imaginary parts separately. Obviously, for the real part, the quantization will have a smaller effect than for the imaginary part, since the real values are clustered whereas the imaginary values are distributed over a larger interval.

Figure 5.5(right) shows the accuracy (a), the computational time in seconds (b), the memory in GB (c), the speedup (d) and the compression factor (e). It can be seen in Figure 5.5(a) that by increasing the number of bins, the accuracy of the matrix-vector multiplication is also increasing, since two rows will less likely be similar. Reduced tolerance λ also contributes to the decrease of the maximum error, because the values of entries in rows are becoming closer to each other. With an increasing number of bins and a decreasing tolerance, the compression factor (Figure 5.5(e)) and speedup (Figure 5.5(d)) decrease and, therefore, the computational time increases (Figure 5.5(b)). The compression factor is decreasing, because the more bins are used, the closer the matrix resembles its uncompressed form. Therefore, the memory usage increases (see Figure 5.5(c)). With larger compression factor, the matrix has a smaller size in the memory, so that the cache effect contributes to the performance increase.

Obviously, there is a trade-off between performance and accuracy. The more accurate the compressed matrix, the slower the matrix-vector multiplication. Based on our experiments above, the most reasonable parameter choice would be a tolerance $\lambda = 0.1$

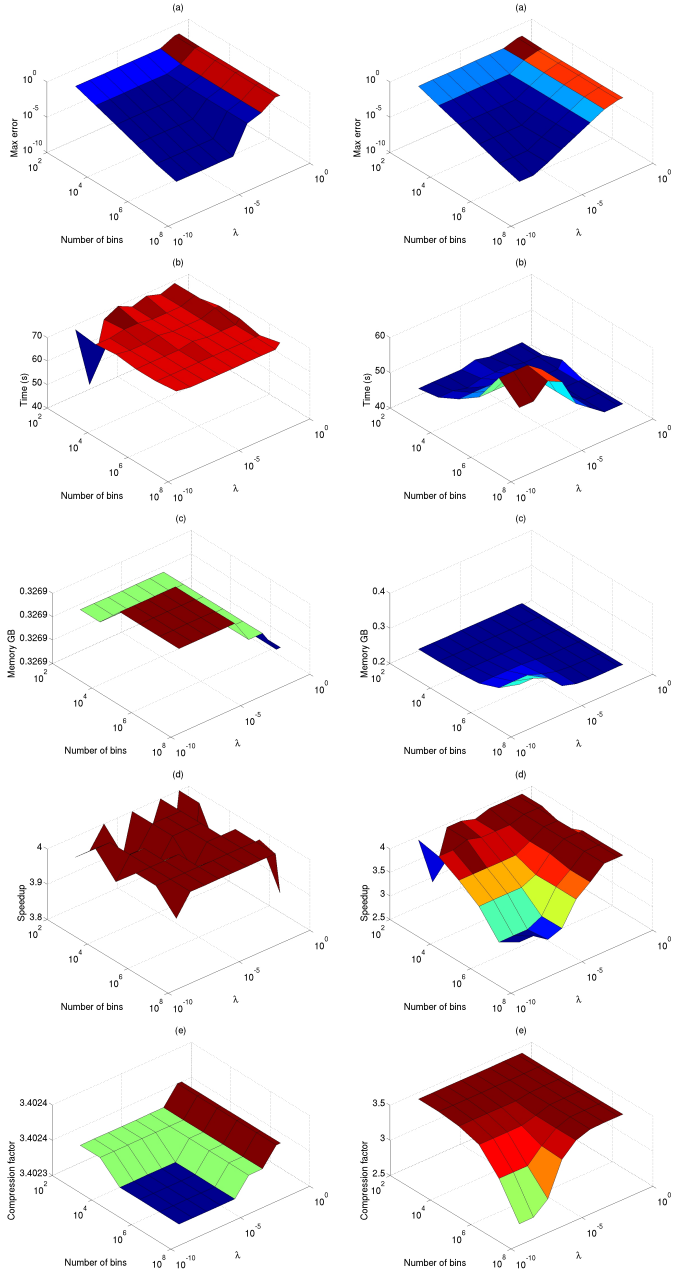


Figure 5.5: Effect of the VCRS format for the matrix-vector multiplication of **MP1** (left) and of **MP3** (right) on the maximum error (a), computational time (b), memory (c), speedup (d), compression factor (e).

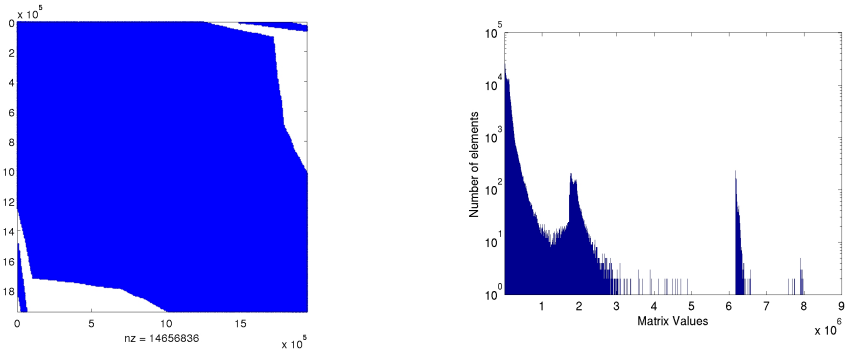


Figure 5.6: The original matrix from pressure solver (left) and its value distribution (right).

and number of bin equals to 10^5. We will investigate the effect of this parameter choice on the multigrid preconditioner and the complete Helmholtz solver.

5

DIFFERENT APPLICATION, RESERVOIR SIMULATION

The VCRS compression can also be used in other applications where the solvers are based on a preconditioned system of linear equations. For example, an iterative solver for linear equations is also an important part of a reservoir simulator, see e.g. [103]. It appears within a Newton step to solve discretized non-linear partial differential equations describing the fluid flow in porous media. The basic partial differential equations include a mass-conservation equation, Darcy's law, and an equation of state relating the fluid pressure to its density. In its original form the values of the matrix are scattered, see Figure 5.6(left). Although the matrix looks full due to the scattered entries, the most common number of non-zero elements per row is equal to 7, however the maximum number of elements per row is 210. The distribution of the matrix values is shown in Figure 5.6 (right). Note, that the matrix has real-valued entries only. It can be seen that there is a large variety of matrix values, that makes the quantization and row classification effective. Using the VCRS format to store this matrix results in two to three times smaller memory requirements and two to three times faster matrix-vector multiplication, depending on the compression parameters. Of course, the effect of the compression parameters on the solver still needs to be investigated.

5.4.3. MULTIGRID METHOD PRECONDITIONER

Since matrix A is implemented in stencil version, it does not require any additional storage. However, the lossy VCRS format may be very useful for the preconditioner, as, because of the matrix-dependent preconditioner, the prolongation and coarse grid matrices have to be stored at each level in the multigrid preconditioner. Note that recomputing those matrices on the fly will result in doubling or even tripling the total computational time.

The multigrid method consists of a setup phase and a run phase. In the setup phase,

Level	Setup (exact)	Run	
		Matrix-free	Compressed
0	M_0, P_0, R_0	M_0, R_0	P_0
1	M_1, P_1, R_1	R_1	M_1,P_1
...	...	...	...
l	M_l, P_l, R_l	R_l	M_l,P_l
...	...	...	...
$m-1$	M_{m-1},P_{m-1},R_{m-1}	R_{m-1}	M_{m-1},P_{m-1}
m	M_m		

Table 5.1: Summary of the exact and compressed matrices used in the matrix-dependent multigrid preconditioner.

the prolongation, restriction and coarse grid correction matrices are constructed, and the matrices are compressed. In the run phase, the multigrid algorithm is actually applied. We assume that the multigrid method is acting on grids at levels l, $l = 0,\ldots,m$, the finest grid being $l = 0$ and the coarsest grid being $l = m$. On each level, except for $l = m$, the matrix-dependent prolongation matrix has to be constructed. The restriction in our implementation is the standard restriction and can be easily used in a matrix-free fashion. At each level, except for $l = 0$, the coarse grid matrix has to be constructed using the Galerkin method 2.6.

We suggest to construct all the coarse-grid matrices exactly in the setup phase and use compression afterwards. This way, errors due to lossy compression will not propagate to the coarse-grid representation of the preconditioned matrix. The algorithm is summarized in Table 5.1. Matrix M_m, on the coarsest level m, is not compressed, since the size of M_m is already small.

To illustrate the compression factor at each level for the matrices M_l and P_l, $l = 1,\ldots,m$, for different values of the row classification parameter λ, let us fix the quantization parameter, i.e. the number of bins is set to 10^3. From Figure 5.5, this choice of bins seems to be reasonable. Table 5.2 shows the compression factors for M and P on each multigrid level. The first column lists the values of the tolerance λ. The multigrid levels are shown in the second column. The third multi-column presents the compression factor, i.e., the percentage of total memory usage of the preconditioner after compression and the maximum error in the matrix elements of the compressed matrix P compared to its exact representation at each multigrid level. The fourth multi-column represents the compression factor, the percentage of the preconditioner memory usage after compression and the maximum error in the matrix elements of the compressed matrix M compared to its exact representation at each multigrid level. The last column shows the total memory usage of the preconditioner for the given tolerance λ. Note, that the total memory usage for the uncompressed matrix is 810 MB.

The compression factor for the prolongation matrix almost does not change on the finest level for different tolerance parameters, since the prolongation coefficients are constructed from the discretization matrix based on the 7-point discretization scheme. On the coarsest level, the stencil becomes a full 27-point stencil and the effect of the compression is more pronounced for smaller tolerance parameter λ. Clearly, the most memory consuming part is the prolongation matrix P_0 on the finest level. However, with

λ	Level	P			M			Memory
		Factor	Memory	Error	Factor	Memory	Error	
0.2	0	3.4	76%	2.7e-2				276 MB
	1	3.2	10%	5.1e-2	24.1	10 %	3.5e-3	
	2	2.9	1 %	1.0e-1	19.3	1.5 %	1.6e-3	
	3	2.3	< 1%	1.6e-1	5.3	< 1%	7.3e-4	
	4				1.9	< 1%	2.4e-4	
0.1	0	3.4	71%	2.3e-2				293 MB
	1	3.3	10%	5.1e-2	20.1	11 %	3.1e-3	
	2	3.1	1.4 %	5.1e-2	6.8	3.4 %	1.5e-3	
	3	2.2	< 1%	6.8e-2	2.4	< 1%	1.0e-3	
	4				1.5	< 1%	2.6e-4	
0.01	0	3.4	57%	4.8e-3				367 MB
	1	2.7	9%	4.8e-3	9.3	19 %	1.8e-4	
	2	2.1	1.4 %	5.1e-3	2.1	10 %	4.7e-5	
	3	1.7	< 1%	4.8e-3	1.6	2 %	1.5e-5	
	4	1.4	< 1%		1.4	< 1%	3.7e-6	
0.001	0	3.4	54%	4.4e-4				382 MB
	1	2.4	9%	4.4e-4	8.4	21 %	4.2e-5	
	2	2.1	1.4 %	5.0e-4	2.1	11 %	1.0e-5	
	3	1.8	< 1%	1.1e-3	1.5	2 %	5.1e-6	
	4	1.4	< 1%	9.2e-4	1.4	< 1%	3.6e-6	

Table 5.2: Model problem **MP3** (3D Overthrust), nbins = 10^3

decreased tolerance λ, the coarse-grid correction matrices need more memory due to larger variety of the matrix entries. As expected, the maximum error is reduced when the tolerance is decreasing in both cases for P and M, respectively. Due to the 27-point stencil on the coarser grids, the coarse-grid correction matrix has a wider spread of the matrix values, which affects the compression factor with respect to the tolerance parameter. The more accurate compression is required, the more memory is needed to store coarse-grid correction matrices. Therefore, to compromise between the accuracy and the memory usage for the multigrid preconditioner, the tolerance λ is chosen equal to 0.1.

A prolongation matrix stored in the VCRS format uses less memory than the original matrix, which can be seen as an approximation. However, the matrix-dependent characteristics of the original prolongation must be preserved for satisfactory convergence of the Helmholtz solver, otherwise it would be easier to just use a standard prolongation matrix. A standard prolongation matrix can be implemented in a matrix-free manner.

5.4.4. PRECONDITIONED BI-CGSTAB

Let us consider the convergence of the preconditioned system,

$$AM^{-1}v = f, \qquad u = Mv,$$

starting with an exact preconditioner and followed by the preconditioner with lossy VCRS compression. From the previous sections, it is clear that standard analysis on a simple homogeneous problem is not sufficient, because here the tolerance and the number of

bins will have a minimum effect on the lossy compression.

TWO-DIMENSIONAL PROBLEM

Let us first consider a two-dimensional variant of the heterogeneous model problem **MP2$_{2d}$**. In this case, we can vary the tolerance and the number of bins and observe the effect on the convergence properties of the preconditioned system. Also, an analytic derivation of the spectral radius of the multigrid iteration matrix is not possible, and therefore, we will use numerical computations to determine it.

In our work we focus on quantitative estimates of the convergence of the multigrid preconditioner, see e.g. [104]. To do this, we construct and analyze the shifted Laplace multigrid operator M^{-1} from equation 2.5 with $\beta_1 = 1$ and $\beta_2 = 0.8$. Two-grid analysis has been widely described in the literature, see [105], [106], [104]. Three-grid analysis has been done in [107]. To see the effect of the lossy VCRS compression, a true multigrid matrix needs to be constructed. The four-grid operator with only pre-smoothing for the F-cycle is given by

$$\begin{aligned} M^{-1} := T_4 &= S_0\,(I_0 - P_0 F_1 V_1 R_0 M_0)\,S_0, \quad \text{with} \\ V_1 &= S_1 (I_1 - P_1 V_2 R_1 M_1) S_1 \\ F_1 &= S_1 (I_1 - P_1 F_2 V_2 R_1 M_1) S_1 \\ V_2 &= S_2 (I_2 - P_2 M_3^{-1} R_2 M_2) S_2 \\ F_2 &= S_2 (I_2 - P_2 M_3^{-1} M_3^{-1} R_2 M_2) S_2, \end{aligned}$$

where S_0, S_1, S_2 are smoothers on the finest, first and second grid, P_0, P_1, P_2 and R_0, R_1, R_2 are prolongation and restriction matrices, respectively. M_0 is the discretization matrix on the finest grid and M_1, M_2, M_3 are respective coarse-grid correction matrices.

Next we compute the spectral radius ρ of AM^{-1}, which is the maximum of the absolute eigenvalues, using the four-grid operator as the preconditioner. The results are summarized in Table 5.3 for different values of λ and different numbers of bins, that are given in the first and second rows. The third row presents the number of iterations of Bi-CGSTAB applied to the preconditioned system, the stopping criterion is 10^{-7} for the relative residual. The fourth row shows the spectral radius of the compressed matrices with VCRS preconditioner. The second column with $\lambda = 0.0$ and #bins= 0.0 represents results for the exact preconditioner. Note that the spectral radius is larger than 1 which means that the multigrid does not converge without the Krylov subspace method for model problem **MP2$_{2d}$**. However, for illustration purposes of the compression it is interesting to consider the spectral radius too. For large tolerance $\lambda = 1.0$ the spectral radius is far from the exact one, meaning that the compressed preconditioned system does not resemble the original preconditioned system and the number of iterations may rapidly increase. For smaller tolerance, the spectral radius of the compressed preconditioned system is very similar to the exact spectral radius and the iteration numbers are almost the same. Note that the number of bins does not have a significant influence on the number of iterations of the Helmholtz solver in this case.

Figure 5.7 illustrates the two extreme cases for **MP2$_{2d}$**, where the number of iterations is most and least affected by the compression. The eigenvalues of the exact operator are

λ	0.0	1.0	0.2	0.2	0.1	0.1	0.01	0.01
# bins	0.0	10^2	10^2	10^6	10^2	10^6	10^2	10^6
# iter	24	124	24	24	24	24	23	24
ρ	1.149	1.966	1.158	1.151	1.155	1.150	1.158	1.150

Table 5.3: Spectral radius ρ for preconditioned system AM^{-1} for four-grids using VCRS format for several λ and number of bins for model problem **MP2$_{2d}$**, $\omega = 10$ Hz. The second column shows the results for the exact preconditioner.

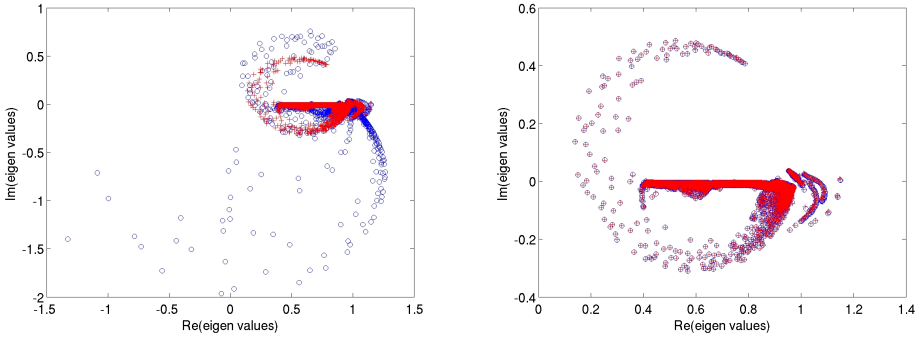


Figure 5.7: Comparison of exact eigenvalues (red crosses) for the preconditioned system AM^{-1} by a four-grid method with approximate eigenvalues (blue circles) using the VCRS format for $\lambda = 1.0$, bins=10^2 (left) and $\lambda = 0.01$, bin=10^6 (right), model problem **MP2$_{2d}$**, $\omega = 10$ Hz.

shown by blue crosses and of the compressed operator by red circles. On the left, the compression parameters are $\lambda = 1.0$, #bins= 10^2, and on the right, $\lambda = 0.01$, #bins=10^6, respectively. Clearly, the more accurate compression (shown on the right) gives a better approximation of the eigenvalues of the exact operator, therefore, the convergence is only slightly affected by the compression. The least accurate compression (shown on the left) affects the eigenvalues of the preconditioned system and therefore, Bi-CGSTAB needs many more iterations.

THREE-DIMENSIONAL PROBLEM

The number of iterations for the more realistic example **MP3** are given in Table 5.4. The results are presented for different values of λ and numbers of bins, that are given in the first and second row. The third row shows the number of iterations of Bi-CGSTAB applied to the preconditioned system, the stopping criterion is 10^{-7} for the relative residual. The second column with $\lambda = 0.0$ and #bins= 0.0 represents results for the exact preconditioner. For tolerance $\lambda = 1.0$ the preconditioned Bi-CGSTAB method does not converge anymore. Therefore, it is advised to use the tolerance smaller than 1.0. In case of **MP3** the number of bins influences the iterations number of the preconditioned Bi-CGSTAB. This happens because the spreading of the matrix entries for the realistic three-dimensional problem is large, therefore the quantization affects many matrix entries significantly. For

λ	0.0	1.0	1.0	0.2	0.2	0.1	0.1	0.01	0.01
# bins	0.0	10^2	10^6	10^2	10^6	10^2	10^6	10^2	10^6
# iter	18	> 400	> 400	60	20	59	20	58	19

Table 5.4: Number of iterations for preconditioned system AM^{-1} using VCRS format for several λ and number of bins for model problem **MP3**, $\omega = 20$ Hz. The second column shows the results for the exact preconditioner.

a relatively small number of bins, the number of iterations is close to the uncompressed case.

5.5. IMPLEMENTATION DETAILS

Presently, a common hardware configuration is a CPU connected to two GPUs that contain less memory than the CPU. By a 'CPU' we refer here to a multi-core CPU and by a 'GPU' to an NVidia general purpose graphics card. We identified the parts of the algorithms that can be accelerated on a GPU and implemented them in CUDA 5.0.

5.5.1. GPU

We consider the GPU as a replacement for the CPU and as an accelerator. In both cases the Bi-CGSTAB algorithm is executed on the CPU, since the storage of temporary vectors takes the most of the memory space. Executing the Bi-CGSTAB method on a GPU would significantly limit the problem size. Therefore, we split the algorithm, where the Krylov solver runs on a CPU and the preconditioner runs on one or more GPUs. We exploited this technique in Chapter 3 and concluded that it reduces the communication between different devices.

When the GPU is used as a *replacement*, the hardware setup consists of one multi-core CPU connected to one GPU. Then the data for the preconditioner, i. e. prolongation and coarse-grid correction matrices, lives in GPU memory to avoid memory transfers between CPU and GPU memory. The process is illustrated in Figure 5.8. A Bi-CGSTAB vector is transferred from the CPU to the GPU, then the multigrid preconditioner is applied. The prolongation and coarse-grid correction matrices are already located in the GPU memory, since they have been transferred in the setup phase. After the preconditioner is applied, the resulting vector is copied back to the CPU memory and the iterations of Bi-CGSTAB continue. As GPUs have generally much less memory available than CPUs, this impacts the size of the problem under consideration. The VCRS format can be used to increase the size of the problem that would fit into GPU memory, moreover, it also leads to increased performance.

When the GPU is used as an *accelerator*, then a multi-core CPU is connected to one or more GPUs, see Figure 5.8. This means that part of the data for the preconditioner lives in the CPU memory, and it is copied to the GPU only for the duration of the relevant computational intensive operation, for example, a multi-colored Gauss-Seidel iteration or matrix-vector multiplication. The data is copied back to the CPU memory once the operation is finished. Note that only the vectors are transferred back-and-forth between

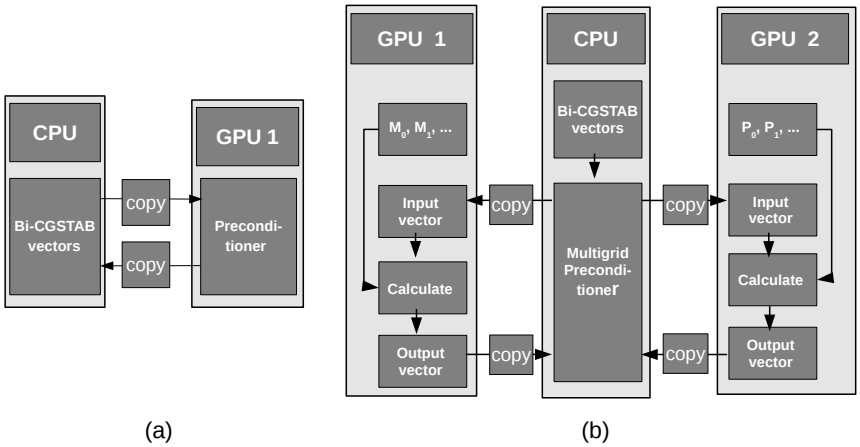


Figure 5.8: GPU as a replacement (a) and as an accelerator (b).

the CPU and GPU memories, the matrices stay in the GPU memory. Also on each GPU, memory for an input and an output vector on the finest grid needs to be allocated to receive vectors from the CPU. Then all the vectors from the preconditioner will fit into this allocated memory. This approach takes advantage of the memory available across GPUs. Using the VCRS format, the matrices become small enough so that they can be evenly distributed across two GPUs. For example, in case of two GPUs connected to one CPU, we suggest to store the prolongation matrices on one GPU and the coarse-grid correction matrices on the other GPU.

Table 5.5 shows the performance of the preconditioned Bi-CGSTAB method for $\mathbf{MP2_{3d}}$ on CPU, GPU as an accelerator and GPU as a replacement. The first column shows the chosen matrix storage format. The second column lists the used hardware. The compression parameters are given in the third and fourth columns. If no compression parameters are given, then lossless compression is applied, where no information is lost due to the matrix compression and the matrix entries are unchanged. Otherwise, the compression parameters belong to lossy compression, where some loss of information is unavoidable. The setup phase, number of iterations and time per iteration are shown in the last three columns of the table. The setup phase for the VCRS format takes longer than the setup phase for the standard CRS because of the construction of additional arrays. The number of iterations does not change significantly for different formats, however it increases slightly in the case of lossy compression. The VCRS format with lossy compression gives the best performance time per iteration without affecting the iteration numbers significantly.

The maximum number of unknowns for the CRS and VCRS formats with lossless and lossy compression are shown for $\mathbf{MP2_{3d}}$ in Table 5.6. The CPU, the GPU as an accelerator and the GPU as a replacement are considered. The first column shows the chosen matrix storage format. The second column lists the used hardware. The compression

Format	Hardware	# bins	λ	Setup (s)	# iter	Time per iter (s)
CRS	CPU	–	–	88	73	5.8
VCRS	CPU	–	–	175	73	4.9
VCRS	CPU	10^3	0.1	150	80	4.6
VCRS	GPU accel	–	–	180	73	3.4
VCRS	GPU accel	10^3	0.1	149	78	3.1
VCRS	GPU repl	–	–	148	73	2.8
VCRS	GPU repl	10^3	0.1	149	76	2.4

Table 5.5: Performance of preconditioned Bi-CGSTAB on CPU, GPU as accelerator and GPU as a replacement for $\mathbf{MP2_{3d}}$ of size 250^3. If no compression parameters are given, then lossless compression is applied.

Format	Hardware	Compression parameters	Maximum size
CRS	CPU	–	74,088,000
VCRS	CPU	$\lambda = 0.1$, # bins=10^3	94,196,375
VCRS	GPU accel	–	19,683,000
VCRS	GPU accel	$\lambda = 0.1$, # bins=10^3	27,000,000
VCRS	GPU repl	–	23,149,125
VCRS	GPU repl	$\lambda = 0.1$, # bins=10^3	32,768,000

Table 5.6: Maximum number of unknowns for $\mathbf{MP2_{3d}}$ for given storage format on different hardware. If no compression parameters are given, then lossless compression is applied.

parameters are given in the third column. If no compression parameters are given, then lossless compression is applied. Across the different hardware platforms, the VCRS format increases the maximum size of the problem compared to the CRS matrix storage. Using GPU as a replacement leads to solving larger problem sizes than using GPU as an accelerator, because the memory needed to store preconditioner matrices is distributed across the GPUs.

Summarizing we can conclude that the VCRS format can be used to reduce the memory for the preconditioner as well as to increase the performance of the preconditioned Bi-CGSTAB on different hardware platforms.

5.5.2. COMMON CODE

The idea of one common hardware code on CPU and GPU has a number of benefits. Just to name few of them, code duplication is kept to a minimum, reducing the possibility to make mistakes, easier maintainability and extensibility. There have been attempts to create a common high-level language for hybrid architectures, for example OpenCL that has been introduced by the Khronos group [38]. For our research the idea of a common code has always been attractive, but its development really started when NVIDIA stopped to support the CUDA-emulator on CPU hardware. By the time our research had started, OpenCL was not commonly available. Therefore, we used our own approach for a common code on CPU and GPU. We assume the code on the CPU is using C++ and the code on the GPU is using CUDA, respectively.

Our implementation is based on the fact that CPUs and GPUs have multiple threads.

Therefore, the multi-threading mechanism can be made abstract on the highest level of the program that describes the numerical algorithms. Code to setup the information about threads is abstracted in macros. On the device level, OpenMP is used for parallel computations on a CPU and CUDA is used on a GPU, respectively. Depending on the device where the part of the program is executed, the high-level functions call sub-functions specific for the device. For example, the synchronization of the threads after computations uses the so-called `omp_barrier()` function for a CPU and `cudaThreadsSynchronize` for a GPU.

Another abstraction technique we use is to simplify argument handling. It uses one structure that contains all arguments of a function. This allows to copy all arguments with one command to a GPU. Examples of a macro and a common code for a simple vector operation are given in Appendices B.1 and B.2. Memory transfers to the right hardware remain the responsibility of the developer.

At the end, there is one code that describes a numerical algorithm, for example the multigrid preconditioner, that compiles for two different architectures, CPU and GPU. Firstly, the developments are done on a CPU in debug mode, where multi-threading is switched off and only one thread is used. This allows to develop on less powerful hardware or when CUDA is not available. The code can be easily expanded to other architectures as long as a subset of all the programming languages is used, in our case meaning the intersection of CUDA, C++, etc. Currently, CUDA is a limiting factor for a program in how many C++ language specific features it can have. As soon as CUDA releases a version that supports more C++ features, it can be immediately used in the common code.

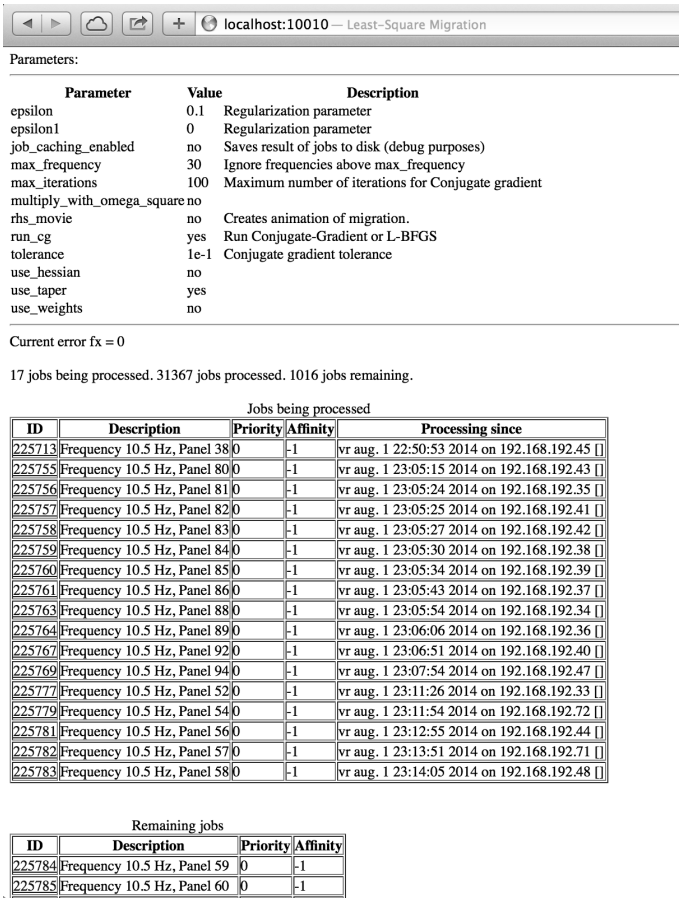
5.5.3. TASK SYSTEM

To run the least-squares migration in parallel we have developed a *task system* that allows to split the work amongst compute nodes and monitor the execution. By a *compute node* we assume a multi-core CPU connected to one or more GPUs, where GPUs can be used as a replacement or as an accelerator.

As we have seen in equation 5.11, the least-squares migration algorithm consists of forward modeling and migrations in frequency domain for each source. Therefore, the highest level of parallelism for LSM consists of parallelization over all sources and frequencies. That means one task consists of computations of one frequency ω_{s_i} for a given source s_i from the set of size ω times s, $s_i \in s$ on one compute node, $\omega_{s_i} \in \omega$, $i = 1, \ldots, N_s$ with the number of sources N_s. In total, we have $N_\omega N_s$ tasks, where N_ω is number of frequencies.

For each frequency, a linear system of equations needs to be solved. We have shown in Chapter 4 that the matrix size and memory requirements are the same for each frequency, but the lower frequencies require less compute time than the higher ones [21]. Here, we assume that one shot for one frequency in the frequency domain fits in one compute node.

For the LSM task system we adapt a client-to-server approach, described in Chapter 4, where clients request tasks to the server. GPU clusters are either heterogeneous or they have to be shared simultaneously amongst the users. For example, the cluster at our disposition, Little Green Machine [108], has the same hardware (with the exception



Parameters:

Parameter	Value	Description
epsilon	0.1	Regularization parameter
epsilon1	0	Regularization parameter
job_caching_enabled	no	Saves result of jobs to disk (debug purposes)
max_frequency	30	Ignore frequencies above max_frequency
max_iterations	100	Maximum number of iterations for Conjugate gradient
multiply_with_omega_square	no	
rhs_movie	no	Creates animation of migration.
run_cg	yes	Run Conjugate-Gradient or L-BFGS
tolerance	1e-1	Conjugate gradient tolerance
use_hessian	no	
use_taper	yes	
use_weights	no	

Current error fx = 0

17 jobs being processed. 31367 jobs processed. 1016 jobs remaining.

Jobs being processed

ID	Description	Priority	Affinity	Processing since
225713	Frequency 10.5 Hz, Panel 38	0	-1	vr aug. 1 22:50:53 2014 on 192.168.192.45 []
225755	Frequency 10.5 Hz, Panel 80	0	-1	vr aug. 1 23:05:15 2014 on 192.168.192.43 []
225756	Frequency 10.5 Hz, Panel 81	0	-1	vr aug. 1 23:05:24 2014 on 192.168.192.35 []
225757	Frequency 10.5 Hz, Panel 82	0	-1	vr aug. 1 23:05:25 2014 on 192.168.192.41 []
225758	Frequency 10.5 Hz, Panel 83	0	-1	vr aug. 1 23:05:27 2014 on 192.168.192.42 []
225759	Frequency 10.5 Hz, Panel 84	0	-1	vr aug. 1 23:05:30 2014 on 192.168.192.38 []
225760	Frequency 10.5 Hz, Panel 85	0	-1	vr aug. 1 23:05:34 2014 on 192.168.192.39 []
225761	Frequency 10.5 Hz, Panel 86	0	-1	vr aug. 1 23:05:43 2014 on 192.168.192.37 []
225763	Frequency 10.5 Hz, Panel 88	0	-1	vr aug. 1 23:05:54 2014 on 192.168.192.34 []
225764	Frequency 10.5 Hz, Panel 89	0	-1	vr aug. 1 23:06:06 2014 on 192.168.192.36 []
225767	Frequency 10.5 Hz, Panel 92	0	-1	vr aug. 1 23:06:51 2014 on 192.168.192.40 []
225769	Frequency 10.5 Hz, Panel 94	0	-1	vr aug. 1 23:07:54 2014 on 192.168.192.47 []
225777	Frequency 10.5 Hz, Panel 52	0	-1	vr aug. 1 23:11:26 2014 on 192.168.192.33 []
225779	Frequency 10.5 Hz, Panel 54	0	-1	vr aug. 1 23:11:54 2014 on 192.168.192.72 []
225781	Frequency 10.5 Hz, Panel 56	0	-1	vr aug. 1 23:12:55 2014 on 192.168.192.44 []
225782	Frequency 10.5 Hz, Panel 57	0	-1	vr aug. 1 23:13:51 2014 on 192.168.192.71 []
225783	Frequency 10.5 Hz, Panel 58	0	-1	vr aug. 1 23:14:05 2014 on 192.168.192.48 []

Remaining jobs

ID	Description	Priority	Affinity
225784	Frequency 10.5 Hz, Panel 59	0	-1
225785	Frequency 10.5 Hz, Panel 60	0	-1

Figure 5.9: Example of a task system monitoring during the LSM execution. The active list contains tasks in table "Jobs being processed". The queue is given in the table "Remaining jobs".

of one node), see Appendix A. Similarly, the GPUs have the same specifications, but one GPU can already be used by a user while the other one remains available. The task system addresses the issue of load balancing. This is also handy when compute nodes are shared between users.

For each CGNR iteration, the server or *master node* creates one task per shot per frequency. Each task is added to a queue. When a client requests a task, a given task is moved from the queue to the active list as illustrated in Figure 5.9. This example shows computation of the right-hand side in 5.11. The active list contains tasks in the table "Jobs being processed". The queue is given in the table "Remaining jobs". The column "Description" shows the task for a given frequency and corresponding source with related receivers called "panel". It can happen that a node will crash due to a hardware failure. In that case, the task will remain on the active list until all the other tasks have

finished. Once that happens, any unfinished task will be moved back to the queue, so that another compute node can take over the uncompleted work. When all tasks have been processed, the master node proceeds to the next CGNR iteration or stops if convergence is achieved.

Using the task system, the frequency decimation in 5.13 can be easily applied, since only the content of the queue will change, the implementation will stay the same. Our implementation has only a single point of failure: the master process. Furthermore, it is possible to adjust parameters on the fly by connecting into the master program using Telnet without having to restart the master program. The Telnet session allows the master program to process commands as well (load/save of restart points, save intermediate results, display statistics, etc.). It is also possible to manipulate the queues with an Internet browser.

5.6. RESULTS

Combining the techniques described above, let us compare the results of the least-squares migration method with and without decimation for the original Overthrust velocity model problem [99]. Both implementations ran on the Little Green Machine [108]. Due to limited resources because of sharing with other students, we take a two-dimensional vertical slice of the Overthrust velocity model for our LSM experiments to make sure it will fit in the available memory. The domain size of the test problem is $[0,\ 20000\ ] \times [0,\ 4650]$, that is discretized on a regular orthogonal grid with 4000 points in x-direction and 930 points in depth.

Both implementations use the VCRS format with the tolerance $\lambda = 0.1$ and the number of bins of 10^7. The decimation is applied with the decimation parameter $\delta = 10$, meaning that the least-squares migration matrix is computed using 10 times less information, and therefore, 10 times faster than without decimation. The stopping criterion for the Helmholtz solver is 10^{-5}. The results for LSM are presented in Figure 5.10, where the implementation without decimation is shown at the top, the implementation with decimation at the center and difference between those two at the bottom, respectively. The results represent the reflectivity of the Overthrust velocity model. The color scale is the same for the three pictures. The results for both methods are very similar, even with the large decimation factor chosen.

The speedup of the LSM with decimation algorithm compared to the original LSM can be calculated theoretically. Assuming the computational time of the right-hand side operator $\mathbf{F^H R^H d}$ from equation 5.12 is equal to t, then the computational time to calculate the least-squares matrix equals $2t$ per iteration of the CG method. In total, the computational time of the original LSM is given by

$$T_{original} = t(2n_{iter} + 1), \tag{5.17}$$

where the n_{iter} denotes the total number of CGNR iterations. There is a possibility to decrease the total time of the LSM. This can be achieved by saving the smooth solution u_0 in 5.6 for each source and frequency to disk while computing the right-hand side and reusing it instead of recomputing for the construction of the LSM matrix. Assuming the

time to read the solution u_0 from the disk negligible, the total time of the LSM reads

$$\hat{T}_{original} = t(n_{iter} + 1). \tag{5.18}$$

Of course, in this case the compression of the solution on the disk becomes important, since the disk space is also usually limited. We recommend to use lossless compression of u_0, to avoid any effects of the lossy compression. This could be investigated in the future.

In case of the LSM with decimation, the time to compute the right-hand side is the same as for the original LSM. However, the computational time of the matrix on the left-hand side of the equation 5.13 is equal to t/δ, where δ is the decimation parameter. Then the total computational time of the LSM with decimation is given by

$$T_{decimation} = t(\frac{n_{iter}}{\delta} + 1). \tag{5.19}$$

The speedup can be defined as the computational time of the original LSM algorithm divided by the the computational time of the LSM with decimation:

$$\text{Speedup} = \frac{T_{original}}{T_{decimation}} = \frac{2n_{iter} + 1}{\frac{n_{iter}}{\delta} + 1} \tag{5.20}$$

For a large number of CGNR iterations, the speedup is approaching the decimation factor 2δ.

Using the VCRS format for the Helmholtz solver gives additional speedup of 4 compared to the standard CSR matrix format. If a GPU is used to improve the performance of the preconditioned Helmholtz solver, then the total speedup can be increased approximately by another factor 3. This brings the total speedup of the decimated LSM to the value of 24δ. In the case $\delta = 10$ the LSM with the above improvements is about 240 times faster than the original algorithm.

5.7. CONCLUSIONS

In this chapter we presented an efficient least-squares migration algorithm using several improvements.

Firstly, a decimation wass done over sources and frequencies to take advantage of the redundant information present in the data during the CGNR iterations, which is used to solve the optimization problem within the LSM framework. This leads to a speedup of the LSM algorithm by the decimation parameter, whereas the impact is kept minimal.

Secondly, we introduced a VCRS (Very Compressed Row Storage) format. The VCRS format not only reduces the size of the stored matrix by a certain factor but also increases the efficiency of the matrix-vector computations. We have investigated the lossless and lossy compression and shown that with the proper choice of the compression parameters the effect of the lossy compression is minimal on the Helmholtz solver which is the Bi-CGSTAB method preconditioned with the shifted Laplace matrix-dependent multigrid method. Also, we also applied the VCRS format to a problem arising from a different application area and showed that the compression may be useful in this case as well.

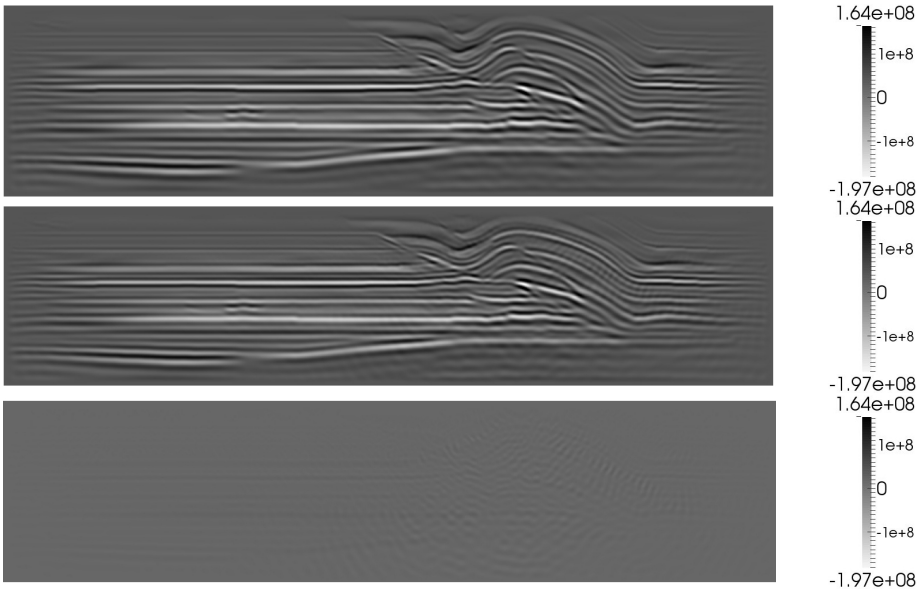


Figure 5.10: The second iteration of LSM for the original Overthrust velocity model without decimation (top), with decimation $\delta = 10$ (center) and the difference between them (bottom).

Moreover, using VCRS allows to accelerate the least-squares migration engine by GPUs. A GPU can be used as an accelerator, in which case the data is partially transferred to a GPU to execute a set of operations, or as a replacement, in which case the complete data is stored in the GPU memory. We have demonstrated that using GPU as a replacement leads to higher speedups and allows to use larger problem sizes than when used as an accelerator. In both cases, the speedup is higher than for the standard CSR matrix format.

Summarizing the effects of each used improvement, it has been shown that the resulting speedup can be at least an order of magnitude compared to the original LSM method, depending on the decimation parameter.

6

CONCLUSIONS

6.1. OVERVIEW

The aim of this thesis is to develop an efficient implementation of the migration algorithm in the frequency domain and the least-squares migration method by using an enhanced Helmholtz solver accelerated on GPUs. This research is a combination of three disciplines: numerical analysis, computational science and geophysics.

First of all, we have considered a two-dimensional Helmholtz equation, for which a GPU implementation of Krylov subspace solvers preconditioned by a shifted Laplace multigrid preconditioner is presented. On CPU, double precision accuracy was used whereas on a GPU computations were in single precision. As Krylov subspace solvers Bi-CGSTAB and IDR(s) have been used. We have seen that both methods are parallelizable on the GPU and have similar speedup of about 40 compared to a single-threaded CPU implementation. It has been shown that a matrix-dependent multigrid preconditioner can be implemented efficiently on the GPU where a speedup of 20 can be achieved for large problems. For the smoothers we have considered parallelizable methods such as weighted Jacobi (ω-Jacobi), multi-colored Gauss-Seidel and damped multi-colored Gauss-Seidel (ω-GS). One iteration of preconditioned IDR(s) is more intensive than one iteration of preconditioned Bi-CGSTAB, however IDR(s) needs fewer iterations so it does not affect the total computation time. To increase the precision of a solver, iterative refinement has been considered. We have shown that iterative refinement with Bi-CGSTAB on the GPU is about 4 times faster than Bi-CGSTAB on the CPU for the same stopping criterion. The same result has been achieved for IDR(s). Moreover, combinations of Krylov subspace solvers on the CPU and GPU and the shifted Laplace multigrid preconditioner on the CPU and GPU have been considered. A GPU Krylov subspace solver with a GPU preconditioner gave the best speedup. For example for the problem size $n = 1024 \times 1024$ Bi-CGSTAB on the GPU with the GPU preconditioner as well as IDR(s) on the GPU with the GPU preconditioner were about 30 times faster than the analogous solvers on the CPU.

Secondly, we have considered a three-dimensional Helmholtz equation, where we presented a multi-GPU implementation of the Bi-CGSTAB solver preconditioned by a shifted Laplace multigrid method. To keep the double precision convergence the Bi-CGSTAB method was implemented on the GPU in double precision and the preconditioner in single precision. We have compared the multi-GPU implementation to a single-GPU and a multi-threaded CPU implementation on a realistic problem size. Two multi-GPU approaches have been considered: a data parallel approach and a split of the algorithm. For the data parallel approach, we were able to solve larger problems than on one GPU and got a better performance than by the multi-threaded CPU implementation. However due to the communication between GPUs and a CPU the resulting speedups have been considerably smaller compared to the single-GPU implementation. To minimize the communication but still be able to solve large problems we have introduced a split of the algorithm. In this case the speedup on the multi-GPUs is similar to the single GPU compared to the multi-core implementation.

As a first geophysical application we have considered migration in the frequency domain based on the enhanced and accelerated Helmholtz solver. Its implementation has been compared to the implementation of the reverse-time migration in the time domain in terms of performance and parallelization. The hardware configuration was a many-core CPU connected to two GPUs that contained less memory than the CPU. The implementation in the frequency domain was using parallel techniques on a many-core CPU and the implementation in the time domain was accelerated using GPUs. The parallelization strategy was based on domain decomposition and dynamic load balancing.

The experiments showed that migration in the frequency domain on a multi-core CPU is faster than reverse-time migration in the time domain accelerated by GPUs, given enough compute nodes to calculate all frequencies in parallel. This observation was based on our own implementation of both approaches, optimization details and the hardware we had access to. Despite such uncertainties, the methods can obviously compete. We expect to have similar results on different hardware since the GPU-CPU performance ratio is not changing dramatically.

As a second geophysical application we have presented an efficient least-squares migration algorithm using several improvements. Firstly, a decimation was done over sources and frequencies to take advantage of redundant information present in the data during the CGNR iterations, which was used to solve the optimization problem within the LSM framework. This led to a speedup of the LSM algorithm by the decimation parameter, whereas the impact was kept minimal. Secondly, we introduced a VCRS (Very Compressed Row Storage) format. The VCRS format not only reduced the size of the stored matrices by a certain factor but also increased the efficiency of the matrix-vector computations. We have investigated the lossless and lossy compression and shown that with the proper choice of the compression parameters the effect of the lossy compression was minimal on the Helmholtz solver, the Bi-CGSTAB method preconditioned with the shifted Laplace matrix-dependent multigrid method. Also, we applied the VCRS format to a problem arising from a different application area and showed that the compression can be useful in this case as well. Thirdly, using VCRS allowed us to accelerate the least-squares migration engine by GPUs. A GPU can be used as an accelerator, in which case the data is partially transferred to a GPU to execute a set of operations, or as

a replacement, in which case the complete data is stored in the GPU memory. We have demonstrated that using the GPU as a replacement led to higher speedups and allowed us to deal with larger problem sizes than when using the GPUs as an accelerator. In both cases, the speedup was higher than for the standard CSR matrix format. Summarizing the effects of each used improvement, it has been shown that the resulting speedup can be at least an order of magnitude compared to the original LSM method, depending on the decimation parameter.

6.2. OUTLOOK

In this work we focused on the Helmholtz equation with constant density, meaning modelling of different rocks using only velocity contrasts. Sometimes in real problems with topography, the air is modelled as a density contrast instead of velocity contrast. In this way, the problem size is reduced since the discretization size depends on minimum velocity. Therefore, as a further development we suggest to include the variable density in the Helmholtz equation. The enhanced shifted Laplace multigrid preconditioner with matrix-dependent prolongation remains unchanged in this case.

To reduce the problem size, one might also use a higher-order discretization of the Helmholtz equation including boundary conditions. It has been shown that these discretizations require overall fewer grid points per wavelength [89] when the underlying velocity model is sufficiently smooth. For this case, we still suggest to use the proposed preconditioner discretized with second-order finite differences as it represents an approximation to the higher-order discretized matrix on the finest level. However, the higher-order discretization will negatively affect the implementation on a GPU since the discretized matrix on the finest grid will need more memory.

Problems with complex geometries sometimes require discretization with finite elements that can lead to an unstructured matrix. In this case, instead of the matrix-dependent prolongation, an algebraic multigrid method within the shifted Laplace preconditioner will be more suitable choice.

During the work on this thesis, GPUs have gradually evolved. The trend is to place more powerful cores on the chip, increase memory size and data transfer bandwidth. However, this progress is rather evolutionary as opposed to revolutionary and more or less predictable. Graphics cards are becoming mainstream for efficient computations at universities. More and more scientists are doing research with computations on GPUs. Some research areas, for instance finance, are better of using GPUs than geophysics, because of the highly computationally intensive (but with low memory requirements) problems occurring. For geophysical applications, the memory size stays one of the biggest challenges for acceleration by means of GPU computations, since it requires adjusting existing algorithms for parallel computations. Many researchers are trying to accelerate the geophysical problems on GPUs, and soon it will also become more popular in the industry.

We anticipate that data transfer will occur much faster in the future. Either by bypassing the PCI-express bus, or by integrating the GPU and the CPU. As far as we know, NVidia does not have a license to produce x86 compatible chips, which means that NVidia must use a different architecture such as ARM (see [109]) in order to integrate

GPUs into a CPU. The high number of GPU cores make fast GPU memory a necessity, and, as a consequence, GPU memory will remain expensive and relatively small compared to the CPU memory. In order to solve large problems by iterative methods on GPUs, we think that the following factors should be present:

- Direct communication between GPUs;
- Highly compressed matrices;
- The compression method used should not impact the performance.

Regarding the implementation of iterative solvers in geophysics, we have the following observations. As compression can not always be lossy, it would make sense to use GPUs for the preconditioner, and let the relatively slow CPU solve the system of equations in a more accurate way. With all the above factors present, solving the elastic wave equation on GPU in the frequency domain becomes feasible in the near future.

7

ACKNOWLEDGMENTS

I would like to thank my advisors Kees Oosterlee and Kees Vuik for giving me the opportunity to do my PhD and for their guidance. We met about 5 years ago at an HPC conference when the subject came up of whether I had a PhD. My advisors understood that it was crucial for me to have a topic that is relevant to my job. This helped me to stay motivated during all this time.

I would also like to thank Wim Mulder for his help especially on the subject of seismic migrations where he took the time to answer my questions.

I would like to thank NVIDIA Corporation, in particular Francois Courteille for giving me access to the latest many-core-multi-GPU architecture where I could test my multi-GPU algorithms on a computer with 8 GPUs.

I would like to thank the Delft University of Technology for granting access to the Little Green Machine, partly funded by the 'Nederlandse Organisatie voor Wetenschappelijk Onderzoek' (NWO, the Netherlands Organisation for Scientific Research) under project number 612.071.305.

Last but not least, thanks to Elena Zhebel for her support, I never thought that having a multigrid specialist at home would be so handy.

A

LITTLE GREEN MACHINE

The Little Green Machine configuration consists of the following nodes interconnected by 40 Gbps Infiniband:

- 1 head node
  - 2 Intel hexacore X5650
  - 24 GB memory
  - 24 TB disk (RAID)
- 1 large RAM node
  - 2 Intel quadcore E5620
  - 96 GB memory
  - 8 TB disk
  - 2 NVIDIA C2070
- 1 secondary Tesla node
  - 2 Intel quadcore E5620
  - 24 GB memory
  - 8 TB disk
  - 2 NVIDIA GTX480
- 1 test node
  - 2 Intel quadcore E5620
  - 24 GB memory
  - 2 TB disk

  - 1 NVIDIA C2050
  - 1 NVIDIA GTX480
- 20 LGM general computing nodes
  - 2 Intel quadcore E5620
  - 24 GB memory
  - 2 TB disk
  - 2 NVIDIA GTX480

B

COMMON CODE

B.1. ABSTRACTION MACROS FOR THE COMMON CODE CPU/GPU

In this appendix we will present an example of abstraction macros for the common code on the CPU and the GPU.

```
#define DECLARE_KERNEL(a,type) \
        extern "C" void a##_CPU(type& args); \
        extern "C" void a##_CUDA(type& args);

////////////////////////////////////////////////
// Macros for C++
////////////////////////////////////////////////

#ifdef COMPILE_FOR_CPP

#include <iostream>
#include <omp.h>

#ifdef _WINDOWS
#define HKPRAGMA(s) __pragma(s)
#else
#define HKPRAGMA(s) _Pragma(s)
#endif

#define KERNEL_SHARED_MEMORY  // empty because on CPU it is a regular memory
#define THREAD_INDEX_INSIDE_BLOCK threadIndex
#define BLOCKIDX_X 0

#ifndef WITH_DEBUG // Release mode
#define KERNEL_THREAD_SYNCHRONIZE HK_OMP_BARRIER
#define BLOCKDIM_X nThreads
#define BEGIN_KERNEL(a,type) extern "C" void a##_CPU(type& args) \
        { \
                HK_OMP_PARALLEL \
                { \
                        const int threadIndex=omp_get_thread_num(); \
```

```
                        const int nThreads=omp_get_num_threads();
#else // WITH_DEBUG // Debug mode
#define KERNEL_THREAD_SYNCHRONIZE
#define BLOCKDIM_X 1
#define BEGIN_KERNEL(a,type) extern "C" void a##_CPU(type& args) \
        { const int threadIndex=0,nThreads=1; {

#endif // WITH_DEBUG

#define END_KERNEL(a,type) } } // to close all brackets

#define KERNEL_SUBFUNCTION

#endif // COMPILE_FOR_CPP

//////////////////////////////////////////////////
// Macros for Cuda
//////////////////////////////////////////////////

#ifdef COMPILE_FOR_CUDA

#define KERNEL_SHARED_MEMORY __shared__
#define THREAD_INDEX_INSIDE_BLOCK (threadIdx.x)
#define BLOCKDIM_X (blockDim.x)
#define BLOCKIDX_X (blockIdx.x)

#define BEGIN_KERNEL(a,type) \
        static __device__ __constant__ type a##_arguments; \
        __global__ void a##_CUDA_KERNEL() \
        { \
                type& args=a##_arguments; \
                const int threadIndex = threadIdx.x+blockIdx.x*blockDim.x; \
                const int nThreads = blockDim.x*gridDim.x;

#define END_KERNEL(a,type) } \
        extern "C" void a##_CUDA(type& args) \
        { \
                cudaMemcpyToSymbol(a##_arguments,&args,sizeof(type)); \
                ...

                // Call the kernel
                a##_CUDA_KERNEL<<<>>>(); \
                ...
        }

#define KERNEL_SUBFUNCTION __device__
#define KERNEL_THREAD_SYNCHRONIZE __syncthreads();

#endif // COMPILE_FOR_CUDA
```

B.2. COMMON CODE CPU/GPU EXAMPLE

In this appendix we present an example of a common code on the CPU and the GPU, that demonstrates the addition of a constant to a vector.

```
#ifdef COMPILE_FOR_CPP
#include <stdio.h>
#endif

struct myfunction_struct
{
        int* _array1;
        int* _array2;
};

KERNEL_SUBFUNCTION static void TestFunction(int* x,int* y,int threadIndex,int nThreads)
{
        *y= *x + 1;
}

BEGIN_KERNEL(MyFunction,myfunction_struct)
{

#ifdef COMPILE_FOR_CPP
        printf("threadIndex=%i\n",threadIndex);
#endif
        for(int i=threadIndex;i<100;i+=nThreads)
        {
                TestFunction(& args._array1[i],& args._array2[i],threadIndex,nThreads);
        }
}
END_KERNEL(MyFunction,myfunction_struct)
```

C

MULTIGRID COEFFICIENTS

The prolongation weights within the multigrid preconditioner are defined as follows.

C.1. MULTIGRID

For Ω_{100}

$$e_{i,j,k} = c^{M00}_{i,j,k} e_{i-1,j,k} + c^{P00}_{i,j,k} e_{i+1,j,k}$$

$$c^{M00}_{i,j,k} = \sigma \frac{d^{M00}_{i,j,k}}{d^{M00}_{i,j,k} + d^{P00}_{i,j,k}}$$

$$c^{P00}_{i,j,k} = \sigma \frac{d^{P00}_{i,j,k}}{d^{M00}_{i,j,k} + d^{P00}_{i,j,k}}$$

$$\sigma = min\left(1, \left|1 - \frac{\sum_{l=1}^{27} a_l}{a_{14}}\right|\right)$$

$$\begin{aligned} d^{M00}_{i,j,k} = max\{&|a_1 + a_4 + a_7 + a_{10} + a_{13} + a_{16} + a_{19} + a_{22} + a_{25}|, \\ &|a_1 + a_{10} + a_{19}|, |a_7 + a_{16} + a_{25}|, |a_1 + a_4 + a_7|, \\ &|a_{19} + a_{22} + a_{25}|, |a_1|, |a_{19}|, |a_7|, |a_{25}|\} \end{aligned}$$

$$\begin{aligned} d^{P00}_{i,j,k} = max\{&|a_3 + a_6 + a_9 + a_{12} + a_{15} + a_{18} + a_{21} + a_{24} + a_{27}|, \\ &|a_3 + a_{12} + a_{21}|, |a_9 + a_{18} + a_{27}|, |a_3 + a_6 + a_9|, \\ &|a_{21} + a_{24} + a_{27}|, |a_3|, |a_{21}|, |a_9|, |a_{27}|\} \end{aligned}$$

C

For Ω_{010}

$$e_{i,j,k} = c^{0M0}_{i,j,k} e_{i,j-1,k} + c^{0P0}_{i,j,k} e_{i,j+1,k}$$

$$c^{0M0}_{i,j,k} = \sigma \frac{d^{0M0}_{i,j,k}}{d^{0M0}_{i,j,k} + d^{0P0}_{i,j,k}}$$

$$c^{0P0}_{i,j,k} = \sigma \frac{d^{0P0}_{i,j,k}}{d^{0M0}_{i,j,k} + d^{0P0}_{i,j,k}}$$

$$\begin{aligned} d^{0M0}_{i,j,k} = max\{&|a_1 + a_2 + a_3 + a_{10} + a_{11} + a_{12} + a_{19} + a_{20} + a_{21}|, \\ &|a_1 + a_{10} + a_{19}|, |a_3 + a_{12} + a_{21}|, |a_1 + a_2 + a_3|, \\ &|a_{19} + a_{20} + a_{21}|, |a_1|, |a_3|, |a_{19}|, |a_{21}|\} \end{aligned}$$

$$\begin{aligned} d^{0P0}_{i,j,k} = max\{&|a_7 + a_8 + a_9 + a_{16} + a_{17} + a_{18} + a_{25} + a_{26} + a_{27}|, \\ &|a_7 + a_{16} + a_{25}|, |a_9 + a_{18} + a_{27}|, |a_7 + a_8 + a_9|, \\ &|a_{25} + a_{26} + a_{27}|, |a_7|, |a_9|, |a_{25}|, |a_{27}|\} \end{aligned}$$

For Ω_{001}

$$e_{i,j,k} = c^{00M}_{i,j,k} e_{i,j,k-1} + c^{00P}_{i,j,k} e_{i,j,k+1}$$

$$c^{00M}_{i,j,k} = \sigma \frac{d^{00M}_{i,j,k}}{d^{00M}_{i,j,k} + d^{00P}_{i,j,k}}$$

$$c^{00P}_{i,j,k} = \sigma \frac{d^{00P}_{i,j,k}}{d^{00M}_{i,j,k} + d^{00P}_{i,j,k}}$$

$$\begin{aligned} d^{00M}_{i,j,k} = max\{&|a_{19} + a_{20} + a_{21} + a_{22} + a_{23} + a_{24} + a_{25} + a_{26} + a_{27}|, \\ &|a_{19} + a_{22} + a_{25}|, |a_{21} + a_{24} + a_{27}|, |a_{19} + a_{20} + a_{21}|, \\ &|a_{25} + a_{26} + a_{27}|, |a_{19}|, |a_{21}|, |a_{25}|, |a_{27}|\} \end{aligned}$$

$$\begin{aligned} d^{00P}_{i,j,k} = max\{&|a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9|, \\ &|a_1 + a_4 + a_7|, |a_3 + a_6 + a_9|, |a_1 + a_2 + a_3|, \\ &|a_7 + a_8 + a_9|, |a_1|, |a_3|, |a_7|, |a_9|\} \end{aligned}$$

For Ω_{110}

$$e_{i,j,k} = c^{MM0}_{i,j,k} e_{i-1,j-1,k} + c^{PM0}_{i,j,k} e_{i+1,j-1,k} + c^{MP0}_{i,j,k} e_{i-1,j+1,k} + c^{PP0}_{i,j,k} e_{i+1,j+1,k}$$

$$c^{MM0}_{i,j,k} = \frac{a^{i,j,k}_{11} c^{M00}_{i,j-1,k} + a^{i,j,k}_{10} + a^{i,j,k}_{13} c^{0M0}_{i-1,j,k}}{a^{i,j,k}_{13} + a^{i,j,k}_{11} + a^{i,j,k}_{10} + a^{i,j,k}_{12} + a^{i,j,k}_{15} + a^{i,j,k}_{17} + a^{i,j,k}_{18} + a^{i,j,k}_{16}}$$

$$c^{PM0}_{i,j,k} = \frac{a^{i,j,k}_{11} c^{P00}_{i,j-1,k} + a^{i,j,k}_{12} + a^{i,j,k}_{15} c^{0M0}_{i+1,j,k}}{a^{i,j,k}_{13} + a^{i,j,k}_{11} + a^{i,j,k}_{10} + a^{i,j,k}_{12} + a^{i,j,k}_{15} + a^{i,j,k}_{17} + a^{i,j,k}_{18} + a^{i,j,k}_{16}}$$

$$c_{i,j,k}^{MP0} = \frac{a_{13}^{i,j,k} c_{i-1,j,k}^{0P0} + a_{17}^{i,j,k} c_{i,j+1,k}^{M00} + a_{16}^{i,j,k}}{a_{13}^{i,j,k} + a_{11}^{i,j,k} + a_{10}^{i,j,k} + a_{12}^{i,j,k} + a_{15}^{i,j,k} + a_{17}^{i,j,k} + a_{18}^{i,j,k} + a_{16}^{i,j,k}}$$

$$c_{i,j,k}^{PP0} = \frac{a_{15}^{i,j,k} c_{i+1,j,k}^{0P0} + a_{17}^{i,j,k} c_{i,j+1,k}^{P00} + a_{18}^{i,j,k}}{a_{13}^{i,j,k} + a_{11}^{i,j,k} + a_{10}^{i,j,k} + a_{12}^{i,j,k} + a_{15}^{i,j,k} + a_{17}^{i,j,k} + a_{18}^{i,j,k} + a_{16}^{i,j,k}}$$

For Ω_{101}

$$e_{i,j,k} = c_{i,j,k}^{P0M} e_{i+1,j,k-1} + c_{i,j,k}^{P0P} e_{i+1,j,k+1} + c_{i,j,k}^{M0P} e_{i-1,j,k+1} + c_{i,j,k}^{M0M} e_{i-1,j,k-1}$$

$$c_{i,j,k}^{M0M} = \frac{a_{22}^{i,j,k} + a_{23}^{i,j,k} c_{i,j,k-1}^{M00} + a_{13}^{i,j,k} c_{i-1,j,k}^{00M}}{a_{13}^{i,j,k} + a_{23}^{i,j,k} + a_{22}^{i,j,k} + a_{24}^{i,j,k} + a_{15}^{i,j,k} + a_{5}^{i,j,k} + a_{6}^{i,j,k} + a_{4}^{i,j,k}}$$

$$c_{i,j,k}^{M0P} = \frac{a_{5}^{i,j,k} c_{i,j,k+1}^{M00} + a_{4}^{i,j,k} + a_{13}^{i,j,k} c_{i-1,j,k}^{00P}}{a_{13}^{i,j,k} + a_{23}^{i,j,k} + a_{22}^{i,j,k} + a_{24}^{i,j,k} + a_{15}^{i,j,k} + a_{5}^{i,j,k} + a_{6}^{i,j,k} + a_{4}^{i,j,k}}$$

$$c_{i,j,k}^{P0M} = \frac{a_{15}^{i,j,k} c_{i+1,j,k}^{00M} + a_{24}^{i,j,k} + a_{23}^{i,j,k} c_{i,j,k-1}^{P00}}{a_{13}^{i,j,k} + a_{23}^{i,j,k} + a_{22}^{i,j,k} + a_{24}^{i,j,k} + a_{15}^{i,j,k} + a_{5}^{i,j,k} + a_{6}^{i,j,k} + a_{4}^{i,j,k}}$$

$$c_{i,j,k}^{P0P} = \frac{a_{15}^{i,j,k} c_{i+1,j,k}^{00P} + a_{5}^{i,j,k} c_{i,j,k+1}^{P00} + a_{6}^{i,j,k}}{a_{13}^{i,j,k} + a_{23}^{i,j,k} + a_{22}^{i,j,k} + a_{24}^{i,j,k} + a_{15}^{i,j,k} + a_{5}^{i,j,k} + a_{6}^{i,j,k} + a_{4}^{i,j,k}}$$

For Ω_{011}

$$e_{i,j,k} = c_{i,j,k}^{0PP} e_{i,j+1,k+1} + c_{i,j,k}^{0MP} e_{i,j-1,k+1} + c_{i,j,k}^{0MM} e_{i,j-1,k-1} + c_{i,j,k}^{0PM} e_{i,j+1,k-1}$$

$$c_{i,j,k}^{0MM} = \frac{a_{11}^{i,j,k} c_{i,j-1,k}^{00M} + a_{23}^{i,j,k} c_{i,j,k-1}^{0M0} + a_{20}^{i,j,k}}{a_{11}^{i,j,k} + a_{23}^{i,j,k} + a_{20}^{i,j,k} + a_{26}^{i,j,k} + a_{17}^{i,j,k} + a_{5}^{i,j,k} + a_{8}^{i,j,k} + a_{2}^{i,j,k}}$$

$$c_{i,j,k}^{0MP} = \frac{a_{5}^{i,j,k} c_{i,j,k+1}^{0M0} + a_{2}^{i,j,k} + a_{11}^{i,j,k} c_{i,j-1,k}^{00P}}{a_{11}^{i,j,k} + a_{23}^{i,j,k} + a_{20}^{i,j,k} + a_{26}^{i,j,k} + a_{17}^{i,j,k} + a_{5}^{i,j,k} + a_{8}^{i,j,k} + a_{2}^{i,j,k}}$$

$$c_{i,j,k}^{0PM} = \frac{a_{17}^{i,j,k} c_{i,j+1,k}^{00M} + a_{23}^{i,j,k} c_{i,j,k-1}^{0P0} + a_{26}^{i,j,k}}{a_{11}^{i,j,k} + a_{23}^{i,j,k} + a_{20}^{i,j,k} + a_{26}^{i,j,k} + a_{17}^{i,j,k} + a_{5}^{i,j,k} + a_{8}^{i,j,k} + a_{2}^{i,j,k}}$$

$$c_{i,j,k}^{0PP} = \frac{a_{17}^{i,j,k} c_{i,j+1,k}^{00P} + a_{8}^{i,j,k} + a_{5}^{i,j,k} c_{i,j,k+1}^{0P0}}{a_{11}^{i,j,k} + a_{23}^{i,j,k} + a_{20}^{i,j,k} + a_{26}^{i,j,k} + a_{17}^{i,j,k} + a_{5}^{i,j,k} + a_{8}^{i,j,k} + a_{2}^{i,j,k}}$$

C

For Ω_{111}

$$
\begin{aligned}
e_{i,j,k} = & c^{MMP}_{i,j,k} e_{i-1,j-1,k+1} + c^{PMP}_{i,j,k} e_{i+1,j-1,k+1} + \\
& c^{MPP}_{i,j,k} e_{i-1,j+1,k+1} + c^{PPP}_{i,j,k} e_{i+1,j+1,k+1} + \\
& c^{MMM}_{i,j,k} e_{i-1,j-1,k-1} + c^{PMM}_{i,j,k} e_{i+1,j-1,k-1} + \\
& c^{MPM}_{i,j,k} e_{i-1,j+1,k-1} + c^{PPM}_{i,j,k} e_{i+1,j+1,k-1}
\end{aligned}
$$

$$
\begin{aligned}
c^{MMM}_{i,j,k} = -\Big[& a^{i,j,k}_{19} + a^{i,j,k}_{10} c^{00M}_{i-1,j-1,k} + a^{i,j,k}_{23} c^{MM0}_{i,j,k-1} + a^{i,j,k}_{22} c^{0M0}_{i-1,j,k-1} + a^{i,j,k}_{20} c^{M00}_{i,j-1,k-1} \\
& + a^{i,j,k}_{11} c^{M0M}_{i,j-1,k} + a^{i,j,k}_{13} c^{0MM}_{i-1,j,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{MMP}_{i,j,k} = -\Big[& a^{i,j,k}_{1} + a^{i,j,k}_{10} c^{00P}_{i-1,j-1,k} + a^{i,j,k}_{11} c^{M0P}_{i,j-1,k} + a^{i,j,k}_{2} c^{M00}_{i,j-1,k+1} + a^{i,j,k}_{4} c^{0M0}_{i-1,j,k+1} \\
& + a^{i,j,k}_{5} c^{MM0}_{i,j,k+1} + a^{i,j,k}_{13} c^{0MP}_{i-1,j,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{MPM}_{i,j,k} = -\Big[& a^{i,j,k}_{25} + a^{i,j,k}_{23} c^{MP0}_{i,j,k-1} + a^{i,j,k}_{26} c^{M00}_{i,j+1,k-1} + a^{i,j,k}_{22} c^{0P0}_{i-1,j,k-1} + a^{i,j,k}_{17} c^{M0M}_{i,j+1,k} \\
& + a^{i,j,k}_{16} c^{00M}_{i-1,j+1,k} + a^{i,j,k}_{13} c^{0PM}_{i-1,j,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{MPP}_{i,j,k} = -\Big[& a^{i,j,k}_{7} + a^{i,j,k}_{17} c^{M0P}_{i,j+1,k} + a^{i,j,k}_{16} c^{00P}_{i-1,j+1,k} + a^{i,j,k}_{8} c^{M00}_{i,j+1,k+1} + a^{i,j,k}_{4} c^{0P0}_{i-1,j,k+1} \\
& + a^{i,j,k}_{5} c^{MP0}_{i,j,k+1} + a^{i,j,k}_{13} c^{0PP}_{i-1,j,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{PMM}_{i,j,k} = -\Big[& a^{i,j,k}_{21} + a^{i,j,k}_{23} c^{PM0}_{i,j,k-1} + a^{i,j,k}_{24} c^{0M0}_{i+1,j,k-1} + a^{i,j,k}_{20} c^{P00}_{i,j-1,k-1} + a^{i,j,k}_{15} c^{0MM}_{i+1,j,k} \\
& + a^{i,j,k}_{11} c^{P0M}_{i,j-1,k} + a^{i,j,k}_{12} c^{00M}_{i+1,j-1,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{PMP}_{i,j,k} = -\Big[& a^{i,j,k}_{3} + a^{i,j,k}_{15} c^{0MP}_{i+1,j,k} + a^{i,j,k}_{11} c^{P0P}_{i,j-1,k} + a^{i,j,k}_{6} c^{0M0}_{i+1,j,k+1} + a^{i,j,k}_{2} c^{P00}_{i,j-1,k+1} \\
& + a^{i,j,k}_{5} c^{PM0}_{i,j,k+1} + a^{i,j,k}_{12} c^{00P}_{i+1,j-1,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{PPM}_{i,j,k} = -\Big[& a^{i,j,k}_{27} + a^{i,j,k}_{23} c^{PP0}_{i,j,k-1} + a^{i,j,k}_{26} c^{P00}_{i,j+1,k-1} + a^{i,j,k}_{24} c^{0P0}_{i+1,j,k-1} + a^{i,j,k}_{18} c^{00M}_{i+1,j+1,k} \\
& + a^{i,j,k}_{17} c^{P0M}_{i,j+1,k} + a^{i,j,k}_{15} c^{0PM}_{i+1,j,k} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

$$
\begin{aligned}
c^{PPP}_{i,j,k} = -\Big[& a^{i,j,k}_{9} + a^{i,j,k}_{18} c^{00P}_{i+1,j+1,k} + a^{i,j,k}_{17} c^{P0P}_{i,j+1,k} + a^{i,j,k}_{15} c^{0PP}_{i+1,j,k} + a^{i,j,k}_{8} c^{P00}_{i,j+1,k+1} \\
& + a^{i,j,k}_{6} c^{0P0}_{i+1,j,k+1} + a^{i,j,k}_{5} c^{PP0}_{i,j,k+1} \Big] / a^{i,j,k}_{14}
\end{aligned}
$$

SUMMARY

REDUCTION OF COMPUTING TIME FOR SEISMIC APPLICATIONS BASED ON THE HELMHOLTZ EQUATION BY GRAPHICS PROCESSING UNITS

HANS PETER KNIBBE

The oil and gas industry makes use of computational intensive algorithms to provide an image of the subsurface. The image is obtained by sending wave energy into the subsurface and recording the signal required for a seismic wave to reflect back to the surface from the Earth interfaces that may have different physical properties. A seismic wave is usually generated by shots of known frequencies, placed close to the surface on land or close to the water surface in the sea. Returning waves are usually recorded in time by hydrophones in a marine environment or by geophones during land acquisition. The goal of seismic imaging is to transform the seismograms to a spatial image of the subsurface. Migration algorithms produce an image of the subsurface given the seismic data measured at the surface.

In this thesis we focus on solving the Helmholtz equation which represents the wave propagation in the frequency domain. We can easily convert from the time domain to the frequency domain and vice-versa using the Fourier transformation. A discretization with second-order finite differences gives a sparse linear system of equations that needs to be solved for each frequency. Two- as well as three-dimensional problems are considered. Krylov subspace methods such as Bi-CGSTAB and IDR(s) have been chosen as solvers. Since the convergence of the Krylov subspace solvers deteriorates with an increasing wave number, a shifted Laplacian multigrid preconditioner is used to improve the convergence. Here, we extend the matrix-dependent multigrid method to solve complex-valued matrices in three dimensions. As the smoother, we have considered parallelizable methods such as weighted Jacobi (ω-Jacobi), multi-colored Gauss-Seidel and damped multi-colored Gauss-Seidel (ω-GS).

The implementation of the preconditioned solver on a CPU (Central Processing Unit) is compared to an implementation on the GPU (Graphics Processing Units or graphics card) using CUDA (Compute Unified Device Architecture). The results show that in two dimensions the preconditioned Bi-CGSTAB method on the GPU as well as the preconditioned IDR(s) method on a single GPU are about 30 times faster than on a single-threaded CPU. To achieve double precision accuracy on the GPU we have used the iterative refinement in Chapter 2.

However, problems of realistic size are too large to fit in the memory of one GPU. One solution for this is to use multiple GPUs. A currently widely used architecture consists

of a multi-core computer connected to one or at most two GPUs. Moreover, those GPUs can have different characteristics and memory sizes. A setup with four or more identical GPUs is rather uncommon, but it would be ideal from a memory point of view. It would imply that the maximum memory is four times more than on a single GPU. However GPUs are connected to a PCI bus and in some cases two GPUs share the same PCI bus, which creates data transfer limitations. To summarize, using multi-GPUs increases the total memory size but data transfer problems appear. Therefore, in Chapter 3 we consider different multi-GPU approaches and understand how data transfer affects the performance of a Krylov subspace solver with shifted Laplace multigrid preconditioner for the three-dimensional Helmholtz equation using CUDA (Compute Unified Device Architecture). Two multi-GPU approaches are considered: data parallelism and split of the algorithm. Their implementations on a multi-GPU architecture are compared to a multi-threaded CPU and single GPU implementation. The results show that the data parallel implementation suffers from communication between GPUs and the CPU, but is still a number of times faster compared to many-cores. The split of the algorithm across GPUs limits communication and delivers speedups comparable to a single GPU implementation.

As a geophysical application which requires an efficient numerical method we consider 3-D reverse time migration with the constant-density acoustic wave equation in Chapter 4. The idea of migration in the time domain is to calculate the forward wavefield by injecting the source wavelet. Secondly, we compute the wavefield backward in time by injecting the recorded signal at the receiver locations. Subsequently, we cross-correlate the forward and backward wavefields at given timesteps. An explicit finite-difference scheme in the time domain is a common choice. However, it requires a significant amount of disk space to store the forward wavefields. The advantage of migration with a frequency domain solver is that it does not require large amounts of disk space to store the snapshots. However, a disadvantage is the memory usage of the solver. As GPUs have generally much less memory available than CPUs, this impacts the size of the problem significantly.

The frequency-domain approach simplifies the correlation of the source and receiver wavefields, but requires the solution of a large sparse linear system of equations. The question is whether migration in the frequency domain can compete with a time-domain implementation when both are performed on a parallel architecture. Both methods are naturally parallel over shots, but the frequency-domain method is also parallel over frequencies. If we have a sufficiently large number of compute nodes, we can compute the result for each frequency in parallel and the required time is dominated by the number of iterations for the highest frequency. Here, GPUs are used as accelerators and not as independent compute nodes. We optimize the throughput of the latter with dynamic load balancing, asynchronous I/O and compression of snapshots. Since the frequency-domain solver employs a matrix-dependent prolongation, the coarse grid operators required more storage than available on GPUs for problems of realistic sizes.

An alternative to the depth migration is least-squares migration (LSM). LSM was introduced as a bridge between full waveform inversion and migration. Like migration, LSM does not attempt to retrieve the background velocity model, however, like full waveform inversion the modeled data should fit the observations.

In Chapter 5 an efficient LSM algorithm is presented using several enhancements. Firstly, a frequency decimation approach is introduced that makes use of the redundant information present in the data. It leads to a speedup of LSM, whereas the impact on accuracy is kept minimal.

Secondly, to store the sparse discretization and matrix-dependent prolongation matrices efficiently, a new matrix storage format VCRS (Very Compressed Row Storage) is presented. This format is capable of handling lossless compression. It does not only reduce the size of the stored matrix by a certain factor but also increases the efficiency of the matrix-vector computations. The study shows that the effect of lossless and lossy compression with a proper choice of the compression parameters are positive.

Thirdly, we accelerate the LSM engine by GPUs. A GPU is used as an accelerator, where the data is partially transferred to a GPU to execute a set of operations, or as a replacement, where the complete data is stored in the GPU memory. We demonstrate that using GPU as a replacement leads to higher speedups and allows us to solve larger problem sizes. Summarizing the effects of each improvement, the resulting speedup can be at least an order of magnitude compared to the original LSM method.

SAMENVATTING

VERMINDERING VAN REKENTIJD VOOR SEISMISCHE TOEPASSINGEN GEBASEERD OP DE HELMHOLTZVERGELIJKING MET BEHULP VAN GRAFISCHE KAARTEN

HANS PETER KNIBBE

De olie- en gasindustrie maakt gebruik van computerintensieve algoritmen om een beeld van de ondergrond te leveren. Het beeld wordt verkregen door het sturen van golfenergie in de ondergrond en door het opnemen van het signaal van de seismische golf dat terugkeert naar het oppervlak van de aarde. Een seismische golf wordt meestal gegenereerd door kleine explosies met bekende frequenties, geplaatst dicht bij de oppervlakte op land of dicht bij het wateroppervlak in zee. Terugkerende golven worden meestal opgenomen door hydrofoons in een maritieme omgeving of door geofoons bij acquisitie op land. Bij seismische beeldvorming worden de seismogrammen getransformeerd tot een ruimtelijk beeld van de ondergrond. Migratie-algoritmen produceren een beeld van de ondergrond gebaseerd op de seismische gegevens gemeten aan het oppervlak.

In dit proefschrift richten we ons op het oplossen van de Helmholtzvergelijking die de golfvoortplanting in het frequentiedomein representeert. We kunnen gemakkelijk van het tijdsdomein naar het frequentiedomein en vice versa transformeren, met behulp van de Fouriertransformatie. Een discretisatie met tweede-orde eindige differenties levert een dunbezet lineair matrixsysteem dat moet worden opgelost voor iedere frequentie. Zowel twee- als drie-dimensionale seismische problemen worden hier beschouwd en opgelost. Krylov-deelruimte methoden zoals Bi-CGSTAB en IDR (s) zijn gekozen voor deze taak. Aangezien de convergentie van de Krylov methoden met een toename van het golfgetal verslechtert, wordt een verschoven Laplace multirooster-voorconditionering gebruikt voor het verbeteren van de convergentie. Hier breiden we de matrix-afhankelijke multiroostermethode uit om complexe matrices op te lossen in drie dimensies. Als multirooster-smoother, hebben we paralleliseerbare methoden zoals de gedempte Jacobi (ω-Jacobi), meerkleuren Gauss-Seidel en de gedempte meerkleuren Gauss-Seidel (ω-GS) methode gekozen.

De rekentijd van de geprecondition eerde oplosmethode op een CPU (Central Processing Unit) wordt vergeleken met die op de GPU (Graphics Processing Units of grafische kaart) op basis van een implementatie onder het CUDA (Compute Unified Device Architecture) systeem. De resultaten laten zien dat in twee dimensies de voorgeconditioneerde Bi-CGSTAB methode evenals de voorgeconditioneerde IDR (s) methode op een enkele GPU ongeveer 30 keer sneller is dan op een enkele CPU. Om hogere nauwkeurigheid op de GPU te bereiken hebben we de iteratieve verfijningstechniek gebruikt in

Hoofdstuk 2.

Problemen van realistische omvang zijn te groot voor het geheugen van een GPU. Een oplossing hiervoor is om meerdere GPU's tegelijkertijd te gebruiken. Een op dit moment veel gebruikte rekenarchitectuur bestaat uit een computer met meerdere rekenkernen, verbonden met één of twee GPU's. Bovendien kunnen die GPU's verschillende kenmerken en geheugengrootte hebben. Een setup met vier of meer identieke GPU's is ongebruikelijk, maar het zou ideaal zijn vanuit het oogpunt van geheugencapaciteit. Het zou impliceren dat de maximale hoeveelheid geheugen vier keer meer zou zijn dan op een enkele GPU. Echter GPU's zijn verbonden met een PCI bus en in sommige gevallen delen twee GPU's dezelfde PCI bus, waardoor een data-overdrachtslimiet gecreëerd zou worden. Samenvattend verhoogt het gebruik van multi-GPU's de totale grootte van het geheugen maar veroorzaakt tevens data-overdrachtproblemen. Om deze redenen beschouwen we in Hoofdstuk 3 verschillende multi-GPU benaderingen en onderzoeken we hoe data-overdracht de rekentijd van een Krylov deelruimte-methode met verschoven Laplace multirooster-voorconditionering beïnvloedt voor een drie-dimensionale Helmholtzvergelijking. Twee multi-GPU implementaties worden beschouwd: data parallellisme en splitsing van het algoritme. De implementaties op een multi-GPU architectuur worden vergeleken met een moderne CPU implementatie. Uit de resultaten blijkt dat zogeheten 'data parallelle implementatie' lijdt onder de communicatie tussen de GPU's en CPU, maar zij is nog een aantal malen sneller in vergelijking met de CPU implementatie. Het uitsplitsen van het rekenalgoritme over de GPU's reduceert de communicatie en levert rekentijdverbeteringen op die vergelijkbaar zijn met een implementatie op een enkele GPU.

Als geofysische toepassing waarvoor we een efficiënte numerieke methode zoeken, kiezen we 3-D inverse tijdsmigratie op basis van de golfvergelijking in Hoofdstuk 4. Het idee achter de migratie in het tijdsdomein is het berekenen van een oplossing door zowel voorwaarts als terugwaarts in de tijd te rekenen. Door middel van kruiscorrelaties van de gevonden oplossingen na de voorwaartse en terugwaartse berekeningen wordt de uiteindelijke oplossing bepaald. Rekenen in het tijdsdomein vereist een aanzienlijke hoeveelheid geheugenruimte voor het opslaan van de voorwaartse oplossingen. Het voordeel van migratie in het frequentiedomein is dat die grote hoeveelheden geheugenruimte niet nodig zijn. Een nadeel is echter het geheugengebruik van de rekenmethode voor de Helmholtzvergelijking. Aangezien GPU's over het algemeen veel minder geheugen beschikbaar hebben dan CPU's, heeft dit aanzienlijke gevolgen voor de omvang van het rekenprobleem in kwestie.

De frequentiedomein-benadering vereenvoudigt de correlatie van de oplossingen, maar vereist de oplossing van een groot dunbezet lineair vergelijkingssysteem. De onderzoeksvraag was of migratie in het frequentiedomein kan concurreren met een implementatie in het tijdsdomein als beide worden uitgevoerd op een parallelle rekenarchitectuur. Beide methoden zijn van nature parallel over de input, maar de frequentie-domein methode is ook parallel over de door te rekenen frequenties. Als we een redelijk groot aantal rekenprocessoren hebben, kunnen we oplossingen voor iedere frequentie parallel berekenen en de vereiste rekentijd wordt gedomineerd door het aantal iteraties voor de hoogste frequentie. GPU's worden hier gebruikt als rekentaakversnellers. We optimaliseren de doorvoer van de rekentaken met dynamische taakverdeling, asynchrone I/O

en compressie van de opnamen.

Aangezien de multirooster gebaseerde oplosmethode in het frequentiedomein op een matrix-afhankelijke prolongatie is gebaseerd, wordt voor problemen van realistische grootte meer opslagruimte benodigd dan beschikbaar is op GPU's. Een alternatief voor algoritmen op basis van de inverse migratie is een kleinste-kwadraten migratiemethode (LSM). LSM werd geïntroduceerd als een overgang tussen volledige golffront inversie en migratie. In Hoofdstuk 5 word een efficiënt LSM-algoritme gepresenteerd op basis van een aantal verbeteringen. In de eerste plaats is een benadering ingevoerd die gebruik maakt van redundante informatie die aanwezig is in de seismische data. Het leidt tot een versnelling van LSM, terwijl de impact op de nauwkeurigheid van de oplossing minimaal kan worden gehouden. Ten tweede, om de discretisatie en matrix-afhankelijke prolongatiematrices efficiënt op te slaan, wordt een nieuw matrixformaat "VCRS"(Very Compressed Row Storage) gepresenteerd. Dit formaat is geschikt voor compressie. Het vermindert niet alleen de grootte van de opgeslagen matrix met een bepaalde factor, maar het vergroot ook de efficiëntie van matrix-vector berekeningen. Ten derde versnellen we de LSM rekentechnieken door gebruik te maken van GPU's. De GPU wordt gebruikt als een rekenversneller, waarbij data gedeeltelijk aan een GPU wordt overgedragen om een set van berekeningen uit te voeren, of als een zogenaamde vervanging, waar alle data wordt opgeslagen in het geheugen van de GPU. We tonen aan dat, met de verbeteringen van de rekentechnieken, het gebruik van GPU's als vervanging leidt tot hogere versnellingen en dat het grotere rekenproblemen laat oplossen. Met de bovengenoemde verbeteringen kan de rekentijd tenminste een orde grootte verkleind worden vergeleken met de oorspronkelijke LSM methode.

CURRICULUM VITÆ

Hans Peter KNIBBE

Born on 07.07.1972 in Reims, France

EDUCATION

2009	–	present	Delft University of Technology, The Netherlands, PhD student in Scientific Computing/Applied Mathematics, subject: *Reduction of computing time for seismic applications based on the Helmholtz equation by Graphics Processing Units*, advisors: Prof.dr.ir. C. Vuik and Prof.dr.ir. C. W. Oosterlee, Department of Applied Mathematics, Faculty of Electrical Engineering, Mathematics and Computer Science
1994	–	1999	Delft University of Technology, The Netherlands, Master of Computer Science
1992	–	1994	IUT Orsay (University Paris XI), France, Degree in Physics (Mesures Physiques, comparable to Bsc)

PROFESSIONAL EXPERIENCE

2013	–	present	Source Contracting, Geophysical Software Consultant
2009	–	2013	Headwave, Senior Algorithm Developer
2008	–	2009	PDS, Senior Software Engineer
2007	–	2008	Fugro-Jason Netherlands BV, Senior Computer Scientist R&D
2004	–	2007	Fugro-Jason Netherlands BV, Computer Scientist R&D
1999	–	2004	LogicaCMG Rotterdam, Computer Scientist
1996	–	1997	QQQ Delft, Programmer

LIST OF PUBLICATIONS

H. Knibbe, C. W. Oosterlee, and C. Vuik. GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method. *Journal of Computational and Applied Mathematics*, 236:281-293, 2011.

H. Knibbe, C. Vuik, and C. W. Oosterlee. 3D Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method on multi-CPUs. *In A. Cangiani, R. L. Davidchack, E. Georgoulis, A. N. Gorban, J. Levesley, and M . V. Tretyakov, editors, in Proceedings of ENUMATH 2011, the 9th European Conference on Numerical Mathematics and Advanced Applications, Leicester, September 2011*, pages 653-661. Springer-Verlag Berlin Heidelberg, 2013.

H. Knibbe, W. A. Mulder, C. W. Oosterlee, and C. Vuik. Closing the per- formance gap between an iterative frequency-domain solver and an explicit time-domain scheme for 3-d migration on parallel architectures. *Geophysics*, 79:47-61, 2014.

H. Knibbe, C. Vuik, and C. W. Oosterlee. Accelerating Least-Squares Migration with Decimation, GPU and New Matrix Format. *Submitted for publication*

REFERENCES

[1] J. G. Hagedoorn, *A process of seismic reflection interpretation,* Geophysical Prospecting **2**, 85 (1954).

[2] J. F. Claerbout, *Towards a unified theory of reflector mapping,* Geophysics **36**, 467 (1971).

[3] D. E. Baysal, D. Kosloff, and J. W. C. Sherwood, *Reverse time migration,* Geophysics **48**, 1514 (1983), http://library.seg.org/doi/pdf/10.1190/1.1441434 .

[4] N. D. Whitemore, *Iterative depth imaging by backward time propagation,* 53rd Annual International Meeting, SEG, Expanded Abstracts , 382 (1983).

[5] P. Lailly, *The seismic inverse problem as a sequence of before stack migration,* in *in Proc. Conf. on Inverse Scattering, Theory and Applications, SIAM* (Philadelphia, 1983).

[6] A. Tarantola, *Inversion of seismic reflection data in the acoustic approximation,* Geophysics **49**, 1259 (1984).

[7] G. Beylkin, *Imaging of discontinuities in the inverse scattering problem by inversion of a causal generalized Radon transform,* Journal of Mathematical Physics **26**, 99 (1985).

[8] G. Beylkin and R. Burridge, *Linearized inverse scattering problems in acoustics and elasticity,* Wave Motion **12**, 15 (1990).

[9] R. E. Plessix and W. A. Mulder, *Frequency-domain finite-difference amplitude-preserving migration,* Geophysical Journal International **157**, 975 (2004).

[10] V. Červený, *Seismic Ray Theory* (Cambridge University Press, Cambridge, 2001).

[11] K. Aki and P. G. Richards, *Quantitative seismology, theory and methods, second edition* (University Science Books, Sausalito, California, 2002).

[12] C. Chapman, *Fundamentals of seismic wave propagation* (Cambridge University Press, Cambridge, 2004).

[13] J. F. Claerbout, *Imaging the Earth's Interior* (Blackwell Scientific, 1985).

[14] J. Virieux and G. Lambare, *Theory and observations - body waves: ray methods and finite frequency effects,* in *Treatise of Geophysics, volume 1: Seismology and structure of the Earth*, edited by B. Romanovitz and A. D. (eds) (Elsevier, 2007).

[15] J. Scales, *Theory of Seismic Imaging* (Samizdat Press, 1994).

[16] J. Virieux, H. Calandra, and R.-E. Plessix, *A review of the spectral, pseudo-spectral, finite-difference and finite-element modelling techniques for geophysical imaging,* Geophysical Prospecting **59**, 794 (2011).

[17] R. Clayton and B. Engquist, *Absorbing boundary conditions for acoustic and elastic wave equations,* Bull. Seis. Soc. America **67**, 1529 (1977).

[18] B. Engquist and A. Majda, *Absorbing boundary conditions for numerical simulation of waves,* Mathematics of Compututation **31**, 629 (1977).

[19] C. Cerjan, D. Kosloff, R. Kosloff, and M. Reshef, *A nonreflecting boundary condition for discrete acoustic and elastic wave equations,* Geophysics **50**, 705 (1985).

[20] B. Saleh, *Introduction to Subsurface Imaging* (Cambridge University Press, 2011).

[21] Y. A. Erlangga, C. W. Oosterlee, and C. Vuik, *A novel multigrid based preconditioner for heterogeneous Helmholtz problems,* SIAM Journal on Scientific Computing **27**, 1471 (2006).

[22] H. A. V. der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems,* SIAM Journal on Scientific and Statistical Computing **13**, 631 (1992).

[23] P. Sonneveld and M. van Gijzen, *IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations,* SIAM Journal on Scientific Computing **31**, 1035 (2008).

[24] J. Gozani, A. Nachshon, and E. Turkel, *Conjugate gradient coupled with multigrid for an indefinite problem,* in *Advances in Computational Methods for PDEs V*, edited by R. Vichnevestsky and R. S. Tepelman (IMACS, New Brunswick, NJ, USA, 1984) pp. 425–427.

[25] R. Kechroud, A. Soulaimani, Y. Saad, and S. Gowda, *Preconditioning techniques for the solution of the Helmholtz equation by the finite element method,* Mathematics and Computers in Simulation **65**, 303 (2004).

[26] A. L. Laird and M. B. Giles, *Preconditioned Iterative Solution of the 2D Helmholtz Equation*, Tech. Rep. 02/12 (Oxford Computing Laboratory, Oxford, UK, 2002).

[27] E. Turkel, *Numerical methods and nature,* J. Sci. Comput. **28**, 549 (2006).

[28] Y. A. Erlangga, C. Vuik, and C. W. Oosterlee, *On a class of preconditioners for solving the discrete Helmholtz equation,* in *Mathematical and Numerical Aspects of Wave Propagation*, edited by G. Cohen, E. Heikkola, P. Joly, and P. Neittaanmakki (Univ. Jyväskylä, Finnland, 2003) pp. 788–793.

[29] Y. A. Erlangga, *A robust and efficient iterative method for the numerical solution of the Helmholtz equation*, Ph.D. thesis, Delft University of Technology, The Netherlands (2005).

[30] Y. Erlangga and R. Nabben, *Multilevel projection-based nested Krylov iteration for boundary value problems,* SIAM Journal on Scientific Computing **30**, 1572 (2008), http://epubs.siam.org/doi/pdf/10.1137/070684550 .

[31] Y. Erlangga and F. Herrmann, *An iterative multilevel method for computing wavefields in frequency-domain seismic inversion,* in *78th Annual International Meeting*, Vol. 37 (SEG, Expanded Abstracts, 2008) pp. 1957–1960, http://library.seg.org/doi/pdf/10.1190/1.3059279 .

[32] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices* (University Press, Oxford, 1986).

[33] Y. Saad, *Iterative Methods for Sparse Linear Systems* (SIAM, Philadelphia, 2003).

[34] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide* (SIAM, Philadelphia, 2000).

[35] Peddy, *Intel gains, nvidia flat, and amd loses graphics market share in q1,* http://jonpeddie.com/press-releases/details/intel-gains-nvidia-flat-and-amd-loses-graphics-market-share-in-q1/ (2014).

[36] K. Rupp, *CPU, GPU and MIC hardware characteristics over time,* http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (2013).

[37] C. D. Riyanti, A. Kononov, Y. A. Erlangga, C. Vuik, C. W. Oosterlee, R.-E. Plessix, and W. A. Mulder, *A parallel multigrid-based preconditioner for the 3D heterogeneous high-frequency Helmholtz equation,* Journal of Computational Physics **224**, 431 (2007).

[38] Khronos Group, www.khronos.org (2014).

[39] K. Karimi, N. G. Dickson, and F. Hamze, *A performance comparison of CUDA and OpenCL,* CoRR **abs/1005.2581** (2010).

[40] P. Du, P. Luszczek, and J. Dongarra, *OpenCL evaluation for numerical linear algebra library development,* in *Symposium on Application Accelerators in High-Performance Computing* (SAAHPC, Knoxville, USA, 2010).

[41] NVIDIA, *Nvidia,* www.nvidia.com (2011).

[42] K. J. Marfurt, *Accuracy of finite-difference and finite-element modeling of the scalar and elastic wave equations,* Geophysics **49**, 533 (1984).

[43] W. A. Mulder and R.-É. Plessix, *A comparison between one-way and two-way wave-equation migration,* Geophysics **69**, 1491 (2004).

[44] H. Liu, B. Li, H. Liu, X. Tong, Q. Liu, X. Wang, and W. Liu, *The issues of prestack reverse time migration and solutions with Graphic Processing Unit implementation,* Geophysical Prospecting **60**, 906 (2012).

[45] Q. Ji, S. Suh, and B. Wang, *GPU based layer-stripping TTI RTM,* in *82nd Annual International Meeting, SEG, Expanded Abstracts*, Vol. 31 (2012) pp. 1–5.

[46] R. G. Clapp, *Reverse time migration with random boundaries,* in *79th Annual International Meeting, SEG, Expanded Abstracts*, Vol. 28 (2009) pp. 2809–2813.

[47] A. Griewank and A. Walther, *Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation,* ACM Transactions on Mathematical Software **26**, 19 (2000).

[48] W. W. Symes, *Reverse time migration with optimal checkpointing,* Geophysics **72**, 213 (2007).

[49] J. Cabezas, M. Araya-Polo, I. Gelado, N. Navarro, E. Morancho, and J. M. Cela, *High-performance reverse time migration on GPU,* International Conference of the Chilean Computer Science Society , 77 (2009), iEEE Computer Society.

[50] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems* (Prentice-Hall, Englewood Cliffs, N.J., 1981).

[51] K. J. Marfurt and C. S. Shin, *The future of iterative modeling of geophysical exploration,* in *Supercomputers in seismic exploration*, edited by E. Eisner (Pergamon Press, 1989) pp. 203–228.

[52] W. A. Mulder and R.-E. Plessix, *Time- versus frequency-domain modelling of seismic wave propagation,* in *Extended Abstract E015, 64th EAGE Conference & Exhibition, 27 - 30 May 2002, Florence, Italy* (2002).

[53] S. Operto, J. Virieux, P. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. B. H. Ali, *3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study,* Geophysics **72**, SM195 (2007).

[54] S. Wang, M. V. de Hoop, and J. Xia, *On 3D modeling of seismic wave propagation via a structured parallel multifrontal direct Helmholtz solver,* Geophysical Prospecting **59**, 857 (2011).

[55] S. Wang, M. de Hoop, and J. Xia, *Acoustic inverse scattering via helmholtz operator factorization and optimization,* Journal of Computational Physics **229**, 8445 (2010).

[56] C. Riyanti, Y. Erlangga, R.-E. Plessix, W. Mulder, C. Vuik, and C. W. Oosterlee, *A new iterative solver for the time-harmonic wave equation,* Geophysics **71**, E57 (2006), http://library.seg.org/doi/pdf/10.1190/1.2231109 .

[57] R.-E. Plessix, *A Helmholtz iterative solver for 3D seismic-imaging problems,* Geophysics **72**, SM185 (2007), http://library.seg.org/doi/pdf/10.1190/1.2738849 .

[58] T. Nemeth, C. Wu, and G. T. Schuster, *Least-squares migration of incomplete reflection data,* Geophysics **64**, 208 (1999).

[59] H. Kuehl and M. D. Sacchi, *Least-squares wave-equation migration for avp/ava inversion,* Geophysics **68**, 262 (2003).

[60] R. Rebollo and M. D. Sacchi, *Time domain least-squares prestack migration,* SEG Technical Program Expanded Abstracts 29 (2010).

[61] A. Guitton, A. B. Kaelin, and B. Biondi, *Least-squares attenuation of reverse-time-migration artifacts,* Geophysics **72**, S19 (2007).

[62] M. L. Clapp, *Imaging under salt: illumination compensation by regularized inversion*, Ph.D. thesis, Stanford University (2005), department of Geophysics.

[63] A. Valenciano, *Imaging by Wave-equation Inversion*, Ph.D. thesis, Stanford University, Department of Geophysics (2008).

[64] G. T. Schuster, *Least-squares crosswell migration,* in *SEG Expanded Abstracts 12, 63 Annual International Meeting* (1993) pp. 25–28.

[65] Y. Tang, *Wave-equation Hessian by phase encoding,* in *78 Annual International Meeting, SEG, Expanded Abstracts*, Vol. 27 (2008) pp. 2201–2205.

[66] D. Wei and G. T. Schuster, *Least-squares migration of multisource data with a deblurring filter,* Geophysics **76**, R135 (2011).

[67] Y. Kim, D. J. Min, and C. Shin, *Frequency-domain reverse-time migration with source estimation,* Geophysics **76**, S41 (2011).

[68] H. Ren, H. Wang, and S. Chen, *Least-squares reverse time migration in frequency domain using the adjoint-state method,* Journal of Geophysics and Engineering **10**, 035002 (2013).

[69] H. Knibbe, C. W. Oosterlee, and C. Vuik, *GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method,* Journal of Computational and Applied Mathematics **236**, 281 (2011).

[70] Y. A. Erlangga, C. Vuik, and C. W. Oosterlee, *On a class of preconditioners for solving the Helmholtz equation,* Applied Numerical Mathematics **50**, 409 (2004).

[71] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, *Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU,* SIGARCH Computer Architecture News **38**, 451 (2010).

[72] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, *Accelerating scientifing computations with mixed precision algorithms, http://dblp.uni-trier.de/db/journals/corr/corr0808.html,* CoRR, abs/0808.2794 (2008).

[73] M. van Gijzen and P. Sonnenveld, *An elegant IDR(s) variant that efficiently exploits bi-orthogonality properties,* ACM Transactions on Mathematical Software **38** (2011), 10.1145/2049662.2049667.

[74] P. Wesseling and P. Sonneveld, *Numerical experiments with a multi-grid and a preconditioned Lanczos type method,* Lecture Notes in Mathematics 771 , 543 (1980).

[75] N. Umetani, S. P. MacLachlan, and C. W. Oosterlee, *A multigrid-based shifted Laplacian preconditioner for a fourth-order Helmholtz discretization,* Numerical Linear Algebra with Applications **16**, 603 (2009).

[76] H. R. Elman, O. G. Ernst, and D. P. O'Leary, *A multigrid method enhanced by Krylov subspace iteration for discrete Helmholtz equations,* SIAM Journal on Scientific Compututing **23**, 1291 (2001).

[77] P. M. de Zeeuw, *Matrix-dependent prolongations and restrictions in a blackbox multigrid solver,* Journal of Computational and Applied Mathematics **33**, 1 (1990).

[78] A. Fog, *Instruction tables: List if instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, http://www.agner.org/optimize/instruction_tables.pdf,* Copenhagen University College of Engineering, DTU DIPLOM, Center for Bachelor of Engineering (2010).

[79] NVIDIACUDA, *Nvidia CUDA^{TM}, programming guide, version 2.2.1,* http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf (2009).

[80] G. Golub and C. van Loan, *Matrix computations*, 3rd ed. (The Johns Hopkins University Press, Baltimore, 1996).

[81] J. M. Elble, N. V. Sahinidis, and P. Vouzis, *GPU computing with Kaczmarz's and other iterative algorithms for linear systems,* Parallel Computing **36**, 215 (2010).

[82] R. E. Plessix and W. A. Mulder, *Separation-of-variables as a preconditioner for an iterative Helmholtz solver,* Applied Numerical Mathematics **44**, 385 (2003).

[83] H. Knibbe, C. Vuik, and C. W. Oosterlee, *3D Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method on multi-GPUs,* in *in Proceedings of ENUMATH 2011, the 9th European Conference on Numerical Mathematics and Advanced Applications, Leicester, September 2011*, edited by A. Cangiani, R. L. Davidchack, E. Georgoulis, A. N. Gorban, J. Levesley, and M. V. Tretyakov (Springer-Verlag Berlin Heidelberg, 2013) pp. 653–661.

[84] E. Zhebel, *A Multigrid Method with Matrix-Dependent Transfer Operators for 3D Diffusion Problems with Jump Coefficients*, Ph.D. thesis, Technical University Bergakademie Freiberg, Germany (2006).

[85] B. Barney, *POSIX threads programming, Lawrence Livermore National Laboratory, online, available,* https://computing.llnl.gov/tutorials/pthreads (2010).

[86] O. Ernst and M. Gander, *Why it is difficult to solve Helmholtz problems with classical iterative methods,* in *Numerical Analysis of Multiscale Problems*, edited by I. Graham, T. Hou, O. Lakkis, and R. Scheichl (Springer Verlag, 2012) pp. 325–363.

[87] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (SIAM, Philadelphia, 1991).

[88] H. Knibbe, W. A. Mulder, C. W. Oosterlee, and C. Vuik, *Closing the performance gap between an iterative frequency-domain solver and an explicit time-domain scheme for 3-d migration on parallel architectures,* Geophysics **79**, 47 (2014).

[89] R. Alford, K. Kelly, and B. Boore, *Accuracy of finite-difference modeling of the acoustic wave equation,* Geophysics **39**, 834 (1974), http://library.seg.org/doi/pdf/10.1190/1.1440470 .

[90] P. Micikevicius, *3D finite difference computation on GPUs using CUDA,* in *GPGPU-2: Proceedings of 2nd workshop on general purpose processing on graphics processing units* (2009) pp. 79–84.

[91] H. Fu, R. G. Clapp, O. Lindtjorn, T. Wei, and G. Yang, *Revisiting finite differences and spectral methods on diverse parallel architectures,* Computers & Geosciences **43**, 187 (2012).

[92] W. A. Mulder and R.-E. Plessix, *How to choose a subset of frequencies in frequency-domain finite-difference migration,* Geophysical Journal International **158**, 801 (2004).

[93] M. A. Dablain, *The application of high-order differencing to the scalar wave equation,* Geophysics **51**, 54 (1986).

[94] B. Fornberg, *Generation of finite difference formulas on arbitrarily spaced grids,* Mathematics of Computation **51**, 699 (1988).

[95] X. Shen and R. G. Clapp, *Random boundary condition for low-frequency wave propagation,* in *SEG Expanded Abstracts 30, 2962* (2011).

[96] A. Louis, P. Maas, and A. Rieder, *Wavelet: Theory and Applications* (John Wiley and Sons, London, 1997).

[97] S. Mallat, *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way* (Academic Press, Burlington, 2008).

[98] W. R. Stevens, *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI* (Prentice Hall, ISBN 0-13-490012-X, 1998).

[99] F. Aminzadeh, J. Brac, and T. Kunz, *3-D Salt and overthrust models* (Society of Exploration Geophysicists, Tulsa, Oklahoma, 1997).

[100] A. Guitton and E. Diaz, *Attenuating crosstalk noise with simultaneous source full waveform inversion,* Geophysical Prospecting **60**, 759 (2012).

[101] A. Gersho and R. M. Grey, *Vector quantization and signal compression* (Springer Science+Business Media, New York, 1992).

[102] K. Kourtis, G. Goumas, and N. Koziris, *Optimizing sparse matrix-vector multiplication using index and value compression,* in *Proceedings of the 5th Conference on Computing Frontiers*, CF '08 (ACM, New York, NY, USA, 2008) pp. 87–96.

[103] Z. Chen, G. Huan, and Y. Ma, *Computational methods for multiphase flows in porous media* (Society for Industrial and Applied Mathematics, Philadelphia, 2006).

[104] U. Trottenberg, C. W. Oosterlee, and A. Schüller, *Multigrid* (Academic Press, New York, 2001).

[105] K. Brackenridge, *Multigrid and cyclic reduction applied to the Helmholtz equation,* in *6th Cooper Mountain Conf. on Multigrid Methods*, edited by N. D. Melson, T. A. Manteufel, and S. F. McCormick (1993) pp. 31–41.

[106] K. Stüben and U. Trottenberg, *Multigrid methods: fundamental algorithms, model problem analysis and applications,* in *Lecture Notes in Math. 960*, edited by W. Hackbush and U. Trottenberg (1982) pp. 1–176.

[107] R. Wienands and C. W. Oosterlee, *On three-grid Fourier analysis of multigrid,* SIAM J. Sci. Comp. **23**, 651 (2001).

[108] LGM, *The Little Green Machine: Massive many-core supercomputer at low environmental cost,* http://www.littlegreenmachine.org (2012).

[109] *Project denver,* http://en.wikipedia.org/wiki/Project_Denver.