Accelerating an Iterative Helmholtz Solver Using Reconfigurable Hardware

by

Art Petrenko

B.Sc. Physics, McGill University, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Geophysics)

The University Of British Columbia
(Vancouver)

April 2014

Abstract

An implementation of seismic wave simulation on a platform consisting of a conventional host processor and a reconfigurable hardware accelerator is presented. This research is important in the field of exploration for oil and gas resources, where a 3D model of the subsurface of the Earth is frequently required. By comparing seismic data collected in a real-world survey with synthetic data generated by simulated waves, it is possible to deduce such a model. However this requires many time-consuming simulations with different Earth models to find the one that best fits the measured data. Speeding up the wave simulations would allow more models to be tried, yielding a more accurate estimate of the subsurface.

The reconfigurable hardware accelerator employed in this work is a field programmable gate array (FPGA). FPGAs are computer chips that consist of electronic building blocks that the user can configure and reconfigure to represent their algorithm in hardware. Whereas a traditional processor can be viewed as a pipeline for processing instructions, an FPGA is a pipeline for processing data. The chief advantage of the FPGA is that all the instructions in the algorithm are already hardwired onto the chip. This means that execution time depends only on the amount of data to be processed, and not on the complexity of the algorithm.

The main contribution is an implementation of the well-known Kaczmarz row projection algorithm on the FPGA, using techniques of dataflow programming. This kernel is used as the preconditioning step of CGMN, a modified version of the conjugate gradients method that is used to solve the time-harmonic acoustic isotropic constant density wave equation. Using one FPGA-based accelerator, the current implementation allows seismic wave simulations to be performed over twice as fast, compared to running on one Intel Xeon E5-2670 core. I also discuss the effect of modifications of the algorithm necessitated by the hardware on the convergence properties of CGMN.

Finally, a specific plan for future work is set-out in order to fully exploit the accelerator platform, and the work is set in its larger context.

Preface

The suggestion that Kaczmarz row projections should be implemented on the FPGA accelerator was made by Felix J. Herrmann. The design of the accelerated kernel was done by the author, with feedback from Diego Oriato of Maxeler Technologies. This feedback consisted of video calls on a semi-regular weekly basis, over the course of a few months. Diego Oriato also assisted in the debugging of the design during a two week visit by the author to Maxeler Technologies, in London. Simon Tilbury, also of Maxeler Technologies provided a crucial contribution to the design by implementing the buffer that enables "accelerator ordering" of the rows of the Helmholtz matrix.

The MATLAB function implementing CGMN, `CARPCG.m` was initially written by Tristan van Leeuwen and modified by the author to interface with the automatically generated MATLAB class that is used to run the accelerator. The C MEX implementation of the Kaczmarz sweeps used in the reference implementation mentioned in Chapter 4 was initially also written by Tristan van Leeuwen, and modified by the author to optimize memory access. Tristan van Leeuwen wrote the MATLAB function that generates the Helmholtz matrix, this code was used unmodified. The design and execution of the numerical experiments measuring performance of the accelerated CGMN implementation were done entirely by the author, with minor suggestions by Rafael Lago. Analysis of the profile and solution data, including making the plots presented, were done entirely by the author.

This work (in its entirety) has led to the following two conference submissions, which have been accepted:

> A. Petrenko, D. Oriato, S. Tilbury, T. van Leeuwen, and F. J. Herrmann. Accelerating an iterative Helmholtz solver with FPGAs. In *OGHPC*, 03 2014. URL https://www.slim.eos.ubc.ca/Publications/Public/Conferences/OGHPC/petrenko2014OGHPCaih.pdf

> A. Petrenko, D. Oriato, S. Tilbury, T. van Leeuwen, and F. J. Herrmann. Accelerating an iterative Helmholtz solver with FPGAs. In *EAGE*, 06 2014. URL https://www.slim.eos.ubc.ca/Publications/Public/Conferences/EAGE/2014/petrenko2014EAGEaih.pdf

The first has been presented and published as a poster at the 2014 Rice University Oil & Gas High-Performance Computing Workshop in Houston, Texas. The second will be presented

as an oral presentation at the June 2014 European Association of Geoscientists and Engineers Conference and Exhibition in Amsterdam, the Netherlands, at which time it will also be published as an expanded abstract. The contributions of the co-authors to the work presented is the same as outlined above for the present thesis. The first author wrote the text and made the figures for both submissions.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all I would like to thank my supervisor Felix Herrmann, whose intuition that implementing a wave equation solver on FPGAs would pay off turned out to be correct. Thank you also to Michael Friedlander and Christian Schoof, members of my committee, and Steve Wilton, my external examiner.

My gratitude goes out to Henryk Modzelewski for his administration and maintenance not only of the Maxeler compute node used for the bulk of this research, but of the entire hardware and software environment used by the SLIM group.

I received guidance about using Maxeler's dataflow computing machine from Diego Oriato of Maxeler Technologies; his help was very useful. Simon Tilbury, also with Maxeler Technologies, developed a crucial piece of code (the row-reordering buffer) that enables the algorithm described in this work to run at full speed. I thank the entire Maxeler team that graciously welcomed me for a three week visit to their London office in the fall of 2013. I learned a lot from this partnership.

Thank you to Eddie Hung for stimulating discussions during the early stages of implementation and to Tristan van Leeuwen for developing the MATLAB versions of many of the algorithms mentioned in this thesis. His work served as a launchpad for my own modifications. Thank you also to Rafael Lago, Lina Miao and Brendan Smithyman for proof-reading versions of the manuscript.

Finally thank you to the entire SLIM group for sustaining an environment where I could succeed.

This work was financially supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grant (RGPIN 261641-06) and the Collaborative Research and Development Grant DNOISE II (CDRP J 375142-08). This research was carried out as part of the SINBAD II project with support from the following organizations: BG Group, BGP, BP, Chevron, ConocoPhillips, CGG, ION GXT, Petrobras, PGS, Statoil, Total SA, WesternGeco, Woodside.

Chapter 1

Introduction

Systems of linear equations are ubiquitous in science and engineering. Being able to solve large systems in a short amount of time is a mathematical and computational challenge. The subject of this work is an implementation of a particular linear system solver on unconventional hardware, namely a system consisting of a CPU host and a reconfigurable hardware-based accelerator. The design, construction, testing and analysis of this implementation are the main contributions of the work. In this chapter I describe the context which motivated the implementation: the field of seismic exploration for oil and gas resources. I then motivate the choice of the reconfigurable computing platform.

1.1 Full-waveform inversion

Seismic exploration aims to characterize the earth's subsurface to assist the discovery and production of hydrocarbon reservoirs. In a seismic survey, vibrational energy is sent into the ground using a *source*. This could be a dynamite explosion, a vibroseis truck, or in the case of marine exploration, an air gun. *Receivers*, which may be geophones or hydrophones, record the ground (or water) motion and pressure changes that result from waves coming back to the surface. The raw data is then analysed, and subsurface properties like acoustic velocity, density or anisotropy are inferred.

Seismic waves are modelled by a partial differential wave equation (PDE) where the input is medium parameters and a source signature, and the solution is a wavefield. In practice, the source signature is not known and must be estimated, although this is not treated in the current work. The problem faced in seismic exploration (and many other disciplines) is the *inverse problem*: given some measurement of the wavefield, find medium parameters that explain the data. The most scientifically rigorous method of analysing seismic data is *full-waveform inversion* (FWI), first proposed by Tarantola [58]. FWI starts from an initial guess at the medium model $\mathbf{m}$ and solves the wave equation for each source $\mathbf{q}_s$. The resulting synthetic wavefields $\mathbf{u}_s$ are sampled at the receiver locations and compared to the measured wavefield data, $\mathbf{d}_s$. Based on the difference, the model $\mathbf{m}$ is updated and the algorithm iterates

until a satisfactory model has been produced. Mathematically this corresponds to solving the optimization problem

$$\min_{\mathbf{m}} \Phi(\{\mathbf{d}_s\}, \mathbf{m}, \{\mathbf{q}_s\}), \tag{1.1}$$

where $\{\mathbf{d}_s\}$ is the set of results for all the source experiments $\{\mathbf{q}_s\}$. A traditional choice for the *misfit function* Φ is the sum of the squared ℓ_2-norms of the data residual vectors:

$$\Phi = \frac{1}{n_{\text{src}}} \sum_{s=1}^{n_{\text{src}}} \frac{1}{2} \left\| \mathbf{d}_s - \mathcal{F}(\mathbf{m}, \mathbf{q}_s) \right\|^2 . \tag{1.2}$$

The symbol $\mathcal{F}$ represents the wave equation solution operator that generates the synthetic wavefields and samples them at the receivers. There are n_{src} source experiments in total. Other misfit functions could be chosen, as described by Aravkin et al. [3].

A full exposition of FWI is beyond the scope of this work, the reader is referred to Virieux and Operto [63] for details. Here it suffices to note that to calculate the model update $\delta\mathbf{m}$, it is necessary to compute the first derivative of the misfit function Φ. By applying the chain rule, we find that

$$\nabla_{\mathbf{m}} \Phi = (\nabla_{\mathbf{m}} \mathcal{F}^*) \left(\frac{1}{n_{\text{src}}} \sum_{s=1}^{n_{\text{src}}} (\mathbf{d}_s - \mathcal{F}(\mathbf{m}, \mathbf{q}_s)) \right), \tag{1.3}$$

where * denotes the adjoint of an operator (complex conjugate transpose for matrices). The operator $\nabla_{\mathbf{m}} \mathcal{F}^*$ maps a perturbation in the measurement of the wavefield at the receivers to one of the many possible perturbations in the model consistent with that measurement. Two wave equation solves are needed to implement $\nabla_{\mathbf{m}} \mathcal{F}^*$, as mentioned by Pratt et al. [49]. Depending on the particular algorithm for minimizing Φ in Expression 1.1, further PDE solves may be required to compute the second derivative (the Hessian) of the misfit function, or during a line-search along the gradient direction $\nabla_{\mathbf{m}} \Phi$. Thus, each iteration of FWI must solve a wave equation at least twice; this forms most of the computational burden of the method, as shown for example by Brossier et al. [5].

When run in industry on modelled domains of $1000 \times 1000 \times 1000$ or larger, FWI can take up to a month to converge to an acceptable earth model (first author of Li et al. [33], private communication). The high computational cost limits how many FWI iterations can be performed, to the detriment of the final result. Research is on-going on various dimensionality reduction techniques to process less of the data without sacrificing accuracy, including work by Li et al. [33] and Aravkin et al. [3]. In addition to fundamental algorithmic improvements in FWI, it is possible to accelerate existing methods with a view to increasing the number of PDE solves that can be performed within a given time budget. Virieux and Operto [63] remark that "efficient numerical modelling of the full seismic wavefield is a central issue in FWI, especially for 3D problems." The authors go on to say that "fields of investigation should address the need to speed up the forward problem by means of providing new hardware [implementations]." This is the direction of the present work.

1.2 Accelerating algorithms

The use of many-core and GPU (graphic processing unit) based accelerators is growing in high-performance computing, as evidenced by the "top 500" list of the world's most capable machines, compiled by Meuer et al. [38]. As of 2013, four of the top ten machines on that list have a CPU host + accelerator architecture.

Commodity machines using GPUs typically do not have very large memories dedicated to the accelerators, relying instead on the PCIe (peripheral component interconnect express) bus to exchange data between the CPU (central processing unit) and accelerator. On the one hand, Johnsen and Loddoch [26] overcome the PCIe bottleneck in data transfer by exploiting parallelism and re-use of data in the *time domain* method of solving the wave equation. On the other hand, Knibbe et al. [30] report that using GPUs in the *frequency domain* method did not result in significant improvements over using the CPU host exclusively, citing the bandwidth of the PCIe bus as the bottleneck. (See the next chapter for a discussion of the relative merits of the time and frequency domain approaches to solving the wave equation.)

The use of reconfigurable hardware accelerators such as FPGAs (field programmable gate arrays) has been more reluctant, due in part to the different programming paradigms necessary to take advantage of such units. However accelerating an algorithm such as FWI by using FPGAs to handle the computationally intensive *kernel* of the algorithm is a possibility worth investigating, due in part to the large dedicated memories available for FPGA-based accelerators (see Chapter 3 for a specification of the system used in this work).

Consider the differences between FPGAs and conventional CPU cores. A CPU core is a general-purpose piece of hardware: no physical part of the core corresponds to the algorithm being run on it. CPUs boast high operational frequency (about 3 GHz), and come with ready-to-use components like caches for frequently used data. Such mechanisms greatly simplify the design of programs since "low level" details are taken care of for the programmer. Consequently it is possible to use a very high level of abstraction when designing an algorithm for a conventional CPU. On the other hand, as mentioned by Dally [8] among others, the increase in processor frequencies that used to guarantee an increase in performance every few years has come to a halt. Increase in performance on today's processors (this includes GPUs) comes by making use of many parallel processor cores, a task that is often far from trivial.

In contrast to a CPU, an FPGA is (almost) a blank slate. It contains building blocks of digital logic that can be configured (and reconfigured) by the end user to execute the task at hand *in hardware*. Once an FPGA is configured it is a hardware representation of the algorithm. FPGAs operate at frequencies an order of magnitude lower than CPUs (about 200 MHz) but consequently consume far less power. Until recently, algorithms for an FPGA had to be designed at a relatively low level of abstraction, which made the development process arduous. However the introduction by Maxeler Technologies [36] of a set of Java classes offering common functionality (like memory controllers) has made FPGA programming much more accessible. (I note that the recent announcement of support of the OpenCL programming framework by

some FPGA manufacturers (Singh [52]) is also progress in the direction of accessibility.) Nevertheless, in the scope of the current project, it was not practical to re-write a full full-waveform inversion algorithm to run on an FPGA. Instead, the *Kaczmarz sweep*, a basic and frequently used routine in a particular implementation of the wave equation solution operator $\mathcal{F}$ and its adjoint, was selected for porting.

CPU cores work by processing a stream of instructions. Modern processors have long instruction pipelines to keep computational units supplied with work, and process several such pipelines in parallel, as documented by Intel Corporation [25]. Nevertheless, an algorithm that has more instructions will generally take longer than a simpler algorithm with fewer instructions. This is not the case for FPGAs. FPGAs work by processing a stream of data, rather than a stream of instructions. All the instructions of the algorithm are hard-wired onto the chip and hence all of them are executed on every tick of the FPGA clock. (I use the terms FPGA and chip interchangeably throughout this work.) The FPGA can be thought of as a filter for data that flows through it. Hence, unless the algorithm is limited by the bandwidth of an interconnect (such as a link to memory), speed of execution is directly proportional to the amount of data to be processed. Apart from their power efficiency, this is the chief advantage of FPGAs in contrast to CPUs.

There are other alternatives besides FPGAs to using CPU or GPU cores. Application-specific integrated circuits (ASICs) are hardware representations of an algorithm that are faster and more efficient in logic-gate use than FPGAs (Kuon and Rose [32]). Unlike FPGAs however, they cannot be reconfigured. Furthermore, due to the fixed, unchangeable nature of an ASIC, development times for ASIC algorithms are longer than for FPGAs (Hauck and DeHon [20]).

The rest of this thesis is organized as follows. Chapter 2 outlines the method I use to solve the wave equation. Chapter 3 describes the implementation of that method on reconfigurable hardware; this is the main contribution of this work. Chapter 4 presents the results of using the implementation, and Chapter 5 presents conclusions and a list of improvements that should be accomplished when the project continues.

Chapter 2

Solving the wave equation: Mathematical background

This chapter starts with a comparison of two approaches to solving the wave equation and then proceeds to describe in greater detail the approach taken in the present work: treating the problem as a system of linear equations. I then describe the chosen solution method, due to Björck and Elfving [4], which is based on a combination of three ideas: the normal equations, projections onto matrix rows, and the method of conjugate gradients.

There are two main approaches to finite difference modelling of seismic waves: formulating the problem in the *time domain* (TD) or the *frequency domain* (FD) (see review by Virieux and Operto [63]). Solving the wave equation in the time domain corresponds to an initial value problem where the wavefield is stepped from one point in time to the next, using information local to each point in the numerical grid. Solving the problem in the frequency domain corresponds to solving a large, sparse system of linear equations for each frequency ω_f to be modelled.

There are two main factors to consider when choosing between these approaches. The first is how the choice of forward modelling scheme effects the overlying full-waveform inversion algorithm. FWI is prone to failing to converge to a reasonable earth model because of getting stuck in local minima of the misfit function Φ, as noted by van Leeuwen and Herrmann [60] among others. To reduce the chance of this happening, the problem can be solved progressively, starting from the easier parts. One way to do this is to invert low frequency data first, followed by higher frequencies. This is particularly simple if the forward modelling step of FWI is performed in the frequency domain, since in that case each solution of the forward model corresponds to one frequency. Another way to regularize the FWI problem, mentioned by Virieux and Operto [63], is to window the data by arrival time, something for which forward modelling in the time domain is appropriate. The time domain method needs storage of the wavefields of some intermediate time steps (a procedure known as checkpointing, described recently by Symes [56]), which is avoided in the frequency domain method. Furthermore,

implementation of the adjoint of the Jacobian of the modelling operator ($\nabla \mathcal{F}^*$) is non-trivial in the time domain; see however the work of Gockenbach et al. [15].

The second factor to consider when choosing between time domain and frequency domain methods is computational expense. In the time domain computational complexity is well defined by the number of time-steps simulated, and is $\mathcal{O}(n_{\text{src}}N)$, where N is the total number of grid-points in the three-dimensional discretized earth model $\mathbf{m}$. This has been shown by Plessix [47]; see also Table 1 in the work of Knibbe et al. [30]. However in the frequency domain formulation, there are two different ways to solve the associated system of linear equations: with a direct or an iterative solver. Direct solvers are theoretically well-suited to problems with a large number of sources, however they use a prohibitively large amount of memory for storing intermediate factors of the matrix that are not as sparse as the original matrix. This limits their use on large 3D FWI problems. Iterative solvers only need storage for the original matrix and a few intermediate vectors, however they typically need a preconditioner to be effective. When comparing execution time for the time domain approach and the frequency domain approach using an iterative solver, Plessix [48] concludes that both approaches have approximately the same performance in terms of execution time requirements for 3D problems. Finally, Virieux and Operto [63] recommend a hybrid method due to Sirgue et al. [53]: model in the time domain but progressively build up the Fourier transform of the wavefield so that when the modelling step is complete, full-waveform inversion can be performed in the frequency domain.

In this work I formulate the forward modelling problem in the frequency domain.

2.1 The Helmholtz system: Discretizing the wave equation

When simulating waves in the frequency domain, the PDE that describes the motion of the wave through a heterogeneous medium can be written as

$$\left(\omega^2\mathbf{m} + \Delta\right)\mathbf{u} = \mathbf{q}, \tag{2.1}$$

and is known as the Helmholtz equation. As written above, the Helmholtz equation represents the special case of a constant density isotropic medium which only supports acoustic waves. Damping effects of viscosity are modelled heuristically by allowing $\mathbf{m}$ to be complex-valued. I ignore the case of elastic and anisotropic media to keep the resulting implementation relatively simple. The symbol Δ represents the Laplacian operator. I take the subsurface earth model to be the slowness squared, $\mathbf{m} = 1/\mathbf{v}^2$, where $\mathbf{v}$ is the sound speed of the medium. $\mathbf{u}$ is the (complex) Fourier transform of the pressure with respect to time, and $\mathbf{q}$ is the amplitude of the source at angular frequency ω. The Laplacian operator here also implements perfectly matched layer (PML) boundary conditions that eliminate reflection artefacts from the boundaries of the domain by setting a damping layer consisting of complex velocities. (See Equation (2) in the work by Operto et al. [42].) In the frequency domain, Equation 2.1 must be solved for each frequency ω that is to contribute to the final wavefield $\mathbf{u}_s$ for a given source.

When $\mathbf{m}$, $\mathbf{u}$ and $\mathbf{q}$ represent quantities that have been discretized on a three-dimensional Cartesian grid with N grid-points, Equation 2.1 can be represented as a large system of linear equations and succinctly written in matrix notation,

$$A(\mathbf{m}, \omega)\mathbf{u} = \mathbf{q} \tag{2.2}$$

where A is the $N \times N$ Helmholtz matrix. The elements of A are calculated by the method of finite differences, following the strategy of Operto et al. [42], which consists of two main parts. First, the Laplacian operator is discretized using a $3 \times 3 \times 3$ *cube* stencil. This 27-point stencil is a weighted average of eight different second-order 7-point *star* stencils, each on their own coordinate grid. The coordinate grids are rotated and scaled versions of the original Cartesian grid. The grids are designed in such a way that even though their axes are not parallel to each other, the grid-point locations of all eight grids coincide. This allows the cube stencil to use all 27 points in the three-dimensional neighbourhood of a given central point. The weighting coefficients are tuned to minimize numerical anisotropy, as described by Operto et al. [42].

Second, the value of the earth model $\mathbf{m}$ at each grid-point is re-distributed to the 27 neighbouring points that make up the cube stencil for that point, a process known as mass-averaging. Mass-averaging is done using a second set of weighting coefficients, and results in a matrix with the same pattern of non-zero entries as the discretized Laplacian. By choosing optimal values for the mass-averaging coefficients, Operto et al. [42] showed that numerical dispersion of the stencil is minimized, which enhances (by a constant factor) the stencil's accuracy. This allows to use as little as 4 grid-points per wavelength [42], although 6 grid points per wavelength are used in this work. The need for accuracy by using finer grid spacings (more grid-points per wavelength) must be balanced against the computational expense of simulating on a larger grid. As noted by Operto et al. [42], 4 grid-points per wavelength is the limit at which the modelling step is accurate, without modelling wavefield features that are in any case too small to be of use to full-waveform inversion in resolving the earth model. A further advantage of the stencil introduced by Operto et al. [42] is that it is compact. A stencil with a large extent in the last dimension (for example as in 5-point star stencil) implies that more intervening grid-points need to be buffered in short-term memory (see Section 3.3 for details).

The mass matrix and the discretized Laplacian are added together to make the Helmholtz matrix A, which is very sparse: while N is at least 10^7 for a realistic model, the number of non-zeros per row, determined by the finite difference stencil, is at most only 27. A is also very structured: its non-zeros are arranged in 27 diagonals. This means that the locations of the non-zero elements, taken together, of each matrix row, do not repeat. In other words, the support of each matrix row is unique (although the support of rows that correspond to grid points adjacent to the edge of the grid is a subset of the support of the rows that correspond to adjacent points). These special properties mean that the rows of the Helmholtz matrix are linearly independent and hence theoretically (disregarding round-off errors) A is invertible.

2.2 The normal equations

A simple method of solving invertible linear systems is the method of conjugate gradients (CG) (Hestenes and Stiefel [24]), which requires the matrix to be symmetric positive definite (SPD) to converge to the unique solution. However the matrix A is not symmetric for arbitrary subsurface models $\mathbf{m}$, and can be indefinite, so CG cannot be applied to it directly. (See Ernst and Gander [13] for a discussion of how the indefiniteness of the Helmholtz equation makes it difficult to solve.) A preconditioner must be used to transform the system into an equivalent SPD form. As has been set out in the work of van Leeuwen et al. [62], desired characteristics for a Helmholtz preconditioner for FWI are independence from the matrix (because the matrix changes on each iteration of FWI) and low storage requirements. These conditions are satisfied by an approach based on the normal equations, as is explained below.

The normal equations for the error (NE) for A are given by

$$AA^*\mathbf{y} = \mathbf{q}, \text{where } \mathbf{u} = A^*\mathbf{y}. \tag{2.3}$$

For invertible matrices the normal equations form a symmetric positive definite system, (see Saad [51] for a comprehensive treatment). In this case the solution of Equation 2.3 will be the same as the solution to the original system, Equation 2.2. However working with the matrix AA^* explicitly is prohibitively expensive because it will be much less sparse than A itself, and thus take up more memory. Instead, it is possible to solve Equation 2.3 without forming AA^* using a *row projection* method, as described by Saad [51].

2.3 Row projections: The Kaczmarz algorithm

The method of solving a system of linear equations $A\mathbf{x} = \mathbf{b}$ by successively projecting an iterate $\mathbf{x}_k$ onto the hyperplane associated with each row $\mathbf{a}_i$ of the matrix A was discovered early on by Kaczmarz [27]. An English translation is available in [28]. A projection onto the i^{th} hyperplane $\langle \mathbf{a}_i, \mathbf{x} \rangle = b_i$ is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda(b_i - \langle \mathbf{a}_i, \mathbf{x}_k \rangle)\frac{\mathbf{a}_i^*}{\|\mathbf{a}_i\|^2}, \tag{2.4}$$

where b_i is the corresponding element of the right hand side vector $\mathbf{b}$. If the Kaczmarz algorithm is directly applied to solve the Helmholtz system (Equation 2.2) $\mathbf{b}$ would correspond to $\mathbf{q}$ and $\mathbf{x}$ would correspond to $\mathbf{u}$. The relaxation parameter λ determines whether the projection is exactly onto the hyperplane ($\lambda = 1$), falls short ($0 < \lambda < 1$) or extends beyond it ($1 < \lambda < 2$). It is set to 1.5 in this work, following the discussion by Gordon and Gordon [16]. The row projections are conventionally done in either a forward sweep from first to last, a backward sweep, or in a double sweep from first to last and back again (k: $1 \to 2N$; i: $1 \to N, N \to 1$). In practice, the rows of A are normalized once at the beginning of the algorithm. The iterations of the Kaczmarz algorithm are shown schematically in Figure 2.1.

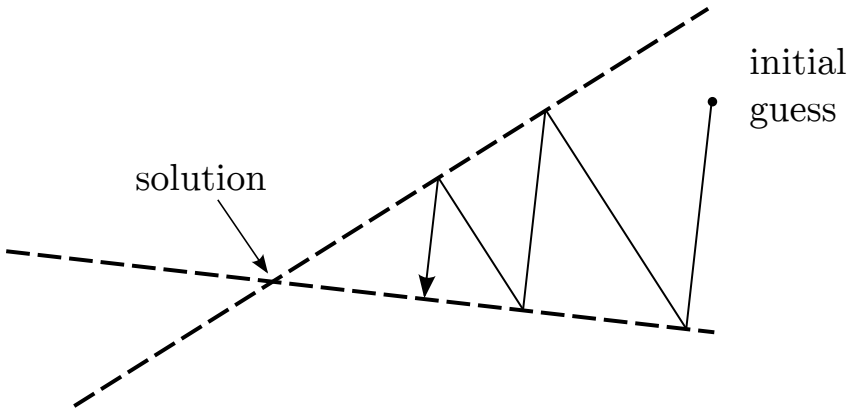

Figure 2.1: The Kaczmarz algorithm solves a linear system by successively projecting the initial guess onto each of the hyperplanes (shown here as dashed lines) associated with the rows of the matrix. Adapted from a similar diagram by the author of van Leeuwen [59].

The connection between the Kaczmarz algorithm and solving the normal equations was made by Björck and Elfving [4], who noted that each minor step in the forward Gauss-Seidel method (see Saad [51], Sections 4.1 and 8.2) applied to Equation 2.3 is equivalent to one Kaczmarz row projection using the original system with λ set to 1. The authors went on to show that varying the relaxation parameter λ away from 1 makes a Kaczmarz row projection equivalent to one minor step of the method of successive over-relaxation (SOR) applied to Equation 2.3. If a forward full SOR iteration is followed by a backward iteration, the result is known as the symmetric SOR (SSOR) method, or SSOR-NE when applied to the normal equations (NE, Equation 2.3) for a matrix. Therefore, one iteration of SSOR-NE is equivalent to a double Kaczmarz sweep: projecting onto the rows of the original matrix from first to last and back again.

The order in which the rows of A are selected to project onto will affect the convergence path of the Kaczmarz algorithm, as well as the overlying CGMN algorithm (discussed in the next section). Kak and Slaney [29] recommend choosing orthogonal (or nearly orthogonal) rows for successive Kaczmarz projections. The impact of different row orderings on the Helmholtz system is examined in Chapter 4. Saad [51] notes that the effect of row-ordering on Kaczmarz algorithm convergence for general sparse matrices is an open problem.

Placed in a modern linear algebra context, the Kaczmarz algorithm is best viewed as a way of implementing SSOR on the normal equations (Equation 2.3) that reduces memory use by not forming the matrix AA^* explicitly. All the convergence properties of SSOR-NE carry over directly. The advantage of the Kaczmarz algorithm lies in the fact that it allows us to work with an SPD system (the normal equations) by using only the non-zeros of the original matrix A. Since it is not necessary to form or store AA^*, the Kaczmarz algorithm is appealing from memory usage and computational complexity points of view.

2.4 Preconditioning conjugate gradients: The CGMN algorithm

Despite the advantages mentioned in the last section, the Kaczmarz algorithm (SSOR-NE) converges slowly, thus it is not suitable for direct application to the Helmholtz system (Equation 2.2). Instead, Björck and Elfving [4] showed that it can be used to accelerate the method of conjugate gradients, calling the resulting algorithm CGMN. Recent studies of how CGMN fares in solving the Helmholtz equation include work by van Leeuwen [59] and Gordon and Gordon [18]. In the latter case, CGMN is equivalent to the sequential (non-parallel, running on only one processor core) version of the algorithm CARP-CG, introduced by Gordon and Gordon [17]. I now describe the CGMN algorithm.

First, it is useful to represent the Kaczmarz sweeps in matrix notation. Following Tanabe [57], let Q_i be the projection matrix onto the hyperplane defined by $\langle \mathbf{a}_i, \mathbf{x} \rangle = 0$:

$$Q_i = I - \frac{\lambda}{\|\mathbf{a}_i\|^2} \mathbf{a}_i^* \mathbf{a}_i.$$

The double sweep can then be written as

$$\begin{aligned} \text{DKSWP}(A, \mathbf{u}, \mathbf{q}, \lambda) &= Q_1 \cdots Q_N Q_N \cdots Q_1 \mathbf{u} + R\mathbf{q} \\ &= Q\mathbf{u} + R\mathbf{q}. \end{aligned} \tag{2.5}$$

Since A is invertible, SSOR-NE will converge to the solution of Equation 2.2, as mentioned by Björck and Elfving [4]. At that point, the iterate $\mathbf{u}$ will be a fixed point of the relation 2.4, which means that Equation 2.5 can be re-written as a linear system:

$$(I - Q)\mathbf{u} = R\mathbf{q}, \tag{2.6}$$

where I is the identity matrix. As mentioned by Björck and Elfving [4] and proved by, for example, Gordon and Gordon [17], the system in Equation 2.6 is consistent, symmetric and positive semi-definite. Björck and Elfving [4] show in their Lemma 5.1 that this is sufficient for CG to converge to the pseudoinverse (minimum ℓ_2-norm) solution of Equation 2.6, which is (by construction) the same as the solution of the original system (Equation 2.2). Note that the matrices Q and R do not have to be formed explicitly, as their action on a vector is calculated using a double Kaczmarz sweep, as in Equation 2.5.

Thus, CGMN is the use of the method of conjugate gradients to solve the SSOR-NE iteration system (Equation 2.6) for the fixed point of that iteration. SSOR-NE is implemented efficiently using Kaczmarz row projections. Björck and Elfving [4] also note that it is possible to view CGMN as solving a variant of the Helmholtz system (Equation 2.2), preconditioned from the left by a matrix derived from a decomposition of AA^*.

Pseudo-code for the CGMN algorithm is given below. Note that the double Kaczmarz sweep on line 1 of the algorithm is performed with an initial guess of zero because only the action of R

is required. Similarly, the double sweep on line 2 is done with a right hand side of zero because only the action of Q is required. Also note that although $\mathbf{u}$ was used as the Kaczmarz iterate in deriving the equivalent system (Equation 2.6), the double sweep in the main while-loop of CGMN (line 5 of the algorithm) is performed on the CGMN search direction $\mathbf{p}$, and not on the wavefield. The double sweep would only be performed on the wavefield if the Kaczmarz algorithm were used to solve the Helmholtz system directly, which is not being done. This double sweep is equivalent to the matrix-vector product with the system matrix of Equation 2.6 that would otherwise be required. References to the *Kaczmarz iterate* indicate intermediate vectors in the calculation of $\mathbf{s}$, in contrast to the *CG iterate*, which is the wavefield $\mathbf{u}$.

Algorithm 1 CGMN (Björck and Elfving [4])

Input: A, $\mathbf{u}$, $\mathbf{q}$, λ

1: $R\mathbf{q} \leftarrow \text{DKSWP}(A, \mathbf{0}, \mathbf{q}, \lambda)$
2: $\mathbf{r} \leftarrow R\mathbf{q} - \mathbf{u} + \text{DKSWP}(A, \mathbf{u}, \mathbf{0}, \lambda)$
3: $\mathbf{p} \leftarrow \mathbf{r}$
4: **while** $\|\mathbf{r}\|^2 > tol$ **do**
5: $\quad \mathbf{s} \leftarrow (I - Q)\mathbf{p} = \mathbf{p} - \text{DKSWP}(A, \mathbf{p}, \mathbf{0}, \lambda)$
6: $\quad \alpha \leftarrow \|\mathbf{r}\|^2 / \langle \mathbf{p}, \mathbf{s} \rangle$
7: $\quad \mathbf{u} \leftarrow \mathbf{u} + \alpha\mathbf{p}$
8: $\quad \mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{s}$
9: $\quad \beta \leftarrow \|\mathbf{r}\|^2_{\text{curr}} / \|\mathbf{r}\|^2_{\text{prev}}$
10: $\quad \|\mathbf{r}\|^2_{\text{prev}} \leftarrow \|\mathbf{r}\|^2_{\text{curr}}$
11: $\quad \mathbf{p} \leftarrow \mathbf{r} + \beta\mathbf{p}$
12: **end while**

Output: $\mathbf{u}$

I remark that it is possible to perform more than one double Kaczmarz sweep, one after the other, in CGMN (line 5 of Algorithm 1). If the Kaczmarz sweeps are viewed as a preconditioner for CG, as mentioned above, multiple sweeps correspond to a more exact preconditioner. Fewer outer CGMN iterations would have to be performed, at the cost of several invocations of DKSWP in the main while-loop. Because the Kaczmarz algorithm takes more sweeps to converge to a solution than CG takes iterations, such a trade-off might be viewed as exchanging a precise tool for a less precise one, and hence undesirable. However as I show in Section 4.1, the characteristics of the Kaczmarz algorithm as implemented on an FPGA-based accelerator might encourage such an adjustment.

In this chapter I have described the mathematical formulation of the approach I take to solving the seismic wave simulation problem. The reader will recall that this problem is relevant because of the role it plays in full-waveform inversion, a key algorithm in seismic exploration. The next chapter will focus on a description of the implementation of this problem on an FPGA-based hardware accelerator.

Chapter 3

Implementation of the Kaczmarz algorithm on reconfigurable hardware

This chapter starts with an introduction to FPGAs and a description of the particular CPU host + FPGA-based accelerator platform used in the work. I then survey previous implementations of linear solvers and seismic processing algorithms on accelerators (both FPGAs and GPUs). The core of the chapter gives details of the implementation of the Kaczmarz algorithm on the FPGA-based accelerator, and describes the technical challenges overcome. This is my main contribution. The chapter concludes with a comparison of this implementation with closely related work by Grüll et al. [19].

The functional unit in an FPGA consists of one or more lookup tables (LUTs) and one or more flip-flops (FFs). LUTs implement functions. For example a lookup table with six inputs, each of which is a wire carrying a binary signal, has 2^6 outputs. Flip-flops implement storage: one flip-flop can store one bit of information. In addition to these *logic elements*, FPGAs also have blocks dedicated for certain common operations. These include digital signal processing (DSP) blocks for multiply-and-accumulate operations, and bulk storage memory blocks (BRAM). Together these components are known as the *resources* of the FPGA. The blocks are embedded in an *interconnect* fabric that carries signals between them. The (automated) process of making sure the signals carried by the interconnect fabric arrive at each computational stage at the right time is known as *meeting timing*. The interested reader is referred to the text by Hauck and DeHon [20] for details. As mentioned in section 1.2, the key advantage of FPGAs is that all of the operations that have been hard-wired onto the chip are executed at once, on different points of the data-stream. Making an algorithm more complex by adding arithmetical operations does not affect the execution time. Of course whether a design will successfully build on an FPGA is subject to resource availability on the particular chip model. FPGAs can be used on their own, or as *accelerators* alongside conventional CPUs. In this case

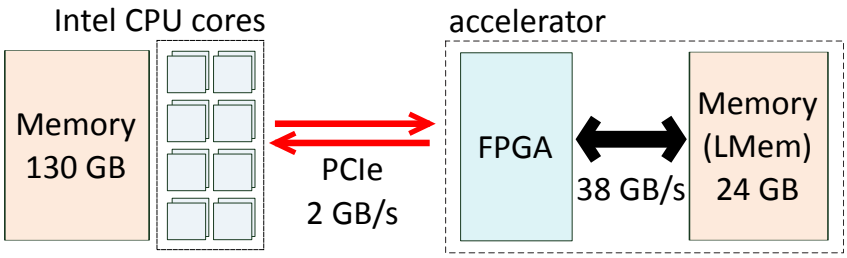

Figure 3.1: High-level overview of the compute node used in this work. Note that the effective PCIe bandwidth is 2 GB/s in each direction independently. The memory bandwidth is total in both directions, and is 96 bytes per memory clock tick. At the maximum memory clock frequency (400 MHz, Diamond [9]) this translates to 38 GB[1]/s. The memory clock is run slower in this work, giving an LMem bandwidth of 29 GB/s.

the CPU handles most of the source code lines of the program, and supplies the FPGA with data to execute in hardware the few source code lines responsible for most of the time spent.

The target platform for our implementation is described by Pell et al. [44] and shown in Figure 3.1. Four accelerators (only one of which is shown in the figure), consisting of FPGAs and their associated dedicated large memory (LMem), are connected via a PCIe bus to the CPU host. The CPU host is an Intel Xeon E5-2670. The accelerator model used in this work is called Vectis. Each Vectis board has two FPGAs. The interface FPGA (Xilinx Virtex-6 XC6VLX75T) is used to communicate with the CPU host, while the compute FPGA (Xilinx Virtex-6 XC6VSX475T) is used for calculation. For a detailed specification of the FPGAs the reader is referred to documentation by Xilinx [64]. The operation of the interface FPGA is completely transparent to the user. When referring simply to the FPGA, I mean the compute FPGA. In addition to the dedicated large memory of the accelerator, each compute FPGA also has 4.6 MB of on-chip BRAM (also termed FMem, for fast memory), which has lower latency of access. The accelerator is operated via a compiled binary that is called from a high level numerical processing environment (MATLAB) using an automatically generated C intermediate layer known as a MEX file.

Conceptually, an algorithm on an FPGA can be described as a directed graph, as is done by Maxeler Technologies [36]. The nodes of the graph are operators, and the edges represent data flowing from one operation to the next along the interconnect fabric. Practically, this graph is described using the Java programming language as documented in [36, 37].

3.1 Related work on accelerators

FPGA-based accelerators have been used in seismic processing for the last decade. An early example of the acceleration of the kernel of pre-stack Kirchoff time migration with FPGAs is given by He et al. [21]. Migration is an alternative to full-waveform inversion for estimating the

[1]Recall that 1 GB = 10^9 bytes.

Earth model $\mathbf{m}$, and He et al. do not solve a linear system but rather use the FPGA as a filter to perform operations on a stream of seismic data. Fu et al. [14] have also used FPGAs to accelerate seismic migration, focusing on reducing floating point precision to increase throughput. More recently, Pell et al. [44] implement the time domain acoustic wave equation on the CPU host + FPGA-based accelerator platform used in this work, and report a performance equivalent to that of over 50 conventional CPU cores.

Solving linear systems using FPGA-based accelerators has also seen recent developments. The Jacobi algorithm (described by Saad [51] and others), being highly parallel and simple to implement, was ported to an FPGA accelerator early on by Morris and Prasanna [39]. That implementation is applicable to general sparse matrices stored in compressed sparse row format. Conjugate gradients is more complex from a computational point of view and work has focused on porting the matrix-vector multiplication that forms the kernel of CG to accelerators. Lopes and Constantinides [35] present such an implementation, and give a brief overview of past efforts in this field. The CG matrix-vector product is parallelizable in various ways; Lopes and Constantinides use multiple linear systems to pipeline their design, which is optimized for small dense matrices. Alternative CG implementations are given by Morris et al. [40] and Strzodka and Göddeke [54]. The latter authors increase the amount of computation that can fit onto one chip by reducing precision of intermediate computations in a process known as iterative refinement. Iterative refinement is also used by Sun et al. [55] in an FPGA-based implementation of a direct solver using LU-decomposition. Another direct solver accelerated with FPGAs, this time via Cholesky decomposition of the normal equations, is presented by Yang et al. [67].

For completeness we also mention some of the work with GPU-based accelerators, although since such units are more common than their FPGA counterparts, this survey is not comprehensive. Elble et al. [12] have examined the improvements that CGMN and related algorithms experience on GPUs. Krueger et al. [31] compares the CPU, GPU and FPGA performance of the acoustic wave equation in the time domain using both computational and energy efficiency metrics. Knibbe et al. [30] compare a GPU-accelerated time domain wave equation solver with an iterative frequency domain solver implemented solely on conventional CPU cores and find the two approaches competitive with each other as long as the number of computational nodes per source is greater than two.

Finally, Grüll et al. [19] present an implementation of the simultaneous algebraic reconstruction technique (SART) on the same hardware platform (CPU host and Vectis model accelerator) used in the present work. SART is closely related to Kaczmarz row projections, and, to my knowledge, this is the most similar previous work to my own contribution. I defer a discussion of that implementation until Section 3.8, and now present the main technical challenges overcome in the course of the present work.

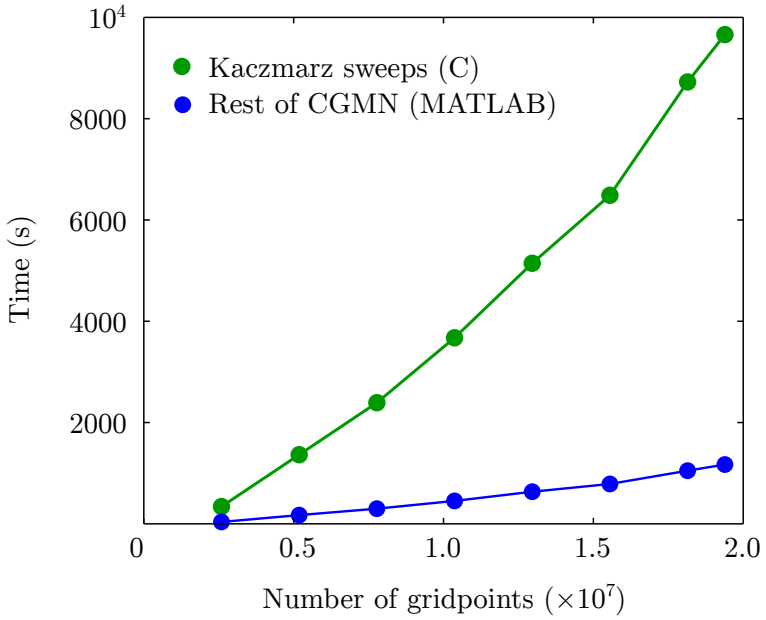

Figure 3.2: Distribution of computation time before acceleration. The time spent in the Kaczmarz sweeps (implemented in C) and the rest of CGMN (implemented in MATLAB) are compared. Both parts of the algorithm are running on a conventional Intel Xeon core. CGMN solved the Overthrust system as described in Chapter 4, with the right-hand side equal to $A\mathbf{1}$. The algorithm was allowed to converge to a relative residual tolerance of 10^{-6}, so each system size ran for a different number of iterations. The fraction of time CGMN spent in the Kaczmarz sweeps for this range of system sizes is $89 \pm 0.4\%$.

3.2 Data transfer

Although the CGMN algorithm is quite similar to the conventional conjugate gradient algorithm, the FPGA-based implementation proposed in the current work differs in an important way from the CG implementations discussed in the previous section. Namely, the CG matrix-vector product is implemented via a double Kaczmarz sweep (line 5 of Algorithm 1). This allows us to apply CGMN to solve systems where the matrix A may be indefinite. Since the double Kaczmarz sweep implements the only matrix-vector operation in CGMN, it is the most time-consuming part of that algorithm. This statement is supported by the results of timing runs of CGMN before any acceleration is applied, shown in Figure 3.2, clearly motivating the identification of DKSWP as the kernel of the PDE solver.

As mentioned in Section 1.2, data transfer is usually the bottleneck when using accelerators. All instances of DKSWP in CGMN share the same matrix $A(\mathbf{m}, \omega)$, so it is transferred to the accelerator's dedicated large memory at the start of execution. The first sweep of the CGMN initialization phase (line 1 in Algorithm 1) requires the seismic source $\mathbf{q}$ as its right-hand side,

while all subsequent sweeps in CGMN require the zero vector as the right-hand side. Hence this zero vector is also only transferred to the accelerator once. Together, these transfers take approximately $(27 + 21 + 1 + 1)N$ ticks of the FPGA clock, where the extra factor of $21N$ is due to zero-padding of the matrix, discussed in Section 3.5.

Dataflow during the double Kaczmarz sweep using the accelerator proceeds in three stages. For concreteness I describe the stages referring to the data they use during the main while-loop of CGMN. First the iterate $\mathbf{p}$ is copied from the CPU host to LMem over the PCIe bus. This takes N ticks of the FPGA clock. Then the forward sweep is performed, during which $\mathbf{p}$, the right-hand side $\mathbf{0}$ and rows of the matrix A are streamed to the FPGA from LMem and $\mathbf{p}$ is updated according to Equation 2.4. The result $\text{FKSWP}(A, \mathbf{p}, \mathbf{0}, \lambda)$ is streamed back out to LMem, replacing $\mathbf{p}$. The third stage consists of the backward sweep, during which the rows of A, the elements of $\mathbf{0}$ and the result of the forward sweep are streamed in reverse order. The result $\text{DKSWP}(A, \mathbf{p}, \mathbf{0}, \lambda)$ is this time streamed out to the CPU host via the PCIe bus. Because of the time necessary to fill and flush the buffers holding a sliding window onto the iterate (see Section 3.3), each of the latter two steps take approximately $N(1 + 2/n_{\text{iter}})$ ticks of the FPGA clock. The matrix and right-hand side remain in LMem, ready to be read during the next sweep which will occur on the next CGMN iteration. The CPU carries out the elementwise and reduction operations necessary for CGMN, such as multiplying a vector (of length N) by a scalar, adding two vectors together, or taking inner products. These operations are on lines 6–12 of Algorithm 1).

3.3 Buffering: Reducing access to slow memory

For an arbitrary matrix (not necessarily sparse), all elements of the Kaczmarz iterate $\mathbf{x}_k$ need to be read to calculate the dot product in Equation 2.4. Each element then needs to be written to with the updated values of $\mathbf{x}_{k+1}$. This would entail two passes over the entire computational grid on which $\mathbf{x}_k$ is defined. However because the Helmholtz matrix arises from a finite difference stencil (see Operto et al. [42]), it has very favourable structure: 27 non-zero diagonals (in the case of the $3 \times 3 \times 3$ cube stencil used in this work), and zeros everywhere else. These diagonals are clustered around the main diagonal at offsets given by

$$i + j \cdot n_{\text{fast}} + k \cdot n_{\text{fast}} \cdot n_{\text{med}} \text{ where } i, j, k \in \{-1, 0, 1\}, \tag{3.1}$$

where indices i, j, k are used for convenience and do not refer to the indices defined in the Kaczmarz row projection formula (Equation 2.4). The quantities n_{fast} and n_{med} are the number of grid points along the first two dimensions of the grid. This terminology refers to the fact that a three dimensional quantity like $\mathbf{m}$ or $\mathbf{u}$ can be stored in linear memory in a variety of ways. One possibility is to store contiguously values corresponding to a line along the first dimension, followed by neighbouring lines that make up a two-dimensional slice, and finally neighbouring slices. Since memory is optimized for consecutive access (see Drepper [11] for a thorough

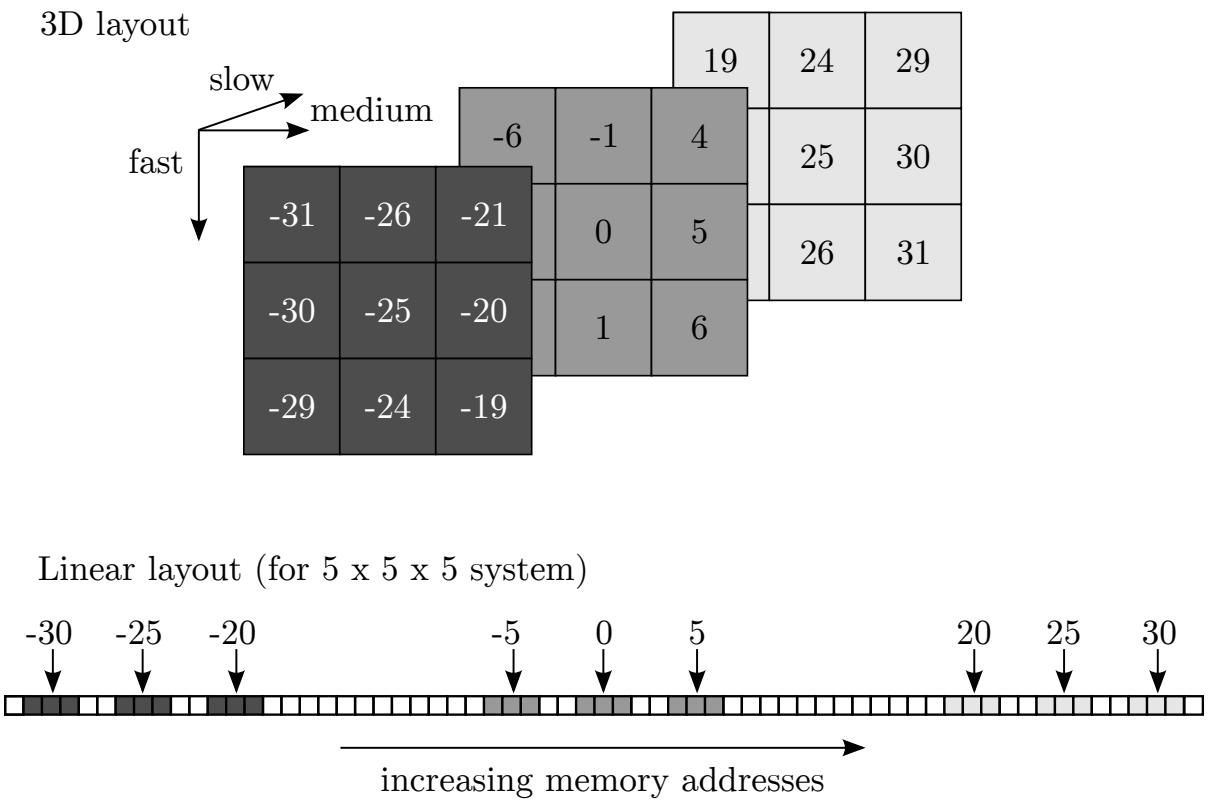

Figure 3.3: Storing a 3D volume in 1D memory. Three adjacent slices of a three-dimensional grid are shown above, along with the corresponding storage locations in memory. The grid locations are labelled with their offsets in memory from the central point (marked with 0).

treatment of issues relating to computer memory), elements along the first dimension can then be accessed quickly, while elements along other dimensions take longer. The correspondence between a three-dimensional grid and its storage in (one-dimensional) memory is illustrated in Figure 3.3. Note that a contiguous 3D sub-volume of the domain is not stored contiguously in memory. Instead, adjacent elements from neighbouring lines along the fast dimension are separated by small gaps of size approximately n_{fast}, and adjacent elements from neighbouring slices are separated by large gaps of size approximately $n_{\text{fast}} n_{\text{med}}$

Due to the "few non-zero diagonals" structure of the Helmholtz matrix, illustrated in Figure 3.4 only 27 elements of the iterate need to be read and updated during each Kaczmarz row projection. These elements correspond to the non-zeros in row $\mathbf{a}_i$. Furthermore, the location of the needed elements changes smoothly as the projections sweep through the rows. This means that rather than needing random access to elements of the Kaczmarz iterate $\mathbf{x}$, a relatively small contiguous working set of the elements is sufficient. This working set can be viewed as a *sliding window* over the iterate elements, or, alternatively, as a FIFO (first in, first out) pipeline. Because communicating with the large memory on the accelerator incurs a significant latency, the elements in the sliding window are buffered in the on-chip BRAM. The size of the

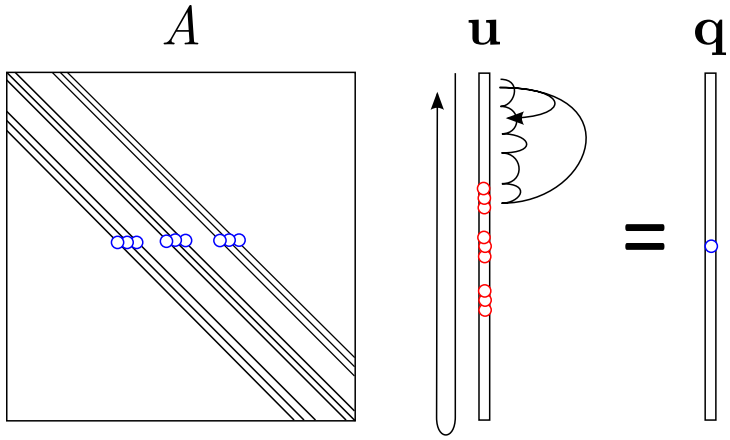

Figure 3.4: The Helmholtz linear system of equations, illustrating that non-contiguous elements of the iterate (shown in red) must be read and written during each Kaczmarz row projection. Blue elements must be read but are not written. The smooth up-and-down arrow represents sequential element access, whereas the arrow that jumps around represents *accelerator ordering*, defined in Section 3.4.

window is approximately equal to $2n_{\text{fast}}n_{\text{med}}$ grid points; it contains the elements to be updated during a given row projection and all of the elements in between. As yet unmodified elements of $\mathbf{x}_k$ are read into the sliding window from large memory, undergo the operations for all the row projections they are involved in (27 projections each) and are written back as elements of $\mathbf{x}_{k+1}$. This is analogous to a CPU cache storing frequently used variables. The important difference is that a CPU cache is managed by transparent low-level processes, whereas the design of buffering structures on the FPGA and orchestrating re-use of data is at least partially up to the programmer. The ability to use a sliding window approach to update elements of the Kaczmarz iterate $\mathbf{x}$ means that rather than needing two data passes over the entire grid for each row projection, only one pass is necessary for each of the forward and backward Kaczmarz sweeps.

The FPGA's fast on-chip memory has two limitations. First, a whole BRAM block (2304 bytes, Xilinx [64]) must be used at a time, even if only a small fraction of that block will actually be written to. More importantly, each block can only have two ports: for example, one read and one write port (see the documentation by Maxeler Technologies [37] for details). Therefore it is not possible to implement the sliding window into the Kaczmarz iterate $\mathbf{x}_k$ as a single buffer. Instead, the contents of the sliding window are distributed among 26 individual buffers, one each for the 27 elements of $\mathbf{x}_k$ (save the first) that are updated during one row projection. The first element can be read directly into the row projection computation from LMem. Similarly, the element from the last buffer is written back to LMem after being updated in the computation, since it will not receive any further updates during the given sweep. The addresses into the buffers and into LMem are incremented every tick of the FPGA clock, with the buffer read address one entry ahead of the write address. The flow of data between LMem and the buffers

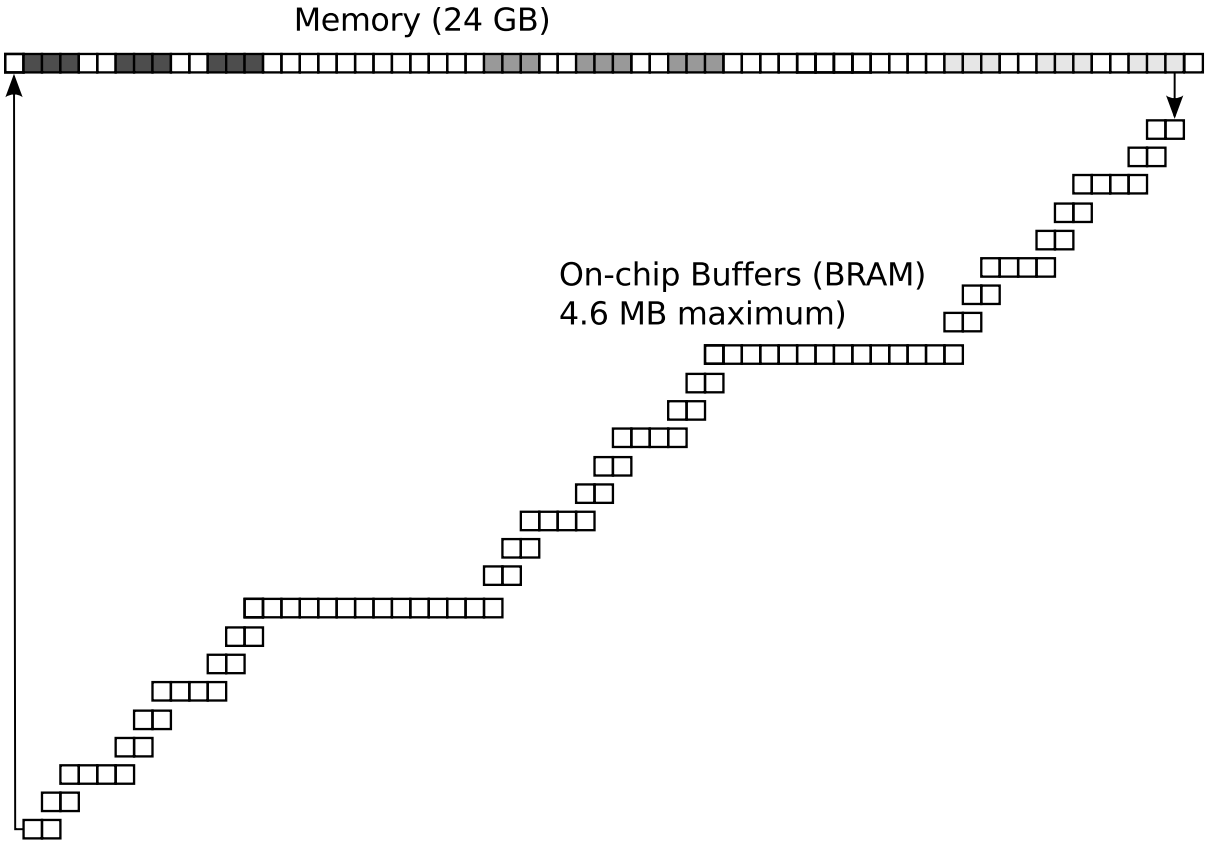

Figure 3.5: Overcoming latency of memory access by the use of sliding window buffers, described in the text for a $5 \times 5 \times 5$ system. At any one point in time, substituting the contents of LMem with the contents of the buffers yields the Kaczmarz iterate $\mathbf{x}_k$. If the rows are processed from 1 to N in a forward sweep, k will be the index of the middle element of the sliding window. The element written back to LMem from the last buffer will then be an element of $\mathrm{FKSWP}(A, \mathbf{x}, \mathbf{b}, \lambda)$. The row projection computation reads the oldest element from each buffer. (In the illustration this is the leftmost element in each buffer.)

is shown in Figure 3.5. I remind the reader that it is necessary to run the FPGA for about $n_{\text{fast}} n_{\text{med}}$ clock cycles before activating the row projection calculation. This phase allows the buffers to fill with valid data. Likewise, a flush phase at the end of each forward and backward sweep drains the buffers while the updating of iterate elements is disabled. Because the size of the buffers must be fixed at compile time, the size of the fast and medium dimensions of the domain must also be fixed.

3.4 Parallelizing the Kaczmarz sweeps: Dealing with row projection latency

If the rows of A are processed from 1 to N, each Kaczmarz iteration depends on the results of the last. Furthermore, each iteration's computation takes many ticks of the FPGA clock.

This is the latency L, and is chiefly due to the pairwise summation of values in $\langle \mathbf{a}_i, \mathbf{x}_k \rangle$ (see Equation 2.4).

One method of dealing with latency is to slow down the rate of dataflow in the FPGA. This can be done by reading a set of inputs (a row of A and elements of $\mathbf{x}$ and $\mathbf{b}$) only once every L ticks. This will give time for the results of one row projection computation to be written to the buffer, so that a consistent state of $\mathbf{x}$ is available for the next iteration. This approach has the FPGA processing data at $1/L$ of its actual operating frequency and is clearly unacceptable.

To avoid having to wait for L ticks between iterations, L independent iterations are used to fill the computational pipeline. Two sources of parallelism can be exploited. First, multiple Helmholtz problems sharing the same matrix A but different right-hand sides $\mathbf{q}$ can be solved on the same accelerator. At each tick, the current element of one of the L different iterate and source pairs would be read and the corresponding row projection started. Because the solution of systems with multiple sources can be treated independently, updating one of the L iterates would not affect the others. As this method requires sliding windows of the kind pictured in Figure 3.5 to be stored in the FPGA's on-chip memory for the wavefield associated with *each* source, the size of fast memory is a limit to how many sources can be used for pipelining. To avoid confusion for readers familiar with direct methods for solving linear systems, I emphasize that working with multiple right-hand sides in this way is not analogous to the favourable scaling of execution time with the number of sources seen in direct solvers. Instead, parallelization over several right-hand sides on the FPGA-based accelerator is a scheme to fully employ the computational facility of the FPGA, in the sense of processing one Kaczmarz row projection every tick of the FPGA clock.

Another approach is to change the order in which DKSWP accesses the rows of A such that consecutive Kaczmarz iterations update disjoint sets of iterate elements. Mathematically this corresponds to picking L mutually orthogonal rows to process. From a geometrical point of view, considering $\mathbf{x}$ as a 3D quantity, this corresponds to processing L successive $3 \times 3 \times 3$ cubes, such that none of the cubes overlap. Once L cubes have began undergoing computation, the row selection wraps back to process the iterate elements it missed the first time. This wrap-around repeats three times in total, until all the intervening cubes have been processed. I term this order of processing rows *accelerator ordering*, in contrast to the *sequential ordering* where the index i proceeds from 1 to N. In Section 4.2 I show experimental results for how such a reordering affects the convergence properties of the CGMN algorithm.

Despite this modified order of row access, memory should be addressed sequentially as much as possible, in order to keep the FPGA's memory controller efficient. Hence, the rows of A and the elements of $\mathbf{q}$ are reordered prior to being sent to the accelerator. However reordering the elements of the Kaczmarz iterate in the same way is not possible. To understand why, consider again the layout of the buffers shown in Figure 3.5. The specific alternation of small, medium and large buffers hard-codes the structure of the finite difference stencil. This structure, shown in Figure 3.3 and expressed as diagonal offsets of A in Formula 3.1, will only produce correct

results if the elements of $\mathbf{x}$ are ordered sequentially (lexicographically). Nevertheless, if L is not too large, it is possible to keep the reordering of the rows local enough that a sliding window onto the iterate can still be maintained, at the cost of a more sophisticated buffer layout. Simon Tilbury of Maxeler Technologies made a significant contribution to this work by devising such a buffer layout and implementing it on the accelerator. In his scheme the fast dimension of the domain should be $3L$ grid points long to enable efficient use of on-chip memory for the buffers. This revised buffering approach enables the full pipelining of the Kaczmarz row projection algorithm, allowing the computation to run at the operational frequency of the FPGA, rather than being slowed by a factor of $1/L$.

Both the multiple right-hand side approach and the row reordering approach could be implemented together, with a multiplicative effect on addressing the latency. For example, if the latency L is equal to 100 FPGA clock ticks, one might process 10 different right-hand sides, while re-ordering the rows of A so that 10 mutually orthogonal rows are chosen successively. However at the present I use the row reordering pipelining method exclusively, and do not use multiple sources to fill the row projection pipeline on the FPGA.

3.5 Number representation and storage

To take advantage of the special structure of A, I do not use a general sparse matrix storage format, but rather follow van Leeuwen et al. [62] in storing the diagonals of A in a rectangular array and the offsets of the diagonals in a separate short vector *idx*. Since the chief operation in the Kaczmarz algorithm is an inner product between a matrix row and the iterate vector, the array of diagonals is stored in row major order (that is, the diagonals are not stored contiguously). This produces an efficient access pattern of the matrix from memory. The compiler, developed by Maxeler Technologies [36], that translates the Java graph-based description of the FPGA design into a lower level hardware description language offers a complex data type. To take advantage of this, it is convenient to store complex vectors and the matrix with the real and imaginary parts interleaved. The data is thus rearranged from its native MATLAB format, which stores the real and imaginary parts separately, before being sent to the accelerator.

Unlike the memory attached to the CPU host, the dedicated large memory attached to each FPGA cannot be addressed at the byte granularity level. Instead, addresses into LMem have to be multiples of the *burst size*, and each chunk returned by the memory controller is also a multiple of the burst size. On the Vectis accelerator model used in this work, a burst size is 384 bytes (3072 bits), as confirmed by Diamond [9]. Furthermore, each stream of input into the FPGA from LMem must be either a multiple or an integer fraction of the burst size. For example, a stream that requests one double precision floating point number (8 bytes) per FPGA clock tick is allowed, but a stream that requests five doubles is not allowed because (3072 bits)/(5 doubles $\times$ 8 bytes per double $\times$ 8 bits per byte) is not an integer.

The Kaczmarz algorithm requires reading all 27 non-zeros in a row of the matrix every tick of the FPGA clock. This means that the row as stored in LMem must take up a multiple (or

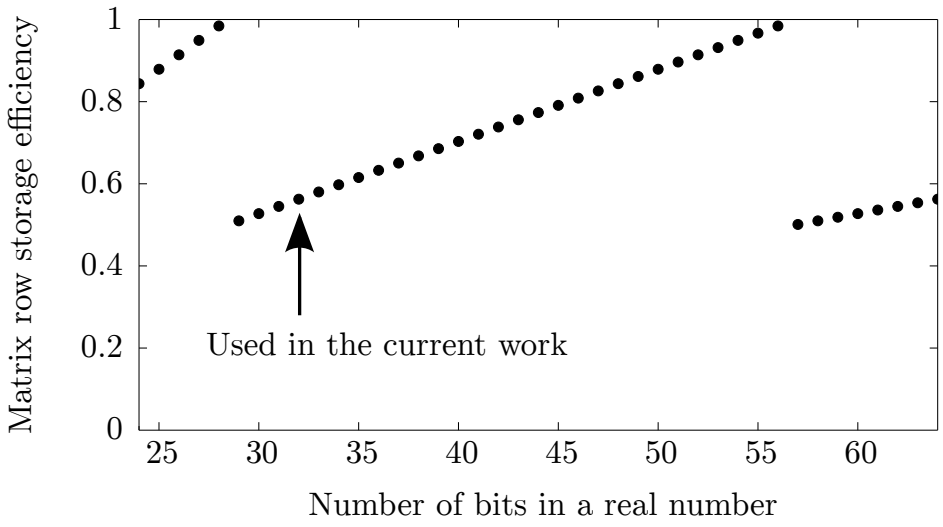

Figure 3.6: Matrix row storage efficiency as a function of the number of bits used to represent the real and imaginary parts of each matrix element. This figure assumes 27 elements are stored per row. Note that the three efficiency "modes" use different amounts of storage for one row: half a burst, a full burst (384 bytes) and two bursts, respectively. Single precision (32 bit real numbers, resulting in one burst per row) is used in this work.

an integer fraction) of the burst size. The row datatype is made up of 27 complex numbers stored at some precision, and zero bit padding that is used to fill out the datatype to the nearest integer fraction or multiple of the burst size. During computation, the zero padding is discarded. There are three factors to consider when deciding on the bitwidth of the row datatype. The first is efficiency of matrix storage in memory: the row datatype should be as wide as possible without going over the nearest integer fraction or multiple of the burst size. Memory efficiency of a range of row datatype bitwidths is shown in Figure 3.6. Note that for row datatypes that take up half a burst, the matrix may have to be padded to give it an even number of rows. The second factor is the available memory bandwidth. Picking a very precise number representation that necessitates two bursts for each matrix row means that the design will be limited by memory bandwidth at lower FPGA operational frequencies. Finally, wide datatypes will use more FPGA resources (like LUTs and flip-flops) and make it more difficult (or impossible) to arrange the design elements on the physical chip. One approach to increase performance on both of these factors is to store and transfer numbers in a compressed format, unpacking them only for the computation. Another is to implement different parts of the algorithm with different precision, which is done by Strzodka and Göddeke [54] and Sun et al. [55]. This adds an extra level of complexity to the development and has not yet been attempted by the author.

The Kaczmarz algorithm also requires one complex element from the iterate and right-hand

Number of bits in a real number	Number of bits in a complex number	Complex numbers per burst
24	48	64
32 (single precision)	64	48
48	96	32
64 (double precision)	128	24

Table 3.1: Possible bitwidths for complex elements of the Kaczmarz iterate and right-hand side vectors. Bitwidths are constrained by needing to be an integer fraction of the FPGA memory controller's burst size. Single precision is used in this work.

side vectors at every clock tick, which means that the vector elements must be an integer fraction of the burst size. Possible options for the bitwidth of the vector element datatype are shown in Table 3.1. As a compromise between the need for an adequately precise number representation and a design that fits onto the FPGA, I settle on 32 bits for a real number. As shown by the work of Strzodka and Göddeke [54], CG will still converge even if the preconditioner (which in our case is implemented with the Kaczmarz sweeps) computes with precision as low as 28 bits. In addition, I perform a toy model experiment comparing the convergence properties of CGMN with double and single precision Kaczmarz sweeps. The results are shown in Figure 3.7, and confirm that single precision is sufficient for Kaczmarz sweeps to ensure CGMN convergence. In contrast to Strzodka and Göddeke, I do not find that using a reduced precision (32 bits) for the Kaczmarz sweeps increases the number of CG iterations that have to be performed. I also note that Nemeth et al. [41] use single precision in their FPGA-based time domain wave equation simulations, while Pell et al. [44] use compression to represent single precision floats in a mere 16 bits. Both authors mention the sufficiency of such precision schemes for (time domain) geophysical applications. Although it is not necessary to store the individual non-zero elements of a row at the same precision as the elements of the iterate and right-hand side vectors, this is being done in the current accelerated implementation for simplicity. I note that this is not optimal in terms of matrix storage efficiency.

3.6 The backward sweep: Double buffering

For the backward sweep, the rows of the matrix and the iterate and right-hand side elements need to be accessed from LMem in reverse. This is simple for the matrix, provided each row takes up one or more bursts in memory; it suffices to have a decrementing counter as the address. The case of the vector is more complicated because while bursts can be accessed in reverse order, the vector elements within each burst will still be in forward order. To resolve this issue I maintain a buffer two bursts long in BRAM. One half of the buffer is written to by the input stream, one vector element per clock tick. The other half is being read from by the Kaczmarz iteration computation. Every T ticks, where T is the number of vector elements in a

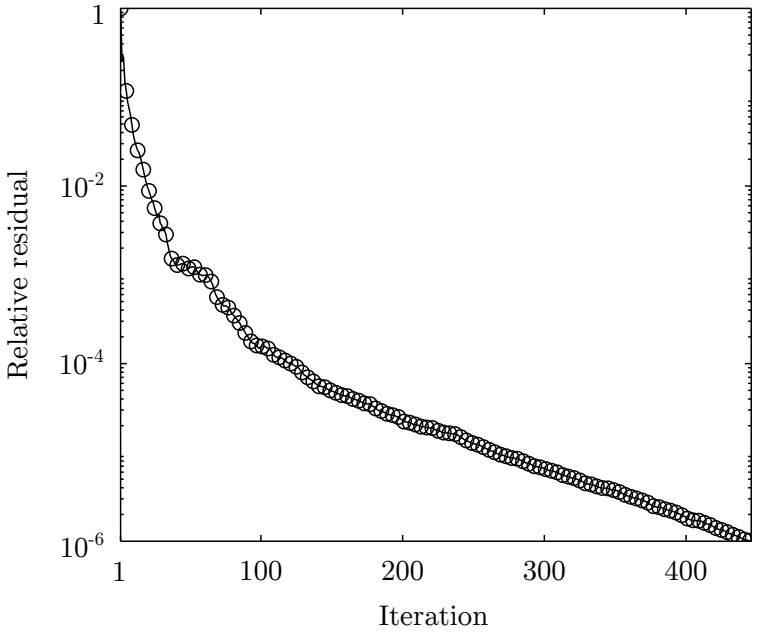

Figure 3.7: Effect of sweep precision on CGMN convergence. The relative residuals of a single precision CGMN run (circles) and a double precision run (solid line) agree to within single precision; the two convergence curves overlap. The system solved is a subsampling of the SEG/EAGE Overthrust model (shown in Chapter 4) that results in a $47 \times 201 \times 201$ grid. A source at $\omega = 3$ Hz, equal to $A\mathbf{1}$ was used.

burst, the role of the two halves of the buffer switches. This is a well known technique known as double buffering; it is illustrated in Figure 3.8. Note that if a row takes up only half a burst, the matrix will also have to be double buffered during the backward sweep. When the result of BKSWP is returned back to the CPU host, it is in reverse order and needs to be post-processed before being used in the rest of the CGMN calculation. Likewise, if multiple sweeps are done by the accelerator before returning back to the CPU host, forward sweeps after the first also need to double buffer the input to return it to forward order.

3.7 Bandwidth requirements

Recall that the advantage of an FPGA is that the algorithm can be made more complex (i.e. more arithmetic operations) without impacting the execution time. This is because of the fact that all of the computations on the chip happen every clock tick. Computation is done "in space" as opposed to "in time". Execution time depends on the amount of data to process. The underlying assumption is that the FPGA always has data to work on. However if the speed with which data is processed exceeds the speed at which data is streamed into the FPGA from external sources, the algorithm is said to be *bandwidth limited*. There are two

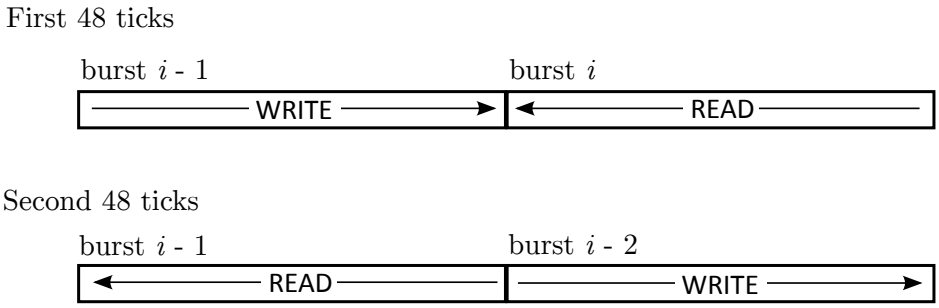

Figure 3.8: Double buffering during the backward Kaczmarz sweep is needed to reverse the order of vector elements which are accessed in burst-sized chunks from memory.

communication channels between the FPGA and the outside world, and hence two potential sources of bandwidth limitation: the PCIe bus connecting the accelerator to the host, and the memory bus connecting the FPGA and its dedicated large memory. This is illustrated in Figure 3.1. It is worthwhile to note that the bandwidth of the PCIe link is actually determined by the connection between the interface and compute FPGAs, not the link between the CPU host and the interface FPGA, as confirmed by Pell [43]. Of course all bandwidths are theoretical maxima and will be impacted by the pattern in which data is accessed. In general, transferring a few large chunks is more efficient than transferring many smaller chunks (see Drepper [11] for details). The general strategy in implementing an algorithm on the accelerator is to minimize communication. Data transferred to the accelerator over the PCIe bus should be stored in LMem and re-used. If several problems share data, it should be loaded into LMem just once. If a stream can be generated on the FPGA, thus doing away with data transfer completely, this should be done. The communication bandwidth required by the current accelerated implementation of the double Kaczmarz sweep is given in Table 3.2. At the memory clock frequency of 303 MHz used in this work, the algorithm is memory bandwidth limited at an operational frequency of the FPGA of 71 MHz or higher. In Section 4.4 I discuss improvements that will ease this bandwidth limitation.

3.8 Related work: computed tomography

In this section I briefly describe closely related work by Grüll et al. [19]. I first introduce the difference between Kaczmarz row projections and the algorithm used by those authors.

The Kaczmarz algorithm (Equation 2.4) has been independently discovered in the medical imaging community by Herman et al. [22], where it is known as the (unconstrained) algebraic reconstruction technique (ART). ART is used in computed tomography to reconstruct a three dimensional image of an object based on two dimensional projections onto a detector. Rather than being derived from a finite difference discretization of the wave equation, the matrix A

	Loading phase	FKSWP	BKSWP
Vector elements from LMem	0	2: $\mathbf{p}, \mathbf{0}$	2: FKSWP($\mathbf{p}$), $\mathbf{0}$
Vector elements to LMem	1: $\mathbf{p}$	1: FKSWP($\mathbf{p}$)	0
Matrix rows from LMem	0	1	1
Size of each vector element	8 bytes		
Size of matrix row	384 bytes		
DRAM bandwidth required	8 bytes	408 bytes	400 bytes
Vectors in from PCIe	1: $\mathbf{p}$	0	0
Vectors out to PCIe	0	1: FKSWP($\mathbf{p}$)	1: DKSWP($\mathbf{p}$)
PCIe bandwidth required	8 bytes	8 bytes	8 bytes

Table 3.2: Kaczmarz double sweep communication bandwidth requirements (per FPGA clock tick) for the current accelerated version. The notation used is applicable to the main while-loop of CGMN; the names of the vectors transferred will differ for sweeps in CGMN's initialization phase. Note that the forward sweep transfers its result to the CPU host even though it is not used in CGMN. Full sweep operator arguments have been left out for brevity.

in computed tomography is the discrete Radon transform, as noted by Yan [66]. The pattern of non-zeros in matrix row $\mathbf{a}_i$ corresponds to those voxels (small volume elements analogous to pixels in 3D) that are projected onto the detector at pixel i. Because the number of pixels in the 2D projection is smaller than the number of voxels in the volume to be reconstructed, the matrix A is underdetermined. It is possible to "stack" several such underdetermined systems, each corresponding to a single 2D projected image, and consider them as one (typically still underdetermined) system. I note that in full-waveform inversion the solution of the linear system 2.1 is only one step in an optimization algorithm to find the Earth model $\mathbf{m}$. In contrast, in computed tomography, solving the Radon system immediately yields the quantity of interest: the three dimensional volume.

A variant of ART is the simultaneous iterative reconstruction technique (SIRT), which is equivalent to the Jacobi iteration on the normal equations for the error (Cimmino's method, as presented by Elble et al. [12]). SIRT proceeds by computing the row-wise update term for all the rows at the same time and averaging the updates before applying them to the iterate, as described by Kak and Slaney [29]. For "stacked" systems it is conventional to apply the update one at a time to each block of rows that corresponds to one projected image. In this case the equivalence to Cimmino's method is lost. Yet another popular variation is simultaneous ART (SART), described by Andersen and Kak [2], which differs from SIRT in how the updates are averaged, as well as by replacing the square of the row norm $\|\mathbf{a}_i\|^2$ in Equation 2.4 by a sum of the row entries. This is the variation used by Grüll et al. [19]. SART is described in a rigorous way by Yan [66].

SART features a higher degree of parallelism than ART and for this reason is favoured for GPU implementations, such as those by Castaño-Díez et al. [6] and Xu et al. [65]. This is

because all the row projections for a SART sweep can be calculated in parallel, averaged and applied at the same time. Compare the inherently sequential nature of the Kaczmarz algorithm, which requires operating on mutually orthogonal rows in order to circumvent the latency of one Kaczmarz iteration. Instead of facing the challenge of pipelining a sequential algorithm, the FPGA-accelerated implementation presented by Grüll et al. [19] overcomes a different challenge. There is an important advantage enjoyed by the Helmholtz matrix (and other matrices derived from finite difference methods) over the matrices used in tomography: regular banded structure. Grüll et al. give two alternatives for dealing with the highly variable location of non-zeros within a matrix row. The first involves row-wise access, but since this requires random access of the iterate elements (which are stored in memory), it is not practical. The other takes advantage of the fact that the matrix has only 4 non-zeros per column. (This is due to the fact that each voxel can contribute to at most 4 image pixels.) The implementation thus proceeds through the matrix column-wise, accumulating all of the N update terms needed for SART into temporary storage. The heart of the implementation is the calculation of projection-ray-voxel intersections that form the non-zeros of the matrix, rather than a buffering scheme to implement a sliding window onto the iterate, as in the present work.

This chapter has conceptually described the implementation of the Kaczmarz row projection algorithm on an FPGA-based accelerator. This algorithm is used as the kernel of an iterative linear system solver, CGMN. While this chapter has outlined my contribution, my work is more accurately represented by the approximately 4000 lines of source code that make up the project, and 84 version control system revisions made from March 2013 to April 2014. The next chapter shows performance results of this implementation when it is used to simulate seismic waves through a synthetic three-dimensional Earth model.

Chapter 4

Performance results and discussion

To test the performance of the accelerated iterative Helmholtz solver, I compared two implementations. The *reference implementation* has CGMN implemented in MATLAB and the Kaczmarz kernel implemented as a C MEX file. Both CGMN and its kernel are run on the CPU host. The *accelerated implementation* also implements CGMN in MATLAB, but the kernel is instead run on the Vectis FPGA-based accelerator at an operational frequency of 100 MHz. It is worth noting that a MEX file is still required to interface between MATLAB and the accelerator; it is automatically generated during the FPGA bitstream build process. As mentioned in Section 3, the host CPU is an Intel Xeon E5-2670.

The reference implementation is run in single-threaded mode, using only one CPU core. On the one hand, this can be seen as a fair comparison with one FPGA-based accelerator, since it means that the degrees of parallelization in both the reference and accelerated implementations are the same. Both implementations are running the serial linear system solver CGMN. On the other hand, the restriction of the scope of the experiment to one accelerator and one core is somewhat arbitrary. Working within a fixed space (1 rack unit), power or budgetary constraint would be an alternative.

The 3D acoustic velocity model used to generate a Helmholtz matrix for these tests is shown in Figure 4.1, it is a part of the SEG/EAGE Overthrust model developed by Aminzadeh et al. [1]. The first quadrant of the full model was used, corresponding to the coordinate ranges $0 \leq x < 432$, $0 \leq y < 240$, $0 \leq z < 187$. The size of the system was varied by changing z, which increases down in Figure 4.1. A perfectly matched layer of 5 grid points was used on all sides of the domain, irrespective of system size. The frequency ω was fixed at 12 Hz to ensure at least 6 grid points per wavelength throughout the model. This is within the limit recommended by Operto et al. [42].

Two types of sources were used in the majority of the experiments: a point source placed near the middle of the top of the domain and a wavefield source. The wavefield source sets $\mathbf{q}$ equal to the solution of the Helmholtz system with a point source for a given domain size. (This solution must be calculated in advance.) The two types of sources emulate the forward and adjoint modes of solution of the wave equation (see Section 1.1). A third source was used

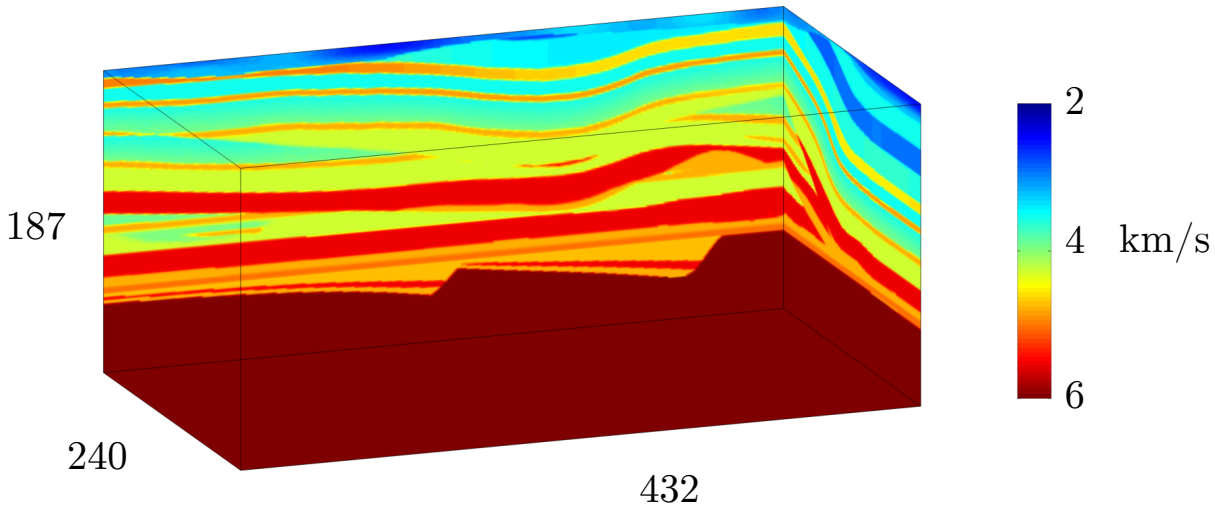

Figure 4.1: A part of the SEG/EAGE Overthrust velocity model was used to construct the Helmholtz matrix $A(\mathbf{m}, \omega)$. This model was used for testing the performance of the FPGA-based accelerator. The number of grid points is shown next to each axis.

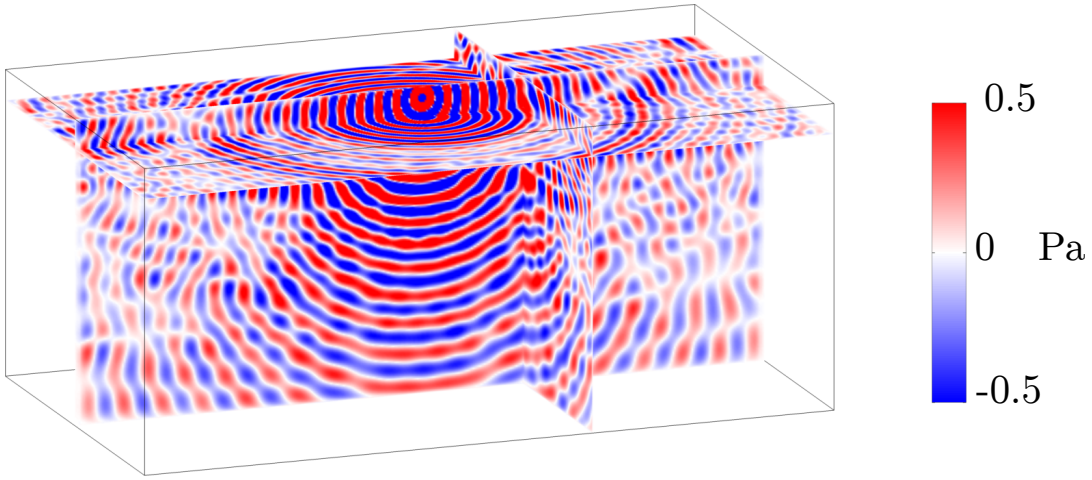

Figure 4.2: An example pressure wavefield solution u. The real part of the Fourier transform of the pressure is shown. The ripple signature of the point source used to create the wavefield is clearly visible. Using the accelerated implementation running on one Intel Xeon E5-2670 core and one Vectis FPGA-based accelerator, the solution shown took 2703 CGMN iterations to converge to a relative residual norm of 10^{-6}. This took 9826 s, or approximately 2:45 hours.

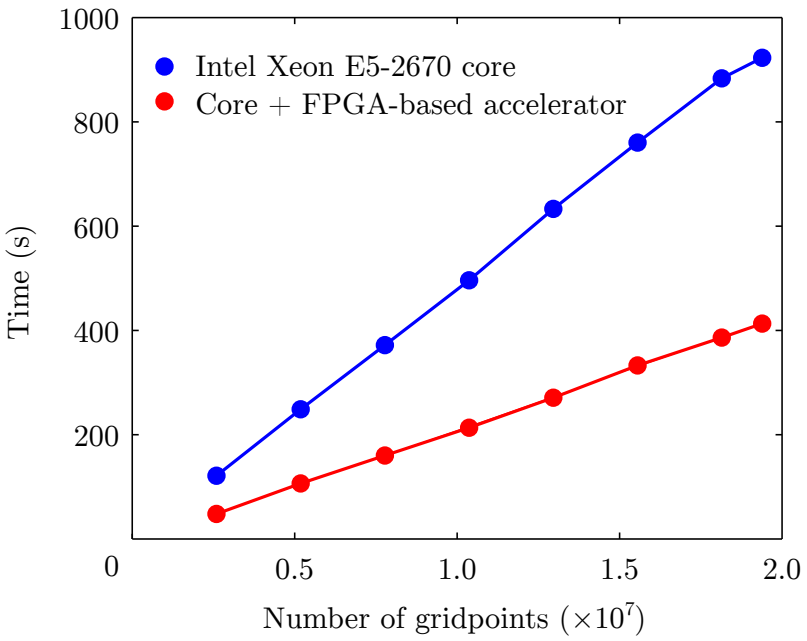

Figure 4.3: Time to execute 100 CGMN iterations on one Intel Xeon E5-2670 core is compared to the time taken on a combination of one such core and one FPGA-based, Vectis model accelerator. The accelerator is running at 100 MHz. The system solved is shown in Figure 4.1, with a source equal to $A\mathbf{1}$.

in the experiments described in the next section: $\mathbf{q} = A\mathbf{1}$. That is, the source was computed by multiplying a vector of ones by the Helmholtz matrix. This is done in order to know the true solution for evaluating how the error of the CGMN iterates behaves. I note that since the problem is formulated in the frequency domain, all sources are time-harmonic, corresponding to a perfect sinusoidal oscillator at 12 Hz. An example solution wavefield is shown in Figure 4.2.

4.1 Execution time and FPGA resource usage

Figure 4.3 compares end-to-end execution times for 100 CGMN iterations using the reference and accelerated implementations. These times do not include the time necessary to calculate the Helmholtz matrix A, but do include the time during initialization necessary to transfer this matrix and other vectors to the dedicated large memory of the accelerator. A speed-up of over $2\times$ is seen when the kernel is run on the accelerator. However when we examine the proportion of time spent in the kernel vs. the rest of CGMN, shown in Figure 4.4, we find that the kernel itself is accelerated by a factor of approximately 30, and accounts for only a small part of the time spent by the accelerated implementation. The rest is taken up by overhead resulting from two inefficiencies in the design. First, the real and imaginary parts of complex vectors produced in MATLAB are stored separately and must be interleaved before being passed to the accelerator. Second, the vector that is the output of the backward sweep is in reverse order,

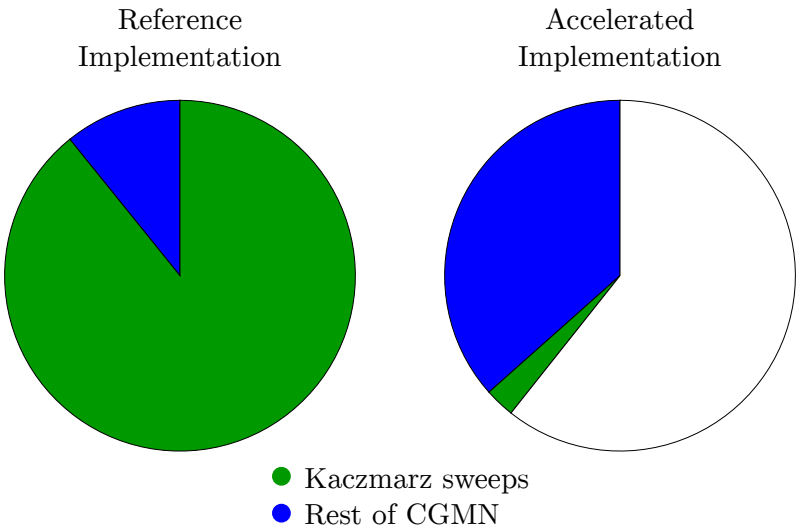

Figure 4.4: Distribution of computation time before and after acceleration. The "rest of CGMN" consists of reduction and elementwise operations. This proportion is stable with varying system size: the kernel takes up $89\% \pm 0.4$ and $3\% \pm 0.3$ of the time in the reference and accelerated implementations, respectively.

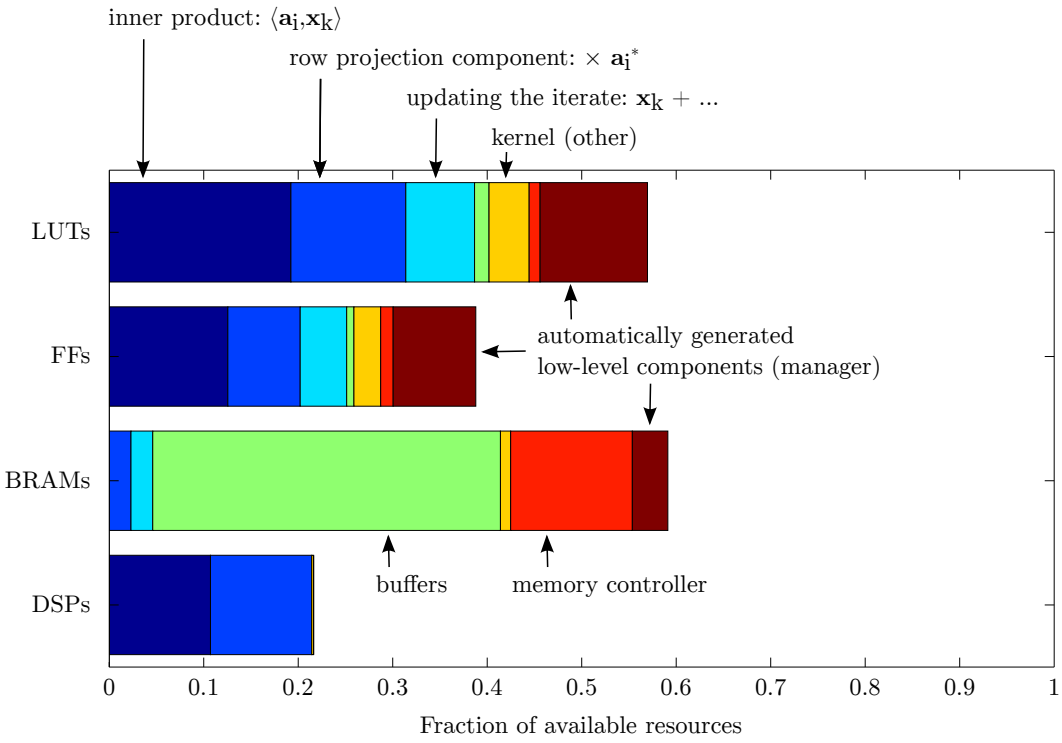

Figure 4.5: FPGA resource usage for lookup tables, flip-flops, block RAM (on-chip memory), and digital signal processing blocks. The design components that use a large fraction of the resources are labelled. Note that the memory controller is generated automatically.

and needs to have its elements rearranged before being passed to the parts of CGMN that are implemented in MATLAB.

The fraction of FPGA resources used by the accelerated implementation is shown in Figure 4.5.

4.2 Effects of reordering rows of the Helmholtz matrix

In Section 3.4 we saw that in order to keep the computational pipeline of the FPGA full it is necessary to access the rows of the matrix A non-sequentially. This accelerator ordering affects the number of iterations CGMN takes to converge. To evaluate whether this is a large effect that might significantly detract from the speed-up per iteration demonstrated in the previous section, I solved the Helmholtz system to a prescribed tolerance of 10^{-6}. Three different system sizes were used: $432 \times 240 \times \{25, 100, 187\}$ and two different seismic sources: a point source and a wavefield source, as described at the beginning of this chapter.

The results of this experiment are shown in Figure 4.6. For both the point and wavefield sources, accelerator ordering actually somewhat improves convergence at lower relative residual norm tolerance. For higher tolerance this effect is reversed, but also becomes less pronounced. In summary, changing the order in which the rows of A are used for the Kaczmarz row projections in the CGMN kernel to accelerator ordering does not detract from the performance of CGMN. In fact it could even be advantageous in cases where imprecise PDE solves are used in the initial iterations of full-waveform inversion, as suggested by van Leeuwen and Herrmann [61] and Herrmann et al. [23].

4.3 Multiple double Kaczmarz sweeps per CGMN iteration

The top plot in Figure 4.7 shows how convergence of CGMN is affected by running more than one double Kaczmarz sweep per CGMN iteration, as discussed in Section 2.4. As expected, doing more double Kaczmarz sweeps decreases the number of CGMN iterations required to converge to a given tolerance, but each of those iterations is now more expensive.

It is clear that since the Kaczmarz kernel is where most of the reference implementation spends its time (see Figure 4.4), doing more than one double sweep would not benefit the reference implementation. However for the accelerated implementation, since the Kaczmarz kernel is no longer the hotspot of the code, it makes sense to ask whether multiple sweeps per CGMN iteration might be advantageous. Because the overhead of each CGMN iteration is large compared to the cost of running the sweeps, running additional sweeps comes with only a small price.

Nevertheless, the current version of the accelerated implementation does not completely eliminate the increase in the time spent doing the reduction and elementwise operations of CGMN as the number of multiple sweeps per iteration is increased. Thus, as seen in the bottom plot of Figure 4.7, multiple sweeps still do not offer a decrease in net computation time.

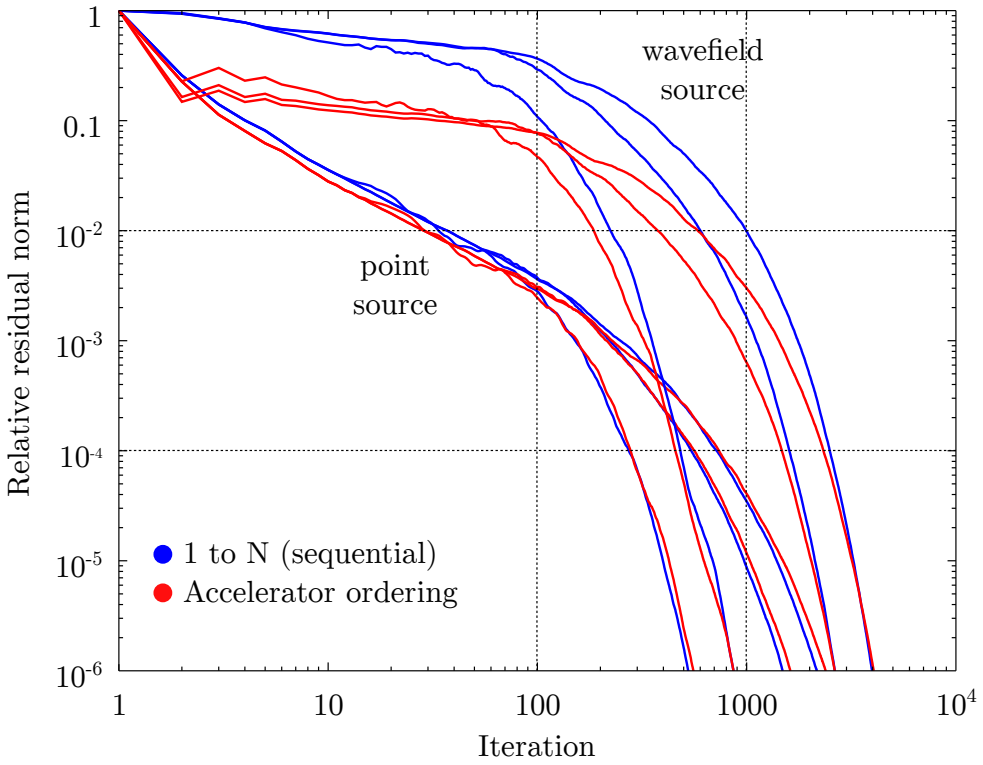

Figure 4.6: Impact of matrix row ordering on CGMN convergence. Two ways of ordering the rows are shown: 1 to N (sequential) and accelerator ordering. Accelerator ordering takes every third row along the fast (x) dimension and wraps back around upon reaching the end of a line along that dimension. (See section 3.4 for details.) The group of six curves toward the upper right of the plot were generated by solving a system with a wavefield source, and the six curves toward the bottom left were generated by solving a system with a point source. Three different system sizes are shown, corresponding to 2.6×10^6, 1.0×10^7 and 1.9×10^7 gridpoints. Larger systems take progressively more iterations to converge for both sources.

It is interesting to note however that *if* the overhead due to each CGMN iteration were held constant as the number of multiple sweeps is increased, performing 2–16 double sweeps would offer the best trade-off. Increasing the number of sweeps per CGMN iteration past this point would result in an increase in computational time.

Finally, I note that running multiple double Kaczmarz sweeps per CGMN iteration is an interim measure to attempt to make the most of the current version of the accelerated implementation. The end goal is an implementation of all of CGMN on the FPGA-based accelerator, as discussed in the next section.

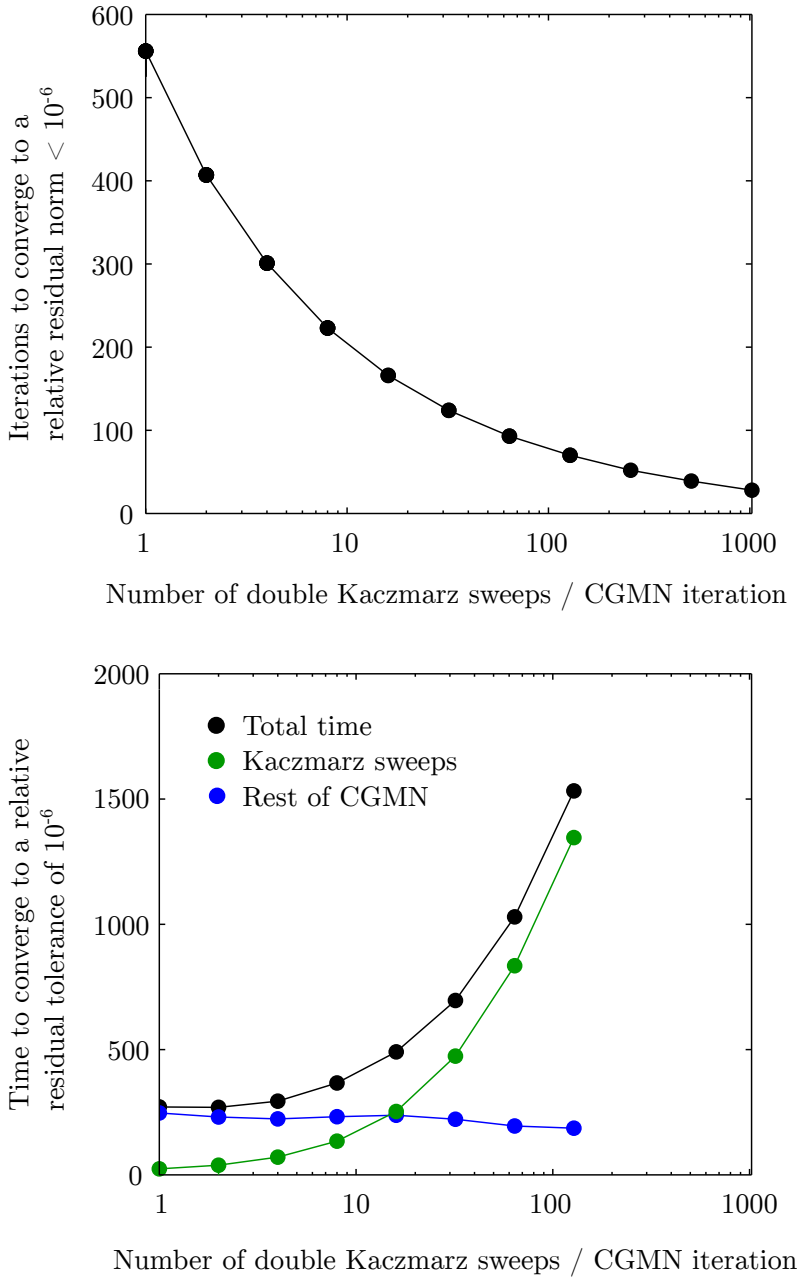

Figure 4.7: Effect of multiple sweeps per CGMN iteration on CGMN convergence. The top plot shows the number of iterations required to converge, and the bottom plot shows the end-to-end time taken (by the accelerated implementation). The total time is the sum of the time spent in the Kaczmarz sweeps and the reduction and elementwise operations that make up the rest of CGMN. Even though fewer CGMN iterations are required when multiple double Kaczmarz sweeps are performed at each iteration, the decrease is outweighed by the corresponding increase in computational time. The system is a $432 \times 240 \times 25$ part of the SEG/EAGE Overthrust system with a point source right-hand side.

4.4 Planned improvements

In the course of this work I have defined a to-do list of specific improvements to the current accelerated implementation. These fall into two categories: making the accelerated CGMN solver more readily useful to my colleagues (measured by uptake of the solver by users), and improving its performance (measured chiefly by end-to-end time taken by the solver to converge to a particular tolerance).

The main impediment to the use of my work in its current state is the requirement that the number of gridpoints along the fast dimension be equal to $3L = 432$, and the compile-time definition of the number of grid-points along the medium dimension. Re-ordering the dimensions of the Earth model before using it to compute A allows to specify one dimension at run time. Ultimately the size of the FPGA's fast on-chip memory limits the size of the first two dimensions of a block that can be processed at one time. A full solution would enable the user to specify an arbitrary domain at run time and leave it to the solver to transparently decompose it into blocks that fit in the on-chip memory.

As is clear from Figure 4.4, the most time-consuming part of the accelerated CGMN solver is no longer the Kaczmarz row projection kernel but the rest of CGMN. The most comprehensive method of dealing with this is to port all of CGMN to the accelerator. Once this has been done, the next step will be to increase the throughput of CGMN by reducing the number of passes over the data to two per CGMN iteration (as discussed in Appendix A) and compiling the design for a higher memory clock frequency and FPGA operational frequency. In order to avoid becoming bandwidth-limited at these higher frequencies, one option will be to change the representation of matrix elements to 28 bits per real number, as suggested by Figure 3.6. I term such a future implementation of CGMN on the accelerator the *efficient matrix storage* implementation. However the most effective way to reduce bandwidth requirements will be to transfer $\mathbf{m}$ to the accelerator and generate the elements of A as they are needed. Such an *on the fly* implementation would have the added benefit of moving algorithm complexity off the CPU host to the accelerator at no extra cost in computation time, and is the best way to realize the full potential of this computing platform. An estimate of the expected speed-up of the "on the fly" implementation can be gleaned from the right plot in Figure 4.4. Getting rid of the reduction and elementwise operations' contribution to the execution time (since they will be implemented on the FPGA) will yield an approximate speed-up of 30–60×, depending on the operational frequency of the FPGA.

Table 4.1 shows the bandwidth requirements of such future implementations of CGMN on the accelerator, they should be compared with Table 3.2. Figure 4.8 illustrates how those requirements affect the range of FPGA operational frequencies.

I note that the current accelerated implementation is limited by LMem bandwidth at an FPGA clock frequency of 71 MHz (94 MHz if the maximum memory clock frequency is set at compile time). The efficient matrix storage implementation will be bandwidth limited beyond 145 MHz. The "matrix on the fly" version is not limited by memory bandwidth within the

	First pass (incl. FKSWP)	Second pass (incl. BKSWP)
Vector elements from LMem	5 ($6^\dagger$): $\mathbf{r}, \mathbf{u}, \mathbf{p}, \mathbf{s}, \mathbf{0}, \mathbf{m}^\dagger$	3 ($4^\dagger$): $\mathbf{p}, \boldsymbol{tmp}, \mathbf{0}, \mathbf{m}^\dagger$
Vector elements to LMem	4: $\mathbf{r}, \mathbf{u}, \mathbf{p}, \boldsymbol{tmp}$	1: $\mathbf{s}$
Matrix rows from LMem	0	0
Size of each vector element	8 bytes	
Size of matrix row	192 bytes, (0 bytes)†	
DRAM bandwidth required	264 bytes (80 bytes)†	216 bytes (40 bytes)†
Vectors in from PCIe	0	0
Vectors out to PCIe	0	0
PCIe bandwidth required	0 bytes	0 bytes

Table 4.1: CGMN (future) communication bandwidth requirements (per FPGA clock tick) for the implementation that puts all of CGMN on the accelerator, entirely doing away with MATLAB for the main while loop. The table applies for the efficient matrix storage implementation; values for the "matrix on the fly" version are marked with a dagger†. Note that these requirements apply for the main while-loop of CGMN; the matrix, iterate and right-hand side still have to be transferred to the accelerator's dedicated large memory before CGMN starts. Likewise the final iterate will be transferred back to the host when the algorithm either converges or reaches the maximum number of iterations.

range of frequencies possible on the FPGA. Arranging the physical resources of the chip (LUTs, flip-flops, DSPs and BRAM) becomes difficult at high frequencies. This ability to meet timing will limit the FPGA frequency for the future "matrix on the fly" implementation of CGMN on the accelerator.

Since CGMN requires a zero right-hand side for most of its double Kaczmarz sweeps, it may be tempting to make a further change to the implementation by substituting a zero-stream generated on the FPGA for reading the vector $\mathbf{b}$ in from LMem. However since the reduction in communication this change would produce would be quite small (compared to improving the efficiency of matrix row storage or generating the matrix elements on the fly) and would decrease the generality of the sweeps, I do not recommend this change.

Finally, I note that the compute node on which this work was done actually has four FPGA-based accelerators, each attached to the CPU host via its own PCIe link, and interconnected with each other in a square topology. Coarse-grained parallelization by solving problems with different right-hand sides $\mathbf{q}$ (and even different Helmholtz matrices A) on each accelerator is immediately possible, and has been done by the author in an ad-hoc, manual manner. The systematic use of all four accelerators for parallelization of frequency domain FWI across sources and frequencies is a straightforward extension. The performance of the accelerated implementation on multiple accelerators is expected to scale in the same way as performance of a host application when using multiple nodes.

I emphasize that the important goal is not to implement one form of parallelization in preference to another, but to maximize the throughput of all computational units available

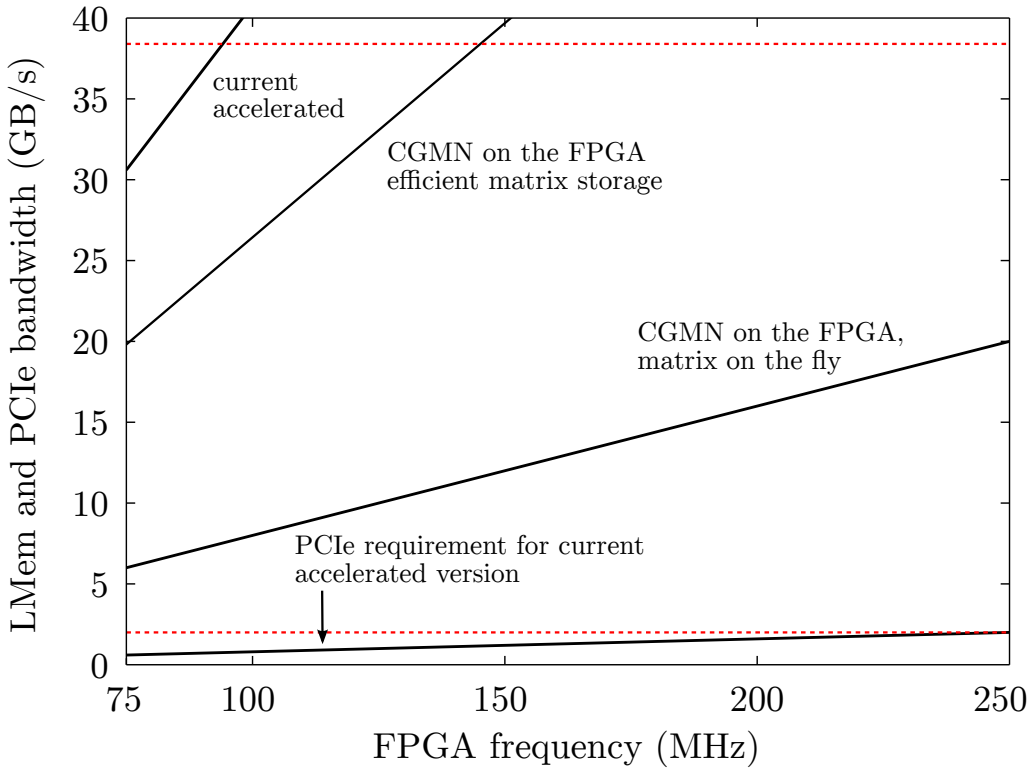

Figure 4.8: Communication bandwidth requirements for LMem and the PCIe bus are affected by the operating frequency of the FPGA and the details of the implementation. The two dashed red lines show the theoretical hardware bandwidth limits for memory (top) and the PCIe bus (bottom). Note that the implementations with all of CGMN on the FPGA have no host PCIe bandwidth requirements after CGMN initialization.

to us. This should be done using the most straightforward mode of parallelism possible. For the platform studied in this work, reordering the rows of the Helmholtz matrix as discussed in Section 3.4, and solving different Helmholtz problems on each of the four accelerators is the approach I recommend as long as the wavefield and other CGMN vectors fit into the 24 GB dedicated memory of one accelerator. I assume here that all of CGMN is implemented on the accelerator. Solving problems whose domains do not fit into one accelerator's memory could be dealt with by using a row-block parallelization approach named CARP-CG, suggested by Gordon and Gordon [17]. Those authors provide a block-parallelization of the Kaczmarz algorithm termed CARP that performs double sweeps on sets of rows in parallel, averaging those elements of $\mathbf{u}$ that are updated by sweeps on more than one set. Such an approach would necessitate exchange of shared elements, which adds a layer of complexity to the dataflow paradigm.

Chapter 5

Conclusions

I have demonstrated a time-harmonic iterative wave equation solver that offloads its computational kernel (the Kaczmarz row projection algorithm) onto an FPGA-based accelerator. In the course of the work, challenges such as reading data from slow memory (including reading the memory backward), orchestrating the flow of intermediate results to maximize data re-use, and seeking out parallelism to fill the computational pipeline of the accelerator were encountered and overcome. The changes to the kernel algorithm dictated by the characteristics of the specific hardware used were found to be benign or have positive side-effects. The end-to-end speed-up of the current accelerated version of the wave equation solver is more than a factor of two compared to one Intel processor core. A planned future version that implements all of the linear solver on the accelerator is estimated to give a combined performance improvement of more than an order of magnitude. The speed-up of the wave equation solver achieved via the accelerator translates directly into being able to run more full-waveform inversion iterations in a given time, which leads to more meaningful FWI results.

The accelerated implementation as presented here is specific to matrices having the banded structure arising from the $3 \times 3 \times 3$ cube finite difference stencil. However now that the design methodology has been explored, generalizing to FD matrices with different stencils will be easier. Indeed, provided that computational grid locality still translates into approximate memory locality, and that some structure is present in the matrix, it should be possible to extend the present approach to more general matrices as well. Due to the availability of a time-stepping library for the accelerator, comparisons with time domain methods are a further avenue of research.

Exciting opportunities lie in better tailoring the algorithm used in the iterative frequency domain solver to the specific hardware. For example, since each pass over the data linearly increases the execution time of the algorithm, it behoves the author to search for an effective algorithm that only requires one pass per solver iteration. Other work involves determining what, if any, advantages may be gained by implementing an asynchronous Kaczmarz algorithm, as suggested by Liu et al. [34]. Such parallel approaches will become increasingly important in the future as dataset sizes increase, since Dongarra et al. [10] make the point that "exascale

computing restricts algorithmic choices, eliminating the use of important techniques such as lexicographic Gauss-Seidel smoothing [for being] too sequential."

Work on this project is ongoing, and while further development is needed in order to better realize the potential for acceleration inherent in the platform, the results presented herein give reason to be optimistic.

Bibliography

[1] F. Aminzadeh, B. Jean, and T. Kunz. *3-D Salt and Overthrust Models*. Society of Exploration Geophysicists, 1997. → pages 28

[2] A. Andersen and A. Kak. Simultaneous algebraic reconstruction technique (SART): A superior implementation of the ART algorithm. *Ultrasonic Imaging*, 6(1):81–94, 1984. ISSN 0161-7346. doi: 10.1016/0161-7346(84)90008-7. URL http://www.sciencedirect.com/science/article/pii/0161734684900087. → pages 26

[3] A. Y. Aravkin, M. P. Friedlander, F. J. Herrmann, and T. van Leeuwen. Robust inversion, dimensionality reduction, and randomized sampling. *Mathematical Programming*, 134(1):101–125, 08 2012. → pages 2

[4] Å. Björck and T. Elfving. Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. *BIT Numerical Mathematics*, 19(2):145–163, 1979. ISSN 0006-3835. doi: 10.1007/BF01930845. URL http://dx.doi.org/10.1007/BF01930845. → pages 5, 9, 10, 11, 48

[5] R. Brossier, S. Operto, and J. Virieux. Seismic imaging of complex onshore structures by 2d elastic frequency-domain full-waveform inversion. *Geophysics*, 74(6): WCC105–WCC118, 2009. doi: 10.1190/1.3215771. URL http://library.seg.org/doi/abs/10.1190/1.3215771. → pages 2

[6] D. Castaño-Díez, H. Mueller, and A. S. Frangakis. Implementation and performance evaluation of reconstruction algorithms on graphics processors. *Journal of Structural Biology*, 157(1):288–295, 2007. ISSN 1047-8477. doi: 10.1016/j.jsb.2006.08.010. URL http://www.sciencedirect.com/science/article/pii/S1047847706002760. → pages 26

[7] A. Chronopoulos and C. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153 – 168, 1989. ISSN 0377-0427. doi: http://dx.doi.org/10.1016/0377-0427(89)90045-9. URL http://www.sciencedirect.com/science/article/pii/0377042789900459. → pages 47, 48

[8] B. Dally. Efficiency and parallelism: The challenges of future computing, March 2014. URL http://rice2014oghpc.blogs.rice.edu/files/2014/03/Dally.pdf. Keynote lecture at Oil & Gas HPC Workshop. → pages 3

[9] R. Diamond. Maxeler developer exchange: Memory architecture, Sept 2011. URL https://groups.google.com/a/maxeler.com/d/msg/mdx/AY-p73PyGXI/0WcXDIE0U2oJ. → pages 13, 21

[10] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. D. Hovland, E. Ng, C. Webster, and S. M. Wild. Applied mathematics research for exascale computing, 2014. URL http://www.netlib.org/utk/people/JackDongarra/PAPERS/doe-exascale-math-report.pdf. LLNL-TR-651000. → pages 38

[11] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 2007. → pages 16, 25

[12] J. M. Elble, N. V. Sahinidis, and P. Vouzis. GPU computing with Kaczmarz's and other iterative algorithms for linear systems. *Parallel Computing*, 36(5-6):215–231, June 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.003. URL http://dx.doi.org/10.1016/j.parco.2009.12.003. → pages 14, 26

[13] O. Ernst and M. Gander. Why it is difficult to solve Helmholtz problems with classical iterative methods. In I. G. Graham, T. Y. Hou, O. Lakkis, and R. Scheichl, editors, *Numerical Analysis of Multiscale Problems*, volume 83 of *Lecture Notes in Computational Science and Engineering*, pages 325–363. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-22060-9. doi: 10.1007/978-3-642-22061-6_10. URL http://dx.doi.org/10.1007/978-3-642-22061-6_10. → pages 8

[14] H. Fu, W. Osborne, R. G. Clapp, and O. Pell. Accelerating seismic computations on FPGAs–from the perspective of number representations. In *70th EAGE Conference & Exhibition*, 2008. → pages 14

[15] M. S. Gockenbach, D. R. Reynolds, P. Shen, and W. W. Symes. Efficient and automatic implementation of the adjoint state method. *ACM Trans. Math. Softw.*, 28(1):22–44, Mar. 2002. ISSN 0098-3500. doi: 10.1145/513001.513003. URL http://doi.acm.org/10.1145/513001.513003. → pages 6

[16] D. Gordon and R. Gordon. CGMN revisited: Robust and efficient solution of stiff linear systems derived from elliptic partial differential equations. *ACM Trans. Math. Softw.*, 35 (3):18:1–18:27, Oct 2008. ISSN 0098-3500. doi: 10.1145/1391989.1391991. URL http://doi.acm.org/10.1145/1391989.1391991. → pages 8

[17] D. Gordon and R. Gordon. CARP-CG: A robust and efficient parallel solver for linear systems, applied to strongly convection dominated PDEs. *Parallel Computing*, 36(9): 495–515, 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2010.05.004. URL http://www.sciencedirect.com/science/article/pii/S0167819110000827. → pages 10, 37

[18] D. Gordon and R. Gordon. Robust and highly scalable parallel solution of the Helmholtz equation with large wave numbers. *Journal of Computational and Applied Mathematics*, 237(1):182–196, 2013. ISSN 0377-0427. doi: http://dx.doi.org/10.1016/j.cam.2012.07.024. URL http://www.sciencedirect.com/science/article/pii/S0377042712003147. → pages 10

[19] F. Grüll, M. Kunz, M. Hausmann, and U. Kebschull. An implementation of 3D electron tomography on FPGAs. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–5, 2012. doi: 10.1109/ReConFig.2012.6416732. → pages 12, 14, 25, 26, 27

[20] S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Systems on Silicon. Elsevier Science, 2010. ISBN 9780080556017. → pages 4, 12

[21] C. He, M. Lu, and C. Sun. Accelerating seismic migration using FPGA-based coprocessor platform. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 207–216, April 2004. doi: 10.1109/FCCM.2004.12. → pages 13, 14

[22] G. T. Herman, A. Lent, and S. W. Rowland. ART: Mathematics and applications: A report on the mathematical foundations and on the applicability to real data of the algebraic reconstruction techniques. *Journal of Theoretical Biology*, 42(1):1 – 32, 1973. ISSN 0022-5193. doi: http://dx.doi.org/10.1016/0022-5193(73)90145-8. URL http://www.sciencedirect.com/science/article/pii/0022519373901458. → pages 25

[23] F. J. Herrmann, A. J. Calvert, I. Hanlon, M. Javanmehri, R. Kumar, T. van Leeuwen, X. Li, B. Smithyman, E. T. Takougang, and H. Wason. Frugal full-waveform inversion: from theory to a practical algorithm. *The Leading Edge*, 32(9):1082–1092, 09 2013. doi: http://dx.doi.org/10.1190/tle32091082.1. URL https://www.slim.eos.ubc.ca/Publications/Public/Journals/The_Leading_Edge/2013/herrmann2013ffwi/herrmann2013ffwi.html. → pages 32

[24] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952. → pages 8

[25] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2012. URL http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. Order Number: 248966-026. → pages 4

[26] T. Johnsen and A. Loddoch. Hybrid CPU-GPU finite difference time domain kernels, March 2014. URL http://rice2014oghpc.blogs.rice.edu/files/2014/03/Johnsen.pdf. → pages 3

[27] S. Kaczmarz. Angenäherte auflösung von systemen linearer gleichungen. *Bulletin International de l'Academie Polonaise des Sciences et des Lettres*, 35:355–357, 1937. → pages 8

[28] S. Kaczmarz. Approximate solution of systems of linear equations. *International Journal of Control*, 57(6):1269–1271, 1993. doi: 10.1080/00207179308934446. → pages 8

[29] A. C. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, 2001. → pages 9, 26

[30] H. Knibbe, W. Mulder, C. Oosterlee, and C. Vuik. Closing the performance gap between an iterative frequency-domain solver and an explicit time-domain scheme for 3D migration on parallel architectures. *Geophysics*, 79(2):S47–S61, 2014. doi: 10.1190/geo2013-0214.1. URL http://library.seg.org/doi/abs/10.1190/geo2013-0214.1. → pages 3, 6, 14

[31] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund. Hardware/software co-design for energy-efficient seismic modeling. In *Proceedings of 2011 International Conference for High Performance Computing,*

Networking, Storage and Analysis, SC '11, pages 73:1–73:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063482. URL http://doi.acm.org/10.1145/2063384.2063482. → pages 14

[32] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM. ISBN 1-59593-292-5. doi: 10.1145/1117201.1117205. URL http://doi.acm.org/10.1145/1117201.1117205. → pages 4

[33] X. Li, A. Y. Aravkin, T. van Leeuwen, and F. J. Herrmann. Fast randomized full-waveform inversion with compressive sensing. *Geophysics*, 77(3):A13–A17, 05 2012. URL https://www.slim.eos.ubc.ca/Publications/Public/Journals/Geophysics/2012/Li11TRfrfwi/Li11TRfrfwi.pdf. → pages 2

[34] J. Liu, S. J. Wright, and S. Sridhar. An asynchronous parallel randomized Kaczmarz algorithm. *arXiv preprint arXiv:1401.4780*, 2014. → pages 38

[35] A. R. Lopes and G. A. Constantinides. A high throughput FPGA-based floating point conjugate gradient implementation. In R. Woods, K. Compton, C. Bouganis, and P. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 75–86. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78609-2. doi: 10.1007/978-3-540-78610-8_10. URL http://dx.doi.org/10.1007/978-3-540-78610-8_10. → pages 14

[36] Maxeler Technologies. Maxcompiler white paper, Feb 2011. URL http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf. → pages 3, 13, 21

[37] Maxeler Technologies. *Multiscale Dataflow Programming*. Maxeler Technologies, 2013. → pages 13, 18

[38] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites, November 2013. URL https://www.top500.org. → pages 3

[39] G. Morris and V. Prasanna. An FPGA-based floating-point Jacobi iterative solver. In *Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, pages 8 pp.–, Dec 2005. doi: 10.1109/ISPAN.2005.18. → pages 14

[40] G. Morris, V. Prasanna, and R. Anderson. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 3–12, April 2006. doi: 10.1109/FCCM.2006.8. → pages 14

[41] T. Nemeth, J. Stefani, W. Liu, R. Dimond, O. Pell, and R. Ergas. An implementation of the acoustic wave equation on FPGAs. In *SEG Technical Program Expanded Abstracts 2008*, pages 2874–2878. Society of Exploration Geophysicists, 2008. doi: 10.1190/1.3063943. URL http://library.seg.org/doi/abs/10.1190/1.3063943. → pages 23

[42] 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(5):SM195–SM211, 2007. doi: 10.1190/1.2759835. URL http://geophysics.geoscienceworld.org/content/72/5/SM195.abstract. → pages 6, 7, 16, 28

[43] O. Pell. Maxeler developer exchange: DRAM and PCIe bandwidth, Oct 2013. URL https://groups.google.com/a/maxeler.com/d/msg/mdx/UWdIf82dvt4/dD_KITGqWHEJ. → pages 25

[44] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. J. Flynn. Finite-difference wave propagation modeling on special-purpose dataflow machines. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):906–915, 2013. ISSN 1045-9219. doi: 10.1109/TPDS.2012.198. → pages 13, 14, 23

[45] A. Petrenko, D. Oriato, S. Tilbury, T. van Leeuwen, and F. J. Herrmann. Accelerating an iterative Helmholtz solver with FPGAs. In *EAGE*, 06 2014. URL https://www.slim.eos.ubc.ca/Publications/Public/Conferences/EAGE/2014/petrenko2014EAGEaih.pdf. → pages

[46] A. Petrenko, D. Oriato, S. Tilbury, T. van Leeuwen, and F. J. Herrmann. Accelerating an iterative Helmholtz solver with FPGAs. In *OGHPC*, 03 2014. URL https://www.slim.eos.ubc.ca/Publications/Public/Conferences/OGHPC/petrenko2014OGHPCaih.pdf. → pages

[47] R. Plessix. Three-dimensional frequency-domain full-waveform inversion with an iterative solver. *Geophysics*, 74(6):WCC149–WCC157, 2009. doi: 10.1190/1.3211198. URL http://library.seg.org/doi/abs/10.1190/1.3211198. → pages 6

[48] R.-E. Plessix. A Helmholtz iterative solver for 3D seismic-imaging problems. *Geophysics*, 72(5):SM185–SM194, 2007. → pages 6

[49] G. Pratt, C. Shin, and Hicks. GaussâĂŞNewton and full Newton methods in frequencyâĂŞspace seismic waveform inversion. *Geophysical Journal International*, 133 (2):341–362, 1998. ISSN 1365-246X. doi: 10.1046/j.1365-246X.1998.00498.x. URL http://dx.doi.org/10.1046/j.1365-246X.1998.00498.x. → pages 2

[50] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985. doi: 10.1137/0906059. URL http://epubs.siam.org/doi/abs/10.1137/0906059. → pages 47

[51] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. → pages 8, 9, 14

[52] D. Singh. Implementing FPGA design with the OpenCL standard. *Altera whitepaper*, 2011. URL http://www.altera.com/literature/wp/wp-01173-opencl.pdf. → pages 4

[53] L. Sirgue, J. Etgen, and U. Albertin. 3d frequency domain waveform inversion using time domain finite difference methods. In *70th EAGE Conference & Exhibition*, 2008. → pages 6

[54] R. Strzodka and D. Göddeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 259–270, April 2006. doi: 10.1109/FCCM.2006.57. → pages 14, 22, 23

[55] J. Sun, G. Peterson, and O. Storaasli. High-performance mixed-precision linear solver for FPGAs. *Computers, IEEE Transactions on*, 57(12):1614–1623, Dec 2008. ISSN 0018-9340. doi: 10.1109/TC.2008.89. → pages 14, 22

[56] W. Symes. Reverse time migration with optimal checkpointing. *Geophysics*, 72(5): SM213–SM221, 2007. doi: 10.1190/1.2742686. URL http://library.seg.org/doi/abs/10.1190/1.2742686. → pages 5

[57] K. Tanabe. Projection method for solving a singular system of linear equations and its applications. *Numerische Mathematik*, 17(3):203–214, 1971. ISSN 0029-599X. doi: 10.1007/BF01436376. URL http://dx.doi.org/10.1007/BF01436376. → pages 10

[58] A. Tarantola. Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, 49(8):1259–1266, 1984. doi: 10.1190/1.1441754. URL http://library.seg.org/doi/abs/10.1190/1.1441754. → pages 1

[59] T. van Leeuwen. Fourier analysis of the CGMN method for solving the Helmholtz equation. Technical report, Department of Earth, Ocean and Atmospheric Sciences, The University of British Columbia, Vancouver, 2012. URL http://arxiv.org/abs/1210.2644. → pages 9, 10

[60] T. van Leeuwen and F. J. Herrmann. Mitigating local minima in full-waveform inversion by expanding the search space. *Geophysical Journal International*, 195:661–667, 10 2013. doi: 10.1093/gji/ggt258. → pages 5

[61] T. van Leeuwen and F. J. Herrmann. 3D frequency-domain seismic inversion with controlled sloppiness. To appear in the SIAM Journal on Scientific Computing (SISC), 03 2014. URL https://www.slim.eos.ubc.ca/Publications/Public/Journals/SIAM_Journal_on_Scientific_Computing/2014/vanLeeuwen20143Dfds/vanLeeuwen20143Dfds.pdf. → pages 32

[62] T. van Leeuwen, D. Gordon, R. Gordon, and F. J. Herrmann. Preconditioning the Helmholtz equation via row-projections. In *EAGE technical program*. EAGE, 01 2012. URL https://www.slim.eos.ubc.ca/Publications/Public/Conferences/EAGE/2012/vanleeuwen2012EAGEcarpcg/vanleeuwen2012EAGEcarpcg.pdf. → pages 8, 21

[63] J. Virieux and S. Operto. An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1–WCC26, 2009. doi: 10.1190/1.3238367. URL http://library.seg.org/doi/abs/10.1190/1.3238367. → pages 2, 5, 6

[64] Xilinx. Virtex-6 family overview, 2012. URL http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. DS150 (v2.4). → pages 13, 18

[65] W. Xu, F. Xu, M. Jones, B. Keszthelyi, J. Sedat, D. Agard, and K. Mueller. High-performance iterative electron tomography reconstruction with long-object compensation using graphics processing units (GPUs). *Journal of Structural Biology*, 171 (2):142–153, 2010. ISSN 1047-8477. doi: 10.1016/j.jsb.2010.03.018. URL http://www.sciencedirect.com/science/article/pii/S104784771000095X. → pages 26

[66] M. Yan. Convergence analysis of SART: Optimization and statistics. *International Journal of Computer Mathematics*, 90(1):30–47, 2013. doi: 10.1080/00207160.2012.709933. URL http://www.tandfonline.com/doi/abs/10.1080/00207160.2012.709933. → pages 26

[67] D. Yang, G. Peterson, H. Li, and J. Sun. An FPGA implementation for solving least square problem. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 303–306, April 2009. doi: 10.1109/FCCM.2009.47. → pages 14

Appendix A

An alternative conjugate gradient formulation

An implementation of CGMN as shown in Algorithm 1 using the streaming dataflow paradigm natural to working with an FPGA accelerator would require three passes over the data for each iteration. The two inner products $\langle \mathbf{p}, \mathbf{s} \rangle$ and $\|\mathbf{r}\|^2$ are reduction operations that must be completed in their entirety (over the entire stream) before the rest of the iteration can continue. These synchronization points were also an inconvenience for conjugate gradient implementations on vector and distributed computers during the 1980's. A solution, found by Saad [50] among others, was to reformulate the calculation of CG's β. This reformulation, applied to CGMN, is shown in Algorithm 2 and allows to fit the algorithm into two passes over the data provided the matrix is banded. It was suggested to me by Diego Oriato of Maxeler technologies and can be found in the work of Chronopoulos and Gear [7] who apply it to CG. Chronopoulos and Gear also note that recalculating $\|\mathbf{r}\|^2$ every iteration directly using $\mathbf{r}$ ensures numerical stability of CG.

Algorithm 2 The CGMN algorithm (Björck and Elfving [4]) split into two passes over the data following Chronopoulos and Gear [7].

Input: $A, \mathbf{u}, \mathbf{q}, \lambda$

$R\mathbf{q} \leftarrow \text{DKSWP}(A, \mathbf{0}, \mathbf{q}, \lambda)$

$\mathbf{r} \leftarrow R\mathbf{q} - \mathbf{u} + \text{DKSWP}(A, \mathbf{u}, \mathbf{0}, \lambda)$

$\mathbf{p} \leftarrow \mathbf{r}$

$\mathbf{s} \leftarrow \text{DKSWP}(A, \mathbf{p}, \mathbf{0}, \lambda)$

$\alpha \leftarrow \|\mathbf{r}\|^2 / \langle \mathbf{p}, \mathbf{s} \rangle$

$\beta \leftarrow 0$

while $\|\mathbf{r}\|^2 > tol$ **do**

  Data pass 1

  $\mathbf{u} \leftarrow \mathbf{u} + \alpha\mathbf{p}$

  $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{s}$

  $\mathbf{p} \leftarrow \mathbf{r}_i + \beta\mathbf{p}_{i-1}$

  $\boldsymbol{tmp} \leftarrow \text{FKSWP}(A, \mathbf{p}, \mathbf{0}, \lambda)$

  Data pass 2

  $\mathbf{s} \leftarrow \mathbf{p} - \text{BKSWP}(A, \boldsymbol{tmp}, \mathbf{0}, \lambda)$

  $\alpha \leftarrow \|\mathbf{r}\|^2 / \langle \mathbf{p}, \mathbf{s} \rangle$

  $\beta \leftarrow \left(\alpha^2 \|\mathbf{s}\|^2 - \|\mathbf{r}\|^2\right) / \|\mathbf{r}\|^2$

end while

Output: $\mathbf{u}$