DISSERTATION

submitted

to the

Combined Faculties for the Natural Sciences and for Mathematics

of the

Ruperto-Carola University of Heidelberg, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

Master of Science Jens-Thomas Krüger

Born in Hilden

Oral examination:

Green Wave: A Semi-Custom Hardware Architecture for Reverse Time Migration

Advisor: Prof. Dr. Ulrich Brüning

Abstract

Over the course of the last few decades the scientific community greatly benefited from steady advances in compute performance. Until the early 2000's this performance improvement was achieved through rising clock rates. This enabled plug-n-play performance improvements for all codes. In 2005 the stagnation of CPU clock rates drove the computing hardware manufactures to attain future performance through explicit parallelism. Now the HPC community faces a new, even bigger challenge. So far performance gains were achieved through replication of general-purpose cores and nodes. Unfortunately, rising cluster sizes resulted in skyrocketing energy costs - a paradigm change in HPC architecture design is inevitable. In combination with the increasing costs of data movement, the HPC community started exploring alternatives like GPUs and large arrays of simple, low-power cores (e.g. BlueGene) to offer the better performance per Watt and greatest scalability.

As in general science, the seismic community faces large-scale, complex computational challenges that can only be limited solved with available compute capabilities. Such challenges include the physically correct modeling of subsurface rock layers. This thesis analyzes the requirements and performance of isotropic (ISO), vertical transverse isotropic (VTI) and tilted transverse isotropic (TTI) wave propagation kernels as they appear in the Reverse Time Migration (RTM) imaging method. It finds that even with leading-edge, commercial off-the-shelf hardware, large-scale survey sizes cannot be imaged within reasonable time and power constraints.

This thesis uses a novel architecture design method leveraging a hardware/software co-design approach, adopted from the mobile- and embedded market, for HPC. The methodology tailors an architecture design to a class of applications without loss of generality like in full custom designs. This approach was first applied in the Green Flash project, which proved that the co-design approach has the potential for high energy efficiency gains. This thesis presents the novel Green Wave architecture that is derived from the Green Flash project. Rather than focusing on climate codes, like Green Flash,

Green Wave chooses RTM wave propagation kernels as its target application. Thus, the goal of the application-driven, co-design Green Wave approach, is to enable full programmability while allowing greater computational efficiency than general-purpose processors or GPUs by offering custom extensions to the processor's ISA and correctly sizing software-managed memories and an efficient on-chip network interconnect. The lowest level building blocks of the Green Wave design are pre-verified IP components. This minimizes the amount of custom logic in the design, which in turn reduces verification costs and design uncertainty.

In this thesis three Green Wave architecture designs derived from ISO, VTI and TTI kernel analysis are introduced. Further, a programming model is proposed capable of hiding all communication latencies. With production-strength, cycle-accurate hardware simulators Green Wave's performance is benchmarked and its performance compared to leading on-market systems from Intel, AMD and NVidia. Based on a large-scale example survey, the results show that Green Wave has the potential of an energy efficiency improvement of 5× compared to x86 and 1.4×-4× to GPU-based clusters for ISO, VTI and TTI kernels.

Zusammenfassung

Im Laufe der vergangenen Jahrzehnte profitierte die Wissenschaft von stetigen Leistungssteigerungen im Hochleistungsrechnen. Bis Anfang des neuen Jahrtausends wurden diese insbesondere durch höhere Taktraten der Prozessoren erreicht. Durch einfaches Austauschen älterer Prozessoren durch eine neue Generation wurde bessere Leistung für alle Codes erreicht. Diese Entwicklung endete im Jahre 2005. Mit 4 Ghz waren Prozessoren an eine Grenze gestoßen, bei der Wärmeentwicklung und Stromverbrauch nicht weiter gesteigert werden konnten um höhere Taktraten zu ermöglichen. Um zukünftige Leistungssteigerungen zu ermöglichen, wurden Taktraten gesenkt und Leistung durch ausnutzen expliziten Parallelismus, innerhalb eines „Shared Multiprocessors“, erreicht. Heute steht die High-Performance Computing Gemeinschaft vor einer neuen, noch größeren, Herausforderung. Um den stetig wachsenden Leistungshunger im wissenschaftlichen Rechnen zu befriedigen, wurden immer mehr Prozessoren in HPC Systemen verbaut. Genau wie Mitte des ersten Jahrzehnts die Leistungsaufnahme eines einzelnen Prozessors an seine Grenzen stieß, gilt dies auch für Großrechner von heute, bei denen die Leistungsaufnahme und damit die Kosten für Energie und Infrastruktur, über ökonomisch und ökologisch, vertretbare Grenzen hinausgehen. Ein radikaler Wandel in der HPC ist deshalb unausweichlich. Auf Grund dessen rücken alternative Ansätze, wie etwa GPUs und „Many-Core“ Systeme, verstärkt in den Fokus von Wissenschaft und Industrie.

Insbesondere die Öl- und Gas-Industrie sieht sich enormen Herausforderungen gegenübergestellt um physikalisch korrekte Abbildungen des Untergrundes, für explorative Zwecke zu erstellen. Diese Arbeit analysiert drei wesentliche Wellenpropagationskernel wie sie für die Reverse Time Migration (RTM) verwendet werden: für isotrope, vertikal transversal isotrope und geneigt transversal isotrope Medien.

Die Analysen dieser Arbeit zeigen, dass auch auf Computersystemen neuester Generation solche Algorithmen, angewandt auf große Explorationsvolumen und kurze Rechenzeiten, von der Leistungsaufnahme für kein Rechenzentrum zu vertreten sind.

In dieser Arbeit wird ein neuartiger „Hardware/Software Co-Design“ Ansatz für HPC Architektur benutzt um signifikante Verbesserungen gegenüber allen evaluierten, markterhältlichen Systemen zu erreichen und RTM selbst für große Gebiete ermöglicht. Es wird eine neue Prozessorarchitektur mit dem Namen „Green Wave“ vorgestellt, welche auf eine Klasse von Algorithmen optimiert ist und sich somit, anders als bei voll angepassten Designkonzepten, nicht auf spezielle Kernel beschränkt. Green Wave basiert auf Tensilica's hoch-effizienten LX4 Prozessor, der das Hinzufügen von kernel-spezifischen Instruktionen ermöglicht. Mit weiterem Anpassen des Chipdesigns durch „Local-Stores“ und einem effizientem „Network-on-Chip“ wird eine bestmögliche Energieeffizienz erreicht. Die weiteren Grundbausteine von Green Wave sind vor-verifizierte, markterhältliche Hardwarekomponenten, um Verifikations- und Produktionskosten möglichst gering zu halten. Anhand der vorgestellten Programmiermodellen für Green Wave, werden mit dem Tensilica „Instruction Set Simulator“ (ISS) zyklengenaue Leistungsbenchmarks erstellt und werden mit den evaluierten Architekturen von Intel, AMD und NVidia, verglichen. Die Ergebnisse zeigen, dass Energieeffizienz-Verbesserungen von ca. 5x gegenüber x86 basierten Architekturen und 1.4x bis 4x gegenüber GPU basierten Systemen erreicht werden.

Acknowledgements

I'd like to thank my advisor Professor Ulrich Bruening, head of the Computer Architecture Group of the University of Heidelberg. He accepted me as his PhD student and his always friendly support and experience made it possible for me to finish this thesis. I greatly benefited from his incredible experience in computer architecture. I will always remember his great advice and valuable lessons.

I'd like to thank Franz-Josef Pfreundt head of the Competence Center High Performance Computing (CC-HPC) department at the Fraunhofer ITWM. With his knowledge about the research taking place at Lawrence Berkeley National Laboratory (LBNL) in Berkeley, California he initiated the cooperation for this thesis. The support through the Fraunhofer scholarship gave me the security to complete my study.

I want to thank Erich Strohmaier, head of LBNL's Future Technology Group (FTG) and John Shalf, head of the Advanced Technologies Group (ATG). I highly benefited from their support and guidance. Especially, I need to thank John Shalf and the great Green Flash team who made me part of the exciting project back in 2009 at my first visit to the LBNL and their support for the Green Wave project later on. I know that John's time is very valuable and appreciate every minute he spent on discussions with me.

Special thanks go to Sam Williams and David Donofrio. I greatly benefited from Sam's incredible experience in software optimization and compute architectures. Only with David's help and expertise in hardware architecture I was able to apply the gained experience from the application analysis to the Green Wave design. He introduced me to the simulation tools and offered a lot of his valuable time for discussions and problem solving. This study couldn't be done without him.

My thank goes to all researches at the Fraunhofer ITWM who advised me in seismic questions. Special thanks go to Norman Ettrich, Daniel Gruenewald. Both spent quite some time with me discussing seismic processing and wave-equation kernels.

I would like to thank Paulius Micikevicius from NVIDIA for his benchmarking and participation. He enabled me to compare my own benchmark results to the fastest wave-propagation kernels on leading-edge GPU hardware.

I want to thank everyone I had the honor to sit in the same room with. I would like to mention Filip Blagojevic and Khaled Ibrahim. I will always remember the advice you gave me and the lively discussions we had.

Finally, I would like to thank all my friends and family who supported and believed in me even when I didn't believe in myself.

Thank you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past the science community greatly benefited from increasing computational performance through advances in compute architectures. New central processing units (CPUs) with higher clock frequencies could simply replace old CPUs to increase application performance. In the consumer market only single-core CPUs were present. Therefore, programming models and software were optimized towards a sequential workflow. In the early 2000's Intel still predicted a 10 GHz processor to be available in 2005 [48]. In the mid 2000's the hardware manufacturing industry hit an upper ceiling, above they were not able to increase clock rates any further. Due to physical limitations, hardware vendors had to perform a paradigm switch to keep processor performance improvements up with Moore's law [59]. Instead of rising clock rates performance improvements were now achieved by exploiting explicit parallelism placing multiple cores on a single socket [47].
The diagram presented in Figure 1.1 shows the flattening of the clock speeds but still rising transistor counts due to smaller silicon feature sizes and multiple cores.

Contrary to the consumer market, the scientific, high-performance computing (HPC) community has used highly parallel systems for some time. First HPC systems exploited performance by increasing the amount of instruction units of the CPU leveraging multiple co-floating-point instruction units like the Solomon supercomputer [88]. This design used 256 *Processing Elements* (PE) of which each could perform a floating-point instruction per cycle. Many designs leveraged this type of vector processing to exploit performance. The downside was that only a limited number of embarrassingly parallel applications were able exploit peak performance on these architectures. In 1976 Sey-

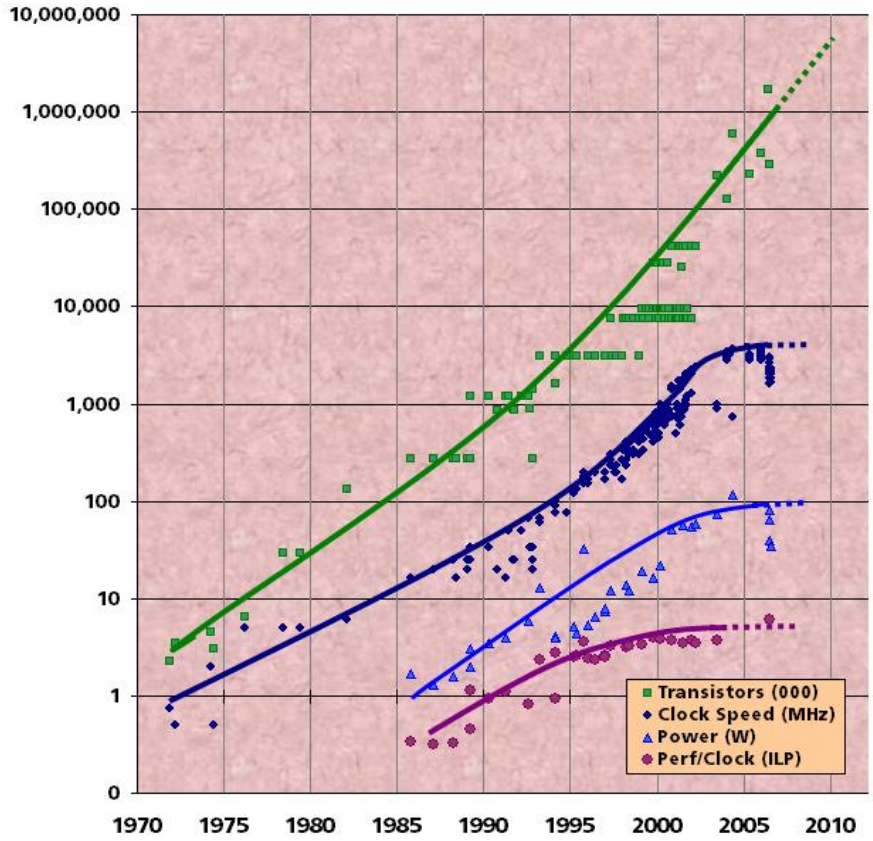


Figure 1.1: Development of Transistor Count. [Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith.]

mour Cray introduced the Cray-1 supercomputer [75]. Different to vector processors, Cray-1 used a single large and complex CPU with instruction pipelining. This way the architecture could fit many different applications. Data parallelism was exploited by connecting many of these CPUs. Cray-1 was the first step of HPC to use general-purpose CPUs in supercomputer clusters.

With the arrival of multi-core general-purpose CPUs the number of cores per cluster hit and passed the number of 200,000 in 2007 [100]. Taking a look at the Top500 [101] list of the last decades it becomes clear that supercomputer growth in performance is even surpassing Moore's Law. In recent years high performance cluster were facing the peta-flop milestone.

Future large-scale systems differ from today's systems in one very important point.

Figure 1.2 shows an extrapolation of HPC systems power consumption under consideration of technology advances how they appeared in the past. Concluding an exascale supercomputer would consume about 200 MW of power [46]. With a price of 10 cents per kW/h this would lead to annual power costs of about US $200M.

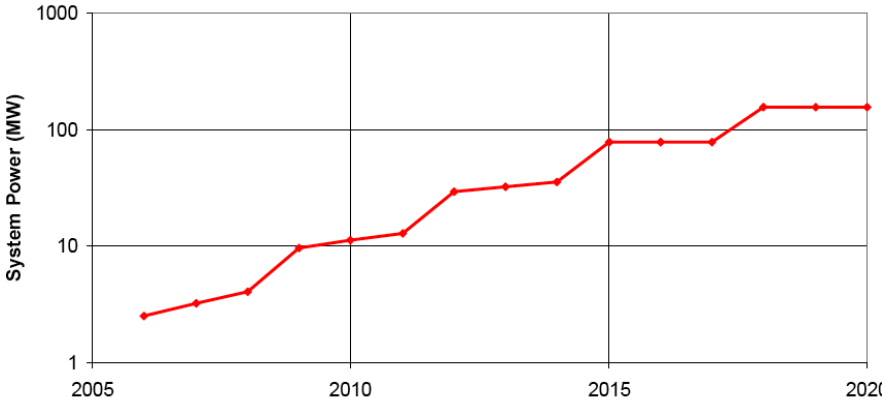


Figure 1.2: Extrapolated Power Consumption of future HPC Systems.[46]

One can see that power consumption is going to be main factor for slowing HPC system performance growth in future. But power consumption is already critical for today's large-scale systems. For many years Top500.org ranks the most powerful HPC systems by billion floating-point operations per second (Gflops/s) running the LINPACK benchmark but does not consider power consumption. For this purpose the Green500 list was introduced in 2006 [82]. It ranks the top 500 most energy efficient high performance compute clusters by Gflops/s per Watt. As for Top500 the amount of Gflops/s is benchmarked with the LINPACK benchmark. Power consumption estimations include power required for CPUs and cooling.

To keep future large-scale compute system in feasible limits in terms of total costs of ownership (TCO) the United States Department of Energy (DoE) officially declared a target operational power consumption of 20MW for exascale system achieved by 2018. Given this energy efficiency challenge the computing industry must discover new ways how to build future supercomputing clusters and another paradigm change is inevitable, again.

This is why private organizations like Intel, AMD and governmental organizations like

DARPA (Defense Advanced Research Projects Agency) in the USA and DFG (Deutsche Forschungsgesellschaft) cooperate with researchers all over the world to point out ways how such an exascale system could look like. Current exascale studies expecting more than 100 million cores, petabytes of main memory and exabytes of hard disk space [1], [46]. But not only new hardware designs have to be developed. One of the main questions is what kind of applications could leverage such high parallel system.
Past approaches focused on developing new hardware architectures and let programmers deal with adjusting software to fit the hardware requirements. The most critical optimization on modern multi-core and many-core architectures are the distribution and parallelization of work. Parallelization in general describes the parallel execution of two or more independent parts of data. The optimal speed-up from parallelization would be a linear reduction of execution time according to the number of parallel processes. Unfortunately, perfect strong-scaling can be achieved only, if no sequential parts exist within the program and all parts can be run completely independent from each other. The maximal achievable speed-up is defined as a relation between the sequential and parallel parts. Amdahl's law [2] estimates the maximum speedup S as:

$$S = \frac{1}{\alpha}$$

where α is the fraction of time spend in the sequential part of the problem. Even if the parallelized part of the program converges towards zero the total runtime is dominated by the sequential part. Therefore the maximum number of processes reasonable working on an application is limited. Having only a limited number of algorithms (e.g. climate simulations, astro physics, seismic) that would be able to leverage exascale parallelism the current approach on system design has to be completely reconsidered. Rather than asking how software could fit the underlying hardware, one should ask how to design a supercomputer that fits the needs of such highly parallel applications. In 2008 John Shalf and his colleagues from Lawrence Berkeley National Laboratory and NERSC introduced the so-called hardware/software co-design methodology to HPC system design. The design methodology is already applied in the mobile and embedded space for a long time and serves well for optimizing performance and energy efficiency. The co-design approach analyzes the requirements of the application and tailors the hardware architecture towards it. This results in a completely balanced system that provides a maximum of energy efficiency for a certain class of application but without

the loss of generality like in full custom designs.
This approach was first applied by the "Anton" supercomputer focused on increasing performance in molecular dynamics [84] and the "Green Flash" project that focused on energy efficient, cloud-resolving climate simulation. Extensive exploration has shown this to be an effective methodology that has the potential for great energy efficiency [21, 107].
As in climate prediction, the seismic community faces huge datasets and time-consuming algorithms to create subsurface images necessary to drive drill decisions. Within the seismic workflow, seismic migration is the most computational intense part that consumes more than 90% of the total computation time. And its significance will even increase with more physically correct modeling in future.
A common migration method is Reverse Time Migration (RTM). Most commonly, RTM uses either isotropic (ISO), vertical transverse isotropic (VTI) or tilted transverse anisotropic (TTI) wave propagation kernels, which serve the needs for high quality imaging. But, as for high-resolution climate codes, common on-market systems cannot address the performance needs that are required to receive RTM migrated subsurface images within a reasonable time and power budget for large survey sizes.
The seismic industry has strong interest in exploring larger areas of the subsurface with a single survey. Larger exploration areas increase the chance of finding oil field and lead to better subsurface image quality to drive drill decision on, which leads directly to money savings. In addition to finer scaling and higher physical correctness, this requires the development of even more computational intense applications then applied today - e.g. the elastic wave equation adding about 100x in complexity. Hence, computational demands are increasing constantly. Up to date imaging average sized $(10 - 20km^2)$ areas require in the order of weeks to month depending on the requested image quality.
This thesis analyzes the requirements of three leading seismic imaging kernels and compares the performance and energy efficiency of leading on-market architectures. Based on the requirements of these kernels, three optimized architectures are derived with the hardware/software co-design approach and their energy efficiency and performance is compared to the evaluated architectures.

The main contributions made by this thesis are:

- the analysis of the requirements of ISO, VTI and TTI seismic wave propagation RTM kernels on hardware,

- the exploration of the potential for single- and multi-node software optimizations on these kernels, for on-market and next generation computer architectures,
- the benchmark comparison of the performance and power efficiency of commercial off-the-shelf architectures.
- the introduction of the Green Wave architecture designed with the hardware/-software co-design approach and tools from the mobile market.
- the introduction of a programming model for Green Wave to hide communication latencies.

1.1 Outline

The remaining work is organized as follows: Chapter 2 gives a general introduction into seismic processing and points out the importance of seismic migration for subsurface imaging. It then introduces one of the most popular migration methods called Reverse Time Migration (RTM) and its different implementations. Three implementations are picked out (ISO, VTI, TTI) as reference RTM wave propagation kernels and their requirements on the underlying hardware are analyzed.
To give an overview of state-of-the art architectures, used in the seismic industry for seismic migration, Chapter 3 introduces different hardware architectures. The ISO, VTI and TTI kernel reference implementations are compared in Chapter 4 in a single-node environment.
Next, Chapter 5 analyzes the potential for further software optimization to achieve a maximum performance and energy efficiency in order to provide a fair comparison between the evaluated architectures. Additional to the single-node benchmarks different multi-node implementations are compared to each other and benchmarked. Finally, Chapter 5 estimates the power consumption of clusters build upon the evaluated node setups running ISO, VTI and TTI kernels.
The hardware/software co-design methodology and past work important for this thesis is introduced in Chapter 6.
Chapter 7 first introduces a programming model to maximize data locality and minimize off-chip data transfers. Further, it derives equations to map kernel requirements to architectural features.

Based on these equations, Chapter 8 presents three different Green Wave architectures optimized for ISO, VTI and TTI kernels, and presents a single-node benchmark. Chapter 9, adds a multi-node analysis and an estimation how the energy efficiency of a Green Wave cluster would compare to cluster setups based on the evaluated COTS architectures.
Finally, Chapter 10 draws conclusions and gives an outlook on future work.

Chapter 2

The Basics of Seismic Processing

This chapter introduces different kinds of seismic processing and gives a short overview of the seismic processing workflow in seismic exploration. It describes the different stages to give an idea what steps have to be taken to receive an accurate subsurface image.

In different fields like petrology, geothermal sciences, geologic sciences and geophysics it is often useful to create images of the subsurface to analyze an area in terms of material variations of the earth interior. Those are useful for earth crust development analysis and exploration of different substances and minerals. For the creation of such a subsurface image it is, in first hand, required to collect seismic data of the area. One may chose between several different methods [9]. Most common is the active data collecting. Therefore, a source energy, like an explosion, is generating acoustic waves that are reflected by impedance contrast between rock layers and received by multiple e.g. geophones along several lines. For up to about 12 seconds those receivers listen to reflections and record arriving amplitudes strengths. After each "shot" the whole system is moved in equidistant offsets until the area of interest is covered. After data acquisition the data is filtered to suppress all energy except first-order reflections, and resorted to the SEG-Y industry standard format. The computational most intense part of the seismic processing workflow is the seismic migration of the data. Migration is used to move recorded data to its true subsurface reflector positions and to cross-correlate the down-going source and up-going receiver wavefield to receive the correct subsurface image.

This thesis uses the coordinate system as it is commonly used in the seismic industry as presented in Figure 2.1 . Whereas, the x direction is referred to as "inline", y as crossline direction and z as depth.

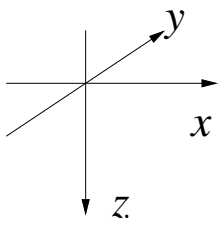


Figure 2.1: The Seismic Coordinate System

2.1 Fundamentals of Seismic Processing

In general three types of seismic applications can be differentiated [112]:

1. **engineering seismology:** down to 1km; research for buildings like bridges or highways and for exploration of minerals and charcoal.

2. **exploration seismology:** down to 10km; searching for gas and oil.

3. **earthquake seismology:** down to 100km; seismology for global earth events like earthquakes, volcano eruption and similar. An example is the tsunami forecast for the Pacific Ocean.

All types use acoustic waves that are propagated through the area of interest to reconstruct the earths' interior. The seismic impulse can be created by using e.g. an explosion, airgun or sledgehammer. The resulting acoustic waves are reflected at places of impedance contrasts. Impedance contrasts are rapid changes in the medium velocity or density in the earths interior. The reflections are recorded by receivers like geophones or hydrophones on the ocean surface for marine seismic acquisitions or boreholes for land seismic. Geophones are most commonly used as receivers in land seismic. They are able to record all three components of displacement of the propagating pressure (P) or shear (S) waves. Hydrophones are preferred in marine seismology, but are restricted to P-waves only. S-waves can be recorded by using so-called *Ocean Bottom Cable* (OBC) setups (for further information see [112]). The direction in which the exploration setup

moves is called *inline-direction* - the perpendicular one: *crossline-direction*. A common setup for narrow-azimuth ocean exploration seismology are six or more parallel receiver-cables (streamers) and two sources towed by a single ship. Each cable can be up to 10 km long with receiver offsets of e.g. 12.5m. The cable separation distance varies from 25m to 100m or more. The actual acquisition setup depends on the subsurface structure and quality demands. If complex subsurface structures are present multiple ships are required. Usually a couple of source injecting ships are patrolled by ships recording the acoustic reflections. By covering a wider area more reflections are recorded, which enables higher accuracy in imaging. Such acquisition setups are e.g. wide-azimuth, full azimuth, multi-azimuth and variations of those.

2.1.1 Seismic Processing Workflow

Even though this thesis focuses on seismic migration methods it is important to understand which role seismic migration plays within the complete seismic processing workflow. This section provides a brief introduction into each processing step.
The following list was partly taken from [112] and gives an overview over the seismic processing workflow:

1. Preprocessing:
    - Demultiplexing
    - Trace editing
    - Geometric spreading correction
    - Setup field geometry
2. Deconvolution, Trace Balancing
3. Common Midpoint (CMP) sorting
4. Normal Moveout Correction (NMO)
5. Velocity analysis
6. Migration
7. Stacking

The creation of subsurface images is an iterative process. Velocity analysis, migration and stacking have to be done multiple times until the desired result is achieved. The number of iterations depends on the complexity of the subsurface structures, acquisition setup and other factors.
The following sections provide a brief introduction into the most important steps of the seismic workflow.

2.1.1.1 Preprocessing

Seismic preprocessing consists of several parts to initially prepare the recorded data. The data recorded by the geophones at the surface is received row-wise. *SEG-Y* as international standard format for seismic data expects traces to be stored in columns. As first step in preprocessing demultiplexing transposes the data. Step two is called trace editing. When recording the wavefield not only clean reflections are received. Scatterers, ground roll, swell- and cable-noise are only a few examples for noise that accrue. Highly noisy traces are filtered out or noise is tried to be attenuated by using for example band-pass and dip-filters. Frequency filtering can be done by defining an amplitude spectrum for the filter and multipling it with the amplitude spectrum of the input seismic trace. With a high-cut amplitude spectrum, smaller separated reflectors can be imaged better. With low-cut frequencies wider separated reflectors improve. Additionally high frequencies are absorbed along the propagation path. It is therefore not possible to image deep subsurface layers with only high frequencies. [112] describes this as follows:

> Just having low or high frequencies does not improve temporal resolution.
> Both low and high frequencies are needed to increase temporal resolution.

Amplitudes decrease at deeper parts of the records. To regain those amplitudes are corrected by a geometric spreading function.
The most important part of preprocessing is merging of the field geometry with recorded seismic data. Through crossline currents cable feathering appears. Cable feathering describes the off-drift of the receiver cables from in-line direction. Therefore the exact coordinates and offsets are corrected and written into the file header, which is then merged with the seismic data.

2.1.1.2 Deconvolution

Deconvolution can be used for improving the temporal resolution by compressing the source wavelet to a spike along the time axis and widening of the amplitude spectrum. It further is applicable to attenuating of multiples and reverberating wavetrains, and therefore improves the signal to noise ratio.

2.1.1.3 Trace Merging

The original coordinates of the seismic data acquisition is in source-receiver format (s, g), given s as source offset and g as offset of the first receiver. Each seismogram or trace position is defined in a three component vector where x, y are the horizontal and z the vertical axis. Since $s = (x_s, y_s, z_s)$ describes the source, $g = (x_g, y_g, z_g)$ the receiver location and $m = (g + s)/2$ the midpoint (see Figure 2.2) for each trace.

For the ease of data processing the source and receiver coordinates are merged together based on the field geometry information in the header. In the case of 2D seismic processing all traces of one shot are assigned to a *Common-Midpoint* (CMP) . CMP gathers all traces based on a midpoint between source and first receiver (see Figure 2.3). *Common-Depth-Point* (CDP) is the same as CMP for horizontal reflectors. Even those midpoints are not the same for dipping reflectors both types are often used interchangeably.

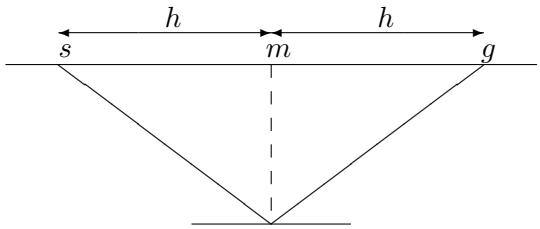


Figure 2.2: Source-receiver coordinates scheme: s as source signal, g as receiver, m is the midpoint and h the offset [40]

Another sorting technique is Common-Shot-Gather. It sorts all traces to the related shot position and saves them consecutively after each other (see Figure 2.4).

In processing of 3 dimensional data this step is called the *binning* of traces. Binning describes checker-boarding of the analyzed area. All traces, located in a square or bin, are merged together.

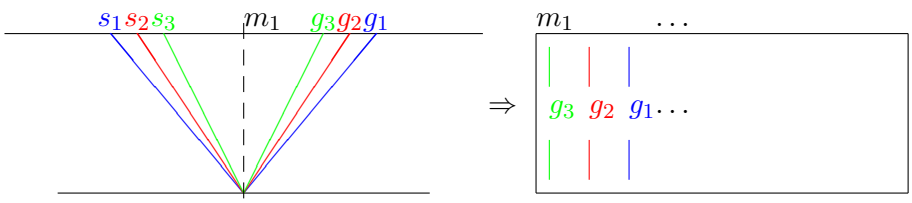


Figure 2.3: Common-Midpoint-Gather sorting [40]

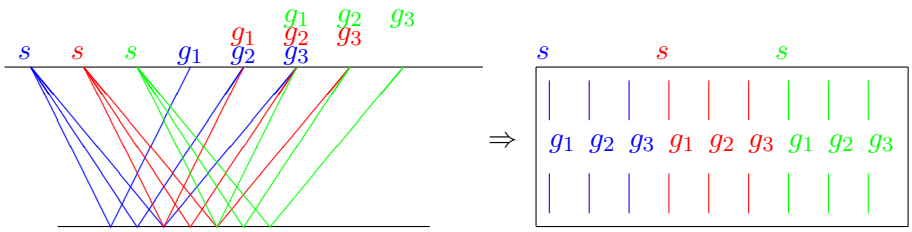


Figure 2.4: Trace to Common-Shot-Gather data sorting [40]

If for each source only one receiver exists with same coordinates it is called "zero-offset" data. Traces like presented in Figures 2.3 and 2.4 is called "prestack" data.

2.1.1.4 NMO

NMO stands for *Normal-Moveout Correction* and describes the removal of the move-out-effect of hyperbolic trajectories. This filter uses the wave travel-times taken from the velocity field to correct each trace. After summing all traces to form a (stretched) CMP-gather, the travel-times might not be valid anymore due to strong lateral velocity variation between the different traces. In this case prestack Migration has to be applied.

2.1.1.5 CMP / CCP Stacking

Stacking greatly reduces the amount of data that has to be processed and therefore execution time. CMP gathers offset stacking means averaging all traces over offsets. [9] Different stacking methods are presented by Figure 2.5.

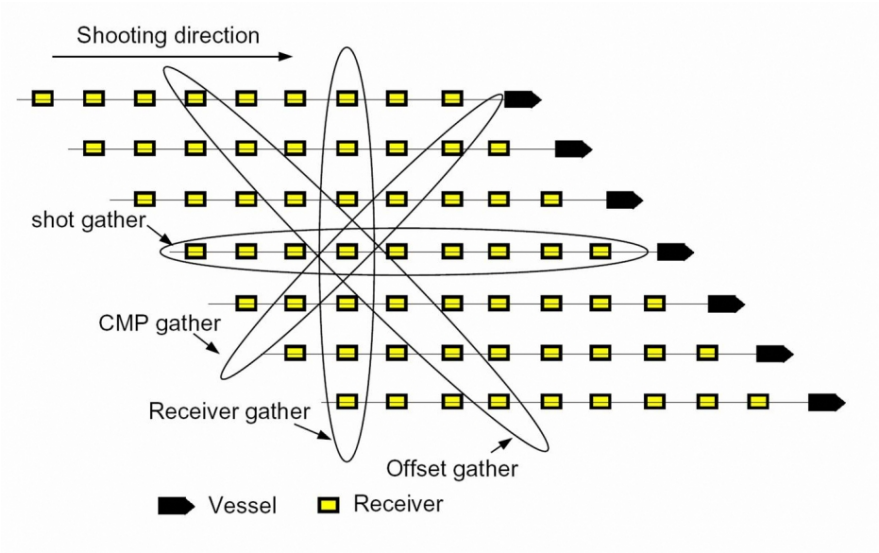


Figure 2.5: Data Gathering Methods [http://petroleumgeophysics.com]

2.1.1.6 Velocity analysis

Due to velocity sensitive imaging algorithm, an important part in imaging subsurface reflectors to their real positions is the creation of an accurate velocity field. Velocity field creation is a complex topic and is just briefly mentioned here.
In general this creation process is done iteratively. The first step is to roughly estimate the acoustic impedance contrasts for different depths of selected CMP gathers. Usually such estimates are made from rock analysis of example drills.
The received velocity spectra are then interpolated for receiving a first coarse velocity field of the subsurface. The knowledge of coarse subsurface velocities complements the information of seismic data. By applying filters and prestack migration methods (see Section 2.2) to the seismic data, the velocity field can be refined by coherency estimation through inversion [9].

$$C, F \rightarrow U$$

$$U, C, F \rightarrow \tilde{F}$$

$$\| F - \tilde{F} \| \rightarrow min$$

Figure 2.6 shows the smooth velocity field of the synthetic 2D Marmousi data set [104]. Smoothed velocity fields suppress internal reflections.

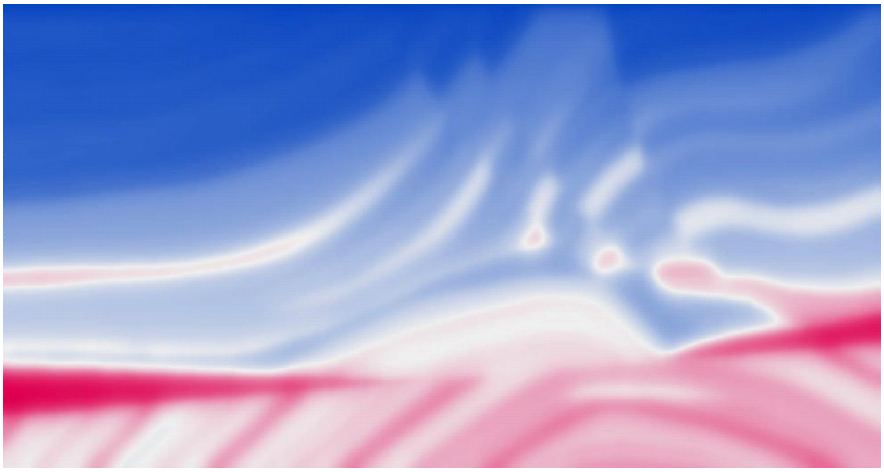


Figure 2.6: The Smooth Velocity Field of the Synthetic Marmousi Data Set

2.2 Seismic Depth Migration

Seismic Depth Migration is a very important step in the seismic processing workflow. With the received data, migration analyzes the amplitude changes and interprets them as reflectors at different depth levels. Dipping events are moved to their true subsurface location, diffractions are collapsed and a final subsurface image is created.
Migration can be done in either 2D or 3D fashion. Nowadays, subsurface imaging is done almost exclusively in 3D that provides higher accuracy compared to 2D imaging of the explored area.
Figure 2.7 presents some of the choices to be considered when it comes to decide which seismic migration method should be used. This section gives an overview of the different decision steps and explained why this thesis is primarily focusing on explicit solutions of the Reverse Time Migration (RTM). Decisions that have to be made for such a solution are highlighted in red and discussed in more detail in later sections. Other migration methods are only briefly mentioned.

All kinds of migration are using the equation of motion to simulate the wave behavior in the subsurface. This equation can be approximated by either using integral methods, e.g. using the Eikonal equation for ray-based methods like the Kirchhoff Migration or a partial differential equation (PDEs) scheme is applied and approximated. In the past forward modeling was done using the one-way wave equation, which delivers

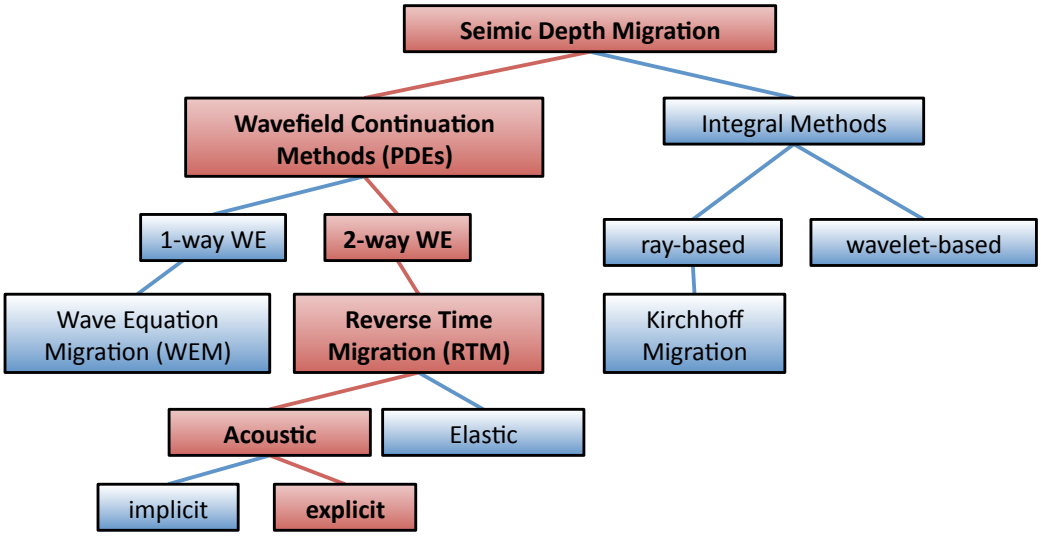


Figure 2.7: An Subset of Methods for Seismic Depth Migration

only limited accuracy. Today, two-way wave equations are the industry standard and are modeled in either frequency, time or depth domain. Dependent which domain is chosen either e.g. FFTs or a finite-difference (FD) scheme can be applied. Reverse Time Migration (RTM) is a migration method that is able to model all kind of waves. It uses wavefield propagation in time domain.

Next, a choice has to be made whether to use the elastic or the much less complex acoustic wave equation to propagate the wavefield. Fully elastic implementations account for all 21 elastic properties. This makes it difficult to apply and requires a tremendous amount of computational power. Hence, the seismic industry uses the acoustic wave equation to propagate the wavefield through the subsurface (see Section 2.2.1).

Depending on the anisotropic properties of the subsurface rock layers the wave equation needs to account for isotropy, vertical transverse isotropy (VTI) or tilted transverse isotropy (TTI). The isotropic wave equation accounts for pressure-wave velocity as the only earth property. For transverse isotropy like VTI, anisotropic behavior is considered symmetric to the z-axis. TTI additionally accounts for the dip of the medium.

Such wave equation implementations can be solved in either implicit or explicit fashion. Whereas implicit implementations are using linear systems to derive amplitudes for the next step in time, explicit implementations create so-called stencil schemes. For each grid point in the volume such stencil calculations are independent from neighboring

stencil computations, which makes parallelization for large compute clusters easier to implement. Explicit finite-difference approximations of the wave equation are therefore commonly used in the seismic community.

2.2.1 The Wave Equation

The fundamental base for the wave equation is Newton's law of motion. It considers full elasticity and anisotropy in inhomogeneous medium. Elasticity describes the amount of deformation if it is exposed to stress through external forces. It can be described by the linear stress-strain relation. Anisotropy is described by [85] by

> [The] variation of seismic velocity depending on the direction on which it is measured.

The law of motion is described by

$$\rho(x)\frac{\partial^2 u_i}{\partial t^2}(x,t) = \frac{\partial \sigma_{ij}}{\partial x_j}(x,t) + f_i(x,t) \tag{2.1}$$

where ρ describes the density, σ_{ij} the stress field, u_i the particle displacement and f_i the body force density, and σ_{ij} being a linear stress-strain relation of the form

$$\sigma_{ij}(x,t) = c_{ijkl}(x,t)\epsilon_{kl}(x,t) \tag{2.2}$$

C_{ijkl} is the stiffness tensor which forms the elastic matrix shown including all properties in Equation 2.3.

$$C_{ijkl} = C_{\alpha\beta} = \begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{12} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{13} & C_{23} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{14} & C_{24} & C_{34} & C_{44} & C_{45} & C_{46} \\ C_{15} & C_{25} & C_{35} & C_{45} & C_{55} & C_{56} \\ C_{16} & C_{26} & C_{36} & C_{46} & C_{56} & C_{66} \end{pmatrix} \tag{2.3}$$

where index conversion is done by

$$\begin{array}{cccccc} 11 & 22 & 33 & 23,32 & 13,31 & 12,21 \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ 1 & 2 & 3 & 4 & 5 & 6 \end{array} \tag{2.4}$$

for simpler notation.

The strain tensor ϵ_{ij} is defined by

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.5}$$

The combination of Equation 2.1 and Equation 2.5 forms the wave equation for particle displacement in an elastic, anisotropic, and inhomogeneous medium:

$$\rho\frac{\partial^2 u_i}{\partial t^2}(x,t) = \frac{\partial}{\partial x_j}c_{ijkl}(x,t)\frac{\partial u_k}{\partial x_l}(x,t) + f_i(x,t) \tag{2.6}$$

In 2.6 the Latin indexes are summed according to Einstein's sum convention.

In an elastic, homogeneous, anisotropic medium the elastic-dynamic wave equation of particle displacement becomes:

$$\rho\frac{\partial^2 u_i}{\partial t^2} = \frac{\partial}{\partial x_j}\left(c_{ijkl}\frac{\partial u_k}{\partial x_l} + f_i\right) \tag{2.7}$$

where u describes the particle system state, t the time, c the Lame parameter and k the number of dimensions. For transverse isotropy the 21 elastic properties are reduced to five independent properties as presented by matrix 2.8.

$$C_{\alpha\beta} = \begin{pmatrix} C_{11} & C_{11}-2C_{66} & C_{13} & 0 & 0 & 0 \\ & C_{11} & C_{13} & 0 & 0 & 0 \\ & & C_{33} & 0 & 0 & 0 \\ & & & C_{44} & 0 & 0 \\ & & & & C_{44} & 0 \\ & & & & & C_{66} \end{pmatrix} \tag{2.8}$$

L.Thomson substitutes the $C_{\alpha\beta}$ from 2.8 to derive the so-called Thomson parameter [98] which enable accounting for weak anisotropy in acoustic media that appears in most marine sediments [106]. The five Thomson parameters are defined as

$$
\begin{aligned}
V_p &\equiv \sqrt{\frac{C_{33}}{p}} \\
V_s &\equiv \sqrt{\frac{C_{55}}{p}} \\
\epsilon &\equiv \frac{C_{11} - C_{33}}{2C_{33}} \\
\delta &\equiv \frac{(C_{13} + C_{55})^2(-C_{33} - C_{55})^2}{2C_{33}(C_{33} - C_{55})} \\
\gamma &\equiv \frac{C_{66} - C_{44}}{2C_{44}}
\end{aligned}
$$

where ϵ measures the difference between vertical and horizontal P velocities and can be described as P-wave anisotropy, γ the amount of S-wave anisotropy and δ the near-vertical anisotropy. V_p describes the pressure wave velocity and V_s the shear wave velocity, respectively. The Thomson parameters are physically given for a certain medium.
Commonly used are three implementations: 1) the isotropic 2) vertical transverse isotropic and 3) the tilted transverse isotropic wave equation.

2.2.2 The Isotropic Wave Equation (ISO)

The acoustic isotropic case has only the pressure field velocity as elastic property and assumes an isotropic propagation along the all space axis. This simplifies the wave equation in three dimensions to:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) u \tag{2.9}$$

with c as velocity, t as time and u as pressure wavefield.

The isotropic wave equation has been extensively used in the seismic industry in the past. It is a well-understood propagation method and provides minimal computational requirements, which was critical due to lack of high performance computing resources. Unfortunately, not all subsurface materials behave strictly isotropic and are therefore not modeled accurately by the isotropic wave equation. Hence, the seismic industry is eager to create more accurate subsurface models accounting for anisotropic properties. Figure 2.8 presents the 3D isotropic wave propagation in a 2D $x - z$ plane (left) and

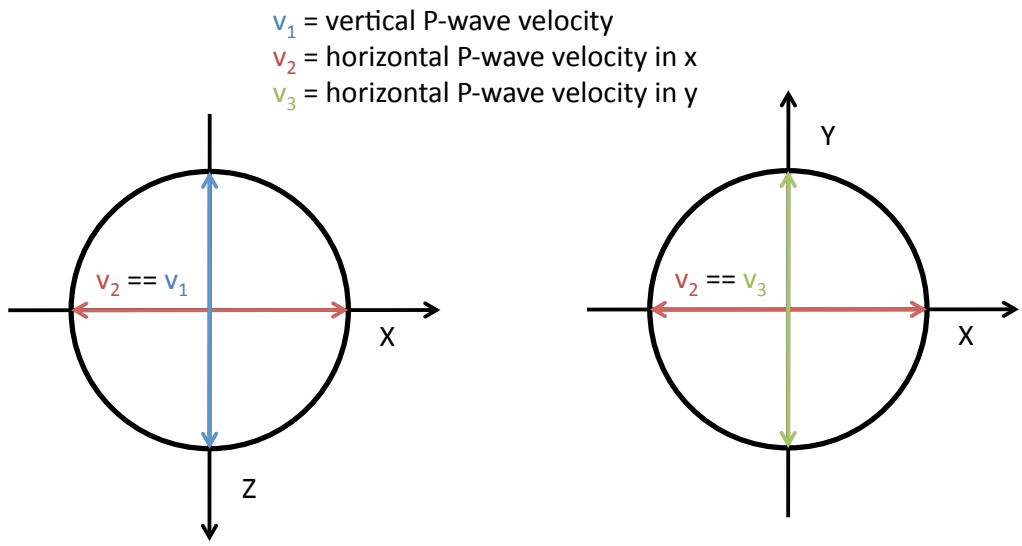


Figure 2.8: Isotropic Wave Propagation

for an $x - y$ plane (right). Only one pressure wavefield velocity exists for all axis $v_1 = v_2 = v_3$.

2.2.3 Vertical Transverse Isotropy (VTI)

Vertical Transverse Isotropy (VTI) accounts for anisotropy along the axis, which can lead to an ellipse shape of the wave. Based on the anisotropic properties of the medium Thomson parameter ϵ and δ describes the difference in velocity along the horizontal axis compared to the vertical pressure velocity. An example PDE for VTI is presented by [23] :

$$\begin{aligned}
\frac{\partial^2 p}{\partial t^2} &= v_{px}^2(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}) + v_{pz} v_{pn} \frac{\partial^2 q}{\partial z^2} \\
\frac{\partial^2 q}{\partial t^2} &= v_{pz} v_{pn}(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}) + v_{pz}^2 \frac{\partial^2 q}{\partial z^2}
\end{aligned}$$

where v_{pz} is the vertical P-wave velocity, $v_p x = v_{pz}\sqrt{1+2\epsilon}$ is the horizontal P-wave velocity and $v_p n = v_{pz}\sqrt{1+2\delta}$ is the P-wave moveout velocity. VTI adds an auxiliary wavefield Q.

Figure 2.9 shows 3D VTI wave propagation in the $x - z$ plane (left) and in a $x - y$ plane (right). Only one pressure wavefield velocity exists v_1. Thomson parameter ϵ

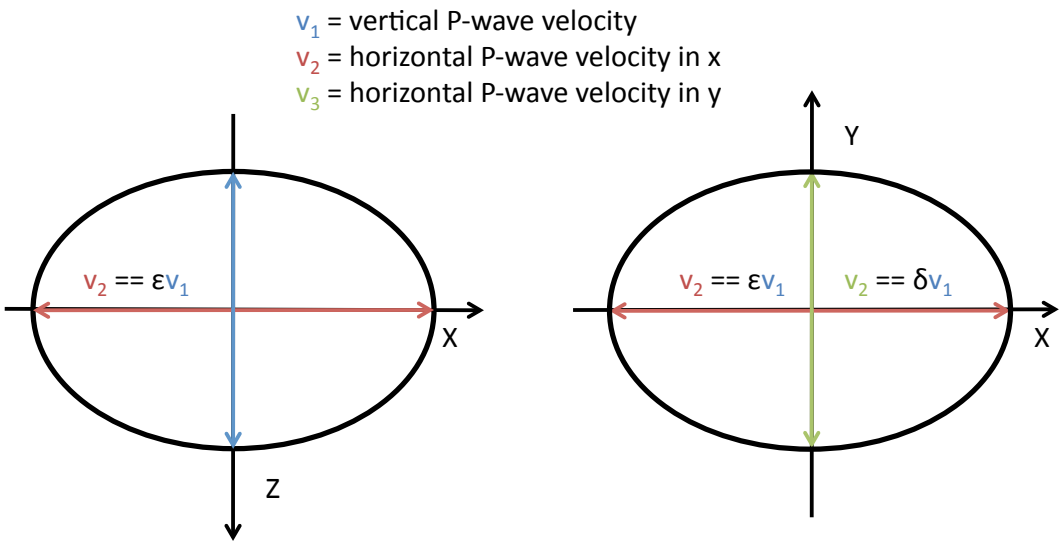


Figure 2.9: Vertical Transverse Isotropic Wave Propagation

describes the difference in x-direction $v_2 = \epsilon v_1$ and δ the adjustment in y $v_3 = \delta v_1$.

As long as the medium is horizontal VTI produces accurate results. For tilted rock layers tilted transverse isotropy must be used.

2.2.4 Tilted Transverse Isotropy (TTI)

For anisotropic and dipping subsurface layers the implementation for TTI media should be used to receive accurate propagation results. TTI implementations consider the angle of a given medium corresponding to the horizontal axis. In a 3D case the azimuth describes the deflection along the y-axis. Chu et al. presented at SEG'10 a PDE for such media [13]:

$$\begin{aligned}
\frac{\partial^2 p}{\partial t^2} &= v_{px}^2 (\frac{\partial^2 p}{\partial \hat{x}^2} + \frac{\partial^2 p}{\partial \hat{y}^2}) + v_{pz} v_{pn} \frac{\partial^2 q}{\partial \hat{z}^2} \\
\frac{\partial^2 q}{\partial t^2} &= v_{pz} v_{pn} (\frac{\partial^2 p}{\partial \hat{x}^2} + \frac{\partial^2 p}{\partial \hat{y}^2}) + v_{pz}^2 \frac{\partial^2 q}{\partial \hat{z}^2}
\end{aligned}$$

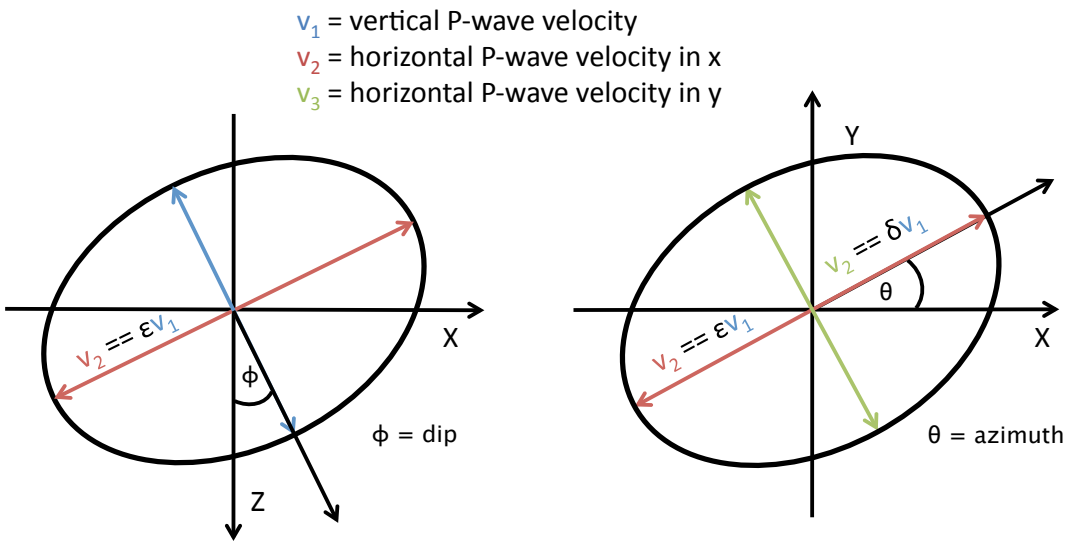


Figure 2.10: Tilted Transverse Isotropic Wave Propagation

where

$$
\begin{aligned}
\frac{\partial^2}{\partial \hat{x}^2} &= \frac{\partial^2}{\partial x^2} cos^2\theta cos^2\phi + \frac{\partial^2}{\partial y^2} cos^2\theta sin^2\phi + \frac{\partial^2}{\partial z^2} sin^2\theta + \\
&\quad \frac{\partial^2}{\partial x \partial y} cos^2\theta sin2\phi + \frac{\partial^2}{\partial x \partial z} sin2\theta cos\phi + \frac{\partial^2}{\partial y \partial z} sin2\theta sin\phi \\
\frac{\partial^2}{\partial \hat{y}^2} &= \frac{\partial^2}{\partial x^2} sin^2\phi + \frac{\partial^2}{\partial y^2} cos^2\phi + \frac{\partial^2}{\partial z^2} sin2\phi \\
\frac{\partial^2}{\partial \hat{z}^2} &= \frac{\partial^2}{\partial x^2} sin^2\theta cos^2\phi + \frac{\partial^2}{\partial y^2} sin^2\theta sin^2\phi + \frac{\partial^2}{\partial z^2} cos^2\theta + \\
&\quad \frac{\partial^2}{\partial x \partial y} sin^2\theta sin2\phi - \frac{\partial^2}{\partial x \partial z} sin2\theta cos\phi - \frac{\partial^2}{\partial y \partial z} sin2\theta sin\phi
\end{aligned}
$$

Figure 2.10 shows 3D TTI wave propagation in the $x - z$ plane (left) and in a $x - y$ plane (right). As for VTI the velocity variation is transverse to the waves midpoint and received by applying Thomson parameters ϵ and δ to the pressure wave velocity v_1. Additionally, the dip or θ describes the deflection in the $x - z$ plane and azimuth the angle deflection in the $x - y$ plane.

2.3 Approximation of the Acoustic 2-way Wave Equation

One-way, two-way and non-reflecting wave equations can be approximated with either partial differential equations (PDEs) or integral methods. Integral methods are further differentiated into ray-based methods like used in Gaussian Beam, GRT or Kirchhoff

migration or in wavelet based methods (see Section 2.3.1 for more details).
Most commonly used are PDEs applying finite-differences for approximating the derivatives of the Laplacian operator.

2.3.1 Integral Methods

Integral methods approximate the wave equation using ray-tracing techniques. In prestack migration each source-receiver pair (traces) is regarded separately for all travel times. Such methods allow many adjustments and high parallel computation.
The widely used Kirchhoff migration gives an example for such an integral method.

Kirchhoff Migration is the most common ray-based, integral or summation based method. It uses the principle of rays traveling through the subsurface and being reflected on layers of impedance contrast.
Imaging using the Kirchhoff migration is separated into several parts. First of all the Eikonal equation is implemented to create travel-time tables based on velocity information. Such a travel time table exists for all source positions and describes the time a wave needs from source to a specific grid point.

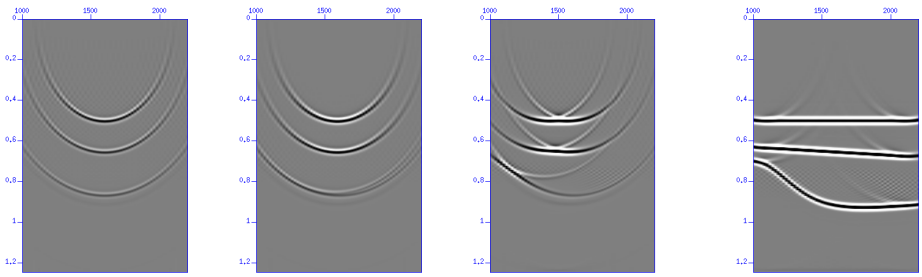
Next, based on the recorded data at a corresponding time τ, for the known source position x_s and receiver position x_r the possible locations of the subsurface reflection point are calculated. Since no single distinct subsurface point x, τ can be derived from that, all possible results define a semicircle[1].
The last step is the constructive summation of all semicircles derived from different source- receiver traces at emphasized reflection surfaces and destruction summation elsewise.

Figures 2.11(a) to 2.11(d) show consecutive increasing numbers of traces for a simple four layer velocity model. The superposition of semicircles is clearly seen.
The Kirchhoff migration methods have several pros and cons:

- [Pros]: Ray-tracing based methods are most often parallelized easily due to independencies between shots, traces, single travel times and subsurface points. Furthermore specific areas can be picked out of the complete survey for higher quality analyses and high adjustability of performance and quality parameters.

[1] A perfect semicircle exists only for homogenous velocity fields

- [Cons]: High frequency approximation limits the image quality in complex areas. Accuracy comparisons have shown that wavefield continuation methods provide, particular in areas where multiple arrivals are required, a superior quality due to no limiting to one particular arrival. Those situations appear below complicated salt bodies. [61]
  High I/O requirements, due to additional data generation during the migration.



(a) Single Trace (b) Few Traces (c) Multiple Traces (d) Fully Imaged Reflectors

Figure 2.11: Ray-Tracing based Imaging

Gaussian Beam *Prestack Gaussian Beam* depth migration (introduced by [38]) advances the Kirchhoff migration method in complex areas with multiple arrivals. Different to Kirchhoff migration it uses local slant stacks of traces with multiple complex valued travel-times and amplitudes tables. The complex values appear from wavefield expression as sum of Gaussian beams [33]. Those approximate the wave equation with a finite-frequency, ray-theoretic solution. Therefore, even Gaussian Beam migration provides more accurate results for complex structures but the additional amount of data required prohibits an efficient implementation as production code.

2.3.2 An example Partial Differential Equation (PDE): The Finite Difference Method (FDM)

The *Finite Difference Method* (FDM) is one way to approximate the wave equation by substitution of the derivatives with difference schemes. Therefore $\mathcal{L}u$ with $\mathcal{L}$ as linear differential operator and $u = u(x)$ as function becomes a discrete expression Λu_h on a finite interval a, b with $a < b$ and grid point offset $h = \frac{b-a}{N}$.

The general second-order differential operator can be expressed as

$$\begin{aligned}\mathcal{L}u &= \sum_{i=1}^{3}\sum_{j=1}^{3} m_{ij}\frac{\partial^2 u}{\partial x_i \partial x_j} \\ &= m_{11}\frac{\partial^2 u}{\partial x^2} + m_{22}\frac{\partial^2 u}{\partial y^2} \\ &+ m_{33}\frac{\partial^2 u}{\partial z^2} + 2m_{12}\frac{\partial^2 u}{\partial x \partial y} \\ &+ 2m_{23}\frac{\partial^2 u}{\partial y \partial z} + 2m_{31}\frac{\partial^2 u}{\partial z \partial x} \end{aligned} \tag{2.10}$$

For the one dimensional case in dependency of time t, the second-order derivative approximated by finite differences becomes:

$$\Lambda_x \sigma(u(x,t)) = \sigma \frac{u(x+h,t) - 2u(x,t) + u(x-h,t)}{h^2} \tag{2.11}$$

with a parameter $\sigma (0 \leq \sigma \leq 1)$.
Attaining Λ for all three dimensions we get a wave equation approximation with

$$\Lambda = \Lambda_x + \Lambda_y + \Lambda_z$$

A general notation for this solution gives

$$\frac{u_{ijk}^{n+1} - 2u_{ijk}^{n} + u_{ijk}^{n-1}}{h^2} = \Lambda(\sigma_1 u_{ijk}^{n+1} - (1 - \sigma_1 - \sigma_2)u_{ijk}^{n} + \sigma_2 u_{ijk}^{n-1}) \tag{2.12}$$

PDEs are discriminated into explicit, implicit and hybrid methods. The choice of σ defines how the linear equation system is solved, which is either explicit ($\sigma_1 = 0$) or implicit ($\sigma_1 \neq 0$).

2.3.3 Implicit Solutions for the Finite Difference Method

Implicit methods for wavefield extrapolation provide high accuracy and stability. Calculating the wave equation in one point of the volume requires solving a linear system with the dimension of the volume. Therefore such methods become very complex for large 3D surveys and require a lot memory and disk access. Parallelization of implicit methods is done by parallel linear solvers. Those decompose the matrix into separate chunks, which are solved on different cores. Figure 2.12 gives an example for a 1D second-order in space and time implicit scheme.

2.3.3.1 Example for an implicit-explicit hybrid Implementation using ADI

For an implicit finite-difference approximation the additive scheme developed by Samarskij, Vabishevich can be used to develop an Alternation Direction Implicit (ADI) scheme [103]. This scheme was used by [41]. This work should serve as an example implementation of an extended implicit solution implementation of the wave equation in a production seismic migration code.

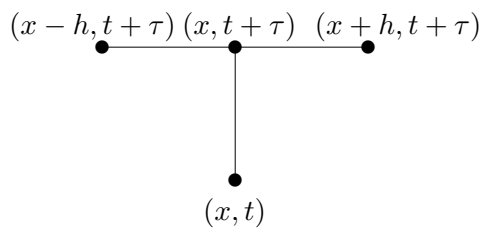


Figure 2.12: Stencil of the implicit first-order Finite-Difference Scheme

Given

$$\frac{\partial^2}{\partial t^2}U - c^2 \triangle U = 0,\ t > 0, x = (x_1, x_2, x_3)^T; \tag{2.13}$$

$$U(0) = u^0; \tag{2.14}$$

$$\frac{\partial}{\partial t}U(0) = v^0. \tag{2.15}$$

we define $-c^2 \triangle U = \mathcal{A}$ with $\triangle$ as Laplace operator $\triangle = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2}$.

The operator $\mathcal{A}$ can be decomposed to

$$\mathcal{A} = \sum_{\alpha=1}^{3} \mathcal{A}^{(\alpha)},\ \alpha = 1, 2, 3 \tag{2.16}$$

where $\mathcal{A}^\alpha = \frac{\partial^2}{\partial x_\alpha^2}, \alpha = 1, 2, 3$ for dimension x, y, z.

For problem (2.13)-(2.15), the following finite-difference approximation of the second-order can be applied for a weight σ and given initial values u^0, u^1

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\tau^2} + \mathcal{A}\left(\sigma u^{n+1} + (1-2\sigma)u^n + \sigma u^{n-1}\right) = 0 \tag{2.17}$$

After the additive scheme is applied the problem description has the form

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\tau^2} + \sum_{\alpha=1}^{p} \left(\mathcal{I} + \mu\mathcal{A}^\alpha\right)^{-1} \mathcal{A}^\alpha u^n = 0 \tag{2.18}$$

where u^n denotes the discretized value of the function U at the timestep $t_n, n = 1, \ldots, N$, respectively. $A^{(\alpha)}$ can be used for the domain decomposition into p domains (usually applied for three-dimensional models) as well as for direction splitting into p directions (usually applied for two-dimensional models)

The finite-difference representation (2.18) contains two parts, i.e. implicit and explicit. In this sequence, Equation (2.18) is to solve

(i) For each $\alpha = 1, \ldots, p$ the linear system of equation is to solve

$$\left(\mathcal{I} + \mu \mathcal{A}^{(\alpha)}\right) w^{(\alpha)} = \mathcal{A}^{(\alpha)} u^n \tag{2.19}$$

(ii) The explicit step is to calculate

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\tau^2} + \sum_{\alpha=1}^{3} w^{(\alpha)} = 0 \tag{2.20}$$

Figure 2.13(b) shows a one dimensional explicit, implicit mixed approach. Read [41] for further details and derivation.

2.3.3.2 Computational Requirements

Implicit solutions to the wave equation approximation have the advantage of advanced numerical stability and are deterministic by construction. Most implicit methods involve huge linear systems that have to be solved. For massive parallel compute clusters this means that the performance of the code relies on the scalability of parallel linear solvers. Up to date such highly scalable and easy to use linear solvers are not available. The linear system is solved for all dimensions (x, y and z) separately. Two main challenges have to be faced for this implicit method.

1. The ADI scheme implies the solving of linear systems over all three dimensions. This includes load and stores to/from main memory. The original data layout has only one fastest direction with a consecutive layout of data in memory. This means for two dimensions that non-consecutive memory loads and stores increase the memory access latency and therefore decrease application performance.

2. The solutions of separate subdomains are not independent from the other. After each iteration over all three dimensions the halo region data has to be shared between subdomains as long as the desired maximal error ϵ is not reached yet. In a worst-case scenario this problem could cause a major extension of runtime due to bad convergence.

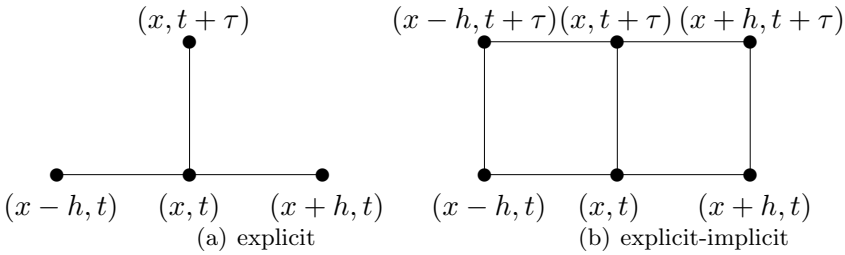


Figure 2.13: Stencil of an explicit (a) and a mixed explicit-implicit (b), first-order in time finite-difference scheme

2.3.4 Explicit Solutions for FDM

Different to the implicit solution which solves linear systems in timestep $t+\tau$ the explicit solution rely only the values of the current timestep t and / or the previous timestep $t-\tau$.

$$u_{ijk}^{t+1} = \tau^2 c^2 \left(\sum_{\alpha} \frac{u_{(i,j,k)+h_\alpha}^{t} - 2u_{ijk}^{t} + u_{(i,j,k)-h_\alpha}^{t}}{h^2} \right) + 2u_{ijk}^{t} - u_{ijk}^{t-1} \tag{2.21}$$

where $\alpha \in \{i, j, k\}$.

This enables explicit methods to propagate the wavefield solving the wave equation stencil for each point of the volume independently. Hence, this is why explicit methods are preferred for large 3D surveys run on computer cluster where domain decomposition is necessary to achieve reasonable time-to-solution. Explicit solutions to the finite difference method form stencil memory access patterns.

A data access pattern for the one dimensional second-order in space and time case is presented by Figure 2.13(a). In higher dimensional solutions, like presented by Equation 2.21 the data access pattern get even more complicated due to larger memory address offsets. The data required by the three-dimensional stencil has only one fast direction in which the data points lie consecutively in memory. In the second and third dimension, single points have to be loaded which lie further apart from each other in memory. This makes software optimization of such stencil kernels a challenging task. Optimal data access methods and programming models for stencil codes are discussed in Section 5.

2.3.5 The Approximation Order

The finite-difference method is an approximation to the analytical solution. The error, or numerical dispersion, compared to the analytical solution decreases with the order of the derivative operator or the number of terms used in the Taylor series representation of the function, respectively [17]. The optimal order is a trade-off between computational complexity, numerical stability and quality requirements. Analytical analysis show that above 8th order the quality improvement does not justify the increase of computational demand [53].
The 8th order finite-difference scheme is used as a seismic industry standard. Further the emphasis of this thesis is on such 8th order derivative operators.

2.3.6 Computational Requirements of Explicit Finite-Difference Kernel

In previous sections ISO, VTI and TTI wave equation kernel were introduced. This section analyzes the computational requirements of the explicit, finite-difference implementations, which are important to predict and understand delivered throughput performance on computing systems.
As described in Section 2.3.2 the finite-difference approximation of the second-order in time derivates form a stencil memory access pattern. PDE's for ISO and VTI media use derivates along the main axis only. Therefore the Laplacian stencil for 8th order in space looks like presented in Figure 2.14

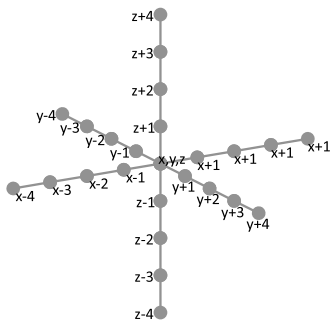
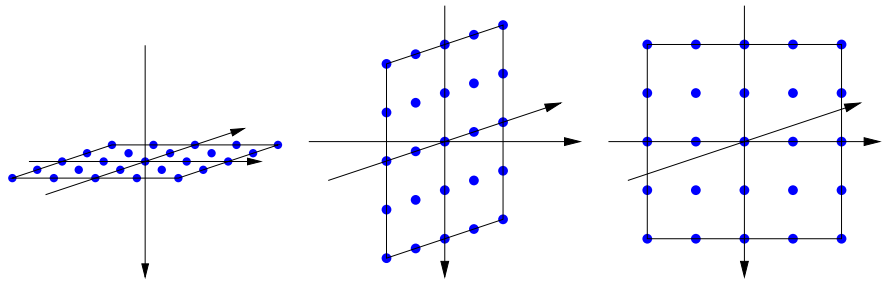


Figure 2.14: ISO, VTI 2nd Order in Time, 8th Order in Space Stencil

The number of points involved in the finite-difference scheme are $2 * r + 1$ per axis with r as radius defined as half the order in space. For ISO the Laplacian stencil re-



(a) Derivations $\partial_x\partial_x$, $\partial_x\partial_y$ and $\partial_y\partial_y$ (b) Derivations $\partial_y\partial_y$, $\partial_y\partial_z$ and $\partial_z\partial_z$ (c) Derivations $\partial_z\partial_z$, $\partial_z\partial_x$ and $\partial_x\partial_x$

quires accessing 25 points and performing a linear combination using 5 weights (one for each equidistant sextet of grid points in $\pm x$, $\pm y$, and $\pm z$, and one for the center). Thus, the Laplacian stencil performs 5 floating-point multiplies and 24 floating-point additions. The wave equation's time derivative requires accessing not only the grid point at the current and previous timesteps, but also the medium's velocity at that point. When the Laplacian and time derivative are combined, the complexity of the inhomogeneous isotropic wave equation's stencil is received. Higher order Laplacian stencils will access neighboring points further from the center and induce a corresponding increase in computation.

The introduced VTI implementation requires two wavefields P and an auxiliary wavefield Q. Whereas, for ISO the second-order derivates are computed in all three dimensions on wavefield P, for VTI the second-order derivates in x and y are computed on wavefield P and in z direction on wavefield Q. Further, additional computation is required to account for derive the different velocities resulting through a linear combination with the Thomson parameters ϵ and δ. The total amount of floating-point operations for a reference VTI kernel implementation increases to a total of 53 flops.

TTI requires second-order, mixed, partial derivates. If all derivations are approximated through finite-differences the resulting stencil consists of three planes. Figure 2.15(a) shows the plane resulting from $\partial^2/\partial x\partial y$, Figure 2.15(b) the $\partial^2/\partial y\partial z$ and Figure 2.15(c) the plane from the derivates $\partial^2/\partial x\partial z$, respectively. Additionally, all figures include the second-order derivates along the axis $\partial^2/\partial x\partial x$, $\partial^2/\partial y\partial y$ and $\partial^2/\partial z\partial z$. For easier visualization the figures show only 4th order in space stencils. The total amount of a reference TTI kernel adds up to about 800 floating-point operations per grid point.

2.4 The Reverse-Time Migration (RTM) Imaging Method

Reverse Time Migration (RTM) is a commonly applied high quality migration method and the most computational demanding part of the exploration seismic workflow. This section describes the RTM imaging method in more detail, introduces different implementation approaches and analyzes the main kernels in terms of computational intensity and their memory capacity demands.

The *Reverse-Time Migration* (RTM) is a wavefield-continuation method in time. It is the only method that is able to simulate all kinds of waves. This leads to the possibility of imaging waves even in areas with steep dipping reflectors and with strong velocity variations caused by utilizing the full wave equation (see Section 2.4.1). It consists of two main steps:

1. the numerical propagation of the wavefield is further separated in two independent propagations:
   (a) **source wavefield**: At first the source wavefield is extrapolated forward in time starting from a synthetic simulated source signal at timestep 0.
   (b) **receiver wavefield**: Starting from the received signals at the surface, the receiver wavefield is propagated reverse in time.
2. The imaging condition cross-correlates source and receiver points at time zero to create the final subsurface image.

So far it yields the highest quality of the resulting subsurface image but it has also the highest computational requirements. Figure 2.15 gives an example for the migrated image of the Marmousi data set. Here the hard velocity field was used due to utilization of the non-reflecting wave equation.

2.4.1 Mathematical Background

The wavefield continuation methods are propagating waves through a given velocity field. Due to low computational performance first approaches in wave modeling used the one-way wave equation. With increasing computational resources the use of the two-way acoustic isotropic wave equation could be accomplished. Compared to the

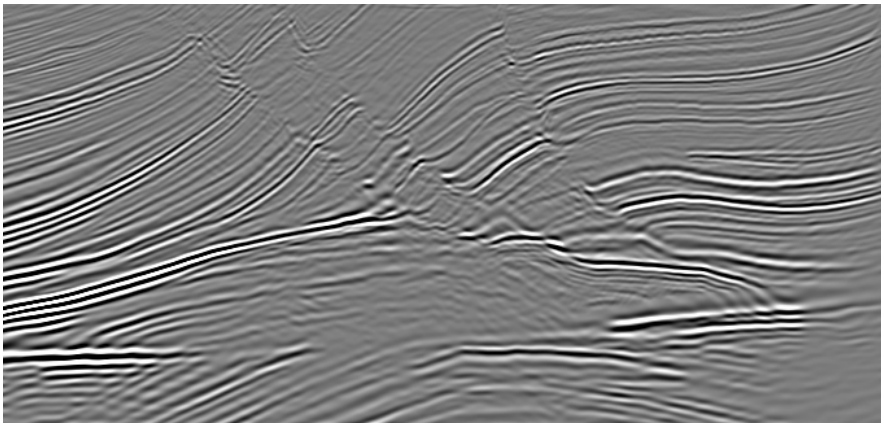


Figure 2.15: Reverse in time migrated subsurface image of the synthetic Marmousi data set

one-way isotropic wave equation it provides superior image quality in complex areas [7].

Commonly used wave propagation kernels are ISO, VTI and TTI implementations. VTI and TTI implementations are preferred since they account transverse anisotropy (see Section 2.2.3, 2.2.4).

The task of migration is now to calculate from the recorded data at the surface $q(x, y; t)$ and the approximated velocity data $c(x, y, z)$ the wavefield u at timestep $t = 0$ ($u(x, y, z; t = 0)$) for each point in the volume.

Source and receiver wavefield can be propagated independent from each other. The following equation simulates the wave propagation of the energy source.

$$\begin{cases} \left(\Delta - \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\right) S = 0, \\ S(x, y, z = 0; t) = r(t) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \delta(x - x_s) \cdot \delta(y - y_s) dx\, dy, \end{cases} \tag{2.22}$$

where $S : (x, y, z; t) \mapsto S(x, y, z; t)$ represent the source wavefield, δ the δ-Dirac function and x_s, y_s source coordinates. As source a synthetical wavelet called Ricker-wavelet $r(t)$ is used. It describes a narrow spike with low to no vibrations 2.23.

$$r(t) = \exp\{-\gamma^2 t^2\} \cos(\omega_{peak} t), \tag{2.23}$$

,with γ as damping coefficient and ω_{peak} the peak circular frequency. The modeling is done in time. Compared to a measured signal at the source the advantages of using a Ricker-wavelet are less data storage requirements and high flexibility on adjusting the wavelet to algorithmic needs. The Ricker wavelet printed as a function of time is shown by Figure 2.16.

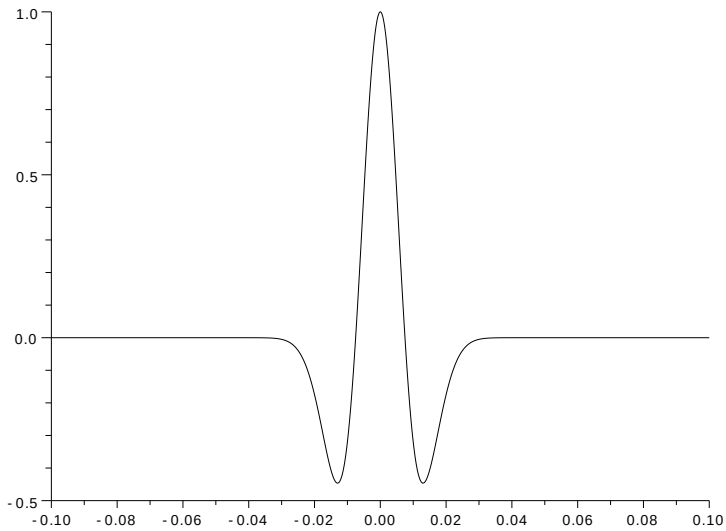


Figure 2.16: The Ricker Wavelet

Next the receiver wavefield is extrapolated reverse in time starting at time t.

$$\begin{cases} \left(\Delta - \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\right) R = 0, \\ R(x, y, z = 0; t) = q(x, y; t), \end{cases} \tag{2.24}$$

where $R : (x, y, z; t) \mapsto R(x, y, z; t)$ represents the receiver wavefield.

For the final result $u(x, y, z; t)$ the imaging condition

$$u(x, y, z; t) = \int_0^T S(x, y, z; \tau) \cdot R(x, y, z; t - \tau)\, d\tau.$$

is used to cross-correlate the source and receiver wavefield.
The final wavefield $u(x, y, z; t = 0)$ is received

$$u(x, y, z; t = 0) = \int_0^T S(x, y, z; \tau) \cdot R(x, y, z; \tau)\, d\tau. \tag{2.25}$$

Figures 2.17(a), 2.17(b), 2.17(c) present a simple example of forward and backward wave propagation, taken from Biondi [9]. It illustrates the propagation of source (a), receiver (b) wavefield and the cross-correlation (c). These figures show three timesteps at t=1.20, t=.75 and t=.30. The two lines visualize areas of impedance contrast between two areas with different velocities. The acoustic wave is therefore partly reflected by them.

2.4.2 RTM Schemes

As the main concept of RTM was introduced in previous sections, this section gives an overview what kind of RTM implementations are commonly allied in the seismic industry.

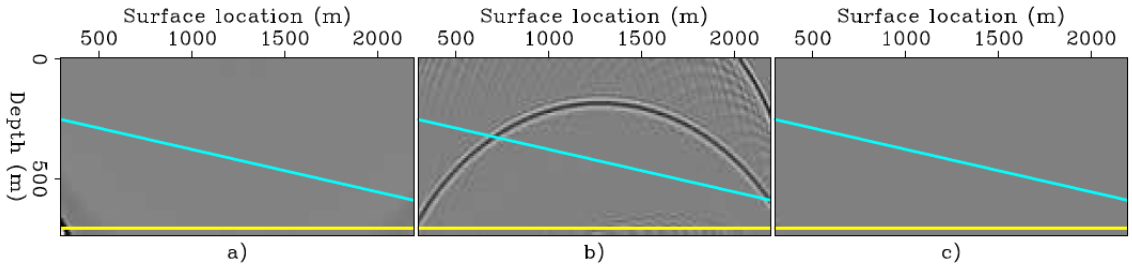
2.4.2.1 Naïve RTM Scheme

The naïve RTM scheme propagates and saves the source wavefield for all timesteps from $t = 0$ to $t = maxtime$. Afterwards the receiver wavefield is propagated backwards in time while the saved source receiver wavefields have to be read from memory to cross-correlate both wavefields for the final image. Code 2.1 shows this approach in pseudo code.

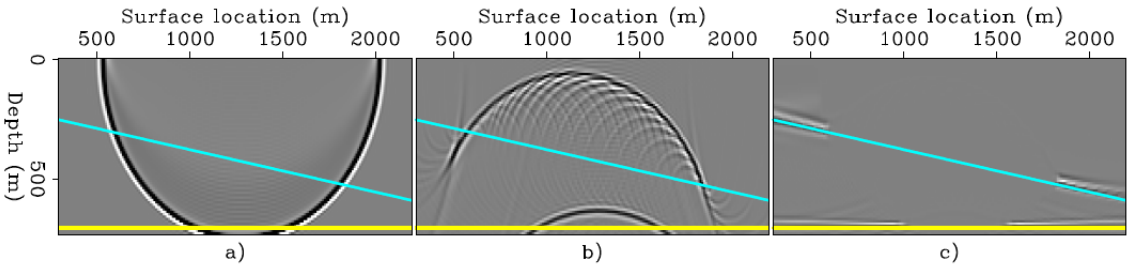
Listing 2.1: Naive RTM approach

```
for all shots do
   for t = 0 to t = maxtime do
        Advance source wavefield +dt
        Write source wavefield at time t
    end for
    for t = maxtime to t = 0 do
        Advance receiver wavefield -dt
        Load source wavefield at time t
        Correlate source and receiver wavefield
    end for
end for
```

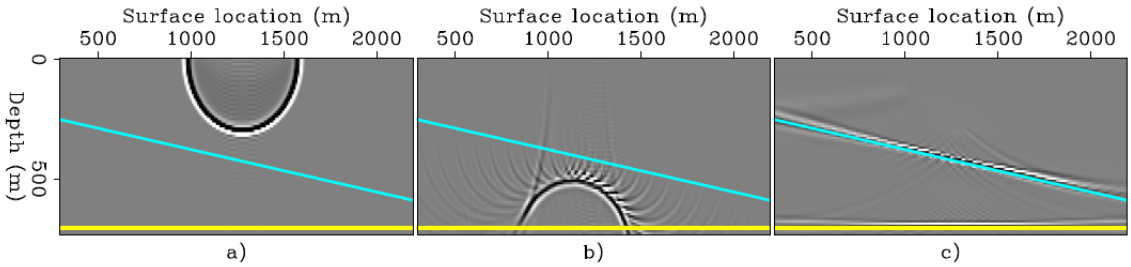
For relieve in bandwidth pressure, data compression algorithms can be utilized. By using compression, less I/O bandwidth pressure is traded against additional computational efforts depending on the compressing algorithm and compression rate. Code 2.2 shows this RTM scheme.



(a) Reverse-time migration with constant (background) velocity function. Snapshots at t=1.20 seconds of source wavefield (a), receiver wavefield (b), and image progression (c). Neither reflector has been imaged yet.



(b) Reverse-time migration with constant (background) velocity function. Snapshots at t=.75 seconds of source wavefield (a), receiver wavefield (b), and image progression (c). The bottom reflector is almost fully imaged, and the shallow reflector is only partially imaged.



(c) Reverse-time migration with constant (background) velocity function. Snapshots at t=.30 seconds of source wavefield (a), receiver wavefield (b), and image progression (c). Both reflectors are fully imaged.

Figure 2.17: RTM with constant Velocity and two Reflectors [9]

Listing 2.2: Naive RTM approach with compressing of data

```
for all shots do
   for t = 0 to t = maxtime do
        Advance source wavefield +dt
        Compress source wavefield
        Write compressed source wavefield at time t
    end for
    for t = maxtime to t = 0 do
        Advace receiver wavefield -dt
        Load compressed source wavefield at time t
        Decompress source wavefield
        Correlate source and receiver wavefield
    end for
end for
```

2.4.2.2 RTM with Interpolation

Saving the source wavefield for each timestep still has high bandwidth requirements. RTM with interpolation therefore stores only every k^{th} source wavefield and interpolates between the kth and $k^{th} - 1$ wavefield during the backward propagation of the receiver wavefield. Code example 2.3 shows how this scheme works.

Listing 2.3: RTM with interpolation between saved source wavefields

```
for all shots do
   for t = 0 to t = maxtime do
        Advance source wavefield +dt
        if (t % k == 0) write source wavefield
    end for
    for t = maxtime to t = 0 do
        Advance receiver wavefield -dt
        if (t % k == 0){
            Load current and next src wavefield
            Interpolate between src wavefields
        }
        Correlate source and receiver wavefield
    end for
end for
```

2.4.2.3 RTM with Optimal Checkpointing

More accurate results are provided by the optimal checkpointing scheme. The method aims on saving only every k^{th} step of the source wavefield forward propagation. Since the wavefield is needed for every timestep during the backward receiver wavefield propagation, the source wavefield has to be propagated later again. This decreases the high requirements on storage space and bandwidth and increases the computational requirements. Since most algorithms are bandwidth bound, this technique increases the overall performance significantly. Code 2.4 gives an example for a RTM with optimal checkpointing.

Listing 2.4: RTM with optimal checkpointing

```
for all shots do
    Define checkpoints c[]
   for t = 0 to t = maxtime do
        Advance source wavefield +dt
        if (t == c[i++]) write source wavefield
    end for
    for t = maxtime to t = 0 do
        if (t == c[i--]){
            Load src wavefield c[i]
            for t = c[i] to t = c[i+1] do
                Advance source wavefield +dt
                Write source wavefield
            end for
        }
        Advance receiver wavefield -dt
        Correlate source and receiver wavefield
    end for
end for
```

2.4.2.4 Simultaneous Backward Propagating

The most optimal algorithm for RTM is using simultaneous backward propagation of source and receiver wavefield. The first step is to propagate the source wavefield to timestep $t = maxtime$ without saving any wavefields to memory or disk. In the second step source and receiver wavefield are both propagated backwards in time. Therefore the wavefields can be cross-correlated directly and don't have to be saved on disk or

memory. Due to minimized bandwidth requirements this approach provides the highest possible performance. Code 2.5 show this type of RTM scheme.

Listing 2.5: RTM with simultaneous backward in time propagation of source and receiver wavefield

```
for all shots do
    Create random boundary around computational domain
    Advance source wavefield to t= maxtime
    for t = maxtime to t = 0 do
        Advance source wavefield -dt
        Advance receiver wavefield -dt
        Correlate source and receiver wavefield
    end for
end for
```

2.5 Seismic Survey

This thesis targets to measure the benefits of a hardware/software co-designed system architecture that minimizes computation time and power consumption for large survey sizes migrated with reverse time migration. Since survey sizes keep growing, the design decisions should be made towards a survey size that belongs to the largest ones proceeded in marine seismic. Together with a fine grained resolution such a survey should provide an upper bound on survey parameters. The introduced architecture should be able to handle such surveys in a reasonable timeframe. Such a survey size is 30 km in streamline or x-direction, 20km in crossline or y-direction and 10 km in depth, which corresponds to z.

The exploration ship is assumed to tow 10 streamer lanes with 1000 receivers, e.g. microphones, each. All receivers have a time sampling interval of 4ms and listens a total of 12 seconds to reflections from the subsurface. Therefore, we get 3000 timesteps and a total of 30,000,000 samples per shot. For the modeling, the 4ms time sampling has to be interpolated to about 1ms to reach numerical stability. Therefore, 12,000 timesteps have to be processed in order to get the wavefield for the next timestep.

With a shot offset of 50 meters in x-direction and 100 meters in y-direction, 120,000 shots are necessary to cover the area of 30 km x 20 km. The calculation is applied on a fine-grained space grid of just 5 meters in all three dimensions. Since such a fine resolution is not needed for imaging of the data, it is written out with a resolution of

20m x 20m x 10m in x, y and z direction, respectively.
With these space-sampling intervals a total volume of 4000 x 4000 x 2000 points has to be processed for each shot over all timesteps. For the ease of handling on a binary based system design, the total volume size is slightly increased to 4,096 x 4096 x 2,048.

2.6 Summary

This chapter provided a brief introduction into different kinds of seismic processing. It is focusing on exploration seismic which goal it is to create subsurface images of the underlying rock layers in depth down to 10km. The chapter identifies that highly accurate migration, including modeling of anisotropic behavior, is critical for high quality subsurface images and takes requires the highest amount of computation time in the seismic processing workflow. Different migration methods are reasoned and Reverse Time Migration (RTM) identified as the current industry standard to focus on.
Three dimensional RTM provides strong qualitative advantages over its predecessors. It has a high computational cost warranting implementation on HPC architectures. While implicit implementations rely on large linear systems, explicit methods approximate the wave equation by solving linear combinations of local points. Since all points can be computed independent from each other, explicit approximations are a better choice for exploiting explicit parallelism on large compute clusters.
This chapter analyzed isotropic, VTI and TTI kernel in terms of computational intensity and their demands in memory capacity and memory bandwidth pressure. Especially complex subsurface layers requiring PDEs accounting for anisotropy are likely pushing modern HPC architectures to their limits which makes it critical to analyze RTM forward modeling kernel performance on different architectures.
Finally, this chapter presented an example large-scale survey, which is further used as example survey to determine performance and energy efficiency of different compute architectures. For that reason Chapter 3 discusses and evaluates different leading-edge, on-market HPC architectures, determines their performance limitations and benchmarks performance running the introduced kernels.

Chapter 3

Compute Architectures: State-of-the-Art

The seismic industry is highly motivated to use wave equation kernels like ISO, VTI and TTI for subsurface imaging. Unfortunately, high computational requirements lead to a reluctant use. Therefore, an important focus of current research in the seismic community is the exploration of performance and energy efficiency of new hardware architectures to improve migration quality and speed.

This chapter first provides a general overview of the state-of-the-art in computer architecture in Section 3.1. A few architectures which are commonly used in seismic processing are selected and discussed in more detail (see Sections 3.2.1ff).

3.1 Computer Architecture: An Overview

In general, the choice of hardware architecture is made by its target application. General-purpose processors like x86 or Power7 are focusing on generality and are most commonly used in mainstream computing systems. They target a high variety of tasks including e.g. running operating systems, encoding, multimedia, etc.

With the rise of tablet computers like the Apple iPad and smart-phones like Apple's iPhone, embedded processors gained big parts of the processor market share. Such processors are designed for maximal energy efficiency to provide a maximum battery lifespan of the portable devices. Embedded processors are more specific than general-purpose processors but are still able to run specific operating systems. In HPC, cluster

architectures based on low-power cores gain performance through high parallelism - an example is IBM's Blue Gene.

More specific are accelerator cards like graphics processing units (GPUs). Initially focused on accelerating graphics performance only, they recently gained new attraction from the scientific community. Since the California-based graphics manufacturer NVidia released its *Compute Unified Device Architecture* (CUDA) programming language for scientific computing in 2008, GPU performance was accessible without implementing with OpenGL or other graphics specific libraries. As GPUs are accelerator cards only they still require a host CPU handling I/O and running the operating system.

Next are *Field Programmable Gate Array's* (FPGAs). Such devices consist of non-predefined arrays of gates, which can be programmed by the user. Manufacturer specific synthesis tools are then connecting these gates to each other to perform the desired tasks. The programming of FPGAs is more complicated since applications have to be mapped onto the FPGA on gate level. This requires not only numerical and software skills but detailed knowledge about hardware, which resulted in a reluctant use in the seismic community so far.

Finally, most specific to an application are *Application Specific Integrated Circuits* (ASICs). ASICs are designed to do a specific task in the most efficient but predefined way. The development of ASICs includes the complete hardware design and fabrication cycle, which includes writing hardware description languages like Verilog, creating a mask and manufacturing of the actual chip. ASICs provide best energy efficiency and performance but are the least flexible and most expensive option.

Figure 3.1 illustrated the introduced architectural approaches. It draws energy efficiency on the y-axis and relates it to the level of specialization from left (least specific) to right (most specific).

3.2 Evaluated Node Architectures

These high computational requirements of forward modeling kernels like TTI forces the seismic community to use large computing clusters and exploring new architectures to produce imaging results in a reasonable amount of time. This section presents several leading-edge, on-market compute architectures. These architectures are benchmarked in terms of throughput and energy efficiency.

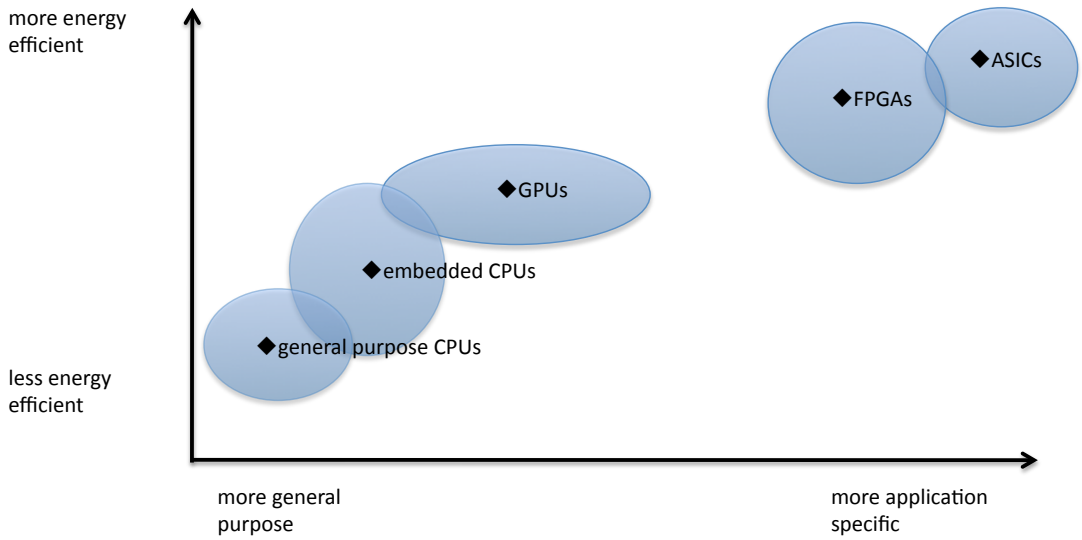


Figure 3.1: Energy efficiency (y-axis) over application specificity (x-axis) for common on-market architecture designs

The following sections provide brief introductions of the different architectural approaches. As general-purpose architectures the Intel Nehalem X5550 (see Section 3.2.1 and AMD's Magny Cours (see Section 3.2.2) CPUs are chosen. The Intel Nehalem X5550 CPU is used in the cluster named "Carver", the Magny-Cours in a Cray XE6 cluster called "Hopper". Carver and Hopper are both located at the NERSC computing facilities in Oakland, California.

For GPUs, Section 3.2.4 gives an introduction into the Fermi M2090 architecture, which is used in a multi-GPU node setup. As GPUs are accelerator cards, they require a host CPU for I/O and the OS . This adds an additional power overhead and has to be considered for total node power consumption. To mitigate the power overhead multi-GPU setup are chosen to provide a fair benchmark on energy efficiency.

Finally, Intel's Xeon-E5 2687W "Sandy Bridge" architecture wraps up the introduction of evaluated COTS architectures (see Section 3.2.3). The Sandy Bridge is Intel's latest development released in March 2012 and is the successor of Intel's Nehalem architecture.

Table 8.1 gives an overview of the main architectural details of all evaluated machines. The interested reader may read the corresponding section to gain further details

Core Architecture	**Intel Nehalem**	**NVIDIA Fermi**	**Intel Sandy Bridge**	**AMD Opteron**
Type	superscalar out-of-order SIMD	dual-warp in-order SIMT	superscalar out-of-order SIMD	superscalar out-of-order SIMD
Clock (GHz)	2.66	1.30	3.40	2.10
SP Gflops/s	21.3	83.2	54.4	16.8
L1 Data $	32 KB	16/48 KB	64 KB	64 KB
L2 Data $/LS	256 KB	N/A	256 KB	512 KB
Threads/core	2	1536 (max)	2	1
Socket Architecture	**Xeon X5550**	**Tesla M2090**	**Xeon E5 2687W**	**Opteron 6172**
Cores/chip	4	16 SMs	8	6
Shared LLC/chip	8 MB	768 KB	20 MB	6 MB
Chips/socket	1	1	1	2
SP Gflops/s	85.3	1331.2	386.8	100.8
DRAM pin GB/s	32.0	177.4	42.4	21.33
TDP	95W	225W	150W	115W
SMP Architecture	**Xeon X5550**	**Tesla M2090**	**Xeon E5 2687**	**Opteron 6172**
Chips/SMP	2	1	2	4
memory parallelism	HW prefetch	Multi-threading	HW prefetch	HW prefetch
DRAM Pin GB/s	64.0	177	84.8	85.33
STREAM GB/s	35	150 (no ECC)	75.2	47
SP Gflops/s	170.66	1331	793.6	403.2
Power under RTM load	298W	180W (GPU-only)	350W	455W

Table 3.1: Details of the evaluated architectures

about the compute node setups.

Further architectural architectures like Intel MIC, IBM Cell and Xilinx FPGAs. Their performance and energy efficiency are not evaluated because of access limitations and time constraints. However, this section still provides a brief introduction into those architectures because of their interesting architectural features and different approaches.

3.2.1 Intel Nehalem X5550 (Nehalem)

The first introduced architecture is Intel's "Nehalem" X5550. The CPU is based on Intel's "Core" architecture, which first entered the market in 2008. The architecture,

reminiscent of AMD's Opteron processors, integrates memory controllers on-chip and implements a QuickPath Interconnect (QPI) inter-chip network similar to AMD's HyperTransport (HT) that replaces the older Front Side Bus (FSB). QPI provides access to remote memory controllers and I/O devices, while also maintaining cache coherency. For QPI the focus was set on throughput and scalability to address the increased communication dependence of current and future multi-core architectures.
Although Nehalem offers two-way simultaneous multithreading (SMT) and TurboMode, both were disabled on test machines. The latter allows a subset of the cores to operate faster than the nominal clock rate under certain workloads. To provide consistent timing and power measurements, TurboMode was disabled. The evaluated system is a dual-socket, quad-core 2.40 GHz Xeon X5550 with a total of 16 hardware thread contexts. Each core has a private 32 KB L1 and a 256 KB L2 cache. The four cores on a chip share an 8 MB L3 cache and three DDR3-1333 memory controllers capable of providing up to 17.6 GB/s per chip.
Using 128-bit wide SIMD instructions and FMAs in each cycle the peak performance per dual-socket node is 170 GFlop/s. This thesis evaluates the performance on the "Carver" Infiniband cluster at NERSC, which is comprised of over 400 X5550 compute nodes. Locally, nodes are connected via a QDR Infiniband fat tree network.

3.2.2 AMD Opteron 6172 (Magny Cours)

AMD's Magny-Cours architecture was released in March 2010. The Opteron 6172 is a 12-core CPU. Each socket (multichip module) consists of two dual hex-core dies with individual memory controllers. As such, the compute node which uses two sockets or 24-cores in total, is conceptually a four-chip NUMA SMP connected via HyperTransport3 (HT3) interconnect. HT3 does not provide as much interconnectivity, which leads to a strong NUMA affine behavior.
Each Opteron chip has six super-scalar, out-of-order cores running a 2.1 GHz. In single-precision, each core can complete one (four-slot) SIMD add and one SIMD multiply per cycle. This provides a peak performance of 403 GFlop/s per node. Additionally, each core has private 64 KB L1 and 512 KB L2 caches. The six cores on a chip share a 6 MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average STREAM[110] bandwidth of 12 GB/s per chip. About 1 MB of L3 is reserved for the probe filter and cannot be used for data. Therefore, effectively 5 MB L3 cache

is shared between the 6 cores. This thesis uses a cluster called "Hopper" which is a Cray XE6 cluster at the NERSC facilities. It is built from over 6000 2P Opteron 6172 compute nodes. Each compute node has up to 64 GB of DDR3-1333 SDRAM and shares one Gemini network chip, which collectively forms a 3D torus interconnect topology. Magny-Cours supports SIMD instruction sets up to SSE3.

3.2.3 Intel Xeon E5-2687W (Sandy Bridge)

Sandy Bridge is Intel's newest commodity CPU architecture, which was introduced in January 2011. The novelties of Sandy Bridge are an integrated GPU and a ringbus that connects the cores to each other. The Xeon E5-2687W node is a dual-socket, 16-core processor setup running at 3.1 GHz. The microarchitecture has been enhanced from Nehalem and now supports the 256-bit AVX SIMD instruction set. Thus, each core is capable of executing one (eight-slot) SIMD add and one SIMD multiply per cycle. This provides a peak node performance of 793.6 GFlop/s (significantly faster than the dual-socket Nehalem). With HyperThreading each core is capable of running 2 threads at a time, which results in a total of 32 threads per node. Unlike Nehalem, Sandy Bridge has two load ports to the L1, thereby doubling L1 read bandwidth — a clear benefit when locality cannot be maintained in the register file. However, the rest of memory hierarchy is quite similar to the X5550 except that Sandy Bridge uses a ring-based network-on-chip and has four DDR3-1600 controllers. Therefore, the E5 has 8 controllers (2Px4) while the 5550 has 6 (2Px3). The node setup uses eight DDR3-1333 memory modules, which reduces peak bandwidth from 102.4 GB/s to 84.8 GB/s. As for Nehalem, Sandy Bridge supports a so-called TurboMode that is capable of increasing the clock frequency if not all cores are used. As for the Nehalem nodes the hardware managed TurboMode was disabled to provide consistent benchmark results.

3.2.4 NVIDIA Tesla M2090 (Fermi)

Graphic Processing Units or GPUs are specialized accelerator cards for graphic processing. Executing the same operations on all pixels enables high parallel processing. Current GPU architectures (NVidia) have 512 [63] SIMD cores, which are organized in so called, thread blocks of multiple cores using a shared memory. All data that should be used in those thread blocks has to be transferred from global memory to the shared memory.

In 2008 NVidia released the Compute Unified Device Architecture (CUDA). CUDA enables programmers to use widely spread programming languages like C or C++ to write software for NVidia GPUs. This enabled for GPUs the way into the HPC market. Several of studies have shown that GPUs can address some problems very efficient and provide enormous speed up rates of 100x and more ([27], [55]).
The M2090 is a Fermi-architecture GPU with 6 GB of GDDR5 memory and 16 Streaming Multiprocessors (SMs). Each multiprocessor is clocked at 1.3 GHz and contains 32 fp32 pipelines (for an aggregate of 512 for the entire GPU), 64 KB of SRAM that can be partitioned by the programmer to be used as shared memory or L1 cache, and a 128KB register file. While an SM can track up to 1536 threads, it implements the *Single Instruction Multiple Threads* (SIMT) execution model: any instruction is issued for a group of 32 threads (referred to as a warp), hardware predication ensures correctness when threads in a warp take divergent control paths. Instructions are issued in-order and execution is pipelined. Switching execution among warps hides memory-access and arithmetic latencies. Context switching between warps is free since the register file and other state is partitioned among the active threads. Theoretical shared memory bandwidth, aggregated across all the SMs, is 1331 GB/s. Theoretical DRAM bandwidth is 177 GB/s, of which 150 GB/s can be sustained with a stream copy. Turning ECC on reduces the amount of bandwidth available to data. While the exact amount of bandwidth consumed by ECC depends on the application, typically it consumes around 20% (note that impact on application performance can be lower if the memory bus isn't continuously saturated).

Figure 3.2(right) shows the *Streaming Multiprocessor* (SM) unit of the NVidia Fermi architecture with its 32 cores (3.2(left)), *Special Function Units* (SFU) and the full cache, memory hierarchy. The four SFUs execute instructions like sin, cosine, reciprocal and square root. As described is the 64 KB shared memory is configurable as 16 KB L1 and 48 KB shared memory or 48 L1 with 16 KB shared memory and 768 KB L2 cache. "LD/ST" refers to 16 load/store units[63].

The highly specialized architecture, targeting mainly graphic processing, causes problems as well. The small, shared-memory doesn't fit well for problems with higher needs on memory capacity for data that is needed frequently. Even the global memory, which provides capacities up to 6 GB (2010), is not enough. Therefore data has to be split between several GPU boards or data has to be transferred from host memory

to the global memory of the GPUs. Since GPUs are accelerator cards they have to communicate via interconnects like PCI-Express. Up to now the fastest available PCI-Express interconnect provides transfer-rates of 16 GB/s, which turns into a bottleneck the faster the GPU becomes.

Since GPUs are using SIMD cores the problem has to fit the requirements of simultaneous computation of several data items at the same time. Not all problems provide this independency. Without full utilization of the SIMD cores the full performance can not be reached.[15]

GPUs memory latency hiding strategies are based on context switching by switching threads. Therefore, many more threads then core per SM has to be provided. The zero-latency thread switching capabilities are achieved by a dedicated register file per thread so that the thread's register file doesn't have to be restored. If a thread's register file is big this can cause capacity issues due lacking of enough entries. This way it might happen that not enough threads can be provided and latencies cannot be hidden.

Application development for GPUs requires careful planning and mapping of the algorithm to the architecture due to the many-core design. SIMT cores and a deeper memory hierarchy add additional complexity to on-node communication considerations. This makes software optimization for GPU especially important and from the beginning a part of kernel development.

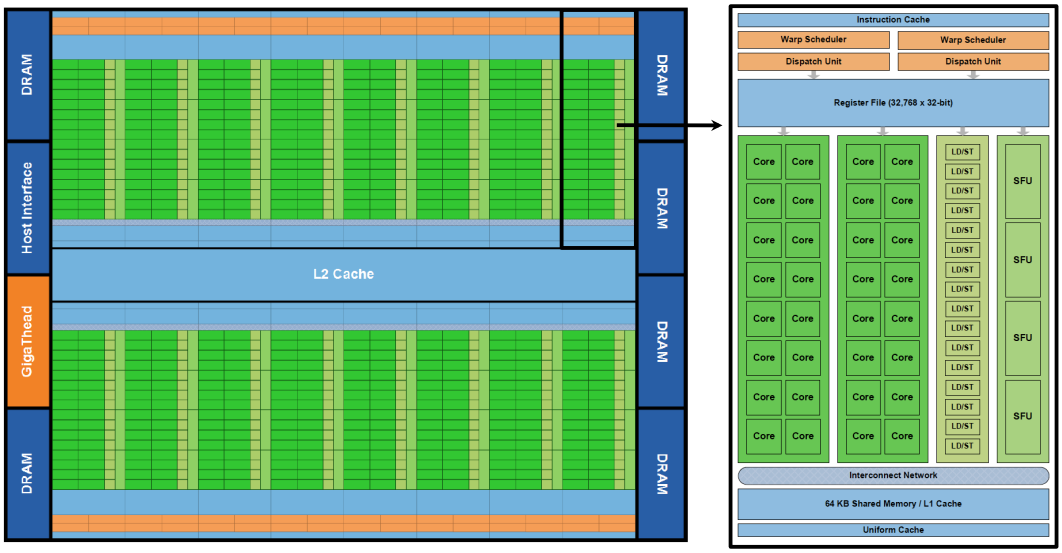


Figure 3.2: The NVIDIA Fermi GPU [63]

In this thesis, results are compared to GPU results obtained by NVIDIA's Paulius Micikevicius on his internal machines with up to four M2090 GPUs. For multi-GPU configuration, PCIe switches are used to arrange the GPUs into a tree topology, as shown in Figure 3.3. Each PCIe link has a theoretical duplex bandwidth of 16 GB/s (8 GB/s per direction), in practice 12 GB/s can be achieved.

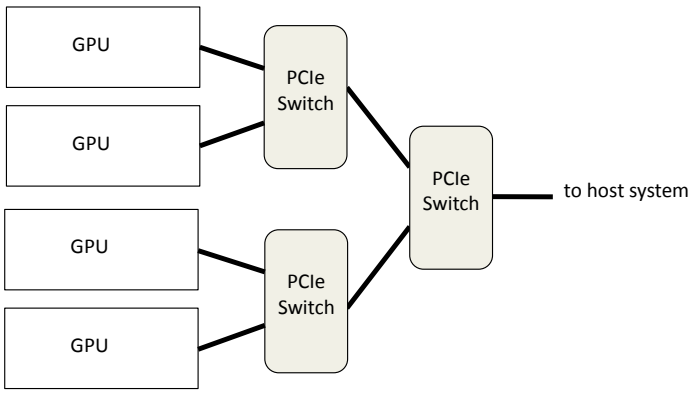


Figure 3.3: Multi-GPU Node Configuration

3.3 Related Architectures

The following architectures are not benchmarked in this thesis. Regardless they are presented because they are either subject of interest in the seismic industry (FPGAs), provide interesting architectural features like the IBM Cell processor or show what manufactures are currently working on (Intel MIC). Lessons learned from these architectures are discussed at the end of this section.

3.3.1 Intel Many Integrated Core Architecture (MIC)

Intel introduced a novel architecture called Larrabee in 2008. The goal was to combine the advantages of many-core architectures like GPUs and the compatibility of CPUs to reach a new dimension in general use of accelerator cards for all kind of problems. Larrabee was renamed to Knights Ferry, Knights Corner and finally published as *Many Integrated Core* (MIC) architecture. The pre-production cards use 30 x86 cores based on the Pentium 1 design. Each core runs at 1 GHz and has private L1 and L2 caches.

Different to the original Pentium approach no prefetch unit is utilized. Cache line prefetching relies on the programmer entirely. Another difference is that those caches are shared between four hyper-threading threads per core. Like on GPUs each thread has its own register file which enables zero latency context switching between threads. As well as on GPUs, threads are used to hide memory access latencies. Each clock cycle an instruction of one of the threads is issued to the pipeline in a round robin fashion. Each MIC card uses 2 GB of on card GDDR5 of which 1 GB is reserved for the operating system, which makes it 1 GB usable capacity. The peak bandwidth is 115 GB/s. About 70-80 GB/s are sustained using the STREAM benchmark.

Motivated by the success of GPUs, the MIC gives a good example how x86 architecture based companies like Intel take new approaches designing novel many-core architectures. It's advantage is the use of almost standard x86 cores which should make application porting more straight forward than porting to pure GPUs. As well as for GPUs, due to the many-core design, even the naive implementation has to be well planned. As all accelerator cards MIC suffers from limited on card memory capacity and the PCIe bottleneck.

MIC is not officially released yet and only research prototypes exist which makes drawing conclusions on that approach too early. One can see that Intel tries to tackle typical problems of GPUs that include backward compatibly and high conversion costs from complex scientific x86 codes to e.g. specific language extensions like CUDA.

3.3.2 Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGAs) are used and subject of research for many years now. The ability to program the hardware through hardware description languages (HDL) and synthesizing tools, it became attractive to hardware designers to test and validate their hardware architectures.

In recent years FPGAs became increasingly more powerful reaching clock-rates of 150+ MHz and high memory capacities and bandwidth that makes FPGAs attractive for custom computing in the seismic community as well. To achieve high performance gains even with low clock rates compared to general-purpose processors, the hardware implementation has to fit on the algorithm needs as much as possible. Usually, algorithm are written in software and executed on different execution units on the CPU. In FPGAs the algorithm is mapped onto hardware, which creates a pipelined design — e.g. an

addition in software would be mapped with an adder functional unit on the FPGA. The performance of the FPGA depends on the number of gate arrays available to the programmer, the frequency of such and the complexity of the algorithm mapped onto it. The more complex the algorithm, the more area the functional units consume on the FPGA and the less pipelines can be created to compute the results. Best performance is achieved if all pipelines produce one result per cycle and the complete area of the FPGA is leveraged.

The main problem with FPGAs is the complicated coding process, including writing in languages like HDL, which most software developers, or scientists are not familiar with. Some new FPGA startup companies, like Maxeler [54] and Convey [18], provide high-level language interfaces for e.g. C/C++ to make FPGA development accessible to a wider range of developers.

FPGAs suffer from typical problems that appear for accelerator cards - e.g. host memory and inter-card communication bottlenecks. Fortunately, the much higher amount of on-board DRAM (96+GB) mitigates capacity issues associated with GPUs.

FPGAs present a good example in how far customizing an architecture design to an application can lead to great performance and energy efficiency improvements. Its major drawback is the implementation and debug process.

3.3.3 IBM Cell Broadband Engine Architecture (CBEA)

First released in 2006 the *Cell Broadband Engine Architecture* (CBEA) developed by Sony, Toshiba and IBM (STI) followed a new approach in hardware architecture. The Cell processor consists of a *Power Processor Element* (PPE) and up to eight *Synergistic Processor Elements* (SPEs). The PPE is a reduced 3.2 GHz PowerPC CPU. Its main task is to run the operating system and to manage threads and I/O operations of the Synergistic Processing Elements. The Cell leverages up to eight *Synergistic Processing Elements* (SPEs), which are "specialized for data-rich, compute intensive SIMD and scalar applications" [39]. A SPE is clocked with 3.2 GHz and provides a specialized instruction set architecture - a new SIMD instruction set specialized for the SPEs. An SPE has its own program counter, four execution units and fetches instructions from its 256KB local store (LS) instead of a cache. The SPE leverages a *Direct Memory Access Controller* (DMAC) to gain access to the main memory. All main memory accesses are software managed and need to be explicitly called by the application.

The CBEA is used in Sony's Playstation 3 console, which was sold over 50 million times worldwide. In 2009 IBM announced that the new Playstation generation would not use the Cell processor. IBM stopped further developments of Cell chips. The CPU is still available for scientific, research purposes.

Figure 3.4 shows an overview of the Cell architecture.

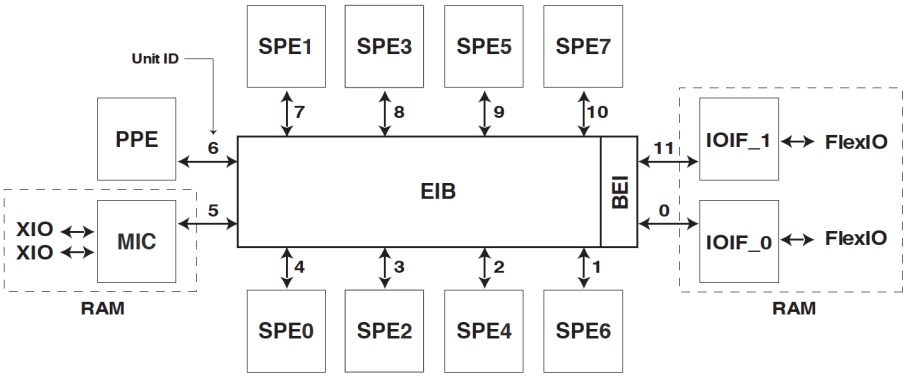


Figure 3.4: The Cell Broadband Engine Architecture (CBEA)

One of CBEA's main disadvantages was the complicated programming process via DMA calls and its non-Harvard architecture. Instructions and data were both hold in the local store which resulted in variating local store sizes utilizable for data. But, the Cell processor showed great performance and energy efficiency when all technical features and optimization options were exploited.

3.4 Summary

The high computational demands of RTM make it critical to analyze and compare performance of on-market architectures running wave propagation kernels. This chapter introduced several x86-based multi-core architectures and Nvidia's leading HPC GPU M2090 in detail. Additionally, non-evaluated architectures were briefly introduced as they provide interesting architectural features and provide a look on current research. The following chapter now analyzes kernel requirements and its performance evaluated compute platforms.

Chapter 4

Evaluated Architectures Performance and Efficiency Analysis

The previous chapters determined explicit RTM as an optimal algorithm for seismic migration on HPC systems and introduced several leading-edge COTS architectures. This chapter now analysis the core propagating kernel of RTM and compares its reference implementation on all platforms.

At the beginning brief introduction into benchmarking is given by Section 4.1 followed by the benchmark setup for this study in Section 4.1.1. Leveraging the Roofline Model, Section 4.2 derives performance limitations for all kernels and platforms. Finally, Section 4.3 presents performance and energy efficiency benchmarks for reference kernel implementations and compares the results to the derived architectural performance boundaries.

4.1 Reference Kernel Benchmark

Benchmarking of compute architectures aims at quantifying attained performance for a given task. There exist different performance metrics, which can be measured for a given architecture. In this thesis performance is compared by the following units:

- **Throughput**: Throughput can be defined as number of outcomes per time period. This thesis uses floating-point operations per second (flops/s) or points

per second (Points/s). Calculation of a single point in the volume means an approximation of the wave equation to attain the amplitude value for the next timestep. This computation consists of a stencil, which complexity depends on its size and shape, and additional operations to account for earth properties. Total throughput is listed as million Points/s (MPoints/s).

- **Energy Efficiency**: Energy efficiency describes the ratio of the achieved throughput and the power consumed. This thesis uses the energy efficiency metric MPoints/s per Watt (MPoints/s/Watt) to quantify energy efficiency.

To ensure a fair comparison the benchmarks have to be normalized over a certain unit. Normalization is applied over:

- **Volume Size**: All compared architectures are working on a fixed volume size. The size of the volume has to be a fair choice and not in favor of a specific architecture.
- **Processing Units**: The number of processing units must be the same. This could mean normalization over the number of cores, SMPs, sockets or nodes.
- **Time or Throughput**: All setups have a fixed time-to-solution and a normalized throughput. This way architectures can be compared in terms of energy efficiency.

4.1.1 Single-Node Benchmark Setup

To provide a realistic and fair comparison between different architectures one has to consider kernel requirements and requirements set through the application area which is seismic wave propagation in our case.
For a single-node comparison the volume size should be normalized for all architectures and performance measured in terms of throughput and energy efficiency. Throughput is measured in million point updates per second (MPoints/s). For each point update, a stencil is applied that resolves from the explicit finite-difference approximation of the wave equation and calculates the wave amplitude value for the next timestep. Energy efficiency is defined by how many Watts are consumed to achieve the given throughput - referenced as "million point updates per Watt" (MPoints/s/Watt). All computations are done in 32-bit single floating-point precision, which is the industry standard for

seismic imaging.

A perfect architectural design solution would be able to process one shot on a single node to avoid inter-node communication. Unfortunately, this is implausible for large survey sizes like introduced in Section 2.5. To attain the optimal node subdomain size several points have to be considered. First the subvolume should be as large as possible to be able to run large shot volumes on a single node. Parallelism would then be exploited between independent shots. Further, a larger node subdomain size also leads to fewer node counts for large shot volumes and therefore requires less communication. On the other hand the volume must be small enough to keep the on-chip memory and node memory requirements in reasonable limits. The memory capacity requirements can be calculated if the kernel type, the domain volume and the finite-difference space discretization are defined. An 8th-order stencil scheme working on an 256^3 volume would require about 275 MB to 900 MB main memory for ISO, VTI and TTI wave-equation kernels (see Section 7.4 for a more detailed description how the main memory requirements are derived). Hence, 2 GB of main memory would be sufficient but would offer limited room for eventual applications with higher capacity requirements. Further, even smallest shot sizes exceed 256^3 and a single node per shot computation would not be possible. Each doubling of the volume sizes leads to an about $8\times$ increase in memory requirements. A 512^3 volume requires 1.6 GB to 7.2 GB and 1024^3 already 17.6 GB to 58 GB . One can see that the gap between the memory requirements of the isotropic and TTI kernel widens for increased volume sizes.
Second, GPUs require a certain volume size that provides enough points to create a minimum number of threads. As GPUs hide latencies through switching between threads a small volume size would lead to decreased performance and a non-fair comparison between architectures. As result a 512^3 is applied for all further single-node benchmarks and all architectures.
As described in Section 2.3.5, a finite-difference scheme with 8th-order in space derivatives offer the best trade-off between computational complexity and quality of the result. For the derivative operator, four points in positive and negative unit-stride direction are accessed in each dimension. These four points are further denoted as stencil *radius*. This expansion in negative unit-stride direction requires adding additional boundary points, called *halo*, to the volume. The thickness of the halo region equals the stencil

radius. The memory layout is one dimensional with x as fastest and z as slowest unit stride. Figure 4.1 shows a simple 2D decomposition with added boundary points for the volume halo (blue) and halo regions shared between subdomains (orange).

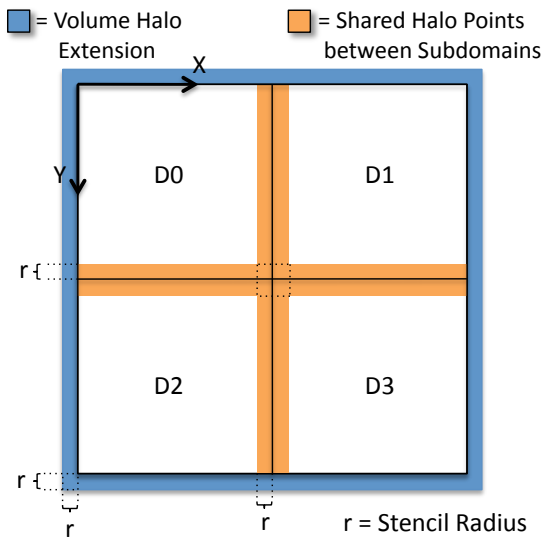


Figure 4.1: Domain Decomposition with Halo Region

4.2 Performance Estimations

Estimating the performance for a given task is critical in many cases. If new hardware should be purchased, performance estimations of applied kernels give a hint how big the performance advantages compared to the old system would be to make investment decisions. In the case of evaluating the performance of a certain kernel, kernel performance estimations are necessary to show if peak performance for a given kernel was reached or if additional effort should be put into optimizing the kernel.

There are two main reasons why a system might not reach peak performance. A first reason is a memory bandwidth restriction, in which the bandwidth from the memory can't deliver the data fast enough to the function units of the cores to fully utilize them. In the second case enough memory bandwidth exists but the complexity of the kernel prevents the functional units to run at peak performance.

To be able to determine if a kernel is memory bandwidth bound one has to analyze

its arithmetic intensity 4.2.1. Once the arithmetic intensity is known for all kernels, Section 4.2.2 studies maximum kernel performance based on memory bandwidth limitations and Section 4.2.3 has its focus on performance limitations set by the instruction performance of each architecture.

4.2.1 Arithmetic Intensity

Arithmetic intensity describes the ratio of compulsory floating-point operations to the total amount of bytes transferred from memory (flops/byte). The total DRAM traffic is the amount of memory requests after filtered by the cache hierarchy. It therefore depends on the spacial locality data points. The lower the ratio the higher the memory bandwidth pressure, the higher the ratio the more computational intense a kernel is. To estimate the maximum number of points per second for each node architecture the number of bytes loaded per stencil computation has to be divided by the total amounts of points processed $P_{stencil}/P_{total}$. In the case of 3D stencil algorithms with dimensions X, Y, Z and radius r the arithmetic intensity of an isotropic kernel is received by dividing the total amount of bytes loaded including the halo region by the amount of points within the volume:

$$AI_{iso} = \frac{(4 * (4 * X * Y * Z + 2 * r * X * Y + 2 * r * Y * Z + 2 * r * Z * Y))}{(X * Y * Z)}$$

for single precision accuracy and four volumes required. Smaller volume sizes or bigger stencil radius' worsen the surface-to-volume ratio and therefore the arithmetic intensity of the kernel. According to the used kernel the amount of data requested from memory varies and the equation must be adjusted accordingly. Table 4.1 shows the number of volumes of size $X * Y * Z$ that must be accessed for a certain kernel per timestep.

Given the PDE for the isotropic implementation we have one pressure wavefield P at timestep t and the velocity field for each point. Additionally, the second-order in time PDE needs the last timestep $t-1$. Since the wavefield of timestep t will be reused in timestep $t+1$, data has to be preserved and cannot be overwritten. For that reason a third volume holds the results (P_{t+1}).

PDEs for VTI implementations require an auxiliary wavefield Q. The same second-order in time derivate scheme applies. Hence, two additional wavefields Q_{t-1} and Q_{t+1} are required. The Thomson parameters ϵ and δ consume another two volumes. Earth properties are constant for a point in the subsurface and are timestep independent.

	ISO	VTI	TTI
0	P(t-1)	P(t-1)	P(t-1)
1	P(t)	P(t)	P(t)
2	P(t+1)	P(t+1)	P(t+1)
3	Velocity	Velocity	Velocity
4		Q(t-1)	Q(t-1)
5		Q(t)	Q(t)
6		Q(t+1)	Q(t+1)
7		ϵ	ϵ
8		δ	δ
9			θ
10			ϕ
	4x	9x	11x

Table 4.1: Kernel Memory Requirements

This way only a single volume is required, which results in a total of 9 volumes for VTI. TTI kernels are adding two more non-timestep dependent values for each point - θ and ϕ.

Including the halo regions to the volumes the compulsory bytes per point on a 512^3 volume are 16.2 for ISO, 36.2 for VTI and 44.4 for TTI. By dividing the amount of floating-point operations by the number of bytes that are loaded per point the arithmetic intensity for ISO is 2.1, for VTI 1.46 and for TTI kernels about 18.
Table 4.2 summarizes the computational characteristics of the three RTM wave propagation kernel implementations.

Kernel(8th)	**ISO**	**VTI**	**TTI**
Points in stencil	25 (P only)	34 (P+Q)	217 (P+Q)
Points in time derivative	2 (P only)	4 (P+Q)	4 (P+Q)
Velocity,ϵ,δ,ϕ,θ	1	3	5
Total points accessed per stencil	27	38	224
Flops/Point	34,	53	804
Compulsory Bytes/Point (512^3)	16.2	36.2	44.4
Arithmetic Intensity	2.1	1.46	18

Table 4.2: Characteristics of ISO, VTI and TTI Wave Equation Implementations

4.2.2 Memory Bandwidth Ceiling

A kernel is memory bandwidth bound if data cannot be delivered to the functional units at the speed the data is requested. The maximum achievable memory throughput is defined by the pin DRAM bandwidth, which is a product of several sources of parallelism: bit-level, memory channels per chip, number of chips, and multiple ranks of the memory module. Raw pin bandwidth can only be reached if hardware and software are able to exploit all architectural features and data access streams through consecutive addresses. The metric of memory bandwidth is billion bytes transferred per second (GB/s).

Memory systems are unlikely to achieve full theoretical memory bandwidth. To determine a realistic upper bandwidth boundary, performance diminishing effects need to be explicitly measured. The *STREAM* [110] benchmark provides simple kernels with varying memory access schemes. This thesis applies a modified version of the STREAM benchmark [109], which adds a simple dot product kernel that delivers peak memory bandwidth results due to exclusive read accesses to memory.

Leveraging the modified version of STREAM should give a good estimate how efficient peak bandwidth can be exploited for a given architecture if full spatial and temporal locality exists. Spatial locality describes the ratio of transferred bytes from memory per requested bytes from the functional units. As the smallest unit of data transferred from memory is one cache line, perfect spatial locality is achieved if all bytes within the cache line are used. Through the use of caches these bytes do not have to be used right away but could be accessed in later computations exploiting temporal locality. Optimal temporal locality occurs only for kernels that stream through memory addresses in the fastest unit stride. The memory bandwidth performance ceiling for a kernel is defined by a product of exploited parallelism through hardware and data locality provided by software.

All discussed architectures are using caches to exploit temporal recurrences. Considering a well optimized cache line replacement policy all points for a stencil computation exist in the lower cache hierarchy, due to temporal locality, and are not requested separately from main memory. The streaming nature of the explicit seismic kernels requires to load a fixed amount of points from memory equivalent to the number of volumes presented in Table 4.1 of Section 2.3.6, plus a halo overhead if implicit halo region

	Carver	NVIDIA M2090	Hopper	Sandy Bridge
ISO	2,065	9,375	2,515	4,119
VTI	904	4,166	1,134	1,778
TTI	687	3,400	822	1,342

Table 4.3: Peak Performance in MPoints/s based on Memory Bandwidth as only Limitation

exchange through main memory is applied. Its size depends on the surface-to-volume ratio of the node volume.
Working with a volume size of 512^3, Table 4.3 presents the maximal performance numbers achievable by the system based on memory bandwidth limitations. These numbers are derived by dividing the maximum memory bandwidth attained with STREAM, by the amount of bytes required to load from memory for each kernel type:

$$\text{Throughput} = \text{Bandwidth} \ / \ \text{Bytes per Points} \tag{4.1}$$

The presented STREAM bandwidth in Table 8.1 is the peak read bandwidth that is normally significantly higher than the write bandwidth. Hence, the performance limitations through main memory bandwidth are a rough estimates only.

The results show that the Sandy Bridge node should be able to deliver about 4100 MPoints/s for ISO, 1800 MPoints/s for VTI and 1350 MPoints/s for TTI, which is twice the performance of Carver nodes. Due to the high-performance GDDR5 used by NVIDIA's M2090 the GPU should be able to deliver more than twice the performance of a Sandy Bridge node based on memory bandwidth limitations.

4.2.3 Floating-Point Throughput Ceiling

For kernels with higher arithmetic intensity the compulsory floating-point operations per transferred byte from main memory increases. Such kernels are unlikely to be bound by memory bandwidth and might achieve peak floating-point throughput. The floating-point throughput ceiling is defined by a product of parallelism and frequency. Parallelism exists on data-level, by SIMD vector units, instruction level through FMA and through multiple cores. Peak throughput can only be achieved if perfect data locality is guaranteed, which means that neither hardware nor compiler is deficient in exploiting all architectural capabilities — prefetching, cache policies and etc.

Additionally, peak Gflops/s depend on the applied floating-point accuracy. As the same floating-point units are utilized for 32-bit single-precision computations and 64-bit double-precision operations, double-precision peak performance is one half of its 32-bit counterpart. Further performance studies are based on single-precision floating-point accuracy.

For a typical Intel x86-based machine with SSE instruction set, this results in one multiply-add issued in each 4-way SIMD slot per cycle. This example would provide a maximum floating-point throughput of:

$$\text{FMA} \times \text{Vector length} \times \text{Clockrate} \times \text{Number of Cores} \tag{4.2}$$

To analyze if a kernel reached peak performance it is necessary to know the number of floating-point operations executed per point in the volume. Counting the number of floating-point operations can either be done by hand for small kernels or by using hardware counters like *The Performance API* (PAPI) [42]. The achieved floating-point throughput is then derived by dividing the total amount of flops for all points of the volume by the execution time. This time can then be compared to the peak performance to see if the floating-point throughput ceiling is reached.

Kernel performance analysis can be challenge due to the high amount of metrics and performance numbers. To make interpretation easier [109] introduced a novel way to combine all metrics into the so-called *Roofline Model*.

4.2.4 The Roofline Model

Analyzing kernel performance for different architectures can be difficult. Many different metrics and performance ceilings need to be combined for interpretations. In 2008 the Roofline Model[109] was introduced by Sam Williams. It provides a visually-intuitive, throughput-oriented diagram for an architecture that

> [...] allows a programmer to model, predict, and analyze an individual kernel's performance given an architecture's communication and computation capabilities and the kernel's arithmetic intensity.

Throughput performance depends on computational intensity and data movement from external storage like DRAM to computational resource like CPUs. The Roofline Model

uses the bound and bottleneck analysis from [50] and determines the attainable kernel performance on a certain architecture according to Equation 4.3.

$$\text{Attainable Performance} = \min \begin{cases} \text{Peak Performance} \\ \text{Peak Bandwidth} \times \text{Arithmetic Intensity} \end{cases} \tag{4.3}$$

Hence, the model draws the arithmetic intensity as ratio of compulsory floating-point operations to the total DRAM memory traffic (flop/byte) on the x-axis and maximum floating-point throughput in Gflops/s for single precision accuracy on the y-axis. Both are plotted in a log-log scale.
Moving from left to right on the x-axis increased the arithmetic intensity which means that less bytes per flop are required - or more flops are performed per accesses byte from main memory. That way the pressure on memory bandwidth is decreased. An improved temporal locality of data improves the flop-to-byte ratio and accordingly the achievable peak performance. Hence, given a certain memory bandwidth (peak, sustained) the maximum achievable performance draws a diagonal *bandwidth ceiling*. If a kernel is not limited by memory bandwidth, it is theoretically possible to reach peak floating-point performance shown as a horizontal, so-called, *in-core ceiling*.

Figure 4.2 presents kernel performances based on their arithmetic intensity and architecture capabilities. One can clearly see the imbalance of architectures between instruction performance and bandwidth. For Opteron with 24 cores and for Sandy Bridge, using 16 cores but 256-bit wide SIMD instructions, peak performance can only be achieved by a limited number of algorithms. The Roofline Model shows that ISO and VTI kernels are likely to be memory bandwidth bound and TTI instruction bound for all architectures. GPUs like the M2090 benefit from high memory bandwidth provided by GDDR5 and great single-precision peak performance. Since GPUs are known for their big difference between peak and sustained and peak performance estimation should be taken with care. The kernel performance in Gflops/s relates to MPoints/s with:

$$\text{MPoints/s} = \text{Gflops/s} \ / \ \text{Flops per Point} \tag{4.4}$$

where the number of floating-points operations per points depends on the propagation kernel.

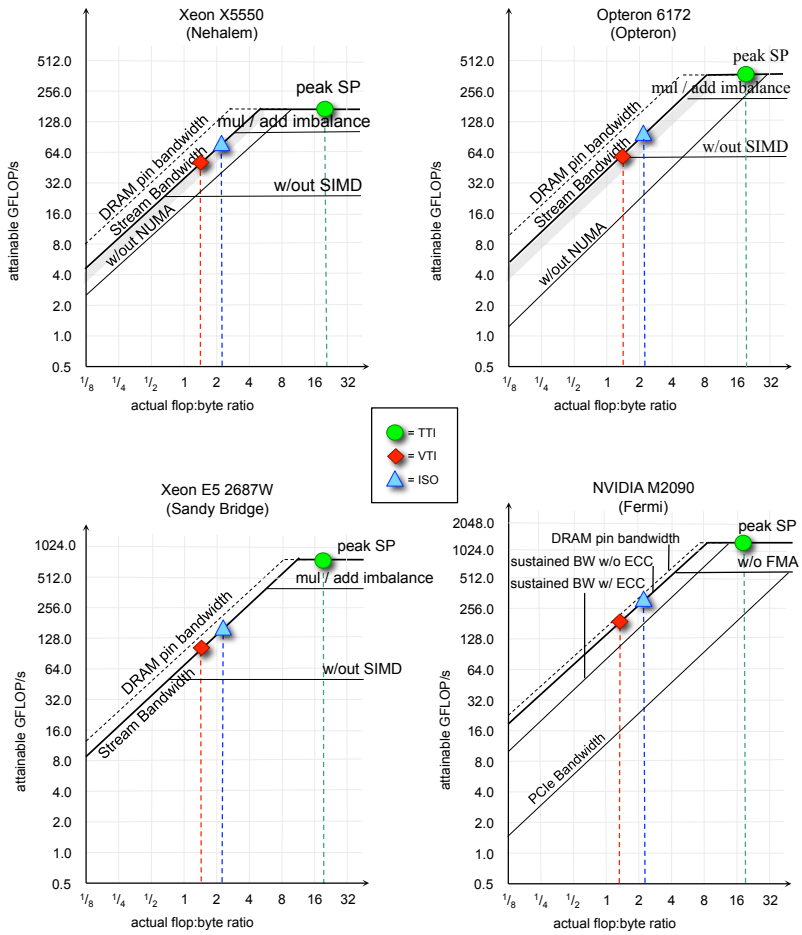


Figure 4.2: The Roofline Model [109]

4.2.5 Summary

The previous section introduced the Roofline Model and showed that ISO and VTI kernels are likely to be memory bound and TTI kernel reference implementations instruction bound for all architectures. Consequently, estimated throughput numbers for ISO and VTI are defined by the memory bandwidth ceiling and TTI kernels by the in-core ceiling.

Table 4.4 summarizes the estimated performance for all architectures in MPoints/s. The reader must be aware that these numbers show approximated system peak performance and only appear if all hardware and software capabilities can be fully exploited.

	Carver	NVidia M2090	Hopper	Sandy Bridge
ISO	2,000	9,000	2,500	4,000
VTI	900	4,000	1,100	1,800
TTI	200	1,600	500	1,000

Table 4.4: Estimated Performance in MPoints/s

4.3 Reference Kernel Benchmark Results

So far this chapter derived peak performance estimations for all architectures. This section now benchmarks the reference kernel implementations. To utilize all cores on a node the node domains decomposed. The estimates should used to quantify the achieved throughput attained in this section.

The following Section 4.3.1 introduces the applied domain decomposition approach, followed by the presentation of the benchmark results of the reference kernel implementations in Section 4.1.

4.3.1 Domain Decomposition

The kernel reference codes are utilizing all cores on the evaluated architectures. As no task-level parallelism exists data-level parallelism is exploited by domain decomposing the node volume into multiple subdomains. Each subdomain is then allocated to a thread. One of the main advantages of explicit solutions is that all points can be processed independently from each other, which makes domain decomposition a relatively easy task. The actual strategy to domain decompose the volume depends on the underlying compute architecture and the parallelized kernel itself.
There are several levels of domain decomposition related to the architecture. The most coarse grained decomposition is the allocating of subdomains for each compute node. This kind of domain decomposition is done if multiple nodes are used to process the volume. The current study is done as single-node comparison only which removes this level of domain decomposition.

The next levels of domain decomposition take place within the node itself, which involves NUMA nodes, and finally, single cores within the NUMA node. Explicit data exchange between distributed memory systems is applied between nodes. This results in an increased latency and reduced bandwidth. To avoid stalls of computation units domain decomposition on NUMA-level is applied in slowest unit-stride direction.

Except for multi-GPU node setups intra-node data exchange is done via implicit communication through main memory accesses. That way latencies decrease and bandwidth increases compared to inter-node data exchange but makes fine-grained synchronization necessary. Furthermore it is feasible to domain decompose in the second fastest unit-stride direction for cores.

For x86-based architectures domain decomposition on node- and NUMA-level is applied in z-direction and in y-direction to exploit data parallelism on core-level. A complete overview of the domain decomposition and how it is implemented on x86-based clusters like Carver (Nehalem) and Hopper (Magny Cours) is presented by Figure 4.3

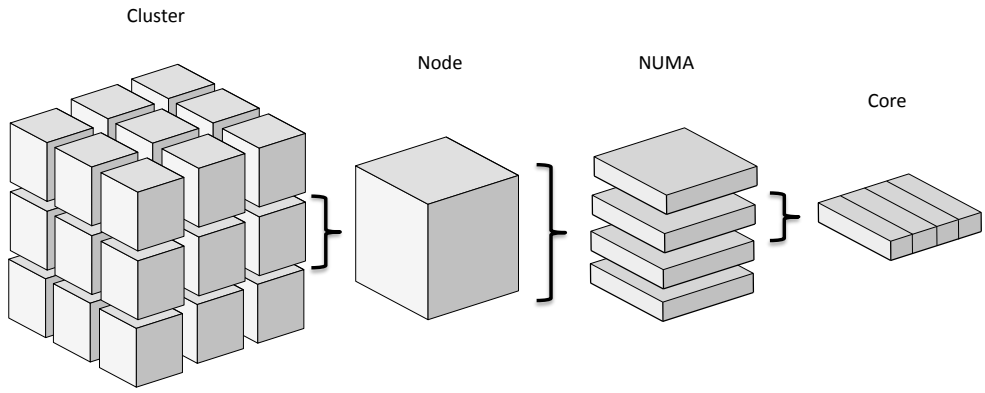
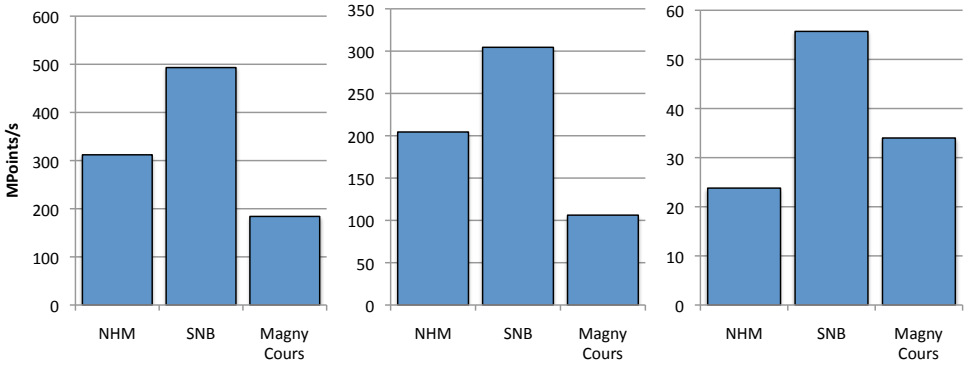


Figure 4.3: The Domain Decomposition on Carver and Hopper

4.3.2 Benchmark Results

This section presents the benchmark results of the reference kernel implementations on all architectures. For ISO and VTI kernels the Nehalem node attains 300 and 200 MPoints/s respectively, whereas the Magny Cours based node sees poor performance of only 170 to 100 MPoints/s. The 24 cores on the Magny Cours node are receiving only half the performance of the Nehalem node for ISO and VTI. Contrary, for TTI Magny Cours provides 1.4x the performance of the Nehalem because of its higher number of cores. Intel's leading-edge Sandy Bridge Xeon E5-2687W outperforms both older architectures by up to 3x for ISO and VTI and 2.4x for TTI. Figure 4.4 presents the achieved throughput numbers.



(a) ISO (b) VTI (c) TTI

Figure 4.4: Reference Single Node Throughput Comparison

4.4 Conclusions

This chapter derived the theoretical performance limits for ISO, VTI and TTI wave propagation kernels for all evaluated COTS architectures. As the plurality of performance metrics makes straight forward interpretation difficult this chapter leveraged the Roofline Model as visually-intuitive way to analyze kernel performance.

Concluding from the estimated maximum performance numbers in Section 4.2.2 and 4.2.3, there is high potential in optimizing the code for the specific architectures to gain better performance. In order to provide a fair comparison between architectures software optimization potential must be exploited. Therefore, Chapter 5 now introduces general and kernel-specific optimization techniques and discusses their effects on throughput and energy efficiency.

Chapter 5

Limits on Performance and Efficiency using COTS Hardware

Comparing the theoretical achievable peak performance estimations of the three seismic wave propagation kernels to the achieved performance of the reference kernel implementations in Chapter 3, shows that, hardware-specific software optimization provides high potential for additional performance gains. It is even a necessity to guarantee a fair comparison of performance and energy efficiency between architectures. This chapter introduces different optimization techniques and analyzes their benefits on kernel performance. It extends the Roofline Model to achieve more accurate performance predictions and provides single- and multi-node benchmarks on the evaluated commercial off-the-shelf (COTS) architectures . Finally, this chapter presents power consumption estimates for complete cluster setups.

Optimization of software has always been an important factor in high performance computing to achieve best performance. Until the mid 2000's the general strategy of the main manufactures like Intel and AMD was to increase the frequency of the cores to improve instruction throughput. The introduction of pipelines, hardware prefetching and a increasingly deeper cache hierarchy to account for rising clock frequencies gave software developers plug-n-play performance improvements. This development ended rapidly with the Pentium4, which experienced enormous problems in heat dissipation caused by a complex chip design necessary to hide memory latencies for high clock frequencies of up to 4 GHz. In order to be able to keep increasing CPU speeds, the industry started using multiple cores per socket. This way it was possible to keep

the transistor growth and performance improvements according to Moore's Law [59]. But the development of multi-core systems made architecture-specific optimizations even more important, because plug-n-play performance increases, like seen in the past decades were not possible anymore. Today's high performance kernels not only have to account for on-chip instruction- and data-level parallelism, but for multiple cores and *Non-Uniform Memory Architecture* (NUMA) properties. Therefore, optimization of applications to the underlying hardware is critical if a maximum speed and energy efficiency should be achieved.

In general one can distinguish between different levels of the optimizations. Low-level optimizations work on concurrent processing of multiple bits, the smallest unit in a system. Over the year the bit level rose from 8Bit to 64Bit today. On x86-based architecture and GPUs, programmers have no influence on bit-level parallelism. Only on FPGA's it is critical to reduce the bit-level accuracy to a minimum to avoid a waste of area. Second is instruction-level parallelism: It exploits possible instruction calculation concurrency between independent instructions. Next follows leveraging of data independencies like vectorization. Independent data values can be executed in parallel via SIMD or VLIW. The two highest levels are core-, and for computing clusters, node-level parallelism. With the introduction of multi- and many-core system architectures the core-level parallelism gained significant importance. It describes how data or tasks can be parallelized using multiple cores on a SMP. In general, node-level describes the parallel execution of tasks, data or both on distributed memory systems. Node-level parallelism is present since the early days of high performance computing and always had significant importance, but is not further discussed in this single-node study. On all optimization levels data locality plays an important role. Every time data is not present when needed additional stalls appear.

The following sections first introduce all applied general optimizations followed by kernel specific optimizations in Section 5.4. The performance of the optimized kernel implementations is then benchmarked and analyzed in a single-node environment 5.7. Since large shot sizes, like the example production-sized survey introduced by Section 2.5, require multiple nodes, Section 5.9 discusses different multi-node communication implementation and optimization approaches, and analyzes their impact on performance.

One of the main challenges for large survey sizes, imaged with highly accurate migration methods like RTM accounting for anisotropy, is the computational complexity of the kernel and to meet runtimes that make high-quality RTM applicable to production. One way to achieve such runtimes is to leverage data independencies e.g. on shot-level. As initial implementation costs for hardware of large-scale compute clusters are diminished by their maintenance costs, especially the energy bill, it is important to compare the power consumption of the evaluated architectures in large cluster setups. A power consumption estimate for migrating the example survey over a fixed time-to-solution of one week is presented in Section 5.10.

5.1 Data-Level Parallelism

There are two main goals for data-level optimizations: data reuse and data locality. Data locality optimizations target at keeping frequently used data as close to the functional units as possible. Unfortunately, registers, which are located closest to the functional units, provide the lowest amount of capacity. The next level in the memory hierarchy is either a level 1 cache or a local store. Both provide superior bandwidth and latency, but also limited capacities. AMD Opteron provides 64 KB, Intel cores 32 KB L1 caches - IBM Cell 256 KB local stores per SPE. The reverse relation between speed and capacity continues through the whole memory hierarchy on x86 architectures.
On GPUs a somewhat inverted memory hierarchy is implemented in which the register file is larger than the L1/shared memory, which again provides more capacity per core than the shared L2 cache. This is required due to latency hiding through thread switching. As GPUs hide memory latencies through switching to a different thread, the thread's register file is kept in shared memory to enable zero-latency switching. The high amount of threads can especially lead to capacity problems within the L1/shared memory for large kernels. Therefore, it is very important to decide which data should reside in registers and low-level caches and local stores, respectively, and which data should not. Examples for data locality optimizations introduced in this section are memory pinning 5.1.1, register blocking 5.1.2, cache blocking 5.1.3, cache bypass instructions 5.1.4 and to a certain extent CSE 5.4.2 which improves data reuse through keeping already computed data in the low-level memories. CSE incorporates or can incorporate all other data-level optimizations. Its data reuse optimizations are important

to minimize the data read into low-level memories as well as for reduction of redundant calculations.
Data locality and data reuse optimizations reduce the pressure on memory bandwidth. Such optimization becomes more and more important on modern architectures since the performance gap between floating-point performance and memory bandwidth keeps expanding. This gap results from memory bandwidths not keeping up with the growth of instruction throughput performance. Therefore, memories are not able to deliver the number of bytes requested by the instruction units - described as being oversubscribed in instruction performance. A very important first optimization to reduce memory access latencies is memory pinning.

5.1.1 Memory Pinning

Modern SMPs use *Non-Uniform Memory Architecture* (NUMA), which means that each socket within the SMP has its own memory controller and attached memory. Figure 5.1 shows such a SMP NUMA setup. For such architecture the latency to get data from memory for a socket depends on data locality. If a socket tries to access data from a remote memory, data has to be transferred through the main bus system, and access latency is significantly higher compared to local requests. Therefore, to leverage optimal memory bandwidth it is critical to place required data NUMA socket specific. Data placement is applied by which socket first touches a specific memory address - the so-called memory pinning through "first-touch policy". Therefore, initialization of memory has to be done by each NUMA socket separately.

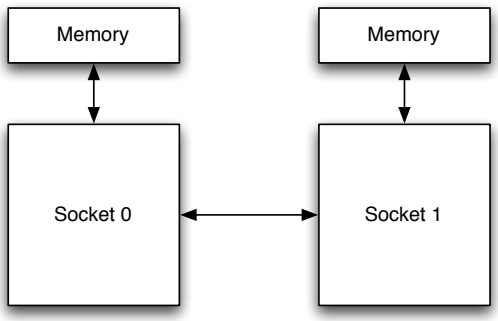


Figure 5.1: A Simple Example for a Non-Uniform Memory Architecture (NUMA))

5.1.2 Register Blocking

The lowest level data optimization technique is *Register Blocking*. Register blocking is used for the innermost kernel loops that are extended for optimal utilization of all available register without oversubscribing them. The way blocking is applied depends on the size of the register file, hence on the underlying architecture. For simple kernels unrolling in several dimensions could lead to a maximum register file use and to a maximum number of independent instructions. An optimal use of registers, leads to best possible temporal and spatial locality whereas, oversubscription of registers leads to register spilling to the stack. In this case, required, but not available registers are offloaded to the stack and reloaded on demand. Hence cache accesses have a higher latency than register accesses, it causes pipeline stalls and is diminishing overall performance. A simple version of register blocking is loop unrolling (see Section 5.2.1). To optimize the register usage for the evaluated seismic kernels a register rotation scheme is implemented. It targets at keeping data, required for subsequent iterations, in the register file to avoid redundant loads from L1. Specifically, $2 \times r + 1$ values are required along the fastest unit-stride for the stencil operations. As the stencil operation moves to the subsequent point $2 \times r$ points are kept in registers and only a single point needs to be loaded - a $2 \times r + 1$ register rotation along the fastest unit-stride.

5.1.3 Cache Blocking

Large working sets can exceed cache size and result in constant cache misses, which again leads to decreased performance caused by increased memory access latency. *Cache Blocking* enables the most efficient use from the cache hierarchy by creating smaller working sets that fit into the desired cache level. Commonly, cache blocking is applied by alternating the order of loops. But, all cache hierarchy levels need different optimization techniques. Each core has a dedicated L1 cache, which is the fastest available cache with the lowest latency to the register file. Unfortunately, L1 caches have low capacity with typical sizes do not exceed 32 to 64 KB. L1 optimization can be difficult and usually focusing on data reduction of within the inners kernels.
L2 caches are usually dedicated to a core and are much larger than L1's. Its usual size is about 256 KB or 512 KB. The programmer needs to analyze the structure of the kernel, to decide whether it's beneficial to apply a cache blocking size that fits into L2

or L3.

A different approach is taken for GPUs. GPUs have a shared L2 cache. Therefore, GPU L2 optimizations are somewhat similar to L3 optimizations on x86 architectures. Higher-level caches like L3 are most commonly shared by several cores (3.2.2, 3.2.1, 3.2.3) and much larger in size. Modern Sandy Bridge servers, like Xeon E5-2687W, have 20 MB of L3 cache. To optimize for L3 the total volume is decomposed further. The actual size of cache blocks depends on the working set size of the actual kernel. The reference kernel implementations decompose the node domain in the z-dimension for NUMA sockets and assigns subdomains for all cores in y-direction within the NUMA socket. The requisite cache working set size scales as $O(X \times Y/nthreads_{NUMA})$. As ISO, VTI, and TTI each operate on multiple arrays, the requisite cache capacity quickly exceeds the cache on any of these machines if the plane size of the static domain decomposition in Y exceeds L3 capacity. The impact is that a large number of capacity misses squander memory bandwidth. The solution is to simultaneously parallelize and create smaller cache blocks (BX, BY). The code is restructured to operate on cache blocks of size $X \times 8$ for Nehalem, $X \times 16$ on Sandy Bridge and $X \times 11$ for Magny Cours. These blocks are parallelized across the cores in a round robin fashion on the compute node. This way the total working set size is reduced to fit the L3 and enables synergistic memory loads from main memory because of shared halo regions between core subdomains. Parallelization among cores in the X-dimension is particularly disadvantageous on CPUs as CPUs rely on hardware stream prefetchers to hide memory latency. Hardware stream prefetchers demand long (multi-KB) unit-stride streams. These long unit-stride streams can only be realized if there is no blocking or thread parallelization in the unit-stride. Such cache blocks march in z-direction creating a pencil. The size of the pencil depends on the number of cores sharing the L3. In the case of Nehalem four cores share the L3 and BY is 8, which creates a pencil size of $X \times 32 \times Z$. An additional loop iterates over all Y/BY pencils to cover all points in the subdomain. Figure 5.2 gives a simple example for L3 cache blocking. It assumes that the volume has x as fastest, z as slowest direction and four cores have a shared L3 cache. Further, a static domain decomposition in y would let the working set exceed L3 capacity. L3 cache blocking makes working sets fit into L3 and enables synergistic memory loads for halo zone points. In this example core 0 loads its data first. Since core 1 needs points from core 0's subdomain these points will generate cache hits in the L3 cache. The

same happens for all cores such that a significant reduction in memory bandwidth and increase of performance is achieved.

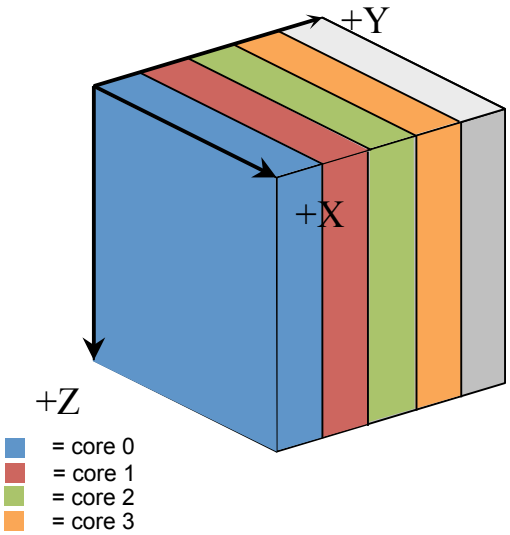


Figure 5.2: Cache blocking for efficient use of L3 cache.

5.1.4 Cache Bypass

Data accessed in main memory is replicated through all cache hierarchy levels for eventual future reuse. But not all data accessed is reused again. The time data sits in the caches and is not used depends on the cache replacement policy and associativity. Each unused cache line eats up cache space that could be used for data that is more important. With special cache bypass instructions the programmer can define explicitly, that data should bypass the complete cache hierarchy. That way it is not being stored and replicated within the different cache levels. The use of such instructions should be made if e.g. a final value is written back to main memory and is not used for a while, or for streaming instruction, which use a certain value only once. This reduces littering of caches and avoids useful data from being replaced with data that is not used in future. Cache bypass instructions were introduced with Intel's SSE 1 instruction set as "streaming" instruction. Examples are `_mm_stream_ps(float *p, __m128 a)` and `_mm_stream_load_si128(__m128i* v1)` which are cache bypassing store and load instruction, respectively.

5.1.5 Translation Lookaside Buffer (TLB) Optimization

The reference implementation uses three arrays to store wavefield data for the previous T_{t-1}, current T_t and next timestep T_{t+1}. The second-order in time stencil scheme requires only two arrays for the actual computation. Since the value of timestep T_{t-1} is accessed only once per timestep, during computation of the corresponding grid point in timestep T_t, the values of array T_{t-1} can be overwritten with the results. This leads to decreased memory capacity requirements and less virtual to physical address translations as only two instead of three pointer swaps are performed after each timestep. Table 5.1 presents the updated number of volumes required for the specific kernels.

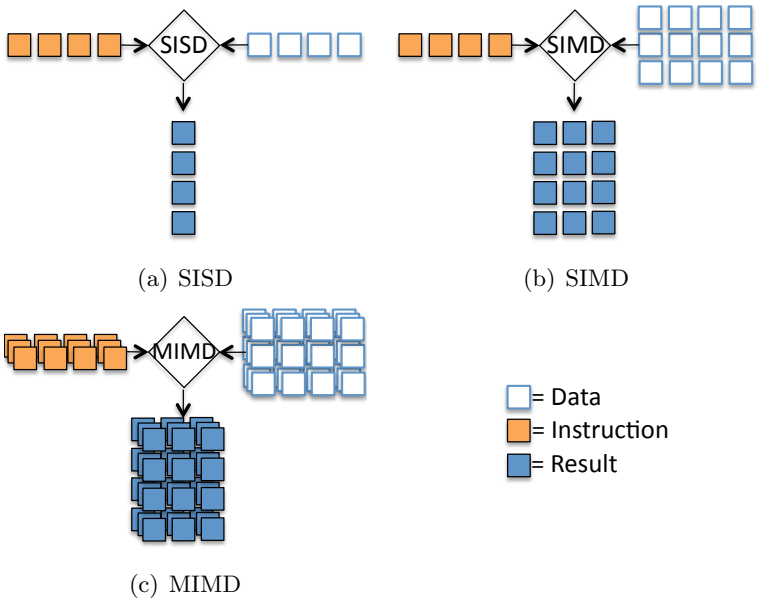
5.1.6 Single Instruction Multiple Data (SIMD)

Data-level parallelism tries to take advantage of independent data for parallel execution. To leverage on-chip data parallelism vectors are used, which means that the same operation is executed on all vector entries. Before 1970 computers used a *Single Instruction Single Data* (SISD) (see Figure 5.8(a)) approach. One instruction was executed for one data value at a time. In the early 1970's Texas Instruments developed the TI-ASC and Control Data Corporation the STAR-100 vector supercomputer [26]. Both took streams of operands from the memory, performed one instruction and wrote the results back to memory. As *Complex Instruction Set Computers* (CISC) one instruction could be e.g. a matrix-vector multiplication.

So far x86 architecture did not use vector operations. At the mid 1990's multimedia applications became more important for the computer industry. First introduced with Intel's Multimedia Extension (MMX) in 1996 and later followed by AMD's 3Dnow! in 1998, manufactures added multimedia ISA extensions to exploit data-level parallelism by executing a single instruction on multiple data items. To make the vector registers compatible to x86 design the vector length had to be shrunk to 64bits, which enabled to fit vector register into microprocessors without compromising other critical components [26].

To the current date all x86 cores support SIMD instructions of 128-bit width. Therefore, four 32-bit or two 64-bit values can be computed at a time (see Figure 5.8(b)). The *Advanced Vector Extensions* (AVX) proposed by Intel in 2008 and introduced for Intel's Sandy Bridge 3.2.3 and AMD's Bulldozer architecture, increases the bit-width

to 256 bits or eight 32-bit wide words. The not-yet released Intel *Many Integrated Core Architecture* (MIC) 3.3.1 accelerator card uses an even wider vector length of 16 32-bit single-precision floating-point values or 512 bits in total.
An extension of the SIMD architecture is *Multiple Instruction Multiple Data* (MIMD) , which can be seen as multiple cores able to process multiple data elements at the same time - such as multi-core CPUs utilizing SIMD units (see Figure 5.3(c)).
For GPUs sometimes *Single Instruction Multiple Thread* (SIMT) is mentioned, which relates to multiple threads applying a single instruction. The number of threads is usually higher than the number of cores to hide latencies 3.2.4. Therefore, these threads are worked through in a mixed parallel and sequential fashion.



(a) SISD (b) SIMD

(c) MIMD

Figure 5.3: Vectorization Approaches

Even though modern compilers are able to SIMDize small loops for complex loop bodies the programmer has to add SIMD specific intrinsics by hand. To leverage the full potential of SSE instructions all data accesses should be cache line aligned to avoid accessing multiple cache lines and data shifting.

Using SSE and AVX instructions for the evaluated architectures is critical to achieve maximum performance. Applying a 2nd-order in time explicit solver for the wave equation kernel, all grid points depend only on the previous and current timestep. All points

can be calculated independent from each other if all required points for the stencil operation are present. Compilers are notoriously challenged to effectively exploit SIMD instructions for complex loop bodies how they appear in TTI kernels. For these single-precision calculations it is a must to ensure 4-way SSE SIMDization and 8-way AVX SIMDization. To maximize performance the operations within a stencil are grouped and mapped to SIMD intrinsics. To facilitate high-performance SSE code, arrays are aligned to 64Byte and no cache line boundaries are crossed which would cause unnecessary pressure on the memory bandwidth. To assure alignment for all memory accesses, additional padding is added for all rows. The size of the padding depends on the order of the stencil scheme in space, row length and the cache line size of the underlying architecture. However, high-order derivatives in the unit-stride (x-dimension) are particularly challenging to SIMDize as they require 8 unaligned accesses per stencil. To minimize this effect, the code is modified to maintain locality within the register file and use shuffle intrinsics to create vectors containing the unaligned data values. This significantly reduces the number of accesses to the L1. Figure 5.4 visualizes in 2D how the SSE and AVX instructions are applied.

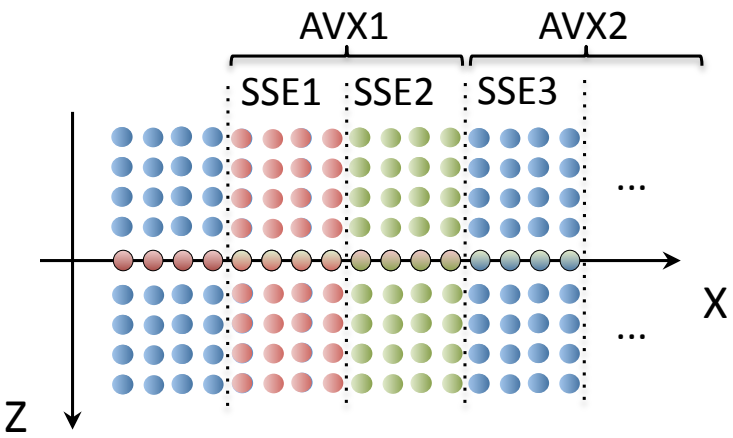


Figure 5.4: A 2D Visualization of SSE and AVX Vectorization

The shuffling is done by SSE/AVX provided shuffle instructions like `palignr` (called by `_mm_alignr_epi8`) which concatenates two 128-bit vectors into a 256bit vector and rotates the contents by a given constant. The lower 128 bits are then extracted for the result. This instruction was first introduced with the SSSE3 extension set. Unfortunately, AMD's Magny Cours cores do not support this extension.

For Magny Cours the `shufps` (called by `_mm_shuffle_ps`) and `movss` (called by `_mm_move_ss` from SSE2 are used. The first instruction takes two 128-bit vectors and shuffles them according to a bit mask provided, whereas `movss` extracts the lowest 32 bits of the second vector and bits 32 to 127 from the first vector to form the result.

5.2 Instruction-Level Parallelism

Instruction-level parallelism optimizations are focusing on supporting the compiler to scheduling assembly instructions most efficiently. Instructions need several cycles until the result is available. The five basic pipeline stages of a simple example pipeline are instruction fetch (IF) which reads the instruction from the instruction cache, the instruction decode (ID) stage, the execution if the instruction (EX), the memory access to access required data in a register (MEM) and the write back (WB) stage which writes the result back to a register. An instruction cannot be executed if the required data is not present in a register yet. This can either be the case because a cache miss appeared and it wasn't able to load the data into a register ahead of time or the instruction requires a result from a previous instruction. In this case a pipeline stall would appear and cycles would be wasted. The compiler's task is to find independent instructions that can be issued to the pipeline while the stalled instruction needs to wait for data. The more independent instructions are now available the better stalls can be hidden and hence, the better the pipeline usage.
An approach to leverage instruction level parallelism is *Very Long Instruction Word* (VLIW) . With a dual issue pipeline it is possible to execute e.g. write, read instruction and arithmetic instruction at the same time. An example is the IBM Cell 3.3.3. A common example how the programmer can support the compiler to find more independent instructions is by unrolling of loops.

5.2.1 Loop Unrolling

Loop unrolling describes a register blocking (see Section 5.1.2) optimization technique in which the body of the innermost loop is replicated for a certain amount of independent iterations to enable more efficient use of the pipelines and registers. This can minimize the effects of loop overhead and eliminate stalls of the pipeline caused by dependencies between consecutive instructions.

Loop unrolling can be performed best if each iteration is completely independent from each other and instructions can be reordered by the compiler or programmer to satisfy all pipeline stages and registers. Even modern compilers are not always able to identify the independence of consecutive loop iterations especially for large loop bodies and non-affine memory access patterns. Examples are sparse matrix computations. Often such matrices are stored in a compact fashion that stores non-zero elements only. The mapping from sparse matrix to compact array is done by an auxiliary index array, which entries determine the array index touched. For such complex loop bodies loop unrolling might not be performed efficiently and the programmer needs to unroll loops and reorder instructions manually. The reordering of instructions helps the compiler to find more independent instructions within his look-ahead window range. Loop unrolling can only be beneficial if the register file is big enough to hold all necessary data elements. Especially, for architectures like the IBM Cell processor with its large register file it makes sense to heavily unroll inner loops to leverage all 128 entries. Figure 5.5 shows a simple example for loop unrolling and reordering.

```
for(i=0; i<=n; i+=1)
{
    statementA(i+0);
    statementB(i+0);
    statementC(i+0);
}
```

(not unrolled)

```
for(i=0; i<=n; i+=2)
{
    statementA(i+0);
    statementB(i+0);
    statementC(i+0);
    statementA(i+1);
    statementB(i+1);
    statementC(i+1);
}
```

(unrolled)

```
for(i=0; i<=n; i+=2)
{
    statementA(i+0);
    statementA(i+1);
    statementB(i+0);
    statementB(i+1);
    statementC(i+0);
    statementC(i+1);
}
```

(unrolled + reordered)

Figure 5.5: Loop Unrolling and Reordering

5.3 Thread-Level Parallelism

Thread-level parallelism describes the concurrent work of multiple threads on a SMP. Most commonly used are implementations with POSIX threads (pThreads) or OpenMP [64]. With POSIX threads each thread has to be created and bound to a core individually. The correct implementation of synchronization, communication and the handling

of global counter can be challenging and is done by using mutual exclusions (mutexes) to provide access to a memory location for only one thread at a time and avoid race conditions. The implementation overhead of using threads is significantly reduced with the OpenMP library that uses simple pragmas to give the compiler a hint what loops should be parallelized. Figure 5.6 presents a simple example how pragmas are used to parallelize "for- loops". The number of threads has to be set via an environment variable and cannot be adjusted dynamically during runtime. Pthreads and OpenMP are the most commonly used alternatives to leverage thread-level parallelism. However, there are different thread library implementations or frameworks as e.g. Intel Cilk Plus [43], which automatically handles load-balancing and adjusts the number of threads called to the number of available threads on the system, or MCTP [28] as a Fraunhofer ITWM development. In the following this section introduces task-parallelism and domain decomposition to leverage concurrency, and analyzes if and how it is applicable to the wave-equation kernel.

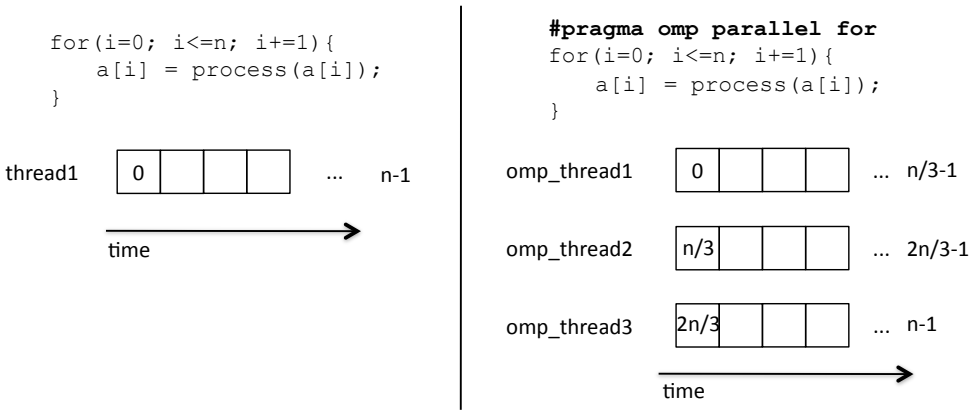


Figure 5.6: OpenMP Example

5.3.1 Task Parallelism

Task parallelism can be used if data is processed by different independent kernels, which are applied consecutively in time. There exist several different implementations. Figure 5.7 shows a task parallelization in a pipelined fashion. This way each process gets a specific kernel, which has to be applied. Each process gets as input the output of its predecessor and sends its own output to the subsequent process until all process stages are completed and data can be written back to disk or memory. Task parallelization

implementations are rarely seen in the HPC community because each task would need an almost identical runtime to avoid load-balancing problems. One exception are FPGAs, which combine task-, data- and instruction-level parallelism to form a multistage pipelines (see Section 3.3.2).

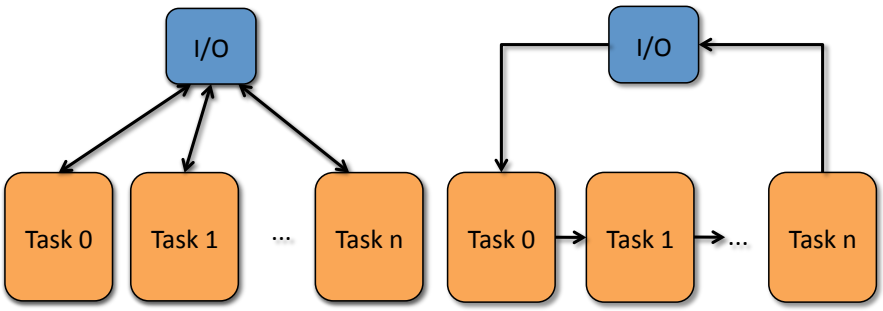


Figure 5.7: Two approaches for task parallelization

Task parallelism is not applicable to wave propagation kernel since only a single kernel is applied to a large dataset. In such cases concurrency is achieved by exploiting data level parallelism.

5.3.2 Domain Decomposition

The choice of an explicit, finite-difference approximation of the wave propagation was done because of superior independence between data points, which enables exploiting data-level parallelism. Parallelization of data processing requires decomposition of the main volume into subdomains. This is done in several different ways. This section presents two of the most common implementations. The most naive version is presented by Figure 5.8 (left) in which a volume is divided into subvolumes which size depends on the number of parallel processes used. This kind of static decomposition can cause problems in load balancing in case of sparse data. Then one process might have only a few non-zero data points. Depending on the architecture zero data values are processed differently which could lead to a significant faster processing time than for dense data. Therefore, process 0 might finish its work early and has to wait for process 3. Further disadvantages of static data decomposition exist in oversubscribing L3 capacity pressure (see Section 5.1.3).

These problems are avoided by scheduling smaller subdomains to the processes. Figure

5.8 (right) presents such a case in which the subdomains are decomposed into tiles. A global counter keeps track how many tiles have been processed already, or are currently worked on, and each time a process requests a new tile the counter is read to receive a tile that hasn't been processed yet. Afterwards, an atomic operation increased the global tile counter by one. The smaller subdomain size and dynamic scheduling can compensate load-imbalances, improve data locality in L3 and lead to synergistic memory loads.

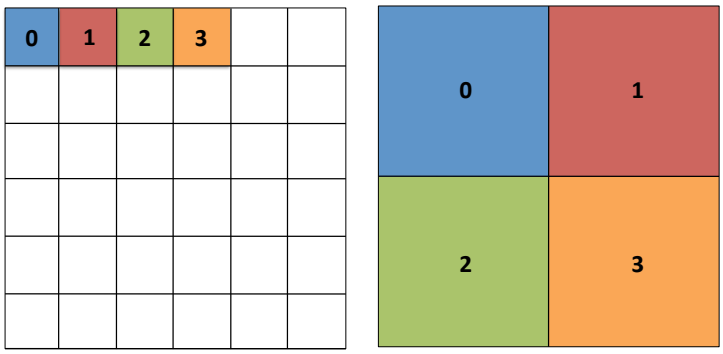


Figure 5.8: Two different implementations of 2D data decompositions for a 3D volume.

The kind of domain decomposition depends on the amount of data processed and the underlying architecture. For codes, like the discussed wave propagation kernels that rely on halo exchange through nearest neighbor communication the reduced volume-to-surface ratio might decrease computational throughput performance and increase the fraction of time spend in communication. Therefore, the domain decomposition scheme has to be chosen individually for each kernel and architecture.

All seismic imaging volumes are dense and no load-imbalance caused by data is expected. The reference domain decomposition uses a static decomposition, where decomposition in z is done between NUMA sockets and decomposition in y for cores on a NUMA socket. The NUMA subdomain is further decomposed to account for best data locality within the shared L3 - called cache blocking. The size of a subdomain depends on the size of the L3. Details about L3 cache blocking can be found in Section 5.1.3.

The domain decomposition scheme used for the reference seismic kernels was described in Section 4.3.1.

5.4 Kernel Specific Optimizations

Kernel specific optimizations are regarding the numerical kernels and its potential for algorithmic optimization. For algorithmic optimizations the underlying numeric kernel needs to be analyzed and eventually be replaced to get better performance or optimization capabilities. The choice of optimal numerical kernels for RTM was discussed in Chapter 2. As a result explicit finite-difference approximations for isotropic, VTI and TTI media were chosen, as it provides independence of data points.

5.4.1 Pre-Computation for TTI

The Roofline Model presented in Section 4.2.4 showed that flop-heavy TTI kernels are likely to be instruction bound rather than memory bound. Therefore, the main goal for optimization must be the reduction of performed floating-point operations per point. Table 4.1 summarized the number of volumes required for all three kernels. Taking a look at the TTI partial differential equation presented in Section 2.2.4 shows that values of volumes θ and ϕ are not timestep dependent and require the application of sine and cosine functions. Trigonometric functions are internally implemented as an iterative approximation that requires many floating-point operations. Since θ and ϕ are constant the sine and cosine computations can be applied in advance of entering the main wave propagation timestep loop. This way the amount of flops-per-point is reduced in expense of higher memory bandwidth pressure and additional memory capacity requirements. Table 5.1 adds *TTI Pre-Comp* and shows a summary of all necessary volumes for each kernel.

5.4.2 Common Subexpression Elimination (CSE)

Common Subexpression Elimination (CSE) leverages computational redundancy of the numeric kernel and was first introduced by John Cocke in 1970 [16]. Compilers commonly apply simple forms of CSE. Such a simple example is presented by the following two equations:

$$\begin{aligned} a &= c \times d + e \\ b &= c \times d + f \end{aligned}$$

	ISO	VTI	TTI	TTI Pre-Comp
0	P(t-1)	P(t-1)	P(t-1)	P(t-1)
1	P(t)	P(t)	P(t)	P(t)
2	(P(t+1))	(P(t+1))	(P(t+1))	(P(t+1))
3	Velocity	Velocity	Velocity	Velocity
4		Q(t-1)	Q(t-1)	Q(t-1)
5		Q(t)	Q(t)	Q(t)
6		(Q(t+1))	(Q(t+1))	(Q(t+1))
7		ϵ	ϵ	ϵ
8		δ	δ	δ
9			θ	$sin(\phi)$
10			ϕ	$cos(\phi)$
11				$sin(\theta)$
12				$cos(\theta)$
	3(4)x	7(9)x	9(11)x	11(13)x

Table 5.1: Kernel Memory Requirements

Both use the expression $c \times d$. By pre-calculating this expression and storing it in a temporary variable a floating-point operation can be saved in sacrifice for a floating-point register. After common subexpression elimination these equation become:

$$\begin{aligned} tmp &= c \times d \\ a &= tmp + e \\ b &= tmp + f \end{aligned}$$

CSE can greatly enhance the performance of instruction bound kernel by reducing the number of floating-point operations (flops). But the reduced number of flops is traded against increased pressure on the low-level memories like register file and local memories or caches. All already computed values kept for future reference are held in such memories to be available quickly when needed. If this cannot be guaranteed and the kernel ends up e.g. cache bandwidth bound, a recalculation of the data value should be preferred. Therefore, the aggressiveness of CSE application depends on one hand on the kernel itself on the other hand on the architecture's low-level memory hierarchy.

5.4.2.1 Common Subexpression Elimination (CSE) for TTI

The nature of second-order mixed derivates requires applying first-order derivates in two steps. At first, first derivates are computed for each point along the axis in all three

dimensions, followed by summing of such values and multiplying a coefficient. Since, the first derivatives do not alter for consecutive points, given x as fastest unit-stride direction, causes redundant computations in the $x-y$, $x-z$ and $y-z$ plane. To avoid redundant computations an additional array is allocated which holds the first derivates for all points and all space dimensions. The size of the arrays depends on the finite-difference approximation order-in-space. In x-direction these arrays are small enough to be kept in L1 cache, in y-direction the size of the array depends on the unit-stride in y.

Hence, when the stencil is applied to the subsequent point along the x-axis, only three new first derivates need to be calculated. Each derivative requires $\delta = 3r$ floating-point operations with r as radius of the stencil. The three applied operations are the subtraction of two points and the multiplication of a coefficient. The result is summed to receive the final derivative. The code snippet 5.1 gives an example for the calculation of a first derivative in the x-y plane where `P` is the array of the pressure wavefield, `idx` a wrapper function that returns the 3D to 1D index mapping and `c` a coefficient.

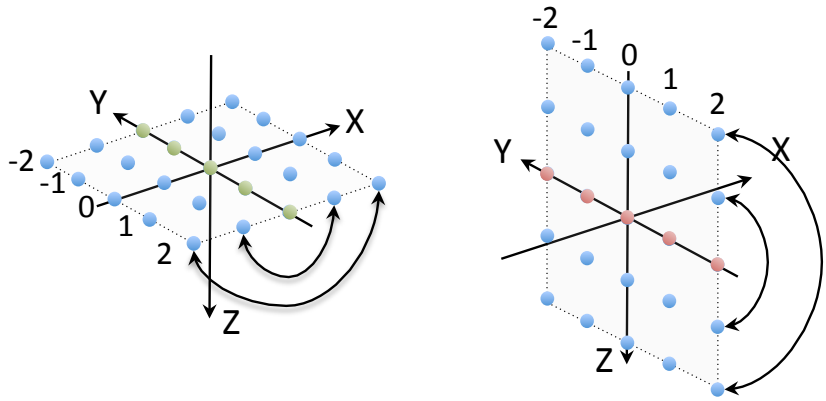
Listing 5.1: First-order derivate calculation

```
for(int r=1; r<=radius; r++){
 cse_xy[r] += c[r]*(P[idx(x, y + r, z)]-P[idx(x, y - r, z)]);
}
```

A total of $(2r+2)*\delta$ flops are required for the partial derivatives, and $\delta * r$ flops for the second-order derivatives along the axis for wavefield P and Q. When CSE is applied, the first derivative needs to be attained per wavefield only twice per axis. In total, a reduction of $6r^2$ flops is achieved for the mixed, partial derivatives. The second-order derivates along the axis cannot be avoided, which adds another $3r * \delta$ flops per wavefield. Finally, an additional fixed overhead of about 70 flops is added for receiving the final values for the next timestep for P and Q.

Figure 5.9 visualizes the CSE scheme for a second-order in space TTI stencil in the $x-y$ 5.9(a) and $x-z$ 5.9(b) plane.

CSE makes the number of floating-point instructions per point dependent on the plane size in x and y. A larger plane improves the flops/point ratio because of an improved volume-to-surface ratio. Adding the additional operations outside the derivate



(a) TTI CSE for x-y plane (b) TTI CSE for x-z plane

Figure 5.9: Common Subexpression Elimination for TTI - (a) for the x-y plane, (b) for the x-z plane.

approximation to the equation, about 290flops per point is the lowest number of flops/point achievable for a volume size of 512^3 and an 8th-order stencil scheme. Assumed all temporal recurrences are caught, a total of 52 bytes have to be accesses in memory per calculation, in case sine and cosine values of ϕ and θ are pre-computed. This decreases the flop-to-byte ratio from ~ 19 to about ~ 5.6. For heavily instruction bound kernels like TTI this means a 3.5x reduction and therefore an enormous increase in performance.

Figure 5.10 in Section 5.6 extends the Roofline Model introduced in Section 4.2.4 and adds a CSE optimized TTI kernel - denoted with the "star" symbol.

5.5 GPU Optimizations

This thesis uses the M2090 as reference for GPUs. The highly optimized GPU code was written by Paulius Micikevicius and follows the approach described in [55]. This section gives a brief introduction into the optimization techniques applied to give the reader an idea how performance is achieved. But since GPU optimization was no part of this thesis this section does not go into more detail.

The domain is tiled with so-called threadblocks along the two fastest varying dimension x and y. Each threadblock then marches along the slowest-varying (z) dimension producing a pencil of output. The "pencil" approach enables the efficient use of reg-

isters and shared memory for storing the input values necessary to compute discrete spatial derivatives. This way values along the z-dimension are kept in registers so that no explicit sharing is needed among threads, values in the xy-plane are stored in shared memory because several threads need to access each value. Other input values like Thomsen parameters, are read when needed, by streaming through those arrays. The use of shared memories is comparable to cache-blocking on CPUs. Its goal is to ensure that stencil values are read once from the off-chip global memory and subsequently are kept and read repeatedly from the on-chip shared memory.

Both ISO and VTI computations can be done in a single pass over the domain, using 2D threadblocks of size 32x16 and 32x8 threads, respectively. TTI is implemented as two passes: the first pass computes the first and second y-derivatives for the P and Q wavefields, the second pass computes the remaining derivatives and the final output. Special hardware units for sine and cosine computations enable computing of such on the fly, which reduces memory bandwidth pressure and a lower working set size.

As for CPU implementations property and wavefield arrays are padded to be cache line size align to achieve highest memory throughput. Additionally, the Fermi architecture provides 64-bit memory-access instructions. Compared to accessing two separate float elements, accessing a single 64-bit vector reduces the number of load/store as well as pointer-arithmetic instructions. To leverage this capability a new 64Bit vector data type is created and two arrays are then interleaved with each other.

Fermi cards have only a limited on board capacity, which makes it necessary to distribute computation to other GPUs for large working sets. For multi-GPU setups the domain is decomposed 1D in z-direction. CUDA 4.0+ enables peer-to-peer (P2P) PCIe communication between GPUs on the same node without involving the host CPU. Together with GPU DMA engines it is possible to asynchronously transfer data and proceed with computation. With using multi-buffering 7.2.1 it is possible to hide communication with computation. First points that need to be exchanged are computed, followed by issuing the memory transfers, which is done at the same time the remaining internal, halo independent, region of the subdomain is computed. A synchronization step at the end ensures that data is current before proceeding with the next time-step.

5.6 Optimization Summary and Extended Roofline Model

In this section the effects of optimization is explored with an extended Roofline Model. The Roofline Model presents a way to visualize which architectural paradigms have to be exploited or software optimizations applied to attain best possible performance for a certain kernel on a specific architecture. The naive Roofline Model was introduced in Section 4.2.4. It took peak single precision performance and peak sustained memory bandwidth to predict the performance of each kernel. As seen in Section 4.3 the reference kernel implementation shows poor performance and comes nowhere near the estimated performance numbers.

Peak Stream DRAM bandwidth is only achieved if multiple levels of data parallelism are exploited and data does not dependent from each other. Memory parallelism exists on bit-level for each channel per memory controller, for multiple memory controllers and multiple chips. Furthermore, compulsory, capacity and conflict misses within the cache hierarchy can appear. Hence, the achieved bandwidth depends on the architecture and the kernel memory access patterns and can be significant lower than the measured Stream bandwidth. This maximum achievable bandwidth is called *Bandwidth Ceiling*[109].

Further an *In-Core Performance Ceiling* is defined. Peak single precision performance is achieved only if all data is present in registers when needed and all cycles are used to compute floating-point instructions while exploiting all architectural capabilities like multiply-adds and SIMD instructions. Only, if the instruction mix provides the same number of independent multiply and add instruction within the compiler look-ahead window, multiply-add capabilities of the FPU can be fully utilized.

Although data level parallelism exists in most kernels, compilers and programmers often have a hard time applying SIMD intrinsics. The performance degradation of not using such instructions depends on the number of lanes per FPU [109]. If SIMD instructions can be performed within one cycle the performance by not utilizing such instructions depends on the width of the SIMD register. For 128-bit register widths or four floating-point values the maximum performance loss is $4\times$.

Through inter-instruction dependencies, not enough instruction level parallelism can be achieved and pipeline stalls might appear which causes performance penalty. Finally, performance is decreased through poor data locality that results in cache and TLB

misses.

The different levels of the *In-Core Performance Ceiling* show how performance is diminished otherwise. Figure 5.10 presents the new Roofline Model. The star symbol

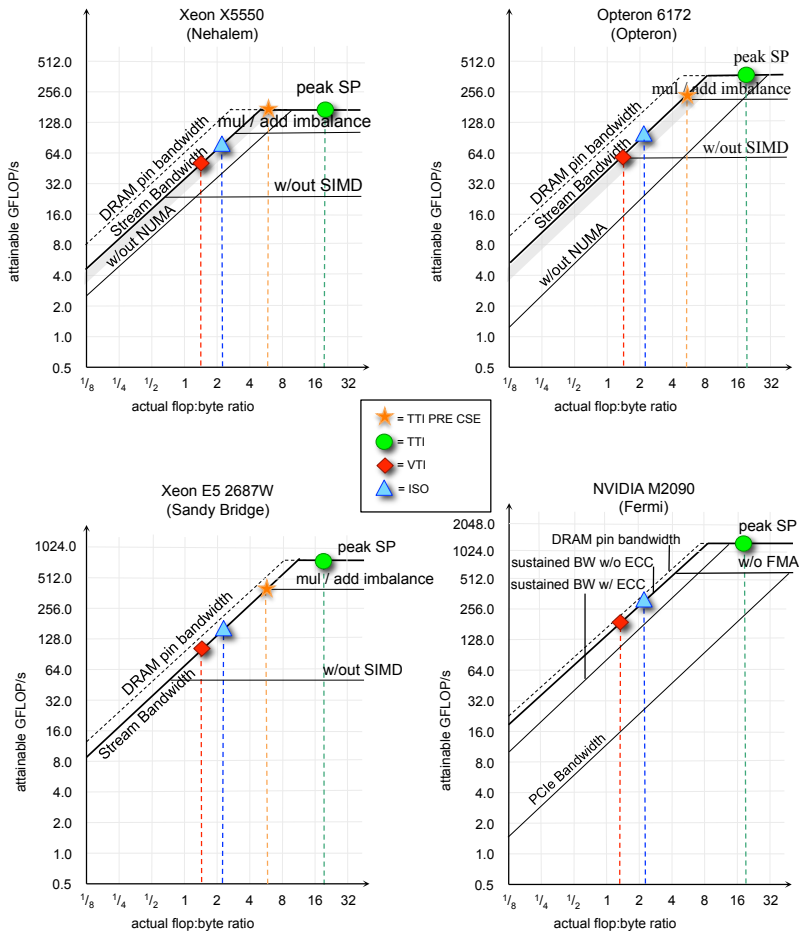


Figure 5.10: Extended Roofline Model.

represents the TTI kernel implementation using CSE and pre-computation of sine and cosine values for dip and azimuth. Note that the number of floating-point instructions per stencil using CSE depends on the plane size. Here, the maximum benefit of CSE for a plane size of 512^2 points is presented which reduces the flop-to-byte ratio to ~ 5.6. By applying cache-blocking techniques the plane size is reduced to fit into local caches. This way the volume-to-surface ration increases and so does the number of flops per point. Hence, it is a trade off between the benefits of CSE and cache block-

ing techniques. Table 5.2 presents a direct translation between Gflops/s, as drawn on the y-axis in the Roofline Model 5.10, and MPoints/s. One can see that higher Gflops/s performance doesn't necessarily mean a higher throughput in MPoints/s - e.g. the optimized TTI implementation with CSE and pre-computation achieves with lower Gflops/s higher number of MPoints/s.

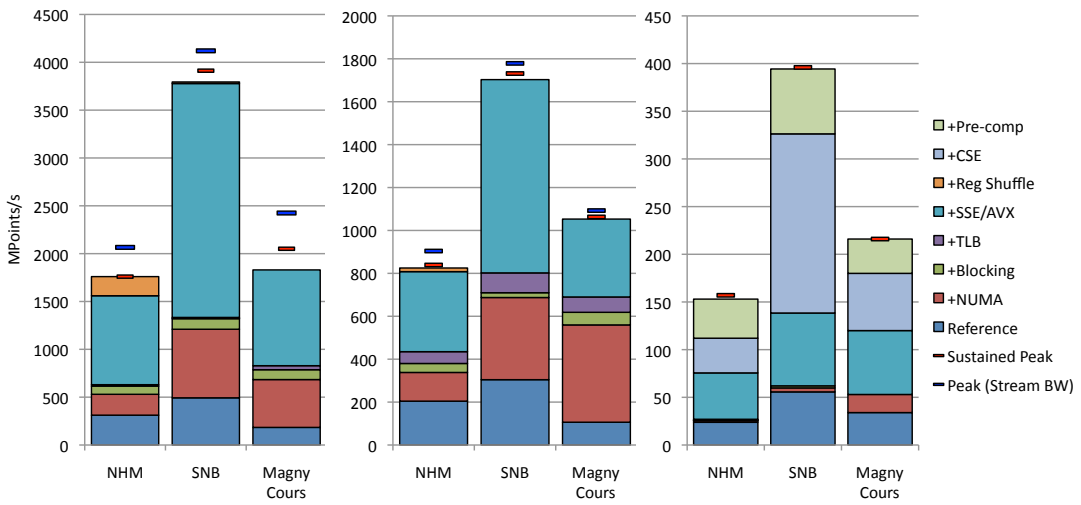
Gflops/s	ISO	VTI	TTI	TTI CSE Pre
0.5	15	9.4	0.6	1.7
1	29	19	1.3	3.5
2	59	38	2.5	7
4	118	75	5	14
8	235	151	10	28
16	471	302	20	55
32	941	604	40	110
64	1,882	1,208	80	221
128	3,765	2,415	160	441
256	7,529	4,830	320	883
512	15,059	9,660	640	1,766

Table 5.2: Unit translation from Gflops/s to MPoints/s

5.7 Benchmark Analysis of Optimized Kernels

Section 4.3.2 presented the benchmark results of the reference kernel implementations. Comparing the achieved performance of all three kernels to the estimated performance limitations in 4.2 showed that software optimization has to be applied to achieve a fair comparison between the different hardware architectures. This section now analyzes the benefits of the introduced optimization techniques to the different architectures and finally compares the achieved performance to the estimated hardware limitations.

Figure 5.11 presents performance as progressively more optimizations are included in the mix. Clearly, the reference implementation delivers poor performance across machines with Sandy Bridge delivering the best performance for all kernels. As all architectures are NUMA architectures, domain decomposition in z-direction for each NUMA socket and in y-direction for the number of cores within a NUMA socket avoids the performance pitfalls of poor data placement. NUMA effects are mitigated when an implementation is heavily compute-bound. Tuning for the appropriate block size further improves performance with best effects on memory intensive VTI kernels. The



(a) ISO (b) VTI (c) TTI

Figure 5.11: RTM kernel performance

TLB optimization applied is reusing the array of the previous timestep for the results. This way the pressure on virtual to physical address translation is decreased which shows strong benefits for memory bound kernels like ISO and VTI but no advantage for flop heavy TTI kernel.

To improve floating-point performance, the manually vectorized code is using SSE or AVX intrinsics including Intel's SVML sine and cosine functions. The benefit is highly dependent on the underlying machine's potential memory or cache bottlenecks as well as the starting point. Especially the new Sandy Bridge with its doubled L1 cache bandwidth benefits greatly from vector instructions. Additional optimization via common subexpression elimination provides significant performance boosts on TTI, but is useless on ISO and VTI where there are no common subexpressions to eliminate. Cache bypass showed increased performance as long as no TLB optimization was applied. With TLB optimization cache bypass instructions even diminish performance and were therefore not used.

Finally, to reduce redundant computation in TTI to a minimum, sine and cosine functions of the spatially varying azimuth and dip constants can be pre-computed. This requires increasing the number of arrays by two, but completely avoids expensive trigonometric calculations in the inner loop. On Magny Cours, this provides a moder-

ate increase in performance of 12.5% and up to 20% on Sandy Bridge.

Comparing the overall performance one can see that the Sandy Bridge nodes deliver the highest total performance of our x86-based systems due its advanced technological features and more effective memory controllers. Moreover, although we obtained more than a 6x increase in TTI performance, we observe that TTI is still instruction bound on all architectures.

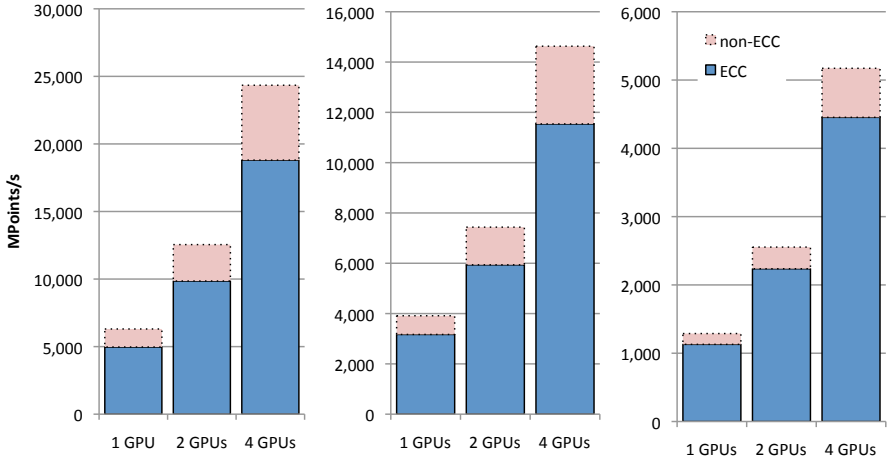
To quantify the results, the achieve optimized kernel performance is compared to the theoretical peak. Peak throughput is determined by the sustained memory read bandwidth divided by the compulsory number of bytes per point. Figure 5.11 presents the the theoretical achievable peak as "Peak (Stream BW)". An additional line above shows the maximum achievable peak performance if cache block and node volume sizes are adjusted to fit the underlying architecture perfectly ("Sustained Peak"). For the 512^3 volume results show that on Carver 85% and 91% of peak could be attained for ISO and VTI kernels respectively. Mangy Cours showed worse performance with 73% and 93% of peak. Our Sandy Bridge node exploits ISO performance the best with 92% of peak and shows similar performance for VTI kernels. Because of its large working-set TTI achieves only 23% to 29% of theoretical memory bandwidth enabled peak.

5.7.1 GPU Performance

Figure 5.12 shows GPU performance as a function of kernel and the number of GPUs per node. Performance scaling is nearly linear with the number of GPUs since communication time is lower than internal computation. Moreover, turning off ECC can further improve performance by as much as 30% due to increased memory bandwidth. Users must therefore carefully weigh the performance benefits with the risks of memory errors.

Table 5.3 examines the impact of array-padding and array interleaving on the GPU code for TTI kernels. When applied individually, padding and interleaving improve performance by 13-15%. However, performance is improved by 39% when optimizations are applied together. Whichever optimization is applied second, gets amplified by the first one - it provides an improvement in excess of 20%.

Since all other architectures use ECC protected memory modules, ECC is enabled for GPUs as well. Further, multi-GPU node setups require explicit exchange of data



(a) ISO (b) VTI (c) TTI

Figure 5.12: GPU performance

	unpadded	padded
non-interleaved	925	1048
interleaved	1067	1288

Table 5.3: TTI Kernel Performance (MPoints/s) with various GPU Optimizations

and are considered as multiple SMPs. To guarantee a fair comparison only SMPs are compared to each other. Hence, a single GPU is compared to the evaluated x86 SMPs.

Interestingly, on the more bandwidth-bound ISO and VTI kernels, we observe that a single M2090 with enabled ECC is only about 1.3× to 1.4× faster than the Sandy Bridge system — a testament to the greatly improved memory subsystem of the Sandy Bridge nodes. Conversely, on the more compute-intensive TTI, the M2090 is about 2.8× faster than Sandy Bridge.

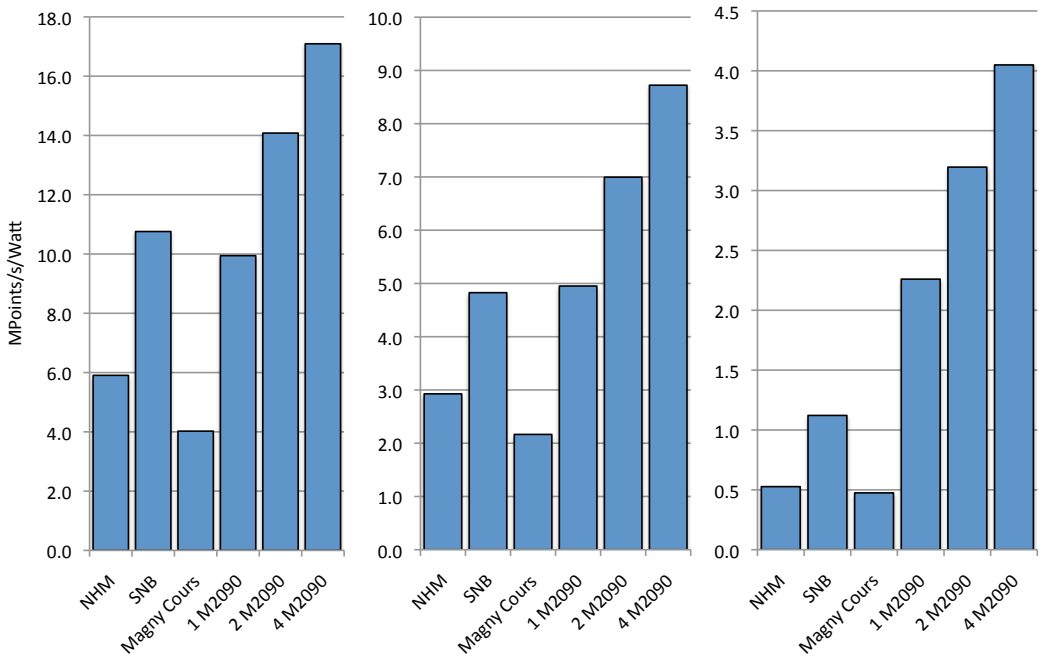
5.7.2 Single-Node Energy Efficiency

While noting the performance differences between the architectures, the commensurate difference in power must be acknowledged. To normalize the differences in machines the energy efficiency of calculating these RTM kernels is examined as a function of machine and configuration.

Sustained system power consumption for the Nehalem and Sandy Bridge machines was measured with an in-line power meter. Using an in-line power meter was not possible for the Magny Cours system due to how power is distributed within a XE6 rack. Therefore, for the Magny Cours system power was estimated by averaging the power per node NERSC measured for their Top500 submission.
GPU-power consumption was measured using the nvidia-smi tool, which reports the power GPU draws from the host system. The highest measured power consumption was 180W. However, since this does not include the system power-supply overhead, the total power was adjusted to 200W assuming a power-supply with 90% efficiency, which is common in modern servers. Since GPUs need a host system to control them, the measured 298W for a dual-Nehalem server were taken as point of reference. Thus, the estimated power consumption for servers with 1, 2, and 4 GPUs results in 500W, 700W, and 1100W, respectively. Contrary to the performance comparison with where only one GPU was taken into to comparison to the x86-based architectures such a comparison would not be fair for a single GPU because of the additional host power overhead. That's why two- and four-GPU node configurations are taken to comparison for energy efficiency.

Figure 5.13 shows energy efficiency (MPoints/s per Watt) across the various platforms. We observe that on ISO, a dual-Nehalem and 1-GPU system with enabled ECC is slightly less energy efficient than a 2P Sandy Bridge. Conversely, a system with 1 GPU is more than twice as energy efficient as the Magny Cours system. Since the host server is not performing computations in GPU systems, the host's power consumption is amortized as more GPUs are added to the node (system-power consumption ratio approaches the ratio between single-GPU and single-socket power ratio). As a result, the 2-Nehalem and 4-GPU solution is about 1.3× more energy efficient than a Sandy Bridge system and is further improved with four GPUs to 1.54x. For the even higher bandwidth bound VTI kernel, GPUs' energy efficiency improves and results in almost 2x four a four GPU setup compared to a Sandy Bridge node.

As we move to more computationally-intensive kernels like TTI, we see the GPU's energy efficiency exploited even better. In fact, for TTI kernels, a single-GPU and 4-GPU systems are now more than 3.5x more energy efficient as a Sandy Bridge system and about 8x as efficient as Nehalem oder Magny Cours nodes.



(a) ISO (b) VTI (c) TTI

Figure 5.13: Energy Efficiency for optimized Kernels

GPU system power efficiency could be further improved by using a Sandy Bridge server as a host. The Sandy Bridge node results could further improve by using DDR3-1600 memory modules, which should result in another 20% performance gain.

5.8 Single-Node Summary and Conclusion

This chapter introduced different classes of optimizations and gave a brief description of commonly used optimization techniques and kernel specific optimizations. Applying the different optimization techniques on the evaluated hardware architectures show enormous performance gains.

Many different optimizations can be applied to the actual kernel in general or for a specific architecture. Each optimization is differently important and its application has to be traded off between effort, sustained accuracy and speed-up that can be achieved with it. In practice application driven optimizations and hardware driven optimization cannot be completely separated from each other. Figure 5.14 describes the relation

between the application driven and the hardware driven optimizations.

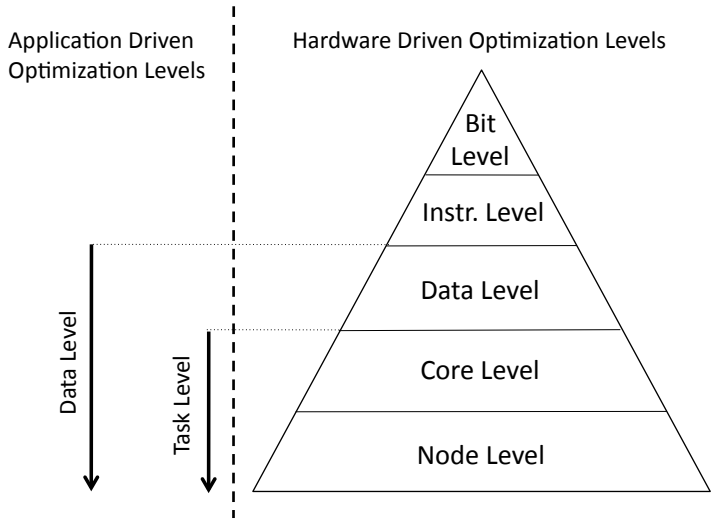


Figure 5.14: The different levels of optimization for application driven and hardware driven optimizations.

Instruction- and lower-level parallelism is hardware specific. The programmer has only limited influence on how ILP is exploited which is done by the compiler. From hardware data level on, all other levels profit from data level parallelism. Independent data points are executed in parallel in SIMD fashion, by multiple cores and for computing clusters multiple nodes (see Section 5.9). Applications exploiting task level parallelism can profit from the hardware core-level on downwards. This way, independent tasks can be distributed to different cores or nodes. As it can be seen hardware and software heavily interact with each other and optimizing a given application to a specific architecture can be a challenging task. The achieved performance comes close to the theoretical performance limits for each kernel and therefore delivers a fair basis for comparing architectural efficiency. TTI kernel implementations are generally instruction bound even after being fully optimized whereas ISO and VTI implementations are likely to be memory bound. Especially, for flop heavy TTI kernel multi GPU node setups show significant better performance and energy efficiency compared to the evaluated x86 node configurations.

5.9 Multi-Node Benchmark

For production-sized seismic surveys a single node is not sufficient for image processing. Even though the single-node study 8 gives a first impression of how such a custom system would perform, compared to on-market architectures more realistic benchmarks are received by a multi-node study.

This chapter introduces the new challenges associated with with production-sized surveys requiring multi-node implementations. It discusses different multi-node implementations for all evaluated kernel. The Green Wave architecture is analyzed and adjusted to serve the needs of a multi-node requirement and is compared to leading-edge cluster implementation like "Hopper" (see Section 3.2.2) as an example for an x86-based processor cluster.

The complete volume is domain decomposed in x-, y- and z- dimension into 3D subdomains (see Figure 5.15) to leverage node level parallelism.

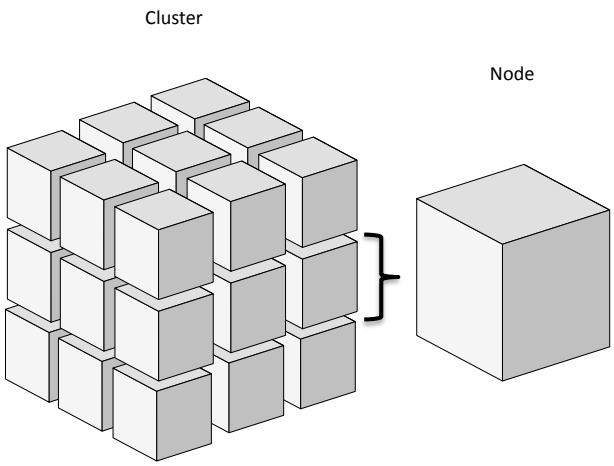


Figure 5.15: The domain is decomposed into subdomains for each node.

5.9.1 Node-Level Parallelism

Node-Level Parallelism describes the concurrent work of multiple nodes. A node describes a symmetric multiprocessor (SMP) organization on which each core can access

all memory addresses existing within the SMP. The use of multiple of these nodes requires communication and the handling of distributed memory since each node can by default access only its own main memory. The most common way to implement communication between nodes is the use of the Message Passing Interface (MPI) [89]. MPI provides two-sided communication, which involves one sender and one receiver. Even though MPI is indented for communication between distributed memory systems it can be used on SMPs as well. Current research explores the advantages or disadvantages of MPI on SMPs versus hybrid implementations, which combine MPI with thread-level parallelization SMP libraries like OpenMP or pThreads (see Section 5.3). The use of MPI on SMPs gained new attraction through the increasing number of NUMA sockets. A NUMA socket describes a collection of cores with a dedicated memory system on a SMP, which is accessible to other NUMA sockets within the SMP. This remote memory access involves an additional latency penalty since data has to be transferred over QPI or HT for each access. Hence, it might by more effective to use something like MPI for specific algorithms.

Although MPI is the most common approach for inter-node communication MPI implementations can suffer from high overhead caused by package headers and two-sided communication for many-core systems and large node counts. To address this problem *Partitioned Global Address Space* (PGAS) libraries and languages have moved into the focus of interest. An example PGAS language is *Unified Parallel C* (UPC) [49] or the *Global Address Space Programming Interface* (GPI) [28] .

The PGAS approach makes distributed memory completely or partially accessible to remote nodes. These remote nodes are then able to use the remote memory without any further knowledge where this memory is physically located via *Remote Direct Memory Access* (RDMA) transfers. Contrary to the two-sided message passing approach of MPI, with RDMA the receiving node doesn't have to be actively waiting to receive data. The cores that physical own the global RDMA accessible memory, have no knowledge if data is being written into its memory at a certain time. This one-sided communication model enables asynchronous communication in which the sending node does not have to wait for the completion of the data transfer. Avoiding strict synchronization and global barriers this approach is able to deal with small load imbalances between nodes without performance degradation of stalling computation. A disadvan-

tage of such message passing programming models is the additional implementation complexity to guarantee data consistency.

5.9.2 Multi-Node Communication Overhead Analysis

As MPI is still most commonly used in the HPC community this multi-node study uses a multi-node programming model based on two-sided MPI communication. The multi-node implementation is divided into four standard steps:

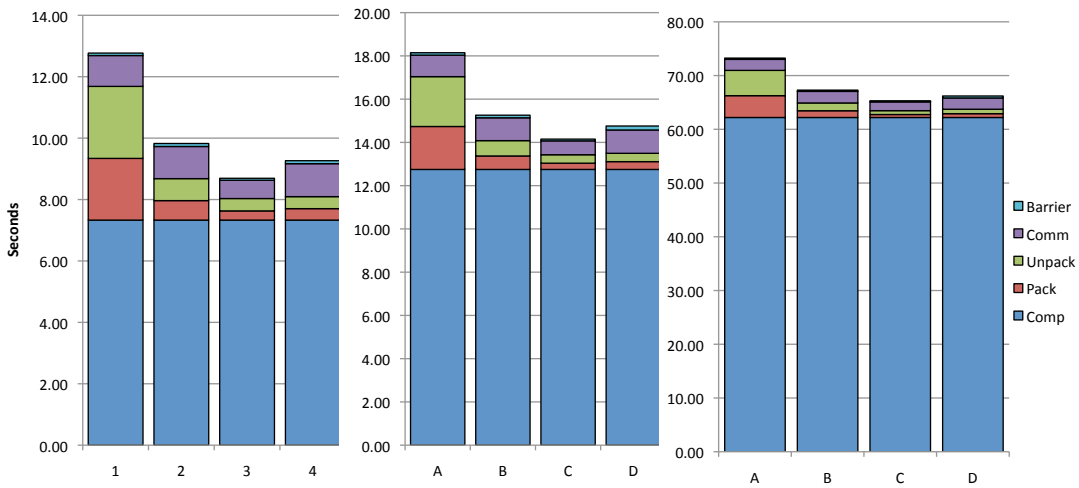
1. Extraction of the halo region into six separate send-buffers for ISO and VTI, and additional 12 buffers for edge halos for TTI.
2. Exchange of halo data with neighboring nodes.
3. Unpacking of MPI receive buffers by scattering data into the main volume array.
4. Applying the wave equation to all points of the volume.

The multi-node communication is nearest neighbor only. To make sure that each neighbor is distinct from each other 27 nodes were allocated forming a 3x3x3 grid using `MPI_Cart`. By enabling rank-reordering `MPI_Cart` tries to achieve the best node to rank allocation for the given grid layout. Each node works on a subdomain size of 512^3 and performs a total of 100 timesteps.

The amount of data transferred during halo exchange is relatively small. About 25 MB is the total amount for ISO and VTI implementations. Even though VTI uses the auxiliary Q, the stencil performs finite-differences in the x and y direction for wavefield P and only the z direction for auxiliary wavefield Q. For TTI implementations 52 MB of data has to be exchanged since volume edges are added.

Figure 5.16 presents how much time is spend for each of these steps as a function of optimization for Hopper. Note that four different MPI implementations labeled "A"–"D" are presented. These implementations vary how threading and NUMA interact with MPI. "A"–"C" represent threaded implementations, which progressively become ever more complex, and NUMA-aware implementations. An additional barrier ("Barrier") is used before the communication takes place to void eventual load imbalance, which could result in inconsistent timing. Thus, the time presented as "MPI Communication" represents pure communication time for sending the ghost zones over the network.

The reference implementation "A" uses one MPI process per node. Only the dedicated core allocates MPI buffers, performs data copies and transfers. Since data is mapped to NUMA sockets for computation, there is a NUMA-effect (buffers to grid) for buffer copies that degrades performance. Clearly, the MPI overhead for the ISO and VTI implementations is significant with 40% and 30% of the total runtime. Although VTI must communicate parts of two grids, the MPI time remains quite similar to ISO as the total volume of data remains similar — the stencil performs finite difference in the x and y direction for wavefield P and only the z direction for auxiliary wavefield Q.



(a) ISO (b) VTI (c) TTI

Figure 5.16: Time Breakdown of the MPI Implementation on Hopper

As seen in Figure 5.11 NUMA-aware allocation is of exceptional importance on strong NUMA affine Hopper nodes. Due to on-node domain decomposition the ghost zones reside on separated NUMA nodes.

To optimize for locality, the MPI buffers should also be allocated in a NUMA-aware fashion based on their corresponding ghost zones. The impact of NUMA-pinned MPI buffers can be seen in implementation "B" of Figure 5.16. It shows that NUMA-aware buffer allocation significantly reduces the time spent in packing and unpacking MPI buffer data. As expected, Hopper benefits greatly with about 3.2x improvement in time for packing/unpacking.

One should note that because MPI was initialized with `MPI_THREAD_SINGLE`, only one thread is allowed to perform the MPI communication. Thus, there is still a NUMA effect as data is transferred from the buffer to this core. To optimize MPI communication even further one should eliminate remote NUMA memory accesses almost completely. The use of `MPI_THREAD_SERIALIZED` ensures each thread sends and receives data from its local buffers. The benefit of this approach can be seen in implementation "C". A 1.9x improved communication time is achieved for the ISO kernel. VTI still sees a respectable improvement of 20% but with only minor improvements to the MPI time for TTI. Additionally, implementation "C" parallelizes the packing and unpacking of MPI buffers to all available cores on a NUMA socket, which results in memory copies improve by about 1.8x for all kernels.

A very common implementation to obviate all remote NUMA memory accesses is to instantiate one MPI process per NUMA socket. This causes additional overhead in terms of data replication and halo communication due to a worse surface-to-volume ratio on each NUMA socket. To minimize these effects data decomposition between processes on a node is done in the Z-dimension only — thereby ensuring the additional surfaces may be accessed at maximum bandwidth. As one can see in implementation "D", this approach to MPI communication actually increased the communication time on Hopper — an artifact of their MPI implementation on their custom network chip. Data copies to/from MPI buffers were parallelized here as well.

Nevertheless, it is clear that obviating NUMA via multiple processes per node can have unpredictable impacts on MPI performance. The overall runtime improvement on Hopper between the basic MPI implementation and the best version are 30% for ISO, 18% and 10% for VTI and TTI, respectively. Still a 17%, 7% and 5% MPI overhead exists for these kernels, which in fact reduces the overall energy efficiency by 15%, 10% and 5% compared to the single node benchmarks.

Generally, in seismic migration strong scaling appears on shot level, weak scaling between nodes. As our x86-based architectures are not limited by DRAM capacity one could increase the total node volume to reduce the influence of the MPI overhead. However, this comes in hand with a reduced shot processing performance. As such, scientists must carefully weigh the compute benefits and the communication penalties for the utilized architecture, especially when it comes to more communication sensitive applications.

5.9.3 Multi-Node Programming Model for GPUs

The GPU accelerator cards require a different multi-node programming model than x86-based CPU node architectures. As GPUs cannot run an operating system on their own they depend on a host CPU to manage e.g. I/O and inter-node communication. GPUs are connected to the host via a PCIe interface and require explicit data transfers to or from host memory to the device memory. A naive multi-node programming approach would consist of two steps: a computation phase followed by a halo exchange phase including halo copies to the host memory. The additional overhead for communication would let GPUs suffer significantly more than CPUs because of their faster computation and limited node volume size. Fortunately, NVidia's M2090 owns a DMA engine which can communicate with the host memory without stalling the cores. Cuda4.x supports asynchronous transfers with a `cudaMemcpyAsync()` function call. It either receives or sends data to or from host memory. All asynchronous memory copies are organized in so-called `cudaStreams`. Within each stream all transfers are done sequentially - concurrency is exploited between stream instances. This way it is possible to read and write MPI buffers to/from the main memory on-the-fly. Under the constrain:

$$\text{Time(Copy)} \;+\; \text{Time(Communication)} \;\leq\; \text{Time(Compute)} \tag{5.1}$$

packing and unpacking of MPI buffers can be completely overlapped with computation.

To test if Equation 5.1 can be satisfied an example program was written that measures the time to transfer the halos. Latency for data transfers between device and host is diminished by address translations for each initiated copy. To improve performance host data arrays can be mapped into the device memory by replacing the standard `malloc` by `cudaHostAlloc(...,cudaHostAllocMapped)`. This way the device creates a page table of the host array and memory copies to can be performed like normal accesses to device memory.

As shown in Figure 5.15 three types of halos need to be exchanged for a 3D domain decomposition between nodes: top, bottom, North, South, East and West halos. Top and bottom halos are send and received as a single block since data lies consecutive in memory. Such a block has the size:

$$\text{Top/Bottom Halo Size} \;=\; \text{Dimension X} \;\times\; \text{Dimension Y} \;\times\; \text{Stencil Radius} \tag{5.2}$$

For North and South halos much smaller pieces are block-wise accessible. Each of these blocks has a size of

$$\text{North/South Block Size} = \text{Dimension X} \times \text{Stencil Radius} \tag{5.3}$$

and needs to be send and received for all $nz \in Z$. The most fine-grained halos exist in directions East and West. Here a single block of data, which lies consecutive in memory is only the size of a stencil radius. For an 8th-order stencil this corresponds to 16 bytes sent and 16 bytes received Dimension Y $\times$ Dimension Z times.
Because no direct access to nodes with M2090s were possible, time measurements were done on the "Dirac" cluster located at the NERSC facilities. A node consists of a dual quad-core Intel X5530 with 24 GB DDR3-1066 and a single Fermi C2050 GPU. Host CPU and GPU communicate via a PCIe x16 Generation 2 interface.
The example code showed that such high numbers of small 16-byte copies add too much pressure on the DMA engine and causes high copy times. In order to keep the time for data transfers smaller than the time for computation node domain decomposition for GPU-based nodes is done in Y and Z only. No real kernels and wave propagation were executed to test the hiding of communication latencies. Measurements show that Equation 5.1 could be satisfied. Neither latency is added for copying data from the GPU mapped memory region to the network interface mapped memory region within the host memory since it can be easily avoided by using the GPUdirect [62] technology. As presented in the multi-node programming model for x86-based architecture, MPI communication can be overlapped with the computation of inner independent points of the the volume. Studies from NVidia [56] showed that multi-node GPU setups perfectly scale with number of nodes and GPUs, respectively. Hence, it is further assumed that no additional overhead applies for packing and unpacking of MPI buffers nor for MPI communication on GPU multi-node setups.

5.9.4 Node Volume Optimization

With asynchronous communication like the MPI `MPI_Isend` and `MPI_Irecv` or PGAS languages it is possible to overlap the communication with computation of inner points of the subvolume which are independent from the halo. Unfortunately, the time for packing and unpacking the communication buffers cannot be hidden. Results from

experiments with MPI and GPI have shown that fine grained communication of small message add a large overhead (MPI) or fill up the asynchronous queues (GPI).

Therefore, the time handling the communication buffers has to be added for each timestep and all architectures to the single-node benchmark results. The overhead for packing buffers was directly measured for all architectures. This overhead of packing and unpacking can be minimized by parallelization and memory pinning that no remote memory accesses slow the buffer access times.

Another important way to minimize communication overhead is to maximize the node volume. This way the surface-to-volume ratio improves and more time is spend in computation relative to packing and unpacking buffers. A maximum node volume is equally important for GPUs to provide enough time to overlap communication times. Hence, according to the memory capacity the node volume is increased. All x86-based nodes own 32 GB of main memory, each GPU owns 6 GB. Table 5.4 presents the memory capacity and volume sizes.

Architecture	max. Memory (GB)	ISO	VTI	TTI
Nehalem	32	2048x1024x1024	1024 x1024x1024	1024x1024x512
Sandy Bridge	32	2048x1024x1024	1024 x1024x1024	1024x1024x512
Magny Cours	32	2048x1024x1024	1024 x1024x1024	1024x1024x512
1 M2090	6	4096x256x256	4096x256x128	4096x128x128
2 M2090	12	4096x512x256	4096x256x256	4096x256x128
4 M2090	24	4096x512x512	4096x512x256	4096x256x256

Table 5.4: Node Volume Sizes

Figure 5.17 presents the percentage of time added for packing and unpacking of buffers - noted as "communication overhead". Packing and unpacking of buffers is highly optimized and the node volume maximized. The additional overhead for x86 architectures is small and adds only 6% overhead to ISO kernels running on Sandy Bridge and about 4% on Nehalem and Magny Cours nodes. For VTI kernels the overhead is 4% or less and for TTI kernels about 1%. Despite significant higher throughput, one can notice a similar total TTI overhead for Sandy Bridge compared to Magny Cours. This results through the higher number of cores per NUMA socket and the improved memory controller architecture.

GPUs are not presented in this diagram as no additional overhead appears for them.

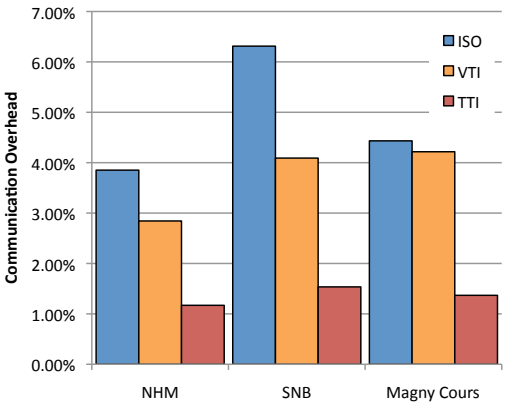


Figure 5.17: Communication Overhead

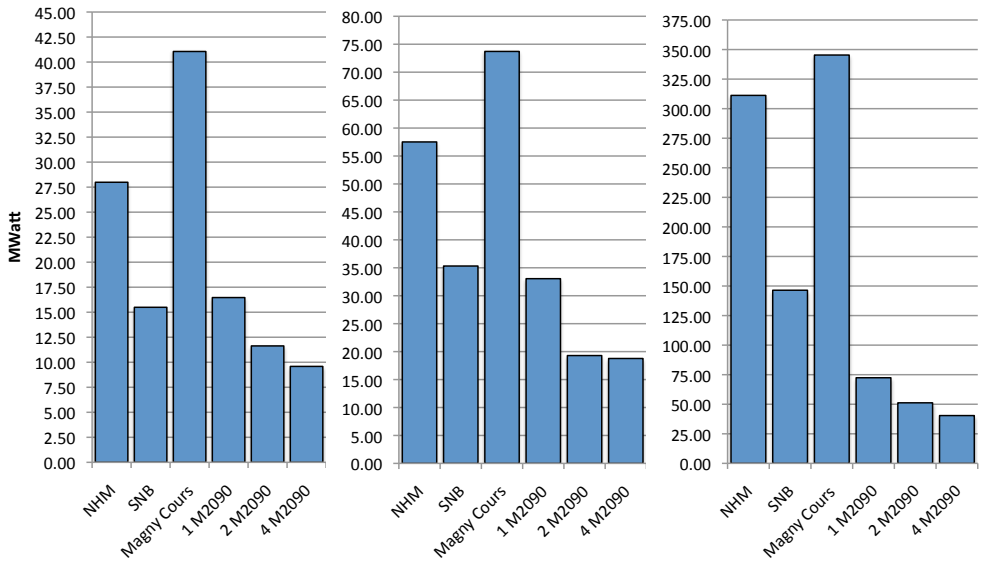
The minimal overhead leads to almost no performance degradation compared to the single-node benchmark results.

5.10 Estimated Cluster Power Consumption

The previous sections discussed the multi-node performance for the evaluated architectures. Based on those performance numbers this section compares the power consumption of cluster setups for a fixed time-to-solution. Within this timeframe, the example survey presented in Section 2.5 is forward- and backward-propagated, including the cross-correlation of wavefields that is necessary for RTM. Simplified assumptions are made and nodes are simply replicated as often as necessary to reach to the required time. For all architecture perfect scaling up to the required number of nodes is assumed.

Figure 5.18 presents the power consumption estimations for cluster setups based on the different architectures. A cluster based on the most energy efficiency x86 Sandy Bridge architecture running the ISO kernel would consume about 16 MWatts, about 7 MWatts more than cluster setups based on nodes with four M2090s but consumes slightly less than GPU node setups with only a single M2090. The much less efficient Nehalem and Magny Cours consume 29 MWatts and 42 MWatts, respectively. VTI kernels are highly memory bandwidth bound on all architectures. One can see that the high bandwidth GPUs provide, leads to an increased energy efficiency improvement. A cluster

based on four M2090s nodes now gains an advantage of about 1.5x compared to Sandy Bridge clusters and would consume about 19 MWatts. Interestingly, no improvement in power consumption is achieved by using four instead of two GPUs per node. The flop-heavy TTI kernel sees further increased energy-efficiency advantage for GPUs. A four M2090s setup is now 3x more energy efficient than Sandy Bridge and a cluster capable of migrating the volume within a week would consume more than 41 MWatts. Older generation architectures like Nehalem and Magny Cours show much worse energy efficiency. Clusters based on those architectures would consume 315 to 350 MWatts for TTI. The leading-edge Sandy Bridge provides more than 2x improvement with roughly 150 MWatts for TTI.



(a) ISO (b) VTI (c) TTI

Figure 5.18: Cluster power consumption estimation for fixed time-to-solution of 6 month, 1 month and 1 week

5.11 Conclusion

After the reference kernel implementation benchmark results in Chapter 3 were far off the estimated performance ceilings for all analyzed kernels, it would not provide a fair comparison of the architectural performance capabilities. Hence, this chapter intro-

duced several software optimization techniques to improve performance. The optimized kernel improved node performance by up to 7x and got close to the prior estimates. The next step introduced a domain decomposition model enabling the processing of larger sized volumes, which requires multiple nodes. With sophisticated, asynchronous inter-node communication it is possible to hide latency for sending and receiving data but packing and unpacking of such cannot be hidden for x86-based architectures. As a result fast computation times suffer the most from the additional overhead. By optimizing the node volume size to the node's memory capacity the overhead could be decreased further but not eliminated completely. Hence, the performance difference between architectures is partly diminished. Based on the multi-node benchmark results a simple estimate of cluster power consumption for an example large-scale survey is given. Normalized over a fixed timeframe of one week, complex kernel like TTI would require clusters that consume at least 41 MWatts, for node setups based on M2090s, and more than 150 MWatts for x86-based node architectures. Such an enormous amount of power is unreasonable to provide or to pay even for large data centers. As the complexity of seismic kernels further increases and larger survey sizes are targeted, such seismic kernels require a new approach designing novel energy efficient compute architectures. A promising design methodology called "hardware/software co-design", adapted from the embedded and mobile market, is presented by Chapter 6.

Chapter 6

Hardware, Software Co-Design Methodology

As in other scientific fields, the use of HPC within the seismic industry keeps growing. It is fueled by the wish for more accurate and physically correct subsurface images. Accurate images are critical to drive drill decision, which can cost US$ 100+ million [80]. Only large compute clusters are enabling application of computational intense, high-quality imaging methods in a reasonable amount of time. As cluster sizes increase, energy bills grow ever higher. The outlook on spending multi-millions of dollars on energy bills for future systems requires for more energy efficiency (see Section 6.1).
In 2009 Franz-Josef, head of the *Competence Center High Performance Computing* (CC-HPC) department, purposed a project called *Green Wave*. The Green Wave project presents one way to tackle this energy efficiency challenge in seismic imaging. Its name originates from *Green-IT* which focuses on energy efficient computing and *Wave* from the wave-equation used for seismic imaging. The hardware/software co-design approach used by the Green Wave project is based on the Green Flash project methodology that was introduced in 2008 (see Section 6.2) . Green Wave uses the hardware simulation platform developed under the Co-Design for Exascale (CoDEx) project to simulate new core and socket designs. The CoDEx project is able to give cycle-accurate FPGA accelerated simulations of node designs running production code including cycle-accurate main memory simulation. CoDEx makes use of the Tensilica Xtensa Processor Generator toolchain (XPG) . With XPG it is possible to quickly implement and test new core designs. With utilization of the projects stated above it is

possible to test different node, socket and core design approaches in much less time than it would take in a regular hardware development cycle.
This chapter gives an overview of projects and tools this thesis is using and / or based on. It first describes the motivation for the Green Flash project and introduces the project itself right after. Section 6.4 introduces the CoDEx project. Later sections are introducing chip power and area modeling which are critical for accurate energy efficiency predictions.

6.1 The Energy-Efficiency Challenge

In 2005 the computing industry was facing one of their biggest challenges. Since then the fulfillment of Moore's Law [59] was achieved by increasing the processor clock rates. Until 2003 studies predicted that by 2005 a core would be able to achieve clock rates of 10 GHz [93]. Instead of 10 GHz cores, like predicted in the early 2000's [48] the Pentium 4 design was hitting the power consumption and heat dissipation ceiling, which made it impossible to further increase clock frequencies. Therefore, the computing industry switched to reduced clock rates but multiple cores per socket. The multi-core age had begun and Moore's law could be fulfilled again.
As described in Section 1 the increase in performance for compute clusters is reached by replicating the number of processing units. Unfortunately, extrapolations studies [46] show that with the pace of architectural advance an exascale supercomputer would consume about 200MW which would directly translate into approximately US$ 200M costs for the annual energy bill. As no data center is willing to spend that much money on power current research is eager to find new technologies to build future supercomputers. The three main technology paths in high performance computing and scientific computing are:

1. The multi-core approach in which we maintain highly complex cores and replicate them. Two examples are x86 and Power7.

2. As alternative GPUs or accelerator cards are used to increase system performance - like the NVIDIA Fermi or the IBM Cell architecture.

3. The many-core or embedded technology path; In this case many simpler, low power cores are used. An example is the IBM BlueGene architecture.

6.2 The Green Flash Project

John Shalf introduced the Green Flash project in 2008 [107]. It focuses on the third technology path (see Section 6.1) using many simple cores to achieve high performance and to keep the power consumption low at the same time. Stating Mark Horowitz of Stanford University & Rambus Inc:

> Years of research in low-power embedded computing have shown only one design technique to reduce power: reduce waste.

The main sources of waste on a chip are:

1. Wasted transistors: Putting transistors onto a chip that are not needed by the software results in an increase surface area and therefore in an increased power dissipation.

2. Wasted computation: Modern complex x86 cores use a complex architecture to hide memory latencies to be able to run at high frequencies. This complex structure leads to useless work if branches are incorrectly predicted and forces the pipeline to flush e.g. useless work/speculation/stalls

3. Wasted bandwidth: Inefficient sized on-chip memory like caches and number of registers can increase data movement to the main memory.

4. Designing for serial performance: Based on evolution and addressing required backward compatibility current x86 cores designs are based on serial performance design.

The Green Flash project takes the co-design methodology, adopted from the embedded and mobile computing industry, and utilizes it for high performance computing. The embedded market chip designs, as found in current mobile devices like tablet computer and cell phones, have to be optimized to minimal power consumption to maximize battery lifetime. And exactly like the HPC community the embedded market faces the same challenges of multi-core design with regards of energy efficiency.
Standard on-market x86 architectures carry many instructions to provide backward compatibility. Most of such instructions take die space and consume power but are not needed by most applications. To reduce wasted transistors Green Flash takes the

basic instruction set architecture (ISA) of Tensilica's Xtensa core 6.3 which consists of only 80 instructions and uses Tensilica's TIE language 6.3.1 to add only instructions that are needed for a certain application. Different than a completely customized chip design or mapping of a kernel to FPGA, the Green Flash methodology just tailors the chip design towards a class of application and not a specific kernel. This way, a core designed this way can deliver good performance and energy efficiency for a broad range of applications within that class.

The problem of creating such a chip or cluster design for specific classes of applications are the design costs. The main costs of building new chips and architectures are the design and verification costs. The mobile and embedded space became the fastest growing market for integrated circuits in recent years. Green Flash leverages the vibrant market for intellectual property (IP) to keep costs low. Such IPs are already pre-verified and tuned for maximized energy efficiency. Hence, to keep costs down the lowest-level building blocks of the Green Flash design are pre-verified IP components from the embedded space. Using the Tensilica Xtensa Processor Generator Toolchain 6.3 it is possible to layer novel processor extensions and communication services on top for greater performance and efficiency based on Tensilica's LX4 core. This approach minimizes the amount of design custom logic, which in turn reduces verification costs and design uncertainty. For Green Flash, the chip itself is not the commodity, but the IPs, the chip is assembled of, are.

Additionally to using IPs for cost effective hardware design, Green Flash leverages commodity processes and tools that were initially directed to the embedded market.

The alternative approach for developing scientific computer system can be summarized in four distinct steps:

1. Choose a class of scientific applications and analyze their requirements.

2. Leverage COTS technologies from the embedded market to design and build power efficient tailored hardware.

3. Design the system based on the requirements of the applications.

4. Use of the hardware/software co-design approach to tune hardware and software to achieve best performance per Watt.

For a rapid turnaround time for design decisions the Berkeley Emulation Engine 3 (BEE3) (see Section 6.4.1.1) is used to test new hardware design configurations. The tool of choice for the core design is the Tensilica Xtensa toolkit 6.3.

6.3 Tensilica Xtensa Processor Generator Toolchain (XPG)

The *Tensilica Xtensa Processor Generator Toolchain* (XPG) introduces a novel approach to rapidly design RISC based core that meet the four main goals. The Tensilica XPG manual [96] describes the main goals of XPG as follows:

1. *Reduced code size*
2. *Improved general-purpose embedded processor performance*
3. *Reduced power dissipation*
4. *Flexible extension of the architecture to address demands of the application*

Reduced code size is not an important aspect of HPC. It can even have negative effects since loop unrolling or other optimization techniques might be disabled. Improved general-purpose performance sets the founding for a broader field of applications but most important the third and fourth point for the purpose of HPC. The low power dissipation combined with the flexibility to extend the core with custom instructions. The highly energy efficient Tensilica LX4 core is a single-issue, in-order instruction processor combined with a floating-point unit that can be customized in several dimensions. Tensilica's Xtensa Processor Explorer and the Xtensa Processor Generator enable rapid prototyping of custom microprocessor cores. At 45nm, the LX4 can achieve a clock rate of 1 GHz. The XPG tool, allows the straightforward addition of new instructions to the base LX4 ISA , as well as additional memory and inter-processor network interfaces. Figure 6.1 presents the Tensilica workflow. After writing the processor configuration with the Tensilica Explorer, XPG automatically generates a complete design, verification and software development environment including C/C++ compilers, debuggers, and functional models that facilitate rapid software porting and testing of each new architectural variant. This environment for rapid prototyping and cycle-accurate emulation environment is central to the hardware/software co-design process.

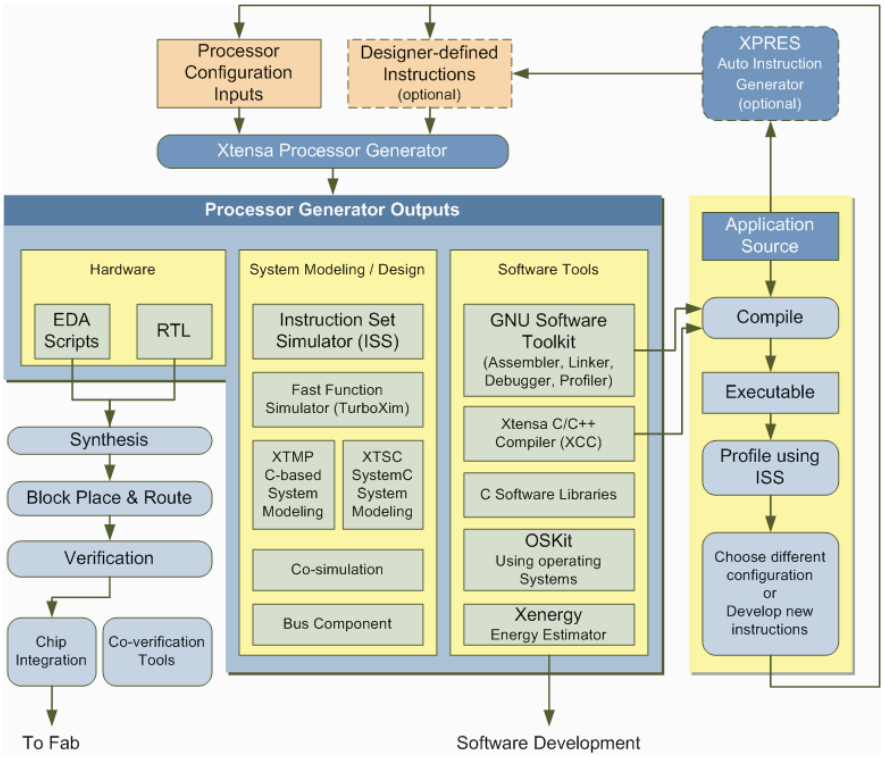


Figure 6.1: The Tensilica Workflow [95]

6.3.1 Tensilica Instruction Extension (TIE)

Dependent on the target field of use a CPU faces different requirements. A way to tailor the chip design towards a minimum number of instructions required for program and instruction encoding is the *Tensilica Instruction Extension* (TIE). With TIE it is possible to extend the basic RISC ISA with application specific instructions to improve performance, minimize cycles per instruction, chip area and power consumption. All RTL blocks added per checkbox or TIE are instantly added to the core design and all necessary changes are made to the environment. The Tensilica TIE whitepaper [97] summarizes the advantages as follows:

- *Firmware programmability to accommodate changes in requirements, specifications, and standards.*
- *Correct-by-construction hardware assembly, which greatly reduces the need for slow hardware verification.*

- *Fast, high-level system simulation through instruction-set simulators running the actual C application code.*

Listing 6.1: Example Tensilica Instruction Extension

```
regfile VR 128 16
ctype vec128 VR

operation RotateIn_LSBs{inout VR v, in FR lsbs}{}{
 assign v = {v[95:0], lsbs};
}

operation RotateIn_MSBs {inout VR v, in FR msbs}{}{
 assign v = {msbs, v[127:32]};
}
```

The code snippet 6.1 shows an example TIE how it is used for Green Wave. The first line defines a new register file `regfile VR 128 16` called `VR` with 16 registers. Each register is 128-bit wide. The second line defines a new variable type called `vec128`. This name can now be used in C/C++ implementations to define 128-bit wide vector. This corresponds to the `__m128` vector type in the SSE instruction set. In the following two custom instructions are defined. The instruction `RotateIn_LSBs` rotates a new 32-bit floating-point value into the upper 32 bits of the vector. All other vector entries are shifted by 32bit to left. Instruction `RotateIn_MSBs` rotates in a new value to the lower 32 bits. Both instructions have two input and one output parameter. The previously defined new register file `VR` is now used to define one input and the output parameter. The seconds parameter `lsbs` or `msbls` respectively, is the 32-bit value that should be rotated into the vector `v`. By assigning the, by curly brackets concatenated, vector to the output parameter `v`, the corresponding result is written into a register in `VR`. A detailed explanation of the usage of such instructions for seismic kernels is given in Section 8.1.3.

The core configuration then needs to be uploaded to Tensilica server that builds the core. For relatively simple single core simulations the *Tensilica Instruction Set Simulator* (ISS) can be used. It is fully integrated into the Xtensa Explorer and offers in depth analysis of assembly code, pipeline usage, instruction distribution and debugging capabilities.

For more complex designs the *Xtensa SystemC* (XTSC) is used. XTSC provides

a programming interface to connect predefined or custom defined building blocks to each other - such as binding of local stores or DMA engines to core interfaces or creating so-called "TIE queue" connections between cores. XTSC supports simulation on *Transaction Level Modeling* (TLM) and pin-level modeling. Listing 6.2 presents a XTSC TLM example code that creates a core and connects a memory to the dataRAM port of the core.

Listing 6.2: XTSC Example

```
#include <xtsc/xtsc_core.h>
#include <xtsc/xtsc_memory.h>

int sc_main(int argc, char *argv[]) {

    xtsc_core      *core0;
    xtsc_memory    *dram0;

    [...]    // Initialize XTSC

    // Create core core0
    core0__parms.extract_parms(argc, argv, "core0");
    core0 = new xtsc_core("core0", core0__parms);

    // Load core program
    const char *core0__argv[] = {"0",NULL};
    core0->load_program("main.out", core0__argv);

    // Create local memory "dram0"
    xtsc_memory_parms dram0__parms(16, 0, 0x5ff80000, 0x80000, 1);
    dram0__parms.extract_parms(argc, argv, "dram0");
    dram0 = new xtsc_memory("dram0", dram0__parms);
    // Connect core0 to port 0 of dram0
    dram0->connect(*core0, "dram0ls0", 0);

    sc_start();
    xtsc_finalize();

    // Delete each sc_module we've created
    delete dram0, delete core0;
    return 0;
}
```

The code example first creates two predefined core (`xtsc_core`) and memory (`xtsc_memory`) objects. The object's parameters can either be set individually or read from the core configuration (`*parms.extract_parms(...)`). The standard constructor for memories requires e.g. the start address, the size and line size. The `connect(*core0, "dram0ls0",0)` function call, connects the memory object to a specific interface of the core. Access behavior and requirements on latency are defined by the interface and how the memory was defined. By calling `sc_start();` the previously loaded binary (`load_program("main.out", core0)` is executed.
The SystemC source code of all XTSC objects is available to the user. This enables the developer to adjust object properties to fit the requirements or to implement completely new objects.

Additionally the XPG tools used to design a new processor core can generate a complete gate-list for the target design (synthesizable register transfer level or RTL), which can target an ASIC design flow or can be uploaded to an FPGA for cycle-accurate emulation of the target chip design. Such a FPGA platform is the Berkeley Emulation Engine 3 (BEE3).

6.3.2 The Green Flash Architecture

The Green Flash project introduced the co-design methodology to the scientific computing space [21, 107]. As a first target climate computing was chosen. Climate computing has high computational demands and timing constraints when it comes to cloud resolving simulations that cannot be solved with current cluster technology. Based on analysis of such code the novel Green Flash network-on-chip (NoC) architecture using photonics [37] and leveraging low power Tensilica cores was introduced in 2008.
Detailed analysis of different NoC designs shows that a two-layered concentrated torus network fits best the requirements of stencil based climate simulations [4]. Using this inter-processor communication fabric enables communication between cores via two methods – one data path is via the memory interface (load-stores and DMA) and the other is TIE Queues (Tensilica Instruction Extension) messaging interface, which transfers small messages like acknowledgements, directly between cores on the same socket, bypassing the memory subsystem. Messages can be pushed into the queue with a single instruction. The queues can be polled to check their depth by both the receiving and

the sending processor, and can operate in blocking mode (if the queue is full) or be programmed to throw an exception if queue depth is exceeded. This mechanism supports fine-grained inter-processor synchronization primitives for global address space memory consistency, feed-forward pipelines for streaming data, and ultra-low-latency word-granularity inter-processor communication. Figure 6.2 presents the Green Flash NoC.

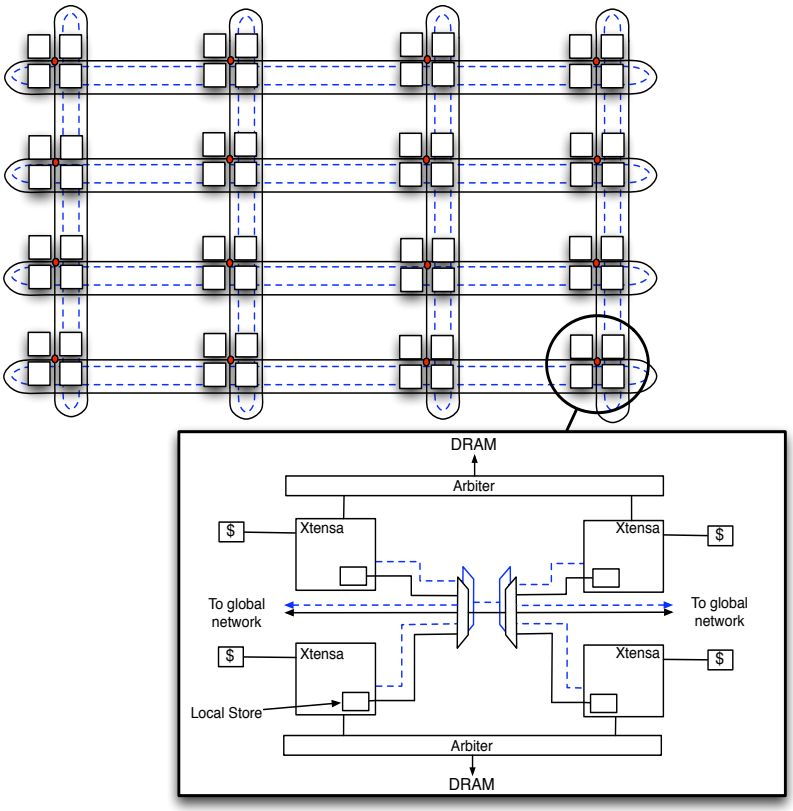


Figure 6.2: The Green Flash on-chip Network with Cores

6.3.3 Green Flash Chip Power Modeling

The motivation for this work is based on the development of a highly energy efficient system design. Therefore a critical point of this work is an accurate power modeling to prove the desired energy efficiency improvement compared to on-market architectures. Energy for events originating in the cores is calculated using the energy estimates provided by the industrial-strength Tensilica tools [96]. These estimates are created

from feedback given to Tensilica from customers who fabricated their processors then measured the actual power consumption. Second, the dynamic energy for the caches and local stores is modeled on a per transaction basis using CACTI5 [99] modeling. Third, on-chip network energy is calculated by starting with the total on-chip network communication requirements and then scaling the energy numbers from recent studies [51] for the target process technology. The NoC traffic patterns for halo exchange and associated power are modeled in detail using PhoenixSim, in collaboration with Columbia University [36, 37]. The network simulation uses router and wire power costs derived from Dally and Balfour's study on electronic NoC modeling [4]. Leakage power is assumed to be 20% of peak power consumed by the processor, on-chip memory and network for any configuration. DRAM energy is modeled via the DRAMSim2 [105] modeling tools and Micron datasheets [57].

6.3.4 Green Flash Chip Area Modeling

The area of a given processor configuration is an important metric due to its effect on the end cost of fabricating and packaging the ASIC design. To this end, the hardware configuration area is modeled within the design space, assuming 45nm chip lithography technology. The Tensilica toolchain provides direct estimates for a 45nm design. For custom processor extensions the Tensilica tools provide area measurement in terms of gate count. As the gate count of the total processor is also provided, it is straightforward to extend the area estimate of the Tensilica tools by the associated instruction extensions overhead. CACTI5 [99] is used to model cache and local store area. NoC area estimations uses the cost models proposed by Dally and Balfour [4]. The quad-channel memory interface adds 20 mm^2 to the chip area regardless of the DIMM frequency. This area estimate is consistent with the specifications for Denali DDR3 memory controller IP blocks from Cadence Inc. [11] together with Silicon Creations [87] Programmable Phase Locked Loop (PLL) for the physical interface.

6.3.5 Green Flash Chip Performance Modeling

Previous performance modeling of local store (LS) architectures [108] has shown that communication (DRAM–LS) can straightforwardly be decoupled from compute (LS–FPU) on double buffered stencil codes. This allows bound and bottleneck analysis to accurately determine performance. Advanced cycle-accurate emulation systems and

rapid design synthesis tools (from Tensilica) play a central role in the Green Wave design. The Tensilica Inc. XTensa Processor Generator (XPG) toolchain provides an end-to-end solution for quickly creating simple, semi-custom processors out of optimized, power efficient building blocks. Moreover, it provides a pathway to both software and FPGA-accelerated hardware simulation capabilities. To model the required time of direct memory access (DMA) data between DRAM and the local store, the thread's ideal block size must first be computed. The XPG toolchain is used to generate a cycle-accurate software model of the configured processor, including any custom hardware extensions added. This model uses the XTensa Instruction Set Simulator (ISS) to provide the number of cycles required to execute a given code assuming the data resides in the local store. All kernel requirements to the hardware to achieve best performance are derived in Chapter 7

6.4 Co-Design for Exascale (CoDEx)

As the energy efficiency challenge requires new hardware architecture design approaches science and industry are eager to explore more and different designs for future supercomputer clusters. Unfortunately, the commonly used hardware design development cycle is expensive and takes lots of human resources and time.

The Co-Design for Exascale (CoDEx) project is a collaborative project at the Lawrence Berkeley National Laboratory (LBNL) , Lawrence Livermore National Laboratory (LLNL) and Sandia National Laboratory (SNL) . Its goal is creating a hardware simulation platform for the hardware/software co-design methodology for maximizing energy efficiency for future supercomputers.

CoDEx couples the cycle-accurate node simulation methodology introduced by the Green Flash project with the ROSE compiler framework from LLNL. With this framework is it possible to extract and extrapolate memory and interconnect traces to large-scale cluster systems, automatically. And finally the Structural Simulation Toolkit (SST) [79] simulator from SNL to simulate massive interconnection networks.

6.4.1 Research Accelerator for Multiple Processors (RAMP)

In computer architecture research, software simulation and emulation or ASICs were chosen to test new designs. The architect had to chose between a cheap and coarse grained software simulation and the accurate but very expensive ASIC option, which does not enable to create a huge variety of different designs. Since hardware designers have to deal with a huge amount of transistors per chip and heterogeneous or homogenous, thread-level parallel architectures the RAMP initiative was founded. The RAMP initiative is a consortium that includes six universities like Stanford, Berkeley, MIT and several industrial partners like Microsoft Research, Xilinx, Sun Microsystems and IBM [58]. Its goal is to provide an FPGA emulation platform for multi-processor design studies. RAMP leverage the Berkeley Emulation Engine 3 (BEE3) as FPGA platform. The emulation is performed directly on the gate-level RTL mirroring the physical design that nominally would be used for place-and-route, mask generation, and chip fabrication. The fact that the emulated logic is precisely the actual circuit design for the target chip provides a superior level of condense in our software-based methods as we can constantly verify our software simulation results against those of the exact model of the hardware platform.

6.4.1.1 The Berkeley Emulation Engine 3 (BEE3)

Berkeley Emulation Engine was created to ease the efforts in hardware emulation [20]. The BEE3 hardware emulation platform and copious performance data provide a fast, accurate performance emulation environment allowing the benchmarking of real codes ensuring the application developers are intimately involved in the hardware / software co-design process.

BEE3 is the platform of choice for the RAMP initiative. It uses a single printed circuit board (PCB) on which four Xilinx Virtex 5 FPGAs are used. Each board can hold up to 64 GB of DRAM and is packed into a 2U enclosure with several I/O ports. With these ports it is possible to connect BEE3 to servers, to use it as an accelerator or to connect several BEE3 boards together. The "Ramp Blue" project from UC Berkeley consists of 1000+ core system build of 21 BEE2's [20].

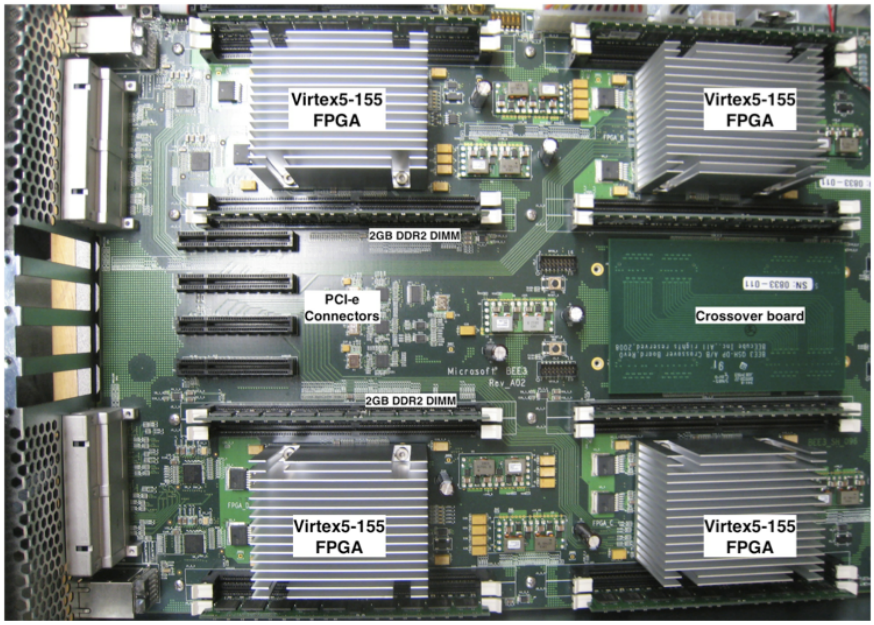


Figure 6.3: The Berkeley Emulation Engine 3.

6.5 Summary

The HPC community is facing one of their toughest challenges in developing future large-scale compute clusters. This chapter introduced the energy efficiency challenge, which requires reconsidering how future compute architectures are designed. Unfortunately, only a limited number of applications are demanding such large-scale systems. Rather tuning an application to the underlying hardware architecture this chapter introduces the hardware/software co-design approach to HPC architecture design. The methodology uses an iterative process in which the requirements of the application after each optimization step are analyzed and hardware being adjusted accordingly. Since the standard hardware development cycle is too complex and cost intensive, new hardware designs must be simulated to receive accurate performance estimations for different design choices ahead of production. This chapter introduces the CoDEx project applying the Tensilica toolchain XPG which enables rapid hardware design studies and enables scientists and hardware architects to run production codes in cycle-accurate fashion on FPGAs. This enables direct impact measurement of hardware optimizations on performance and energy efficiency.

The preceding Green Flash project proved this design concept to be beneficial for cli-

mate codes. The recent challenges the seismic industry is facing with physical accurate subsurface modeling with RTM, it serves as a good target for exploring the gains of co-designed architectures in terms of energy efficiency. To receive a co-designed architecture for seismic propagation kernels, Section 7 first presents a programming model that provides maximum temporal data locality and analyzes the requirements of the kernels applying this model.

Chapter 7

The Green Wave Programming Model and Requirement Analysis

The hardware/software co-design methodology targets tailoring a machine design to optimal efficiency. In order to achieve a balanced architecture design, a detailed analysis of the application requirements is required.
This chapter presents the steps that need to be applied to receive an energy efficient hardware design that serves the requirements of the application. First the "plane-scheme" programming model is introduced (see Section 7.1) as it provides optimal data locality for stencil-based kernels on software-managed local store architectures. Following, a domain decomposition approach is presented that works best for this programming model (see Section 7.1). And finally, this chapter analyzes the requirements of the introduced seismic kernels and derives equations to determine the minimum requirements derived from kernel type and programming model.

7.1 The Plane Scheme Programming Model

The plane-scheme programming model describes an effective programming model for architectures with software-managed local caches running stencil kernels. Considering dimensions $X = Y = Z = N$, a subdomain can be seen as Z planes of size N^2. Given the nature of stencils, the data access patterns produce a high amount of cache misses for hardware-managed caches due to poor spatial locality. Spatial locality is defined as used data values within a single memory access e.g. a cache line. The worst spatial lo-

cality occurs if a complete cache line is accessed but only a single word is used, whereas the best spacial locality incorporates all the words within the cache line.

Memory addresses of points in Z-direction are too distant to be captured within the same cache lines of points in X or Y-direction. Therefore, additional cache lines need to be brought into low-level caches while utilizing only a limited number of bytes. In worst case such cache lines are replaced because of the replacement policy, although temporal locality could be exploited for subsequent points.

With software-managed local stores, the right amount of N^2 planes are kept in the local store to serve the demand of the stencil computation in all dimensions. The size of the planes has to be adjusted appropriately to the local store size to achieve a maximum spatial and temporal locality. The complete domain is then computed by streaming through all Z planes.

Each Green Wave node owns dedicated main memory to hold the required node subdomain. This subdomain is further 2D domain decomposed into columns along the z-axis. The number of columns correlates with the number of cores per socket. Each core then applies the plane scheme, which streams through all planes in z-direction. Each cores' subdomain is extended by a halo region that holds the neighboring data required for the stencil computation. Figure 7.1 shows the Green Wave domain decomposition and plane scheme.

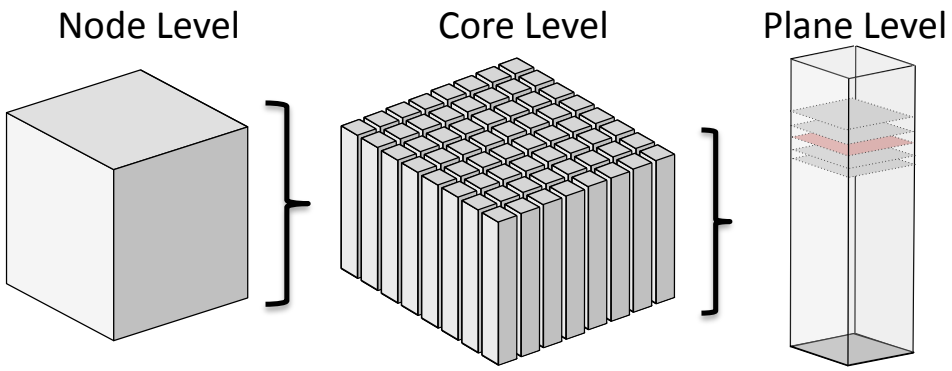


Figure 7.1: Domain Decomposition for Green Wave

7.2 Estimation of the Number of Processing Units

In the last decade average processing power grew according to Moore's Law. Unfortunately, memory bandwidth does not follow the trend, which lead to an oversubscription in terms of computing power for a high amount of scientific kernels and therefore a waste of energy. Hence, it is critical that energy efficient hardware architecture be balanced in terms of computational power and memory bandwidth and where one main advantage of the hardware/software co-design approach comes in. Since hardware is tailored towards certain classes of applications the computational as well as the bandwidth requirements are known.

To retrieve a balanced system the amount of cores should be adjusted so that neither an oversubscription in computational resources nor memory bandwidth appears. Because the co-design approach targets only a class of applications and not a specific one, a perfect balanced system is difficult to achieve being dependent upon how generalized the hardware design should be.

The required memory bandwidth depends on the application and the amount of cores per socket. A few questions need to be answered to determine the right amount of cores per socket:

- What is the problem size to work with ?
- Which kernel is used to retrieve my results ?
- What are the time constraints for solving a given problem size with a specific kernel ?

Three different kernels are defined as target for Green Wave. The first kernel is the isotropic wave equation, the second a wave equation kernel accounting for VTI, and the third kernel accounts for TTI earth properties. A detailed analysis of these kernels was done in Section 2.2.1.

Analyzing the arithmetic intensity then retrieves the optimal target volume size per core or socket. Arithmetic intensity describes the how many operations are performed per byte transferred from main memory (see Section 4.2.1). For stencil codes the subdomain needs to be extended by a halo region to be able to compute points on the edges and faces of the volume. The width of the halo region corresponds to the radius width

of the applied stencil. These halo points are retrieved from either neighboring cores working on subsequent or precedent domains via direct exchange or through loads from shared memory. If halos are exchanged on-chip between cores the arithmetic intensity remains constant independent of the domain size. With applied halo exchange through main memory the arithmetic intensity increases at a rate dependent upon the surface-to-volume ratio. Given the amount of data that has to be present to compute the next timestep of the volume, it is possible to retrieve the arithmetic intensity for different subdomain sizes per core. The larger the volume, when compared to the halo size, the better the flop-to-byte ratio. The maximum size of the subdomain is limited by capacity and latency requirements of SRAM and the amount of cores per socket. Determining the local store size, the chip designer has to consider the chip area for a single chip to stay within reasonable limits. Now that the number of bytes accessed for each point computed is determined, the next factor that influences the amount of cores that can feasibly be put on a socket is contingent upon the core speed itself. The faster the total floating-point throughput per core, the faster the data needs to delivered to the local memories.

The last variable that is critical to achieve a balanced system is the number of floating-point operations per point required by a kernel. This can be derived either by counting manually or by using hardware counters like PAPI.

Given all necessary variables it is now possible to calculate the optimal number of cores per socket saturating the memory bandwidth. First, the memory bandwidth requirement of a single core is calculated by dividing the number of bytes requested per point by the time it takes to compute one point in the volume:

$$\text{Core Bandwidth} = \text{Bytes per Point} \ / \ \text{Time per Point}$$

A balanced socket is now attained by dividing the total memory bandwidth provided by the memory controller with the bandwidth requirements of a single core:

$$\text{Number of Cores} = \text{Sustained Memory Bandwidth} \ / \ \text{Core Bandwidth}$$

The number of cores per socket then needs to be adjusted to receive an efficient NoC layout, however, this can lead to a slight over- or undersubscription in memory bandwidth. In an optimal case the transfer of bytes from main memory to local store

will not stall the processor. This is possible by using an asynchronous DMA engine and an applied multi-buffering communication model.

7.2.1 Multi-buffering via Direct Memory Access (DMA)

One of the main design choices of the IBM Cell processor, is the use of the SPE dedicated asynchronous *Direct Memory Access* (DMA) engines. Since SPEs do not fetch data from main memory by themselves, like in a cache-based architecture, the software is required to issue DMA transfers in order to transfer data from main memory into the local store. The DMA engine works asynchronously and remains independent from the core which makes it possible to compute and communicate data concurrently via multi-buffering, which can be performed as double- or triple-buffering. The standard way is a sequential ordering of computation and communication in three steps.

1. required data is loaded,
2. computation on this data is performed,
3. results are stored back to memory.

This order is repeated for all loop iterations. Unfortunately, utilizing this methodology stalls the functional units during the communication with main memory. To avoid stalls while loading data, double-buffering can be performed with the DMA engine. The usual way to perform double-buffering is to load data required for loop iteration $i + 1$ in a second buffer b_1 simultaneously with the computation of the iteration i on buffer b_0. The computation of iteration $i+1$ is then performed on buffer b_1 where buffer b_0 is loaded with data for $i + 2$. Computation and communication can be overlapped only when no dependencies between iterations or data exist and when the time spend in the computation is greater or equal the time spend in communication. In the same fashion, results from iteration i are overlapped with the results of the computation of iteration $i + 1$. This can be done $N\times$ loop iterations ahead and is thereby called multi-buffering. Figure 7.2 gives an example of the double-buffering approach.

Figure 7.3 visualizes the necessary planes for a 4th-order Laplacian stencil and a 2nd-order time derivative, which is how it appears in an isotropic wave-equation kernel. The double-buffering is applied along the z-axis and while results for depth plane z are

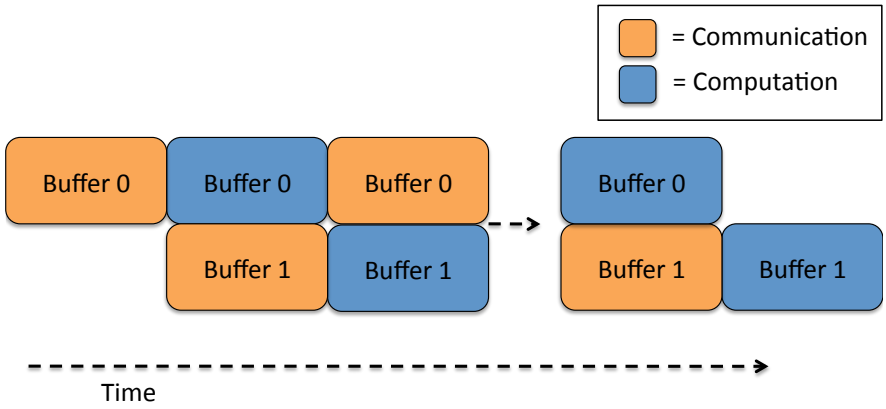


Figure 7.2: Double Buffering

computed the required planes for $z + 1$ are loaded while results from $z - 1$ stored to main memory. This way $4 \times N^2$ additional local store capacity must be provided.

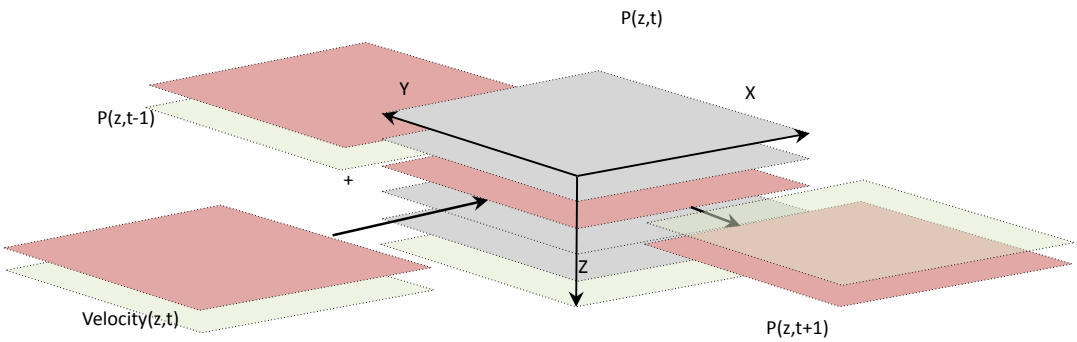


Figure 7.3: Streaming Planes including Multi-buffering and all ISO Planes

The disadvantage of multi-buffering is that additional buffer space that has to be allocated to hold multi-buffered data in the local memories. Secondly, the programming complexity increases as data transfers are explicitly software-managed; therefore the programmer might need to rethink kernel optimization strategies in order to secure adequate computation time capable of overlapping communication with computation.

7.3 Estimation of Local Memory Size

It is critical for the Green Wave architecture to have the data as close to the core as possible. Software-managed local memory architectures have proven advantages over cache-based architectures in terms of raw performance and energy efficiency [108], [109]. To derive an optimal size for the local store, it is necessary to determine the data requirements for a given kernel.

Section 2.3.6 summarizes volume requirements for all kernels analyzed in this thesis. During the backwards-in-time propagation part of the RTM algorithm, additional to the data needed for the wavefield propagation, the previously stored source data has to be read, cross-correlated with the receiver wavefield and the result stored back to main memory (see Section 2.4). Applying the plane scheme, the local store needs to be able to hold enough $x - y$ planes that all derivatives in z-dimension can be calculated. Therefore, the number of necessary planes depends upon the order in space of the derivation. Additional memory space is accounted for the velocity plane, a plane for the previous timestep, the results and the source wavefield. The results and source planes require reduced space sampling because fine-grained space sampling is needed only for computation and not for the final image.

To hide all latencies it is necessary to overlap computation with communication using asynchronous DMA transfers applying double- or multi-buffering. For this purpose additional memory space must be allocated to hold buffers for data that is loaded for the next timestep and results stored back for the previous timestep (result). The total amount of memory required (S_{LS}) can be described as follows:

$$S_{LS} = nx * ny * P$$

with

$$\begin{aligned}
P &= P_P + \alpha * P_Q + Pl_{V(z)} \\
&+ \alpha(Pl_{\epsilon(z)} + Pl_{\delta(z)} + \beta(Pl_{\theta(z)} + Pl_{\phi(z)})) \\
&+ \gamma(Pl_{sin(\theta(z))} + Pl_{cos(\theta(z))} + Pl_{sin(\phi(z))} + Pl_{cos(\phi(z))}) \\
P_P &= \left(\sum_{n=-Radius}^{Radius+1} Pl_{P(z+n,t)}\right) + Pl_{P(z,t-1)} + Pl_{P(z,t+1)} \\
P_Q &= \left(\sum_{n=-Radius}^{Radius+1} Pl_{Q(z+n,t)}\right) + Pl_{Q(z,t-1)} + Pl_{Q(z,t+1)}
\end{aligned}$$

where nx, ny are the core subdomain dimension in X and Y, P the total number of planes which is the sum of all planes required for computing the derivatives for pressure wavefield P, the velocity V, for VTI and TTI implementations the auxiliary wavefield Q and necessary Thomson and tilting parameters. Coefficients $\alpha, \beta, \gamma \in 0, 1$ where $\alpha = 1$ in case of VTI and TTI implementations, $\beta = 1, \gamma = 0$ in case of using TTI without pre-computing trigonometric functions for ϕ and θ and $\beta = 0, \gamma = 1$ if pre-computation is used.
In case of double buffering one plane has to be added for each volume and each timestep which means

$$\begin{aligned} S_{LS} \quad += \quad & X * Y * (Pl_{V(z+1)} \\ + \quad & Pl_{P(z+radius+1,t)} + Pl_{P(z+1,t-1)} + Pl_{P(z-1,t+1)}) \\ + \quad & \alpha(Pl_{Q(z+radius+1,t)} + Pl_{Q(z+1,t-1)} + Pl_{Q(z-1,t+1)}) \\ + \quad & \alpha(Pl_{\epsilon(z+1)} + Pl_{\delta(z+1)} + \beta(Pl_{\theta(z+1)} + Pl_{\phi(z+1)}) \\ + \quad & \gamma(Pl_{sin(\theta(z+1)}) + Pl_{cos(\theta(z+1))} \\ + \quad & Pl_{sin(\phi(z+1))} + Pl_{cos(\phi(z+1))}) \end{aligned}$$

In addition to the size above the halos need to be added for $Pl_{P(z,t)}$ and $Pl_{Q(z,t)}$. Because the on-chip domain decomposition is done in two dimensions four different halos are exchanged. From now on these halos are called by the direction they are facing from the a core's view, which is either West (H_W), East (H_E), North (H_N) or South (H_S). Their sizes are defined by:

$$\begin{aligned} H_W, H_B &= r * ny \\ H_N, H_S &= r * nx \end{aligned}$$

where r is the radius defined as half the order in space of the finite-difference approximation.
Applying the FD approximation to ISO and VTI kernels results in star-shaped stencils. Contrary, the stencil resulting from TTI forms three intersecting planes caused by the partial mixed derivatives. This requires additionally halo data for the edges adding $4 * r^2$ in memory capacity requirements. Furthermore, additional space is required for the backward propagation: the current plane of the forward propagated source wavefield ($Pl_{FWD(z)}$), the resulting image plane ($Pl_{R(z)}$) and ($Pl_{FWD(z+1)}$),($Pl_{R(z-1)}$) and

$(Pl_{R(z+1)})$ when double-, multi-buffering is applied.

7.4 Estimation of Main Memory Size

The required size of the main memory per Green Wave socket is calculated similar to what we have seen for local memories. The complete wavefield of size X, Y, Z is 3D domain decomposed into cubes of size X_c, Y_c, Z_c with

$$\begin{aligned} X_c &= X/\#Nodes_X \\ Y_c &= Y/\#Nodes_Y \\ Z_c &= Z/\#Nodes_Z \end{aligned}$$

Volumes of size $X_c * Y_c * Z_c$ have to be kept in memory for $P(t-1)$,$P(t)$, $P(t+1)$ and velocity V for the isotropic kernel. For VTI volumes $Q(t-1)$,$Q(t)$, $Q(t+1)$, ϵ and δ are added. Finally, for TTI implementations memory for θ and ϕ and eventually $sin(\theta), cos(\theta), sin(\phi), cos(\phi)$, if the trigonometric functions are pre-computed, has to be accounted for.

For the 3D decomposition the cube needs halos on six sides - West (H_W), East (H_E), North (H_N), South (H_S), top (H_T) and bottom (H_B). The sizes of these halos are derived by:

$$\begin{aligned} H_W, E_{BN} &= r * ny * nz \\ H_N, H_S &= r * nx * nz \\ H_T, H_B &= r * ny * ny \end{aligned} \tag{7.1}$$

with $r = radius$ or half of the order of the stencil. For TTI, 12 edges are required additionally.

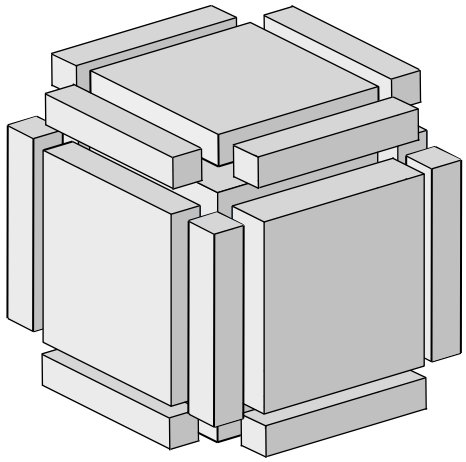


Figure 7.4: Node Subdomain with TTI Halos

$$
\begin{aligned}
E_{NS}, E_{NW} &= r * r * nz \\
E_{SW}, E_{SE} &= r * r * nz \\
E_{TW}, E_{TE} &= r * ny * r \\
E_{TN}, E_{TS} &= nx * r * r \\
E_{BW}, E_{BE} &= r * ny * r \\
E_{BN}, E_{BS} &= nx * r * r
\end{aligned}
$$

where N=North,S=South, E=East, W=West, T=top and B=bottom. Corners points are not required. No diagonal points are used by the stencils calculations of the discussed kernels. Figure 7.4 illustrates the main volume including all halos needed for TTI.

7.5 Estimation of Main Memory Bandwidth Requirements

The required memory bandwidth to and from local memories and the main memory depends heavily on the programming scheme used. The plane scheme (see Section 7.1) has proven to provide best locality and data reuse for local store based architectures. As described in Section 2.3.6 multiple volumes are required to compute the next timestep for each point of the volume. Applying the plane scheme and multi-buffering,

one plane has to be loaded for all volumes necessary for the kernel from memory while computation takes place on previous planes in z-direction. The memory bandwidth, between local- and main memory, needs to be sufficient that data is delivered to the core in time it is required. Because main memory bandwidth is set by the DDR3-1600 memory interface used by a Green Wave socket, the estimated memory bandwidth shows if the socket is balanced or if either the number of cores or the core frequency has to be adjusted to achieve a balanced system.
The amount of data per core (D_{Core}) is defined by the by the plane size and the kernel used. In Section 7.4 the data requirements of the different kernels were introduced. The same naming convention is used here. Assuming double-buffering is applied and halos are shared on-chip and not via shared memory to decrease memory bandwidth pressure a plane size is defined as $S_{pl} = nx * ny$. Given all required planes required to compute the derivatives for the plane in z-position z are already present in the local memory the following equation describes the amount of data to be loaded for each $z \in Z$:

$$\begin{aligned} S_{Read} &= S_{Pl} * (Pl_{V(z+1)} \\ &+ Pl_{P(z+radius+1,t)} + Pl_{P(z+1,t-1)}) \\ &+ \alpha(Pl_{Q(z+radius+1,t)} + Pl_{Q(z+1,t-1)} + Pl_{Q(z-1,t+1)}) \\ &+ \alpha(Pl_{\epsilon(z+1)} + Pl_{\delta(z+1)} + \beta(Pl_{\theta(z+1)} + Pl_{\phi(z+1)}) \\ &+ \gamma(Pl_{sin(\theta(z+1)}) + Pl_{cos(\theta(z+1))} \\ &+ Pl_{sin(\phi(z+1))} + Pl_{cos(\phi(z+1))}) \end{aligned}$$

As before P is the pressure wavefield, V the velocity and Q the auxiliary wavefield necessary for VTI and TTI implementations. ϵ,δ,ϕ and θ are Thomson and tilting parameters. Coefficients $\alpha, \beta, \gamma \in 0, 1$ where $\alpha = 1$ in case of VTI and TTI implementations, $\beta = 1, \gamma = 0$ in case of using TTI without pre-computing trigonometric functions for ϕ and θ and $\beta = 0, \gamma = 1$ if pre-computation is used.
Additionally $S_{Write} = Pl_{P(z-1,t+1)}$ has to be written back to memory. If the time to compute the current plane (T_{Pl_z}) is greater or equal the time to read ($T_{S_{read}}$) and write ($T_{S_{write}}$) the necessary data to and from main memory the communication can be completely overlapped with computation:

$$T_{Pl_z} \geq T_{S_{Read}} + T_{S_{Write}}$$

7.6 Estimation of NoC Requirements

Explicit stencil codes are computed in requirement of neighboring halo points. Therefore, the stencil code communication scheme is nearest neighbor based. Dependent on the size and shape of the stencil 6, 18 or 27 neighbors are required in a three dimensional domain. The main volume domain decomposition for Green Wave is 3D which means that every node works on a given subdomain cube of size $X_{sd} * Y_{sd} * Z_{sd}$. This subdomain cube is further decomposed with an 2D scheme in x and y in which the number of cores and subdomain size per core depends on the applications parameters in terms of computational intensity and memory bandwidth requirements (see Sections 7.2 ff.). The core subdomain size is defined by X_c, Y_c and Z_c where as

$$X_c = X_{sd}/\#Cores_X$$
$$Y_c = Y_{sd}/\#Cores_Y$$
$$Z_c = Z_{sd}$$

with $\#Cores_X$ defines the number of cores in x-direction for each node and $\#Cores_Y$ in y-direction. Through applying the plane scheme, each core streams through its subdomain planes in z-direction.

To achieve optimal energy efficiency a maximum compute performance needs to be achieved. Stalls caused by halo exchange should be avoided. Additionally, the communication model needs to account for the kernel. A typical approach is that each core starts processing the inner, halo-independent points first. At the same time required points are send and received to or from neighbors, respectively. Figure 7.5(left) presents this scheme. It works for isotropic and VTI kernel implementations.

The performance of TTI depends on the common subexpression elimination (CSE) optimization and therefore on the surface-to-volume ratio of the plane. Hence, a communication model that reduces the plane size to the amount of the inner points, like applied to ISO and VTI, would decrease TTI performance. Another approach is presented by Figure 7.5(right). As before, the plane computation is done in two steps.

First, computation is applied to all points with $y < ny/2$ where $0 < y < ny$. During the computation, halos for $y > ny/2$ are transferred. In the second step points $y > ny/2$ are computed and the lower half are transferred.

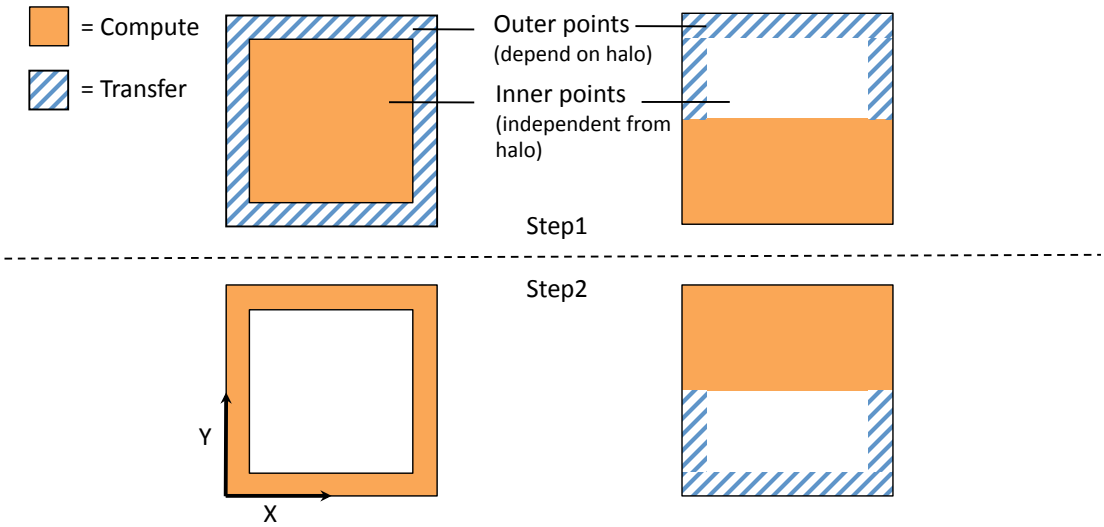


Figure 7.5: Inter-Core Communication Schemes

Certainly, the version that overlaps communication with the computation of the inner points has a higher pressure on NoC bandwidth. If the time to compute these points (T_{inner}) is greater or equal the time to exchange data (T_{ex}) with the neighbors no stalls appear. Therefore Network-on-Chip bandwidth must provide exchange of halos in a time less than T_{inner}:

$$\text{NoC Bandwidth} \geq \text{Size of Halos} \ / \ T_{inner} \tag{7.2}$$

If Equation 7.2 is true this leads to $T_{inner} \geq T_{ex}$ which is the requirement to hide communication. Figure 7.6 illustrates a complete summary of processing a plane. The single plane is decomposed into the different dependency region. Blue shows the actual halo region that depends on data from the neighboring cores. Red marked points depend on the halo region and cannot be processes until the halo is present. Green colored inner points are independent from the halo. Additionally, the figure shows which data is read from memory and is written to memory during the processing of a plane.

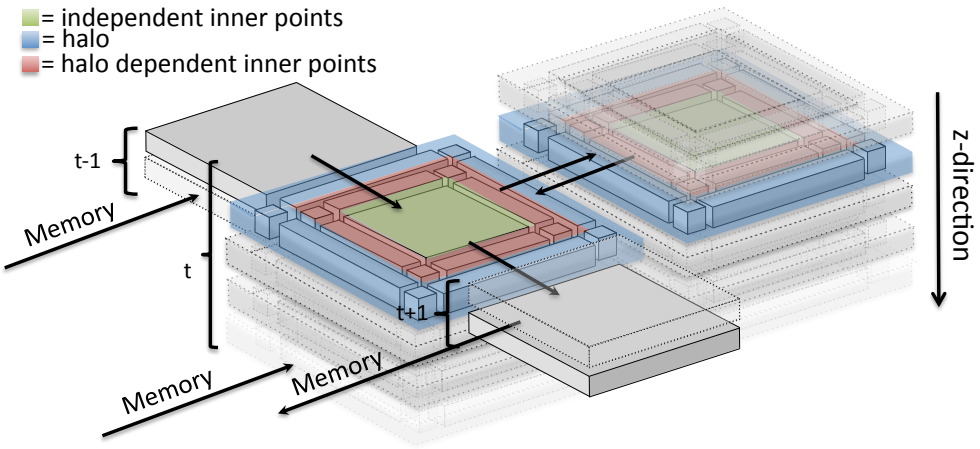


Figure 7.6: Illustration of the plane scheme, including double buffering and halo exchange

Even the co-design approach targets on retrieving a balanced system a slight over-subscription in memory bandwidth can appear. It is then possible to share halo points through shared memory. This way each core reads the required halo points when loading the corresponding plane from memory. This increases local store capacity requirements and memory bandwidth pressure but reduces programming complexity significantly.

7.7 Summary

One foundation of the hardware/software co-design methodology is to analyze the requirements of the application. This chapter introduced the plane scheme programming model which has proven best capturing all temporal recurrences for stencil-based kernels on software-managed local store based architectures. Given a fixed programming model and the initial limitations on hardware this chapter maps the requirements of the application into equations. These equations now allow easy adjustment to changes in software or hardware parameters how it appears in the iterative manner of the hardware/software co-design approach. The following Chapter 8 now uses these equations to design a balanced architectures that fit the requirements of the isotropic, VTI and TTI kernels.

Chapter 8

The Green Wave Architecture and Single-Node Study

This chapter applies the hardware/software co-design methodology introduced in Section 6 and the equations describing the kernel requirements from Section 7 to derive Green Wave design parameters for isotropic, VTI and TTI kernel implementations.
After introducing initial limitations to the design (see Section 8.1) it is explained how the application requirements affect the size of on and off-chip memories, core counts, core optimizations, bandwidth requirements and the on-chip network. Next, the Tensilica Instruction Set Simulator (ISS) is used to receive cycle-accurate performance results. The Green Wave benchmark results are then compared to single-node throughput performance and energy efficiency of the evaluated COTS architectures (see Section 3).

8.1 The Green Wave Node

In this section the Green Wave node architecture is described. Based on the Green Flash architectural design that was designed for climate codes, the requirements of seismic wave propagation kernel are taken (see Section 7), system parameters adjusted and new custom instructions introduced. A Green Wave node is considered a single socket. Main memory is dedicated to a socket and not shared by default.
An important point of the hardware/software co-design methodology is to keep verification and design costs low by using commodity IPs instead of full-customized hardware (see Section 6.2). This forms initial limitations to the hardware design. The Green

Wave design is based on Tensilica's Xtensa core and designed with the *Xtensa Processor Generator* (XPG). Currently, the Xtensa LX4 supports up to 1 GHz clock rate and a silicon feature size down to 45nm. The basic core design owns a floating-point unit capable of FMA. With a 5-stage pipeline it is capable of 2flop/cycle peak. Furthermore, the Green Wave socket uses a COTS quad-channel, DDR3-1600 memory interface that restrains the total number of cores per socket due to a maximum bandwidth of 51.2 GB/s.

8.1.1 Local-Store Size and Core Count

The first adjustment that takes place is selecting the on-chip memory size appropriately to be able to capture all temporal recurrences for the high-order stencil kernel how they appear for ISO, VTI and TTI. As previous studies on the IBM Cell processor have demonstrated, substantial efficiency benefits can be achieved by using software-managed local stores instead of hardware-managed caches [108] — particularly for stencil computations. Therefore, Green Wave utilizes a local store architecture as the primary on-chip cache memory. A small L1 data cache remains for each processor in order to support convenient code porting, but the local store is the primary approach for latency hiding and capturing temporal recurrences. One of the drawbacks of the IBM Cell processor is that data and instructions are both located within the local store. This caused problems since application developers had no fixed local store size to program against. To keep Green Wave a strict Harvard design [91], data and instructions are kept separately. A small 8 KB L1 instruction cache prefetches instructions and works concurrent and independent to the data fetch of either local store of data cache. The local store size depends on the implementation of the plane scheme 7.1 and the corresponding plane size. The optimal core plane size depends on many factors. One has to make sure that all data needed for the stencil computation fits into the local store. At the same time the plane size needs to be big enough to provide enough inner points to be able to overlap halo exchange communication with computation. Last, the size of the local store needs to be small enough to keep the final die size in reasonable limits. For isotropic kernel the best plane size results in 64x32 points; for VTI kernel this optimal ratio is retrieved for a 2D plane of 64x32 points as well. The TTI plane size is further reduced to 32x16. Additionally, the halo region has to be added which makes the required local store size dependent on the stencil-order in space.

In total the 8th-order in space ISO kernel has memory requirements of about 120 KB, VTI requires about 250 KB and TTI about 80 KB already according for additional buffers for common subexpression elimination and precomputed sine and cosine value of dip and azimuth constants.
Once Green Wave is configured with sufficient on-chip memory to capture all recurrences, the number of processors is determined that is required to saturate the off-chip memory bandwidth. Note that an iterative optimization process is necessary, as changes in the core count requires a resizing of the local-stores (to incorporate the halos from the blocked implementation), which in-turn impacts the optimal core count (to effective capture temporal recurrences). This analysis resulted in a design choice of approximately 120 processor cores and 120 KB of local-store per core for ISO, 250 KB and about 96 cores for VTI and 160 KB with 380 cores for TTI. The requirements of ISO and VTI are completely different to the requirements of TTI. As ISO, VTI are more likely to be memory bandwidth bound, TTI depends on high floating-point throughput. Due to the high demands on local store size for TTI kernels the number of cores has to be limited to keep the chip area and power consumption within reasonable limits (see Section 8.1.8 and 8.1.9). All figures were adjusted to simplify the layout of the SRAM mats on chip as well as the NoC topology. This results in 128 cores with 128 KB local store, 96 cores with 256 KB and 256 cores with 128 KB for ISO, VTI and TTI kernel optimized Green Wave designs, respectively.
All cores on a socket are able to write directly into each other local store. For that the local store is mapped into the main memory. Each time a core writes into a specific address range that belongs to a local store of a core the data exchange is performed on the NoC and data is directly written into the corresponding local store.
A load instruction from local store to a register reads 128 bits at a time. For 32-bit requests only the lower 32 bits are used. Besides the Harvard design the local store technology is taken from the IBM Cell processor.

8.1.2 Local Store Access Latency

The number of floating-point instructions per cycle heavily depends on the location of operands. Every time an operant is not present in one of the registers, the local memories has to be accessed. This can be either the stack that resides in a small data cache or the local store. The local store is attached to the dataRAM interface of the Xtensa

LX4 core. Using a five-stage pipeline the core requires a fixed latency of 1 cycle per memory access (t_{acc}). SRAM of 256 KB are unlikely to achieve 1 cycle t_{acc} at 45nm, which makes it necessary to hide the latency. This can be done several ways.
A way is used by IBM's CBEA 3.3.3. The IBM Cell processor has eight synergistic processing units, which leverage software-managed local stores of 256 KB size. These local stores are pipelined to achieve minimal access latency. Even though the architectural complexity increases, combined with a low access time of 1ns to 2ns, it is possible to provide one access per cycle with pipelined memory.
Further, the local store could be divided into multiple smaller SRAMs. This way the access latency could be cut in half by alternating accesses.
Another idea is to set a prefetch buffer between local store and register file. Due to the streaming access behavior only one access to the local store would appear with higher latency and further requests would be read from the buffer. This way loads of four 32-bit words would require only one cache line sized access with >1 cycles t_{acc}.
This thesis does not assume a specific implementation. Since the Xtensa core demands a single cycle latency for its dataRAM interface, for further core simulations a single cycle latency is assumed.

8.1.3 Core Design Optimizations

The Tensilica design flow enables to add hardware optimizations specifically tailored to RTM-based stencil computations. For such hardware optimization special instructions are created which are directly accessible though intrinsics in high-level languages. Because the Green Wave architecture includes a correctly sized local store for capturing all temporal recurrence of data, the performance model only requires a fixed latency model for memory accesses, and focuses the optimization effort on reducing instruction count. After application of one optimization it is important to check whether the design is memory bound already or if further attempts reducing the instruction count could further increase performance.

8.1.3.1 Custom Instruction Design

An approach towards reducing instruction count is the creation of custom instructions that allow "fusing" of commonly used operations. Here it is possible to leverage a key feature in the Tensilica LX4 design flow that allows the creation of custom instructions

Listing 8.1: Original Version of the Code.

```
for (z,y,x)
    value1 = C[1] *(
      src[idx-1] + src[idx+1] +
      src[idx + YStride1] + src[idx - YStride1] +
  src[idx + ZStride1] + src[idx -  ZStride1]);
     [...]
```

and data types [96]. These instructions are written in a language similar to Verilog that Tensilica calls via the TIE interface. These custom instructions are fully supported by the Xtensa compiler and become native intrinsics for software development in high-level languages.

The first custom instruction allowed the parallel computation of Y and Z loop indices for a given stride. These indices are stored in special registers for later use as offsets into the data array. The original code version would calculate these offsets individually and then do pointer arithmetic to fetch the correct data point from the array. By pre-computing these values via custom instructions allows the user to pass a pointer to the start of the array, then select the direction (Y or Z) as well as the offset (1 through 8) of the desired value. This instruction fetches the pre-computed offset from the register, calculates the address, feeds this new address to the processor's load/store unit and then returns the value — essentially collapsing two instructions into a single fast array index operation. The following code presents an example of the ISO stencil calculation before 8.1 and after 8.2 custom instructions are used.

The reader should be aware that the following code snippets are written in pseudo code only to help understand the different parts of the co-design process.

The software optimized kernel precomputes the strides `YStride` and `ZStride` and adds them accordingly. To use the special function units the software developer simply has to call the intrinsic functions `GetYValue` or `GetZValue`. These function calls are directly translated into assembly instructions, which use special purpose hardware defined with Tensilica's TIE language. An example code snippet of the TIE implementation is presented by listing 8.3.

Listing 8.2: Optimized Green Wave Code Version.

```
float[] YSrcValues, ZSrcValues;
vec128 current, current_m;  //128-bit vector register
int idx;       // array index

for (z,y,x)
    current = *(vec128*)(&src[idx+1]);
    current_m = LoadVec128_Reverse(&(src[idx-3]));
    ComputeIndicies(idx);
    value1 = C[1]*(
       GetValue(current,0) + GetValue(current_m,0)
    + GetYFrontValue(YSrcValues, 0) + GetYBehindValue(YSrcValues, 0)
    + GetZFrontValue(ZSrcValues, 0) + GetZBehindValue(ZSrcValues, 0));
    [...]

    RotateIn_LSBs(current_m, GetValue(current, 0));
    RotateIn_MSBs(current, src[idx + 4]);
```

Listing 8.3 presents a code snippet from the Green Wave custom TIE instructions. It shows a custom instruction `GetZFrontPoint`. The Xtensa compiler automatically creates a corresponding assembly instruction that can be pipelined and a corresponding function call for C/C++ like seen in code listing 8.2. The presented function returns the corresponding value of a point that lies `sel` z-planes "in front" of the current point. The instruction takes two kinds of parameters - explicit and implicit used ones. The explicit parameters need to be given with the function call by the high-level language call. Enclosed within the first pair of brackets, such parameters are denoted with the keyword `in` - return values with the keyword `out`. The parameters in the second pair of brackets describe the implicit parameters, like registers. Given the `StrideSel` parameter the instructions chooses the desired index from the 128-bit `ZPOINTS_FRONT` registers and determines the array offset. The address is written back into a special register and the data value is returned to the high-level function call. Temporary variables of the instruction are defined with the `wire` keyword. TIE-defined instructions are performed within a single clock cycle if no dependencies exist and all required data is available in registers.

Note that the TIE language does have some limitations here: while the TIE compiler uses auto multi-porting of register files to support multiple reads per instruction, each

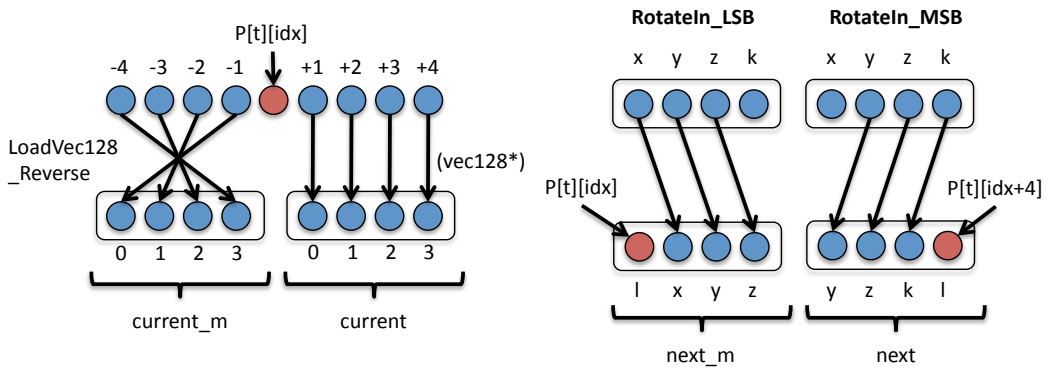
Listing 8.3: Green Wave Index Instruction TIE Extension Example

```
[...]
operation GetZFrontPoint {out FR value, in AR* addr, in StrideSel sel}
      {out VAddr, in MemDataIn32,
      in CURRP_SHORT, in ZPOINTS_FRONT_1,
      in ZPOINTS_FRONT_2}
{

 wire [2:0] MuxSel = StrideSelValues[sel];
 wire [31:0] index = TIEmux(MuxSel, ZPOINTS_FRONT_1[31:0],
   ZPOINTS_FRONT_1[63:32], ZPOINTS_FRONT_1[95:64],
   ZPOINTS_FRONT_1[127:96], ZPOINTS_FRONT_2[31:0],
   ZPOINTS_FRONT_2[63:32], ZPOINTS_FRONT_2[95:64],
   ZPOINTS_FRONT_2[127:96]);

 wire [31:0] offset  = (index + CURRP_SHORT) << 2;
 wire [31:0] address = addr + offset;
 assign VAddr = address;
 assign value = MemDataIn32;
}
```

instruction is limited to one write per register file. Thus allowing for a variety gather operations, but no scatter support.

Next, a register rotation scheme is implemented. It is applied in the fastest unit-stride and reuses already loaded data values to avoid redundant loads between consecutive stencil calculations. This can easily be done in hardware. The Itanium architecture [81] first introduced hardware register rotation to keep data values in registers and the use of renaming to avoid register copies. Different to Itanium and because of the limited number of only 16 floating-point registers for LX4, a second 128-bit register file was created used as additional space for more temporaries. The register file can be accessed in non-traditional ways, such as rotating a 32-bit float in or out of an individual register, or register loading with four 32-bit values starting from the most or least significant bit. This way it is possible to stream through the fastest unit stride without constantly performing redundant loads.

Code listing 8.2 allocates two 128-bit registers: `current` and `current_m`. The `vec128` datatype is defined with all other hardware definitions in TIE and can be



(a) Vector Load

(b) Vector Rotate

Figure 8.1: Vector Register load and rotate Instructions

used as a native datatype in high-level languages like C. A simple pointer cast from `float*` to `vec128*` is sufficient to store 128 bits beginning from the pointer address into the register. In case of an 8th order stencil the four values ahead of the current point and all values succeeding the current point at position `idx` are required for the finite-difference scheme and written into the registers accordingly. The register rotation can be performed by calling `RotateIn` instructions. The first parameter defines the 128-bit vector, the second parameter the 32-bit value, which is rotated into the vector to receive the necessary data for the subsequent stencil computation. Figure 8.1 visualizes the register rotation. A detail description of the corresponding TIE code was given in Section 6.3.1.

Additional to the novel register rotation instructions a `GetValue` instruction is introduced that returns a 32-bit values from a 128-bit vector depending on the position given to the instruction via the second parameter. This is necessary since no SIMD instructions are available up to date and all computations are done in 32-bit accuracy.

8.1.3.2 VLIW Extensions

Another way to reduce the instruction count is to leverage compiler's ability to bundle instructions into VLIW, allowing for co-issue of instructions. Because forward modeling kernel computations are floating-point intensive, the base LX4 processor is configured to support maximum instruction dispatch width of 64-bit and data load-/store width of 128bit, thus allowing multiple floating-point instructions to be con-

currently issued. The Xtensa compiler automatically bundles opcodes depending on the designers' specifications. From a hardware perspective, the processor generator tool creates parallel pipelines capable of executing the various instructions in each slot. Similar to IBM Cell, one pipeline is responsible for load/store, the other for arithmetic instructions. No parallel execution of either is possible due to a single-ported local store and only a single FPU. VLIW is a simple and effective optimization that requires no code changes while providing a potentially significant performance boost. To enable VLIW via TIE the core designer adds a new e.g. 64-bit wide instruction format (`format f64_2 64 {slot_a, slot_b}` and allocates instruction into one of the two slots. Green Wave uses `slot_a` for loadstore and `slot_b` for arithmetic instructions. The corresponding TIE entries are:

```
slot_opcodes slot_a {xt_loadstore, [...],
    RotateIn_LSBs, RotateIn_MSBs}
slot_opcodes slot_b { xt_shift, [...], ADD.S, MADD.S}
```

An assembly excerpt presented in 8.4 shows how the compiler is able to not only bundle standard instructions but also the newly defined custom instructions.

Listing 8.4: Green Wave Assembly Example using VLIW

```
[...]
256  6000283a  { getzfrontpoint f4, a2, 3; addi a11, a9, 5 }
256  60002842  { getyfrontpoint f3, a2, 3; getvalue f8, v1, 3 }
256  6000284a  wur.currp_short a11
256  6000284d  { getvalue f11, v1, 2; getvalue f0, v2, 2 }
256  60002855  { getyfrontpoint f12, a2, 2; add.s f0, f0, f11 }
[...]
```

8.1.4 Fused Multiply-Add Support

Green Wave is using a FPU that supports fused multiply-add $(A \times B + C)$ - `fma` - per clock cycle. Even if dependencies between instructions and a mismatch in the ratio of multiply and additions is preventing to leverage this capabilities Green Wave is exploiting `fma` using a fused multiply-add extension to the FPU. Such an extension is inexpensive in terms of transistor count and the fusing of instruction is completely

done by the compiler. Therefore, it provides a cheap way to increase performance when applicable without additional programming overhead.

8.1.5 Instruction Profiling

The Tensilica tools provide the programmer with powerful tools to analyze code performance. It presents the generated assembly code, the pipeline usage, an overview of fraction of time spent per function and it provides a summary of instruction usage. This summary helps to understand with what kind of instruction, or which part of the code the largest amount of time is spent and where optimization efforts should be focused on. Figure 8.2 presents the instruction profile of the TTI kernel implementation. As presented in Figure 8.2(a), a high percentage of cycles is spent in loading e.g. `lsi` and storing e.g. `ssi` single-precision values. Because the LX4 core owns only 16 floating-point registers, this leads to a great amount of register spills to the stack. Each of these values has to be loaded and written back to memory. The use of single-ported memories for Green Wave makes it impossible to the compiler to bundle load-store instructions via VLIW. Hence, only one read- or write-access can be executed at a time. By applying the Green Wave special register rotation instruction one can see that the total amount of instructions and the ratio of load-store and arithmetic instructions is much better distributed (see Figure 8.2(b)). This way VLIW bundling is exploited more often, which results in a significant performance increase (see Figure 8.6).

8.1.6 Main Memory

A number of fixed design choices are adopted as boundary conditions for the Green Wave co-design process. As one of them a commodity quad-channel, DDR3-1600 memory subsystem is used, which presents a low-risk design point from the standpoint of practical ASIC packaging and power dissipation, and reflects the memory performance of existing mainstream products, such as Intel Nehalem. The choice of conventional memory also simplifies the power model as DDR components have a well-characterized power profile.

Section 7.4 introduced the main memory capacity estimation. If the equation is applied to the evaluated seismic kernels this results in about 1.7 GB for ISO, 2.9 GB for VTI and about 3 GB for TTI kernels. The TTI implementation considers additional capacity for pre-computed sine and cosine functions of dip and azimuth earth properties. By

Format	Slot	Opcode	Total (%)	Occurrences
f64_2	0	lsi	14.69	2055080
x24	0	lsi	12.58	1759909
x24	0	ssi	9.91	1387469
x24	0	sub.s	7.49	1048576
f64_2	1	addmi	6.87	960942
f64_2	1	madd.s	6.3	882184
f64_2	1	add.s	5.15	721048
f64_2	1	mul.s	3.76	526464
f64_2	0	l32i	3.16	441826
f64_2	1	add	2.96	414272

(a) before

Format	Slot	Opcode	Total (%)	Occurrences
f64_2	0	lsi	18.09	1958025
f64_2	1	addmi	8.45	914725
f64_2	1	madd.s	7.36	796208
x24	0	sub.s	6.66	721408
f64_2	1	add.s	6.01	650744
f64_2	1	mul.s	6.0	649224
x24	0	lsi	5.22	565145
f64_2	0	l32i	4.32	467333
x24	0	ssi	2.99	323921
f64_2	0	ssi	2.96	320097

(b) after

Figure 8.2: The TTI instruction profile before (a) and after (b) using Green Wave register rotation instruction.

rounding the memory capacity requirements to a power-of-two value this results in 4 GB of DDR3-1600 memory per Green Wave node.
To gain maximum energy efficiency no DIMMs are used but single memory modules are hardwired onto the Green Wave hardware.

8.1.7 Network-on-Chip (NoC)

The Green Wave on-chip interprocessor communication fabric is derived from the Green Flash design. In particular, Green Wave adopts Green Flash's tiled processor architecture and interprocessor communication mechanisms 6.3.2.
The NoC is used primarily for communication between cores and memory controllers (for data load/store) and to facilitate energy efficient halo-exchanges between cores to

further reduce memory bandwidth requirements by eliminating redundant loads of halo regions for large-radius stencils.

The lightweight processor cores are interconnected via a scalable 4-way concentrated torus interconnection topology NoC, which is parameterized allowing networks of different performance and scale. This topology was adopted because recent cycle-accurate NoC studies [36, 37] have shown that this topology provides the most energy efficient solution for problems — such as the RTM forward modeling kernels — where the communication pattern is predominantly nearest neighbor.

Clustering of cores reduces the effective hop count for on-chip core-to-core communication and minimizes latencies [4]. All cores are additionally connected through a secondary on-chip network that is used for the exchange of control messages. The main on-chip network is used for actual data transfer like nearest neighbor halo exchange. Each core's local store is mapped into a shared memory region. If such a memory region is accessed in read or write manner the communication is done directly between the local stores on-chip. By allowing the clustered cores to write directly in each other local stores the pressure on main memory bandwidth is decreased. To achieve a maximum of computational efficiency the communication of halos between cores should be overlapped with computation of the inner points of a plane. The resulting bandwidth requirements for the NoC depend on core speed and kernel, which includes the order-in-space of the finite-difference approximation and if either an ISO, VTI or TTI implementation is used. For ISO and VTI only the faces of the plane have to be communicated - for TTI edges must be added.

Four cores share a single router to access the NoC. The 128-core ISO design results in a 16x8 NoC grid layout, 96 cores for VTI in a 12x8 and the 256 cores of the TTI Green Wave design in a 16x16 grid layout. Since the applied domain decomposition decomposes the node volume in only two dimensions, halos exchange takes place in x and y-direction only. Top and bottom halos in z-direction are exchanged between Green Wave nodes via distributed memory communication protocols (see Section 5.9). Each access point has to provide a maximum of about 350 MB/s for the isotropic and 200 MB/s for VTI kernel. The programming model for TTI kernel shares core halos through shared memory accesses. This is possible because 256 cores do not satisfy the main memory bandwidth. This way no NoC pressure besides control messages appears

and programming complexity is reduced significantly. The corresponding total NoC bandwidth is about 12 GB/s for ISO and 4.6 GB/s for VTI.

8.1.8 Power Consumption Modeling

Power modeling was done by combining several modeling tools. The core power was taken directly from the Tensilica tools [96]. As described in Section 6.3 the core power consumption is instantly presented when the core design is changed. It is assumed that the cores are operating at >80% efficiency, so the effect of clock gating idle portions of the chip are negated. Besides the quad-channel DDR3-1600 memory interface the silicon feature size of 45nm limits the Tensilica LX4 to maximum clock rate of 1 GHz. One of the main power consumers in high performance computing system is main memory. In order to get an accurate power consumption estimation for Green Wave, it is critical to model the leakage and dynamic power of the main memory.

- leakage power: The leakage power is called the static power consumption that is independent from the workload. Leakage power for main memory is caused by refreshing memory rows to keep data valid.
- dynamic power: The dynamic power is the amount of power consumed for accesses to the memory rows. Each access, like reads and writes trigger events that result in additional power consumption. Hence, dynamic power depends on the memory access pattern.

DRAM energy is modeled using the current profiles from the Micron datasheets [57] for a 1 Gb DDR3-1600 memory module and the cycle-accurate DRAMsim2 memory architecture simulator [105]. DRAMsim2 is able to model the leakage and dynamic power for a specific type of memory. The type is defined by a memory description file that is fed to the application. Additional to the description file of the simulated DRAM module, DRAMsim2 gets an application trace that represents the memory access pattern, which was collected through Tensilica's XPG simulation of the Green Wave core executing the wave propagation kernels. The memory is build upon 1 Gibit chips of organization 128M x8. Internally a chip uses eight banks (DDR3 standard) of 8192 rows and 2048 columns. One column entry holds 8 bits. In order to keep the burst length to the JEDEC standard of 64 bits 8 chips are combined to a memory rank of size 1 GB. This

memory configuration uses one rank per memory controller channel, which supports 'unganged' memory accesses of 4x 64bits or 256 bits (quad-channel). Consequently for ranks are required in total to leverage all four memory controller channels.
The trace data was collected via a XTSC implementation of a simplified socket design using local stores. This way only required main memory accesses appear in the trace. Traces were collected for up to 32 cores. Unfortunately only a single channel is supported by DRAMsim2 and it is assumed that power consumption and bandwidth scales with the number of memory channels.
The results show an interesting behavior in which memory pressure of 32 cores is below the pressure of a single core. To receive conservative power estimates the trace of a single core is taken and one access per cycle assumed. This way peak power consumption for a given memory organization is achieved.
The final DRAMsim2 output looks like 8.5. It lists the memory access patterns to the specific memory banks, the total memory bandwidth used, the access latencies and power consumption. One can see that the rank reaches full bandwidth with 11.25 GB/s and a peak power consumption of 3.6 Watts. By scaling these numbers up to four ranks we receive a bandwidth of 45 GB/s and a power consumption of 14.4 Watts for the memory chips. Power consumption numbers presented by DRAMsim2 do not include the memory controller. Further eight additional Watts are assumed as power consumption for the memory controller, which leads to a total of 22.4 Watts.

Listing 8.5: Example DRAMsim2 Output

```
=======================================================
============== Printing Statistics [id:0]==============
Total Return Transactions : 236 (15104 bytes) aggregate average bandwidth
     11.253GB/s
-Rank   0 :
-Reads  : 236 (15104 bytes)
-Writes : 0 (0 bytes)
-Bandwidth / Latency  (Bank 0): 1.383 GB/s              252.026 ns
-Bandwidth / Latency  (Bank 1): 1.001 GB/s              260.298 ns
-Bandwidth / Latency  (Bank 2): 1.144 GB/s              178.594 ns
-Bandwidth / Latency  (Bank 3): 1.335 GB/s              172.188 ns
-Bandwidth / Latency  (Bank 4): 1.621 GB/s              232.904 ns
-Bandwidth / Latency  (Bank 5): 1.669 GB/s              259.286 ns
-Bandwidth / Latency  (Bank 6): 1.621 GB/s              265.404 ns
-Bandwidth / Latency  (Bank 7): 1.478 GB/s              269.153 ns
```

```
== Power Data for Rank        0
Average Power (watts)     : 3.636
-Background (watts)     : 0.803
-Act/Pre    (watts)     : 0.724
-Burst      (watts)     : 2.108
-Refresh    (watts)     : 0.000
```

Further reading about the power modeling tools can be found in Section 6.3.3.
A breakdown of Green Wave power components is shown in Figure 8.3(right). Given the low-power nature of the Tensilica cores, it can be observed that most power is consumed by on- and off-chip memory. The large amount of memory is necessary to capture all temporal recurrences. SRAM memories like local stores and caches consume 18% and DRAM power constitutes roughly 51% of the node's total power for VTI design that has the highest memory capacity and bandwidth requirements. The power distribution mirrors the kernel requirements. Since TTI is heavily instruction bound the amount of power consumed by the cores increases from only 22% of the VTI design to 36% for TTI. The Green Wave nodes' total power consumption is 55 Watts, 51 Watts and 84 Watts for the ISO, VTI and TTI designs. An on-board Infiniband (IB) 4x QDR interface required for off-node communication is not included in this power estimation diagram.

8.1.9 Chip Size Estimation

The chip size is a critical factor that has to be considered and kept within feasible limits if a realistic chip design is the goal. The Green Wave chip size estimation is done by combining the output of several modeling tools. The size of one Xtensa core including caches and TIE instructions is directly given by Tensilica's XPG. The area consumed for local stores is taken from the CACTI6.5 memory-modeling tool. Since a standard DDR3-1600 controller is used the area of such is taken from current on-market memory controller IP's. Last the NoC's area consumption is estimated by [4].
Given each XTensa core is only 1 mm^2 and each local store about 0.19-0.72 mm^2, the area consumed by memory controllers is quite substantial. As such, there is a clear economy of scale by incorporating many cores to share the memory controller resources. The result is a 246 mm^2 for ISO, 224 mm^2 for VTI and 472 mm^2 for the TTI 45nm chip, making Green Wave larger than a Nehalem processor but quite smaller than the

C2050 GPU that weighs in at 576 mm^2. Figure 8.3(left) presents the area breakdown of Green Wave components.

8.1.10 Green Wave Architecture Summary

The Green Wave chip design is based on Tensilica's LX4 and uses VLIW, in-order cores, including novel instruction extension for stencil operations. The number of cores per socket is chosen accordingly to attain a balanced system between memory bandwidth and computational performance. This results in the most energy efficient architecture design for running a specific optimized kernel implementation. This lead to 128, 96 and 256 cores for ISO, VTI and TTI designs, respectively. A minimum local store size was chosen to provide lowest power and area consumption but appropriately sized to capture all temporal recurrences. This trade-off resulted in local store sizes of 128 KB for ISO, 256 KB for VTI and 128 KB for TTI Green Wave cores. In order to attain a Harvard architecture with clear separation of instructions and data an 8 KB L1 instruction cache is used. An additional 8 KB L1 data cache holds the stack and enables easier porting of standard x86 code. Each core has a general and floating-point register file of 16 entries each. Additionally a vector register file was created that provides 16 entries of 128-bit width.

All cores are leveraging a quad-channel DDR3-1600 memory interface that provides up to 51.2 GB/s peak pin bandwidth. Single precision peak performance utilizing FMA's reaches from 192 Gflops/s for the VTI up to 512 Gflops/s for the TTI design consuming 51 Watts to 84 Watts including off-chip DRAM. With a process technology of 45nm, the die area consumes about 246mm^2 for ISO, 224mm^2 for VTI and 472mm^2 for TTI. All architectural details are summarized in Table 8.1.

Figure 8.3: Green Wave Die Area and Power Breakdown

Core Architecture	**Tensilica LX4**	**Tensilica LX4**	**Tensilica LX4**
Type	VLIW in-order customized	VLIW in-order customized	VLIW in-order customized
Clock (GHz)	1.00	1.00	1.00
SP GFlop/s	2.00	2.00	2.00
L1 Instr. Cache	8 KB	8 KB	8 KB
L1 Data Cache	8 KB	8 KB	8 KB
L2 Local Store	128 KB	256 KB	128 KB
SMP Architecture	**GreenWave ISO**	**GreenWave VTI**	**GreenWave TTI**
Threads per core	1	1	1
Cores per socket	128	96	256
Sockets per SMP	1	1	1
memory parallelism	DMA	DMA	DMA
Aggregate on-chip RAM	≈16 MB	≈24 MB	≈32 MB
Aggregate DRAM GB/s	51.2	51.2	51.2
Aggregate SP GFlop/s	256	192	512
Power under RTM load	55W	53W	81W
Total Die Area	246mm^2	224mm^2	472mm^2
Process Technology	45nm	45nm	45nm

Table 8.1: Summary of Green Wave Designs

8.2 A Single-Node Study

In this section first estimates of achievable performance for a single Green Wave socket should clarify if the novel architecture can achieve enough throughput performance to compete the evaluated COTS architectures. From analyzing the Roofline model it is possible to identify the amount optimization effort for Green Wave specific kernels. Later the *Tensilica Instruction Set Simulator* (ISS) is used to attain a cycle-accurate kernel benchmark result running Green Wave optimized implementations including all previously introduced novel hardware features.

8.2.1 The Green Wave Roofline Model

As for the evaluated COTS architecture, the Roofline model is used to visualize the bandwidth- and in-core ceilings for a Green Wave socket (see Figure 8.4). The theoretical peak performance is achieved by issuing one multiply-add instruction per cycle for all cores. Assuming no fused multiply-add instruction the in-core ceiling drops to half of its peak performance, which draws the second horizontal line into the diagram.

To attain the bandwidth ceiling, one has to consider that Green Wave is using a DDR3-1600 quad-channel memory controller that delivers 51.2 GB/s as peak pin memory bandwidth. As the bandwidth-ceiling is a product of the memory bandwidth and the arithmetic intensity this draws the first diagonal line into the model. Running a slightly modified version of the STREAM benchmark [110] shows that for modern DDR3-1600 memory controller like used on Intel Sandy Bridge sockets, about 88% of this bandwidth can be sustained. This lowers the bandwidth-ceiling to about 45 GB/s and draws the second diagonal line. An additional drop in memory bandwidth can occur if data is not cache line aligned. Based on the experience with IBM Cell [109], which uses a similar local store architecture, the same bandwidth penalty is assumed for Green Wave and is represented by the third diagonal bandwidth ceiling.
Comparing Green Wave to the evaluated COTS architectures, one recognizes that less code optimization is required to reach high throughput performance. No SIMDization or unknown cache policies prevent Green Wave kernels attain good performance. Code optimization for Green Wave is reduced to hide communication and memory latencies through double-buffering and asynchronous communication schemes.

8.2.2 Instruction Performance Limitations

The Green Wave core architecture described in Section 8.1 uses a clock rate of 1 GHz, one floating-point unit and a five stages deep pipeline. To estimate the instruction performance limitations one has to account for the amount of floating-point operations to be executed per grid point (N_{flops}). Isotropic and VTI kernel apply a finite-difference scheme along the axis only, which makes the amount of flops-per-point independent of the plane size. Different to ISO and VTI it is necessary to apply common subexpression elimination for TTI kernel to achieve good performance. As explained in Section 5.4.2 CSE reuses the already computed first-order derivates along the fastest unit-stride direction on x and y-axis. Since these derivates have to be computed for halo points as well, the number of floating-point operations per point increases with decreasing plane size, assuming the halo width to be constant. For each point in either the first row ($x = 1, y, z$) or column ($x, y = 1, z$) all mixed derivates for all points have to be calculated. Points with the coordinates $x = 2..n, y, z$ or $x, y = 2..n, z$ are able to reuse precomputed subexpressions. It follows that, the smaller n gets, the higher the amount

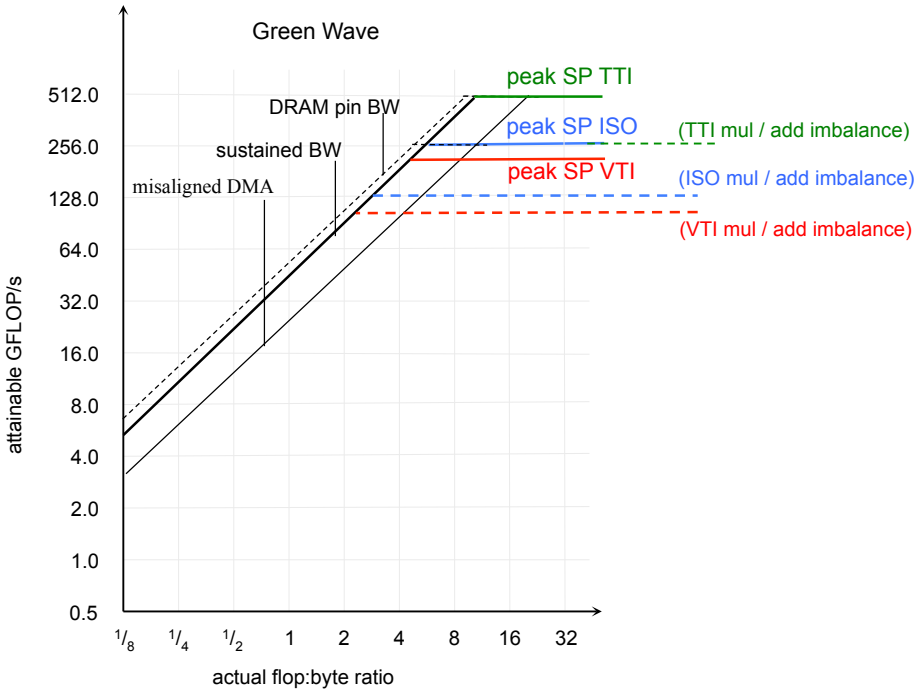


Figure 8.4: The Green Wave Roofline Model

of relative flops/point. Switching from a plane size of 512^2 as applied for COTS architecture, to a plane size of 32x16, as applied for TTI kernel implementations on Green Wave, means a modest 14% increase in flops per point. Despite the increase in flops, CSE still has huge performance benefits.

Whether the compiler is able to use the FMA capabilities of the FPU or not cannot be fully predicted and needs to be analyzed by the assembly output. To give a conservative, but more realistic, performance estimation the achieved instruction throughput is assumed to be one flop per cycle. The time spent in applying the finite-difference scheme is derived by dividing the number of floating-point instructions by the achieved number of flops-per-cycle, multiplied with the clock frequency:

$$\text{Time} = \text{Number of Flops} \,/(\text{Clockrate} \times \text{Flops per Cycle}) \tag{8.1}$$

The estimated performance, if not other limitations apply is 3,879 MPoints/s in the isotropic case, 1,811 MPoints/s for VTI and 768 MPoints/s for TTI respectively,

Kernel	ISO	VTI	TTI	TTI PRE CSE
Flops/Point	33	53	800	333
MPoints/s/Socket	3,879	1,811	320	768

Table 8.2: Green Wave Performance Estimation (1flop/cycle)

Kernel	ISO	VTI	TTI	TTI PRE CSE
Bytes/Point	16	36	44/47	52/55
MPoints/s	2,816	1,252	1,024/957	866/818

Table 8.3: Estimation of Green Wave Performance

assuming perfect scaling. Table 8.2 summarizes the performance limits for all kernel implementations for an 1 flop/cycle sustained performance.

8.2.3 Memory Bandwidth Performance Limitations

This section estimates the maximum performance achieved under the circumstances that no other limitation apply but memory bandwidth.

To estimate the achievable performance it is mandatory to know how many bytes are required to be accessed in main memory per point computed. For all cores the local memories are appropriately sized to capture all temporal recurrences. Under the assumption that halo regions are shared between cores on-chip, rather than through shared memory, the ISO kernel reads data values for timestep t, $t-1$ and the velocity and writes the value for timestep $t+1$. Hence, ISO's compulsory number bytes accessed is 16. Corresponding are 36Bytes accessed for VTI, 44 Bytes for TTI without pre-computed trigonometric functions and 52 Bytes with pre-computation. If halo exchange for TTI kernels is performed through shared memory loads this numbers increases to 47 Bytes for TTI and 55 Bytes with for TTI with applied pre-computation. Table 8.3 presents the memory bandwidth limited peak performance. The maximum achievable performance for Green Wave is calculated according to Equation 8.2.

$$\text{Performance} = \text{Sustained Bandwidth}/(\text{Compulsory Bytes per Point}) \tag{8.2}$$

8.2.4 Energy Efficiency Comparison based on Performance Estimations

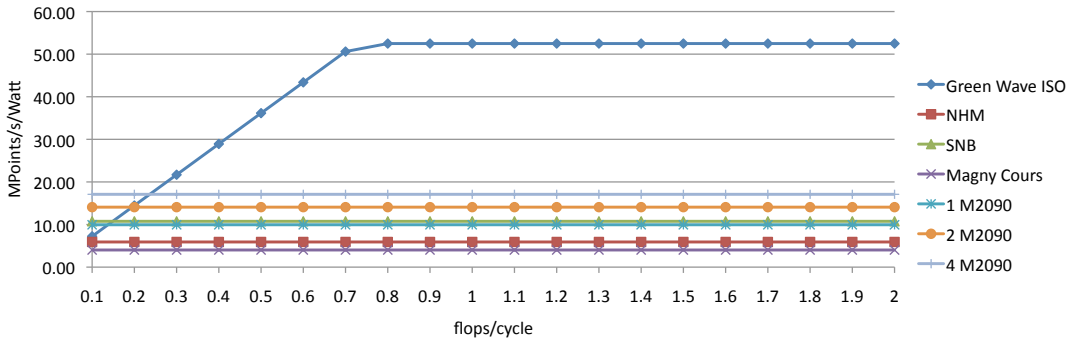
Before any effort is put into implementing optimized kernels for Green Wave an energy efficiency estimation should tell if a significant benefit can be achieved. Due to the uncertainty how many flops/cycle the Green Wave design is able to maintain Figure 8.5 explores how a changing computational efficiency influences the overall energy efficiency of the design. The diagram draws performance in flops-per-cycle on the x-axis and MPoints per second per Watt on the y-axis. The straight lines represent the benchmarked energy efficiency of the optimized kernels on the evaluated COTS architectures. These lines do not depend on the flop-cycle count but are added as orientation at what point Green Wave would achieve comparable or better energy efficiency.

The impressive results in Figure 8.5 show that with only 0.45 flops/cycle the Green Wave TTI node would achieve the energy efficiency of a node configuration running four M2090 GPUs. For the ISO and VTI Green Wave node designs only about 0.25 flops/cycle are necessary. Further the diagrams clearly show the bandwidth ceiling at which a further increase in flops/cycle would not have any effect on neither performance nor energy efficiency. ISO and VTI design reach that ceiling at 0.7 flops/cycle - contrary TTI reaches is bandwidth bound maximum energy efficiency at 1.1 flops/cycle.
The results from this study should provide a point of reference to evaluate achieved performance of Green Wave kernel implementations.

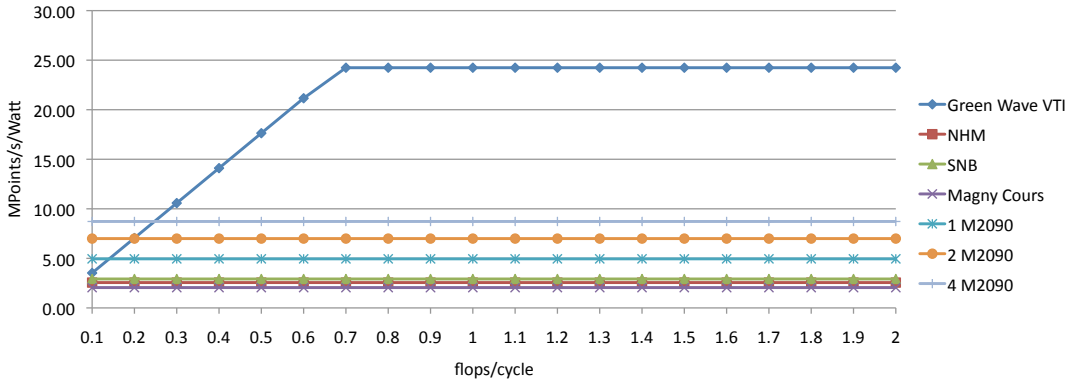
8.2.5 Summary and Conclusion

Concluding from the estimated performance numbers Green Wave implementations will likely be slightly memory bandwidth bound for ISO kernel and insignificantly oversubscribed in computational performance for VTI kernel implementations. The TTI design will likely be instruction bound. The expected performance is 2816 MPoints/s, 1252 MPoints/s, and about 768 MPoints/s for ISO, VTI and TTI for 1 flop/cycle, respectively. Based on these assumptions one Green Wave socket should be able to compete easily in terms of MPoints/s with the evaluated Nehalem and Mangy Cours x86 architectures. Only the GPU configurations and Sandy Bridge nodes are outperforming Green Wave in a single-socket comparison. Considering the amount of cores per GPU

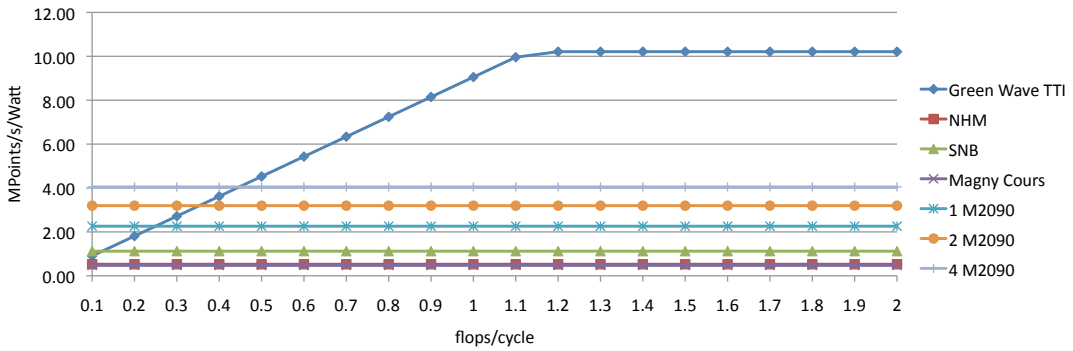
and a sustained memory bandwidth of Sandy Bridge and the M2090s this isn't a big surprise.



(a) ISO



(b) VTI



(c) TTI

Figure 8.5: Green Wave energy efficiency in relation to achieved flops per cycle

When comparing Green Wave's energy efficiency, the balanced architecture design shows its strength. The designs show great energy efficiency potential for all kernels. These results serve as motivation to implement optimized kernel implementations for Green Wave and run the cycle-accurate ISS to see what performance can be achieved.

8.2.6 A Single-Node Benchmark

The former sections derived a Green Wave design based on the requirements of common forward modeling RTM kernels. The predicted performance 8.2.3 shows significant energy efficiency improvement compared to COTS architectures. This chapter uses the Tensilica's Instruction Set Simulator (ISS) to receive accurate benchmark results for the introduced core design running the desired kernels. The achieved performance within the simulator is then compared to the estimated performance, which provides a guideline of how well the design is performing for the given kernel and if the hardware design should eventually be adjusted.

With Tensilica's ISS it possible to run only one simulated core at a time. The Green Flash performance modeling approach (see Section 6.3.5) had shown that communication (DRAM–LS) can straightforwardly be decoupled from compute (LS–FPU) on double buffered stencil codes. Therefore, for this benchmark analysis we assume perfect scaling using a halo exchange programming scheme that overlaps halo exchange with computation. Given the single core benchmarks results it is then possible to refine the assumption that computation and communication can be completely overlapped.

The following benchmark evaluates the benefits of hardware and software specific optimizations. Three types of implementations were developed for each kernel type to distinguish their effects on performance.

1. **The Reference Implementation** has no optimization applied and uses an 8th order finite-difference scheme.

2. **The Software Optimized Implementation** uses software optimizations techniques, which are not unique to Green Wave. As the inner most loop is very short it is fully unrolled and the branch moved outside the grid sweep for ISO and VTI.

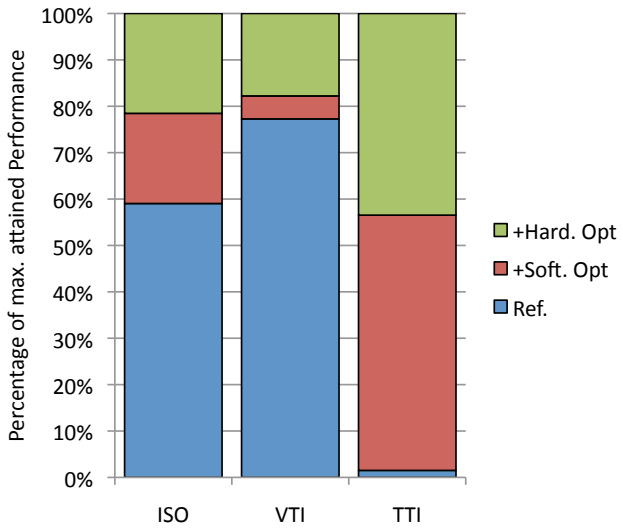


Figure 8.6: Effects of Green Wave Software and Hardware Optimization

For TTI common subexpression elimination and pre-computation of sine and cosine functions is applied. Additionally all kernels use the register rotation scheme that targets at keeping data values within the register file for later reuse.

3. **The Green Wave Specific Implementation** uses novel Green Wave instructions to leverage the special hardware units. Hardware optimizations include the improved index calculation, rotation instructions and VLIW implementation.

Even though the Green Wave custom instructions were introduced by analyzing the requirements of the ISO kernel, VTI and TTI implementation benefit heavily from the introduced optimizations. Figure 8.6 shows the effects of software and hardware optimizations on performance for the different kernels and Green Wave designs.

Not surprisingly, the unoptimized reference kernel implementation shows best performance for VTI. VTI depends the most on memory bandwidth and has the lowest arithmetic intensity of all kernels. Hence, software optimizations show only about 6% performance increase for VTI but already 33% for ISO. The approximation of sine and cosine function is highly inefficient and explains the slow performance of the reference TTI implementation. Pre-computation of such functions and the applied CSE scheme results in a huge increase in performance of more than three magnitudes. Extending the

software optimized kernel implementations by Green Wave specific custom instruction results in a 28% performance increase for ISO and 22% for VTI. ISO and VTI kernel using the rotation scheme involving working on 128-bit register file for the wavefield P and Q. For TTI the rotation scheme cannot only be used for the two wavefields but also for all CSE buffers. This drastically reduces pressure on the small floating-point register file of the Xtensa LX4 core. All hardware specific optimizations greatly increase TTI performance another 77%.
The fully optimized Green Wave kernels achieve 0.78flops/cycle for ISO, 0.69flops/cycle for VTI and 0.72flops/cycle for TTI. This results in a throughput performance of 2860MPoints/s, 1200MPoints/s and 560MPoints/s, respectively.

8.2.7 Benchmark Analysis

The promising results are now analyzed by Figure 8.7, which draws new in-core ceiling for attained performance into the Green Wave Roofline Model. As core numbers are adjusted to the attained performance it shows that for ISO and VTI designs the architectures are almost perfectly balanced in throughput performance and memory bandwidth. Even VTI is slightly under subscripted in compute performance, concluding from the previous flop-cycle energy efficiency analysis in Section 8.2.4 a further increase in flops-per-cycle would not lead to greatly increased performance or energy efficiency due to memory bandwidth limitations.
TTI achieves 0.73 flops/cycle. As previously presented in Figure 8.5(c) about 1.1flops-per-cycle would result in a perfectly balanced architecture. Concluding the optimal number of cores would be 380. As 256 cores already consume an area of $472mm^2$ such a high number of cores is not reasonable with a silicon feature size of 45nm. Instead of increasing core numbers it is more advisable to study the possible benefit of a 2-way SIMD unit. This way it could be possible to achieve higher throughput but keeping the same amount of cores per socket. Such a SIMD floating-point unit is currently utilized in all x86 processors. Vector processing on COTS hardware is available since the mid 1990's, which makes the technique well understood and available as IP. Hence, it is reasonable to take further studies to adapt SIMD technology. Enabling the core to execute 2 FMA's at a time could at best double performance to 1.4 flops/cycle. As one FPU does not achieve its peak of 1flop/cycle it is likely to expect less than 2x improvement by utilizing a 2-way SIMD FPU. This way the Green Wave TTI design

is likely to end up perfectly balanced if SIMD units are applied. Implementing such a FPU design could be done with the Tensilica toolkit.

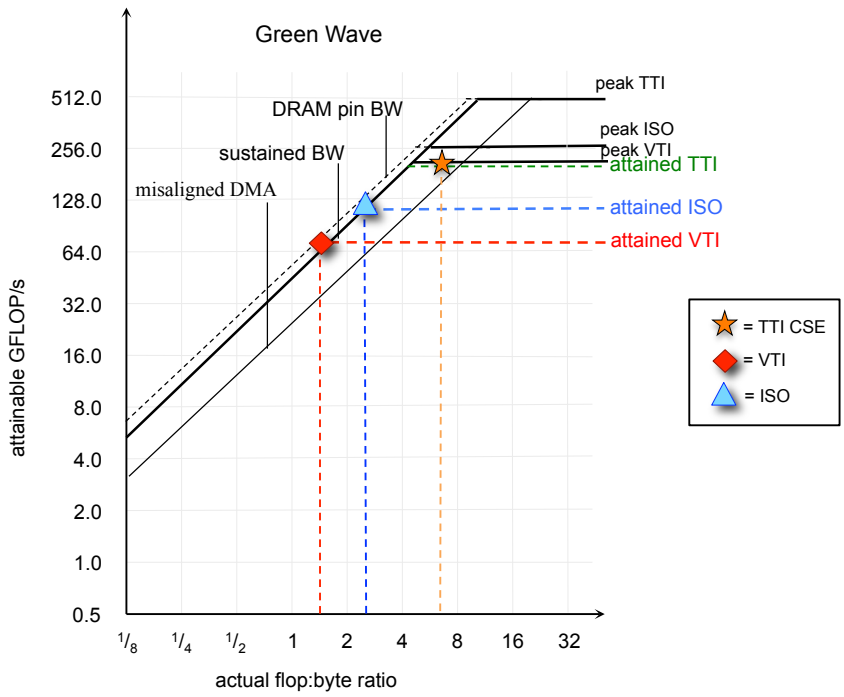
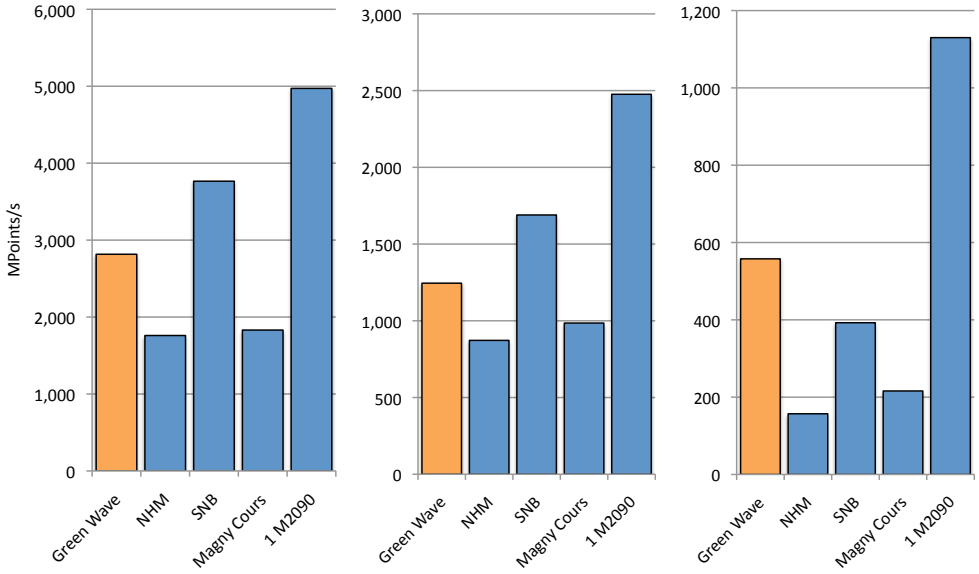


Figure 8.7: The Green Wave Roofline Model including kernels

In the following the achieved Green Wave performance is compared to the evaluated COTS architectures in Section 3. Figure 8.8 presents the throughput in MPoints/s achieved for Green Wave compared to the previously benchmarked COTS architectures. This benchmark compares SMPs to each other. For GPUs only one M2090 is considered since multiple GPUs already require distributed memory communication via PCIe.
It can be seen that Green Wave shows a superior performance of 2860 MPoints/s, 1200 MPoints/s and 560 MPoints/s for ISO, VTI and TTI, compared to Nehalem and Magny Cours nodes. Sandy Bridge provides better performance for ISO and VTI. As expected the GPU still performs better in terms of raw throughput. It shows a performance advantage of 1.7x for ISO and 2x for VTI and TTI.

As Green Wave was developed targeting optimal energy efficiency and not single-node floating-point throughput performance the above results are not surprising. Fig-



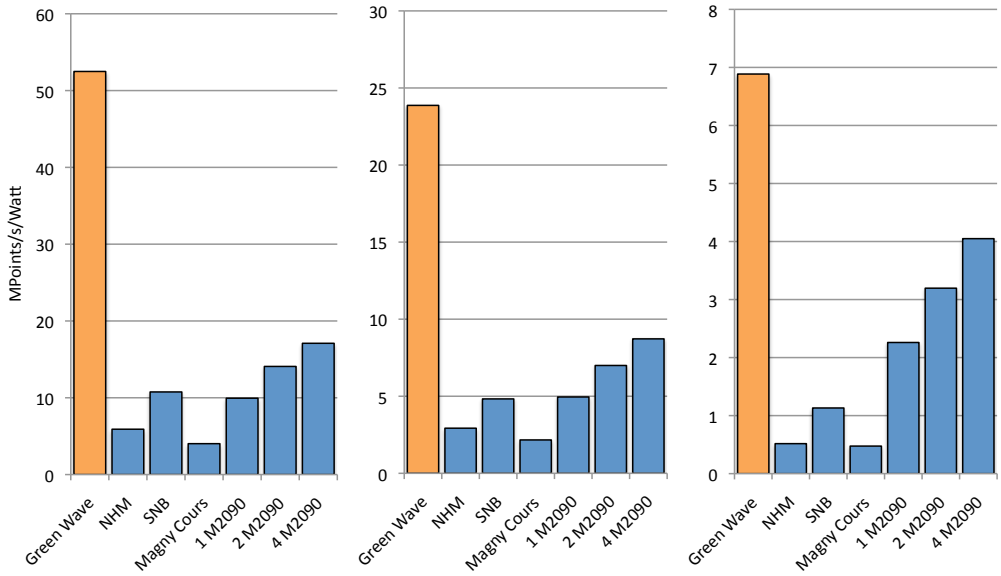
(a) ISO (b) VTI (c) TTI

Figure 8.8: Green Wave Throughput in MPoints/s

ure 8.9 now presents the achieved energy efficiency. Green Wave offers great energy efficiency advantage of 5x to 12x for ISO compared to x86-based architectures and 2.9x-5x to GPUs. For VTI the advantages stay at about the same level with 4.5x-11x for x86 and 2.7x-4.5x to GPUs. Contrary, the TTI kernel shows an improved performance to x86-based architectures of 6x to 14x but is less ahead of GPUs with now 1.6x improvement to a four-GPU node setup using NVIDIA's M2090.

8.2.8 Single-Node Study: Conclusion

This chapter took the derived formulas from Chapter 7 and the experiences of former studies (see Chapter 6) to derive different Green Wave node designs tailored for ISO, VTI and TTI kernels. The introduced Green Wave hardware design for ISO uses 128 cores per socket and a 128 KB local store. For VTI kernel this had to be extended to 256 KB to capture all temporal recurrences. The lower flop-to-byte ratio of the VTI kernel caused an oversubscription in floating-point performance, which resulted in a reduced number of only 96 cores per node to achieve a balanced system. The TTI design leverages 256 cores and 128 KB local store to serve the need of high floating-point



(a) ISO (b) VTI (c) TTI

Figure 8.9: Green Wave Energy Efficiency in MPoints/s/Watt

throughput. This chapter analyzed the theoretical performance limitations of the Green Wave socket for all three kernel implementations and compared them to benchmarks taken from cycle-accurate emulation running the forward modeling kernels. Applying the novel instructions introduced through Green Wave's special hardware units showed great performance benefits for all kernels. To address the still instruction bound TTI kernel a possible implementation of a 2-way SIMD floating-point unit was discussed and would likely result in a balanced design. For the already memory bound ISO and VTI kernel implementations the additional increase in throughput would neither increase performance nor energy efficiency.

The overall Green Wave single-node performance easily competes with Nehalem and Magny Cours nodes but is outperformed by GPU and Sandy Bridge setups due to their amount of cores and high sustained memory bandwidth. When comparing Green Wave to Sandy Bridge the reader needs to remember the smaller silicon feature size of 32nm compared to Green Wave's 45nm design. When it comes to energy efficiency Green Wave shows great advantages compared to all evaluated on-market architectures even with only one floating-point unit.

8. THE GREEN WAVE ARCHITECTURE AND SINGLE-NODE STUDY

For production-sized surveys a single node does not provide enough performance or memory capacity to serve the needs of the seismic industry. The good single-node benchmark results serve as motivation to take the Green Wave study further. The following chapter introduces a multi-node benchmark and analyzes the effects of distributed memory communication on the throughput and energy efficiency of the Green Wave designs.

Chapter 9

A Green Wave Multi-Node Study

In Section 8.2 the performance of a single Green Wave node was compared to the evaluated COTS architectures. The results showed a significant improvement in energy efficiency. Unfortunately, production-sized surveys are too large to make a single shot fit into a single node's memory. Especially to meet the time constraints by leveraging parallelism, multiple nodes have to be utilized. The most basic multi-node programming model computes the results for a timestep, exchanges data, waits for transfers completion and then begins the calculation for the next timestep. The inter-node communication study in Chapter 5 showed that this would add significant overhead and results in a decreased performance and worse energy efficiency, respectively. An optimal communication scheme would keep the overhead as small as possible and try to avoid stalls of the processing units. Hence, inter-node communication topology, bandwidth and programming model has to fit the application requirements. This chapter extends the single-node study to a multi-node comparison between all architectures. It analyzes the effects of inter-node communication overheads on throughput and energy efficiency. At the end of this chapter this multi-node study is further extended with external I/O devices. Two commodity options for mass-storage are discussed and the bandwidth and capacity requirements for all architectures derived. This completes a simplified cluster architectures based on the discussed COTS systems and all three Green Wave node designs. Finally, including the additional overhead caused by I/O the power consumption of different cluster setups is analyzed.

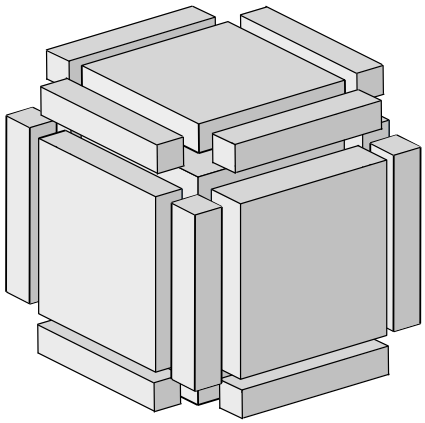


Figure 9.1: Volume including Halos

9.1 Green Wave Node Volume Analysis

As for the on-chip subdomains, certain neighboring points are required to apply stencil computation on the edges and faces of the node volume. This leads to the requirement of extending the node volume by a halo region. The width of the halo depends on the order of the finite-difference scheme. Its size equals the radius of the stencil width - 4 points for an 8th order stencil. The shape of the stencil defines which halos are needed. For the ISO and VTI kernel have star-shaped stencil and only the faces of the volume are extended halos. Wave propagation kernels for TTI media use mixed derivates and create a stencil consisting of three intersected planes in all space dimensions. These planes contribute diagonal points, which require to exchange subdomain edges additionally to the faces. Figure 9.1 shows the volume including all halos that are required for TTI.

In total 19 different distinct domains can be differentiated:

Size	#	Name
$(N-2r)^3$	1	Volume
$(N-2r)^2 r$	6	Face
$(N-2r)r^2$	12	Edge

The four corner halos are not required since the stencil operator work on the two-dimensional plane domain only.

Contrary to the x86 architecture nodes Green Wave nodes cannot easily increase the node volume without decreasing performance. Hence the node volume size was adjusted

so that a plane size would not exceed local store capacities. This leads to relatively small volume, which makes Green Wave vulnerable to communication overhead, and entails the necessity to look into inter-node communication models more carefully.

9.2 Green Wave Inter-Node Communication

Green Wave uses the same two-sided, message-passing protocol MPI, as it was used for COTS architectures in the previous multi-node study in Section 5.9. As previously mentioned the typical communication model consists of three steps:

1. Packing halo data from the working array into MPI send buffers.
2. Data exchange from send buffers to receive buffers
3. Unpacking data from MPI receive buffers to the working array

This basic programming model has the downside of adding additional overhead to the computation time and computational units are stalled until halo data is received from the neighbors. The approach to hide communication on the other node architectures was to compute inner points while halo regions are transferred. This requires $t_{inner} > t_{comm}$ with t_{inner} as the time to compute the inner, independent points and t_{comm} the time to transfers the halos. The Green Wave node subdomain gets further 2D domain decomposed in 128 smaller 2D subdomains. Accordingly, the significantly smaller domains have a worse surface-to-volume ratio and computation time of inner points cannot overlap communication time anymore ($t_{inner} < t_{comm}$). This would cause the Green Wave node to stall a large number of cores and result in decreased throughput and energy efficiency. Hence, Green Wave needs a different communication model.

Each Green Wave node has a dedicated Infiniband interface that is able to work concurrent to the cores. Together with the inter-node programming model presented in Figure 9.2 it enables Green Wave to hide halo data exchange. To do so the computation of the node subdomain is decomposed into two steps. The first step computes planes z_0 to $z_{n/2}$ followed by planes $z_{n/2}$ to z_n in the second step (see Figure 9.2(left)). Each step owns dedicated MPI buffers (`MPI_buffer_1`, `MPI_buffer_2`), which are allocated separately from each other. The two-step approach is necessary since scatter-gather requirements of packing and unpacking MPI buffers do not let Green Wave hide

latencies for those operations. For the first timestep (t_0) it is assumed that data including corresponding halos resides in the node's main memory and computation of step one can start without additional communication. Step one interrupts computation after finishing plane $z_{n/2}$ and starts packing its MPI buffers with halo data for the neighboring nodes. The exchange of data from step one is then overlapped with computation of the volume in step two. At the beginning of all following timesteps ($1..nt$), `MPI_buffer_1` is unpacked and `MPI_buffer_2` packed. Concurrent to computing of step one `MPI_buffer_2` is transferred - and vice versa for $z > z_{n/2}$. To hide communication it is necessary that the computation time of step one is higher than the communication time of `MPI_buffer_2` ($t_{step1,2} > t_{comm}$, with $t_{step1,2}$ as computation time of step one or step two).

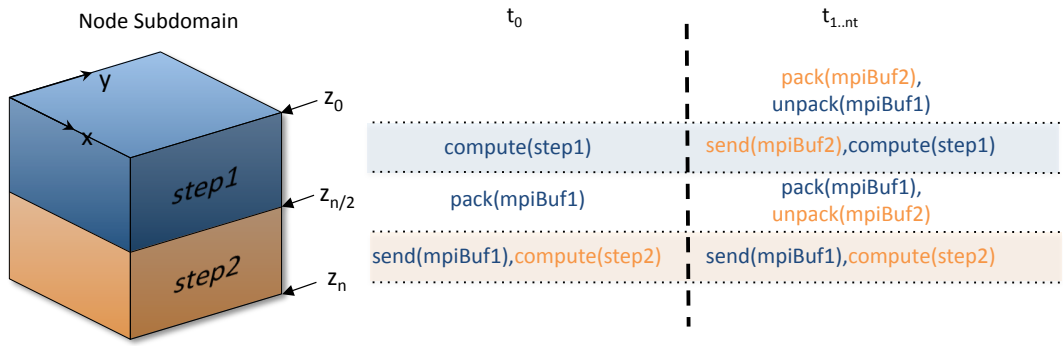


Figure 9.2: Green Wave Inter-Node Communication Scheme

This inter-node communication model requires a bandwidth of 406 MB/s for ISO, 266 MB/s for VTI and about 74 MB/s for TTI kernel, which is easily achieved by common Infiniband or even 10Gbit Ethernet interconnects.

Green Wave's vulnerability to communication overhead due to small node volume sizes makes it necessary to further optimize not only the inter-node data exchange. As the data transfer can be overlapped the main cause for stalled cores is the packing and unpacking of MPI buffers. The inter-node communication model presented in Section 5.9.2 used a basic approach in which communication took place between buffers and all data had to be copied to/from buffers to make it available to MPI.

This copy overhead can be further reduced by reading top and bottom buffers directly from their location within the working array and writing them remotely into

the corresponding position within the working array of the neighboring node, and skips copying top and bottom halos to/from separated MPI buffers. This is possible because top and bottom halo regions reside on consecutive addresses in memory. To ensure that no load imbalances cause overwriting of data that is still required by computation, the `I_Recv` command, which signals MPI the readiness to receive data, is started when computation of plane $r - 1$ is done. All planes from $r + 1$ on do not require any more points from the top halo. The receive command for the bottom halo is called accordingly when no access to bottom halo points is required anymore.

The Green Wave communication overhead was derived by benchmarks applied on the Sandy Bridge node. Green Wave uses the same COTS memory controller but DDR3-1600 instead of DDR3-1333 memory chips. Unfortunately no node configuration with DDR3-1600 was accessible for experiments. To attain an estimate times for buffer copies were measured on a system equipped with DDR3-1066 and again with DDR3-1333. Benchmarks showed about 12% improvement. The same improvement is expected for using DDR3-1600 instead of DDR3-1333 on Green Wave.
Despite all optimization efforts, in comparison to the overhead added to the evaluated x86 architectures, the fraction of communication overhead on the total runtime is significantly higher with 19% for ISO, 10% for VTI and 6.5% - a testament to the higher performance and smaller node volume size.

Table 9.1 summarizes the node volume sizes, the required memory capacity and communication overhead for all three Green Wave designs.

	Req. Memory	Node Volume Size	Comm. Overhead	Req. Bandwidth
ISO	1.7	512x512x512	18.9%	406 MB/s
VTI	2.9	512x384x512	9.7%	266 MB/s
TTI	3.1	512x256x512	6.5%	74 MB/s

Table 9.1: Green Wave Node Volume and Communication Overhead

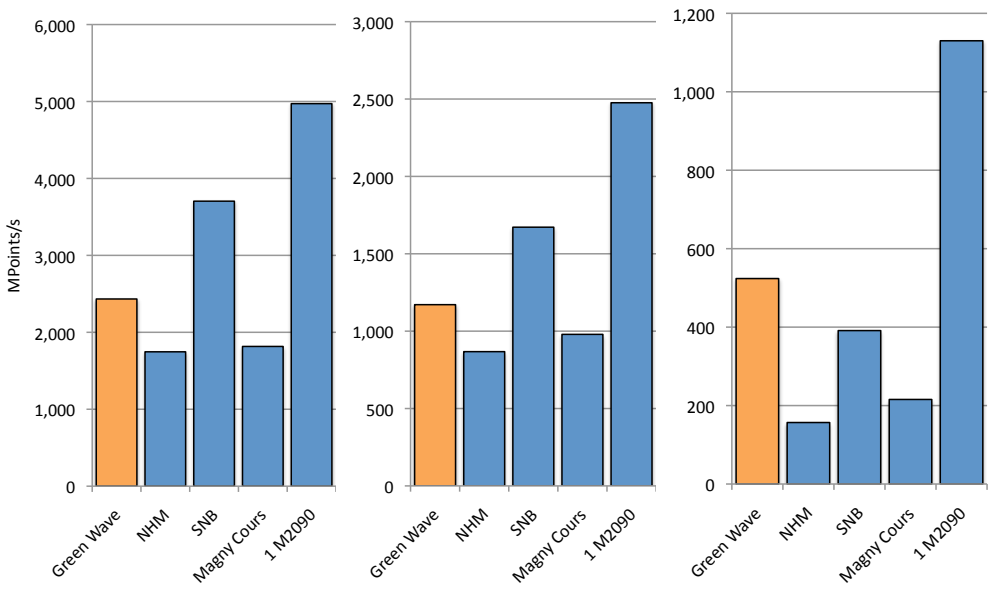
9.3 Green Wave Cluster Topology

Seismic imaging requires processing of large amounts of shots. All shots are independent from each other, which means that nearest neighbor communication must appear only between nodes computing on the same shot. To achieve a minimum communication overhead, nodes are organized in tidily-coupled sub- or shot-clusters. Only at the

beginning receiver data and earth properties are scattered to the corresponding nodes and results gathered after applied computation. Scaling over large node counts is achieved by increasing the amount of shot computation concurrency. The amount of communicating nodes interconnected is limited and depends on the shot size. This makes the choice of the interconnect topology a non-critical part. Therefore, this thesis gives no special recommendation for a specific Green Wave cluster topology.

9.4 Green Wave Multi-Node Performance Estimations

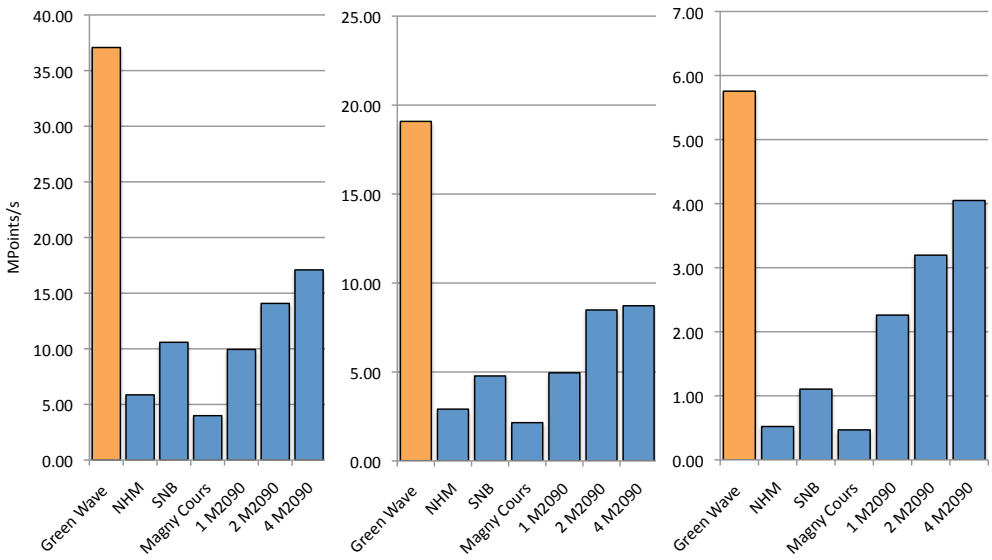
As explained in the multi-node comparison of COTS architectures in Section **??**, Green Wave cannot hide packing and unpacking of MPI buffers. Therefore, a communication overhead according to the volume size is added to single-node results in order to provide a fair comparison. Figure 9.3 adds Green Wave to the multi-node benchmark results previously presented for COTS architectures. As for x86-based architectures Green Wave performance suffers from the additional overhead and diminishes performance by 14%, 4% and 6% for ISO, VTI and TTI kernel, respectively.



(a) ISO (b) VTI (c) TTI

Figure 9.3: Green Wave Multi-Node Performance Benchmark

The modeled power consumption of a single Green Wave socket did not include the network interconnect (NIC). Since this power overhead is included for all other architectures an additional NIC power consumption of 10 Watt is assumed for Green Wave. The 10 Watts are combined by a 4 Watts power consumption of a QLogic QDR Infiniband host channel adapter [72] and another 6 Watts for a single-ported QLogic 12200 IB switch [71]. Figure 9.4 presents the multi-node energy efficiency estimations including Green Wave. Green Wave energy efficiency is decreased by the additional power overhead added by the NIC. However, the higher throughput lets Green Wave suffer more from multi-node communication and causes decreased energy efficiency. The benchmark shows that Green Wave offers great energy efficiency advantage of now 3.5× to 9.3× for ISO compared to x86-based architectures and 2.2× to 3.7× to single and multi-GPU nodes. For VTI kernel Green Wave achieves 4× to 8.8× compared to x86 and 2.2× to 3.8× to GPUs - TTI energy efficiency advantage stays similar with 5x to 12x compared to x86 architectures and 1.4× up to 2.5× to NVIDIA's M2090.



(a) ISO (b) VTI (c) TTI

Figure 9.4: Green Wave Multi-Node Energy Efficiency Benchmark

9.5 Green Wave Multi-Node Optimizations

The halo data exchange can already be hidden by overlapping the communication with computation via the two-step method. Unfortunately, the packing and unpacking of the communication buffers with the data points required by the neighboring nodes cannot be hidden. Only two of the six halo regions (ISO, VTI) don't have to be copied. To be able to hide the packing and unpacking as well, a scatter- gather address calculation unit could be added to the DMA engine. This way the DMA engine could gather data with different strides from the working array into the buffers and scatter received data from buffers back into the working array in one pass. The implementation and of such an scatter-gather unit and its influence on power consumption or area demands needs further studies and is the subject of future work.

9.6 Multi-Node Estimation Summary & Conclusion

This chapter extended the Green Wave single-node study and introduced a multi-node programming model. Additionally, the requirements on cluster topology and network interconnects were discussed. As for the evaluated COTS architectures, Green Wave cannot hide packing and unpacking of communication buffers, which reduces overall performance. Further, this chapter extended the Green Wave power model by a network interconnect.

The higher node performance and smaller node volume of Green Wave let it suffer more from the additional overhead than Nehalem or Magny Cours based nodes. The results show that for multi-node setups Green Wave still offer a great energy efficiency advantage to its competitors, which could even further advance with a proposed scatter-gather address calculation unit for the DMA engine and additional implementation of SIMD capable FPUs .

9.7 The Green Wave Cluster Architecture

This section compares energy efficiency based on simplifying cluster architecture assumptions. The different cluster setups are based on the evaluated node architectures including Green Wave designs. Based on the example large-scale survey introduced in Section 2.5, this section studies the forward and backward in time wave modeling kernel,

including cross-correlation of wavefields. For cross-correlation, the forward propagated source wavefield has to be written out and read back in during the backward in time modeling of the receiver wavefield. This makes it necessary to take a look into options for external storage configurations and how node setups affect external storage capacity and bandwidth requirements. The final benchmark assumes a fixed time-to-solution of one week. As cluster sizes need to be increased to stay in that timeframe the metric of comparison is the total power consumption in million Watts (MWatts).

9.7.1 Mass Storage Requirement Estimation

A mandatory design point for a system that is build for seismic modeling is that it provides enough external storage capacity to hold the complete data set, plus additional data that must be stored during the modeling process. External- or mass-storage is called any memory that does not reside directly on a compute node, like e.g. main memory, and is accessed via dedicated I/O calls. This storage is used to hold the initial data set, which size depends on the memory requirements per shot and the total number of shots. The size of a single shot changes with the acquisition setup. Per shot a certain number of receivers aligned in a two dimensional grid are used. Therefore, a total of $Receiver_X \times Receiver_Y$ receivers are listening for acoustic signals for a predefined time of T_{listen} milliseconds. Every, T_{rec} milliseconds all receivers record the amplitudes of the subsurface reflected waves. Thus, the total number of samples per shot can be calculated as:

$$\text{Number of Samples } = \text{Number of Receivers} \times T_{listen}/T_{rec}$$

Each sample is stored in single-precision accuracy of 32 bits and multiplying the number of samples by 4 (Bytes) gives us the amount of data stored per shot.

As well as the memory requirements per shot the acquisition setup determines the required number of shots. For the example survey of size X_{Survey} and Y_{Survey} and shot offsets of X_{off} and Y_{off} the total amount of shots can be calculated by:

$$\text{Number of Shots} = (X_{Survey}/X_{off}) \times (Y_{Survey}/Y_{off})$$

Adding the velocity field and additional arrays for earth properties like Thomson parameters, the amount of data per survey is

$$\text{Survey Memory Requirements} = \#\text{Shots} \times \#\text{Samples} + \text{Earth Properties}$$

The amount of additional storage that is required depends on the RTM scheme used (see Section 2.4.2). The highest requirements on capacity occur for the "basic" RTM scheme 2.4.2.1. This scheme writes full wavefields out on disk and doesn't use any compression. The amount of data written depends on the time discretization accuracy of the final image. Usually a finer time discretization is needed for numerical stability than for the final image. During the backward propagation part of the receiver wavefield, the data is read back in and cross-correlated with the receiver wavefield, which finally repositions reflectors to their true subsurface position.
As for time discretization, numerical stability might require a finer discretization in space as well. Not all of these additional grid points are needed for the final subsurface image, which reduces the overall size per wavefield written out and read back in from external storage devices. We define TS_{res} as the required number of timesteps for the final result and TS_{size} the corresponding size. This leads to additional I/O capacity requirements of

$$TS_{res} \times TS_{size} \times \text{Number of Shots.}$$

Now, the equations are used to derive the capacity requirements of I/O storage for the example large-scale survey introduced in Section 2.5. The example survey uses 1000 receivers listening for 12 seconds and recording every 4ms the amplitudes of the reflected waves. This results in $3 * 10^7$ samples per shot. With single-precision floating-point accuracy (4 bytes per data value) one shot requires 0.12 GB of storage. Given 120,000 shots this adds up to a total of 14.4 TB. In the case of a basic implementation scheme of RTM (see Section 2.4.2.1) the complete source wavefield has to be stored to external storage devices for certain timesteps, e.g. 4ms time sampling and a reduced space sampling interval. The example survey uses 1ms time sampling interval and a 5m x 5m x 5m space discretization for wave propagation and a reduced space grid of 20m x 20m x 10m summed every 4ms for the final image. This leads to a 32x reduction in volume size. As a result, additional 13 TB of data storage is allocated for the source wavefield for all shots. In the case of just one shot computed at a time such a large survey consumes about 27.4 TB of external storage space.

For this study we assume the basic RTM scheme. As previously described, the storage devices hold the initial data which includes the velocity field, the results, etc. - and the wavefields written out during the forward propagation step. Since this data is only of local interest dedicated storage devices are directly connected to the nodes. This way I/O bandwidth and access latency is reduced. An optimal system should be balanced, which means neither oversubscribed in I/O bandwidth nor in capacity, to avoid unnecessary power consumption.

Two main on-market commodity alternatives are considered for node-attached mass-storage:

- **Hard Disks (HDD)** are the most common alternative for mass-storage. The advantages are low costs and high capacity but HDDs generally provide low bandwidth. In this study a platter size of 3 TB and an effective bandwidth of 125 MB/s is assumed.

- **Solid State Drives (SSD)** are the second alternative that are explored in this study. Solid State Drives became the subject of interest in recent years as they provide high bandwidth and low power consumption. SSDs are build on non-volatile NAND flash chips to store data even if not attached to a power source. SSDs have the disadvantage of low capacity and high costs per GB. For this study it is assumed that one SSD has a capacity of 480 GB and a sustained bandwidth of 450 MB/s.

I/O bandwidth of HDDs and even SDDs is often not sufficient enough to satisfy the demands of scientific applications. One way to achieve reliability and high bandwidth for external storage devices is described by so-called *Redundant Array of Inexpensive Disks* (RAID) levels. In 1988 D. Patterson proposed techniques to improve I/O performance and reliability by using multiply commodity, and therefore cheap, disks ([68], [12]). The paper originally introduced RAID level 1 to 5 with each level using redundancy to achieve better *Mean Time Between Failure* (MTBF) - the common metric of reliability for storage devices. Later on, RAID level 0 was introduced. Level 0 has no data redundancy and simply distributes data between multiple disks to achieve high throughput. By using an I/O controller for each disk the read and write performance scales along with the number of disks.

This cluster performance analysis assumes a simplified RAID level 0 storage model,

whereas the required number of disks $N_{Capacity}$ for each node are calculated by dividing the total node capacity requirements for external storage by the capacity of a single disk:

$$N_{Capacity} = \text{Node Capacity Requirements / Capacity per Device}$$

In a similar way the number of required disks to achieve the I/O bandwidth requirements $N_{Bandwidth}$ is derived. Here, the total required node I/O bandwidth is divided by the bandwidth achieved by a single disk:

$$N_{Capacity} = \text{Node Capacity Requirements / Capacity per Device}$$

In this study bandwidth is defined as successfully transferred requested bytes per second, and a product of the actual I/O bandwidth and additional read-write or access latencies .

The number of storage devices is then chosen by:

$$\text{Number of Devices} = \max \begin{cases} \text{Number of Devices to reach Capacity } (N_{Capacity}) \\ \text{Number of Devices to reach Bandwidth } (N_{Bandwidth}) \end{cases}$$

This study does not exploit effects of different file-systems nor fault resilience, which is subject of future work.

Table 9.2 lists capacity and bandwidth requirements for all kernels and all architectures. One can see that bandwidth requirements are generally low. It ranges from 15 to 100 MB/s for ISO, 7 to 60 MB/s for VTI and 1 to 30 MB/s for TTI. The bandwidth requirements directly scale with node performance. Neither a single HDDs nor SDDs should have problems to provide it. A different picture is drawn for capacity requirements. All node volumes had been maximized to lower the percentage of the communication overhead in relation to the time spend in computation. This leads to large capacities required for x86-based architectures. As Nehalem, Magny Cours and Sandy Bridge nodes own 32 GB of main memory the node volumes are the same and therefore external storage capacity requirements. Each x86 architecture requires 604 GB for ISO, 302 GB for VTI and 151 GB for TTI. For GPUs the node volume size depends on the number of GPUs. Consequently, higher numbers of GPUs require more external storage capacity. Still, the much smaller node volumes require only 50 to 200

GB per node for ISO, half of it for VTI and one fourth for TTI. The two-pass approach for TTI (see Section 5.5) requires two additional arrays, which increases the total number to 13. Given these numbers, it is now possible to calculate the additional power overhead added by the external storage devices. Due to the high capacity requirements HDDs are better suited for large node volumes - as seen for x86-based architectures - and SSDs for smaller node volumes and faster computation times - as seen for GPUs and Green Wave. The additional overhead added per node is marginal with 1 to 2 Watts.

	ISO		**VTI**		**TTI**	
	GB	MB/s	GB	MB/s	GB	MB/s
NHM	604.0	13.6	302.0	6.8	151.0	1.2
SNB	604.0	28.9	302.0	13.1	151.0	3.1
Magny Cours	604.0	14.2	251.7	7.7	151.0	1.7
1 M2090	50.3	29.6	25.2	15.8	12.6	8.2
2 M2090	100.7	53.8	50.3	33.8	25.2	14.6
4 M2090	201.3	93.2	100.7	59.9	50.3	27.2
Green Wave	25.2	19	18.8	9.1	12.6	4.1

Table 9.2: Node I/O Capacity and Bandwidth Requirements

9.7.2 The Green Wave Cluster Memory Hierarchy

The most distant memory, from the cores, are storage nodes that hold the initial survey data set (receiver data, velocity, etc...). Latency and bandwidth are non-critical since the survey data set is scattered to node-dedicated storage devices only once at the very beginning. Afterwards, access to non-node attached storage is not required anymore for the rest of the computation. These storage nodes have to provide about 15 TB of capacity for the example survey. The next storage hierarchy level contains node-attached storage devices, which could be either solid state drives or hard disks. The capacity of node attached I/O devices depends on the node subdomain size. For Green Wave design 15 GB is sufficient, however, such storage drives can be shared to avoid oversubscription in bandwidth or capacity. The following hierarchy level contains node-dedicated main memory modules. The size depends on the node volume as well. Green Wave node configurations optimized for ISO kernels require only 2 GB, VTI kernel 2.9 GB and TTI designs slightly more than 3 GB. The two hierarchy levels closest to the

core are its own local store of either 128 KB or 256 KB size, its 16 KB cache and its register files.

Each node holds its own results in main memory. After finishing the migration of one shot the results are written back to the storage nodes. Note that non-node attached storage nodes are not included in this study. Figure 9.5 visualizes the Green Wave cluster memory hierarchy.

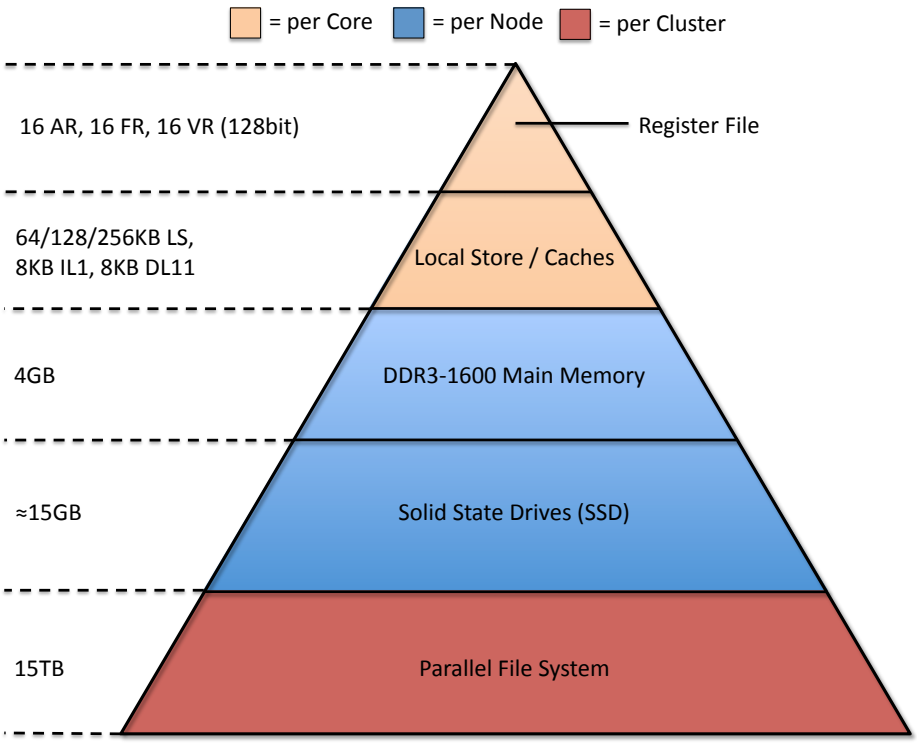


Figure 9.5: The Green Wave Memory Hierarchy

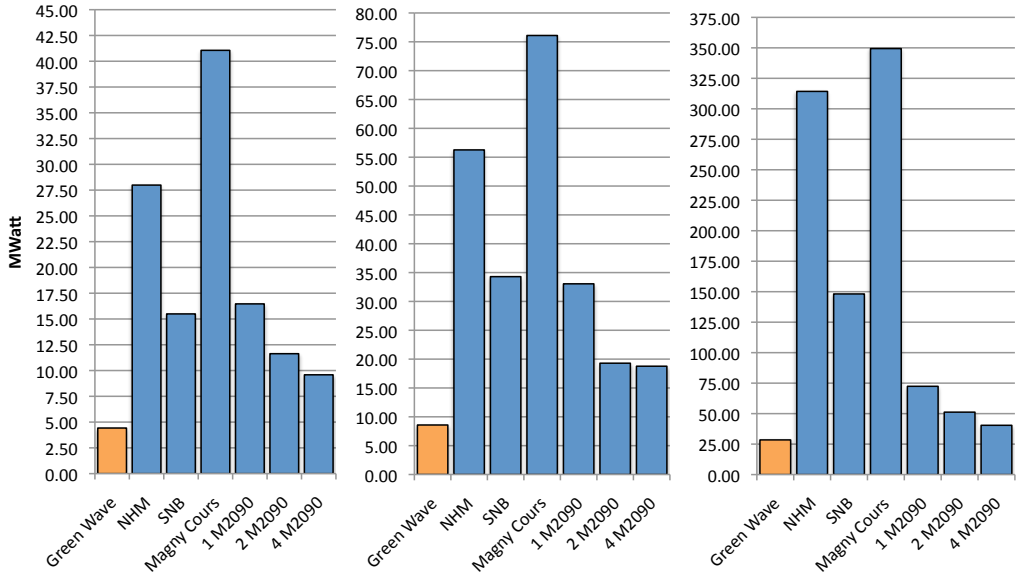
9.7.3 Cluster Power Consumption Comparison

This section compares the power consumption of simplified clusters architectures running a basic RTM. Power modeling includes external storage, network interconnects, main memories and compute sockets. Dependent on the node architecture, different requirements in speed and capacity exist. Four-GPU node setups require more bandwidth for the backward propagation part to read the forward propagated source wavefield in time to overlap computation and communication. For all architectures

the required amount of storage devices per node is derived and the additional power overhead added. This should provide a first good estimate in how far hardware/software co-designed node architectures are affecting overall cluster energy efficiency. Not included in the estimates are facility costs, cooling, fault tolerance and etc.
All architectures are normalized to a fixed time-to-solution of one week. As shots are completely independent from each other, shot-cluster concurrency is increased accordingly to achieve the one week computation time. Figure 9.6 presents the results of the total system power consumption in MWatts. Green Wave offers best energy efficiency that is 3.5× better than the most energy efficient x86-based cluster. The ISO Green Wave cluster design would consume only about 4.4 MWatts compared to 15 MWatts for Sandy Bridge based clusters. Compared to GPU clusters Green Wave still shows an impressive 2.2× improvement.
Green Wave cluster setups specialized for VTI show comparable power consumption and energy efficiency improvements of 4× to x86 and 2.2× compared to GPUs.
Comparing the cluster power consumption running TTI kernel Green Wave advantage increases to 5× compared to x86 and about 1.4× to clusters based on four M2090's per node. A TTI Green Wave cluster would consume about 29 MWatts of power where four-GPU cluster setups end up consuming 41 MWatts, which would directly translate in annual money savings of $12 million (see Section 6.1).

9.8 Green Wave Design Comparison

For all three seismic wave propagation kernels the hardware/software co-design methodology was applied and the most energy efficient architecture designs derived. As it is unlikely for a company to buy three kernel dedicated cluster setups, this section analyzes the energy efficiency improvements achieved by kernel specialization. Figure 9.7 presents the power consumption variation, relative to the kernel specific design on the y-axis and the three different Green Wave architectures on the x-axis. The different columns represent the three kernels running on the specific Green Wave designs. An y-value greater than one means a higher total power consumption of the cluster. The power consumption is normalized for all designs to the best energy efficiency, achieved by the specific design - e.g. an ISO kernel running on the ISO specific Green Wave design is taken as reference ISO power consumption and has therefore a y-value of *1x*.



(a) ISO (b) VTI (c) TTI

Figure 9.6: Power Requirements of Cluster Setups for a fixed Time-to-Solution of one Week.

One can see that, especially for the instruction bound TTI kernel, both, ISO and VTI Green Wave design show significantly higher power consumption. As those architectures do not deliver enough throughput more nodes are required to achieve the desired time-to-solution of one week. At a first glance 1.6x higher power consumption does not seem too impressive, but contributes another 17 MWatts to the cluster design. The VTI specific Green Wave design achieves only marginal better energy efficiency than running VTI kernels on Green Wave ISO architecture. It is important to remember the smaller local store size of the ISO design and an accordingly smaller node domain result in an increase in nodes required. Contrary, running ISO kernels on VTI specific design results in a 1.13x increase in power consumption. Such a combination would lead to an unbalanced node design, which is oversubscribed in memory bandwidth.

9.9 Summary & Conclusion

This chapter took the results from the multi-node study and extended them to a simplified cluster study including node-external mass-storage devices. For all architectures

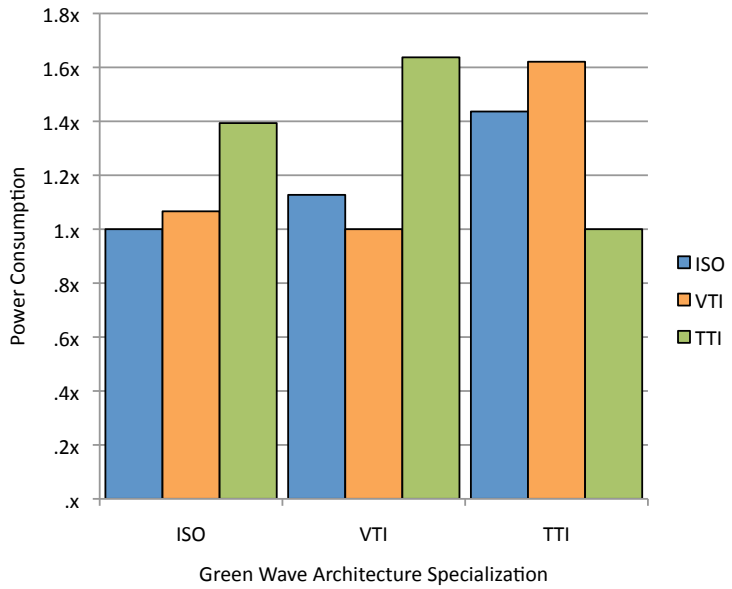


Figure 9.7: Green Wave Architecture Design Comparison

the requirements on capacity and bandwidth were analyzed, based on the example survey introduced in Section 2.5. The additional power overhead was added accordingly and the Green Wave cluster power consumption compared to the evaluated COTS architectures for a fixed time-to-solution of one week.

The numbers show that the efficiency advantage of Green Wave is diminished by the additional overhead. Still, Green Wave delivers great energy efficiency improvements of 1.4× to 5× compared to the most energy efficient COTS architectures and could further be improved with additional proposed optimizations.

Finally, this chapter quantifies the benefits of have Green Wave tailored to specific seismic kernels. It concludes that a Green Wave ISO design would provide almost the same energy efficiency for VTI kernels as the VTI specific design but would require higher number of nodes. Differently, TTI requires a specific design to achieve best energy efficiency.

Chapter 10

Conclusion & Future Work

The goal of this work was to explore the potential of the hardware/software co-design methodology applied to seismic kernels, like they appear in Reverse Time Migration imaging methods, to achieve maximum energy efficiency for large-scale computing clusters. The main contributions of this work is the analysis of the requirements of the partial differential equations, to derive the corresponding system design parameters and programming models, describe optimization capabilities and performance on commercial off-the-shelf hardware and to introduce a custom chip design to achieve that target. Finally, this thesis presents estimations how clusters, build upon the evaluated architectures, would compare in terms of throughput and energy efficiency for large-scale surveys.
The Tensilica Processor Generator (XPG) was used to achieve a rapid software analysis, hardware optimization turn-around time. The good energy efficiency improvement predictions from this study motivate to take this approach into further detail and could present a way to design future compute architectures.

The individual kernels analyzed in this thesis are an isotropic (ISO), vertical transverse isotropic (VTI) and tilted transverse isotropic (TTI) version of the wave equation. All three kernels are analyzed in terms of their computational demands. Based on their individual requirements the effectiveness of different software optimization techniques was exploited on current and next-generation node and cluster architectures from Intel, AMD and NVidia. With the introduction of multi-core SMPs, the importance of optimizing software to a specific node design became even more critical to gain top

performance and energy efficiency. This study highlights the necessity on architecture-specific code optimization for higher-order stencil based kernels on future SMPs that will use even higher number of cores. The performance improvements achieved with software optimization is up to 10x for AMD Magny Cours nodes and up to 7x for Sandy Bridge based nodes for TTI kernels.

Production survey sizes are too large to fit on a single node which makes distributed programming necessary. The increasing number of cores per node and therefore the increasing node performance introduces a new challenge for distributed memory applications: The less time spend within the computational part of the program the higher the percentages of time spend in communication. Therefore, this thesis analyzes different ways to optimize the inter-node communication via the Message Passing Interface (MPI). The results show that not only the computational kernel needs to be optimized to the underlying node architecture. Asynchronous messaging can hide data transfers but packing and unpacking of communication buffers cannot be overlapped. This thesis finds that it is critical to optimize the communication model to account for NUMA affinity.

The design of Green Wave is based on the experience gained from the analysis of on-market systems. Understanding the strength and weaknesses of different architectural approaches enables to pick energy efficiency and performance improving design choices while leaving out all performance diminishing aspects.
This thesis leveraged the hardware/software co-design methodology known from the embedded and mobile community for HPC. Using experience from the Green Flash project that applied the co-design methodology on climate codes, this thesis derives the appropriate architectural parameters first for a single-node, next for a multi-node environment, and finally provides a performance and energy efficiency estimation for a complete Green Wave cluster architecture for an example large-scale sized seismic survey over a normalized time-to-solution.

A single Green Wave socket tailored for explicit RTM isotropic wave propagation kernels uses 128 highly coupled cores, which are connected via a 2D concentrated torus

NoCs. Additionally, each core uses a 128 KB software-managed local memory to capture all temporal recurrences, which minimize main memory bandwidth pressure. A design optimized for VTI increases the local store size to 256 KB and decreases the number of cores to 96 for a balanced and most energy efficient architecture design. A Green Wave system design tailored for TTI utilizes 256 cores with a 128 KB local store. Each socket has a COTS quad-channel DDR3-1600 memory interface that delivers a bandwidth up to 51.2 GB/s.
To optimize the overall throughput even further, the RISC ISA is extended to support stencil-based codes. Next, by using Tensilica's cycle-accurate, market-strength instruction set simulator (ISS), the novel architecture was directly compared to the evaluated on-market hardware solutions in single and multi-node setups.
The novelty in this approach is not the custom instructions taken in isolation, but rather the contribution of these instructions in the context of a co-design methodology where only the specific functionality needed to efficiently solve a problem is added to the hardware. Although, many of these features are available in other existing architectures, co-design is able to provide only the subset that improves performance — thus maximizing power efficiency while maintaining general programmability. In addition, the general-purpose nature of these instructions allows them to be applied to other stencil-based computations, allowing the Green Wave solution to be applicable to a wide-variety of high-order methods. Finally, these custom instructions allow high performance with a simpler programming methodology than Intel intrinsics or NVIDIA's CUDA.

The thesis concludes that application-tailored architecture design can provide a superior energy efficiency and performance by keeping general programmability and portability. Nevertheless, programming model that are able to hide communication and memory latencies are highly important for many-core, local store based architectures like Green Wave. An example programming model for on-chip and off-chip communication was introduced accordingly. Figure 10.1 shows where such a co-design approach fits in compared in terms of application specificity and energy efficiency to COTS architectures. Similar to GPUs, hardware/software co-designed architectures fit best for a certain class of applications where they can perform at their highest efficiency. Basing Green Wave on highly energy efficient cores from the mobile market the design

is not limited to its specific class of applications but keeps a certain level of generality. As the Green Wave node is not an accelerator card with dedicated memory, it avoids the main disadvantage of current GPUs, which require additional memory copies and communication with a host CPU.

Based on a large-scale example survey, the results show that Green Wave provides the potential of an energy efficiency improvement of 3.5x to 12x compared to x86 and about 1.4x to 4x to M2090 based clusters for ISO, VTI and TTI kernels.

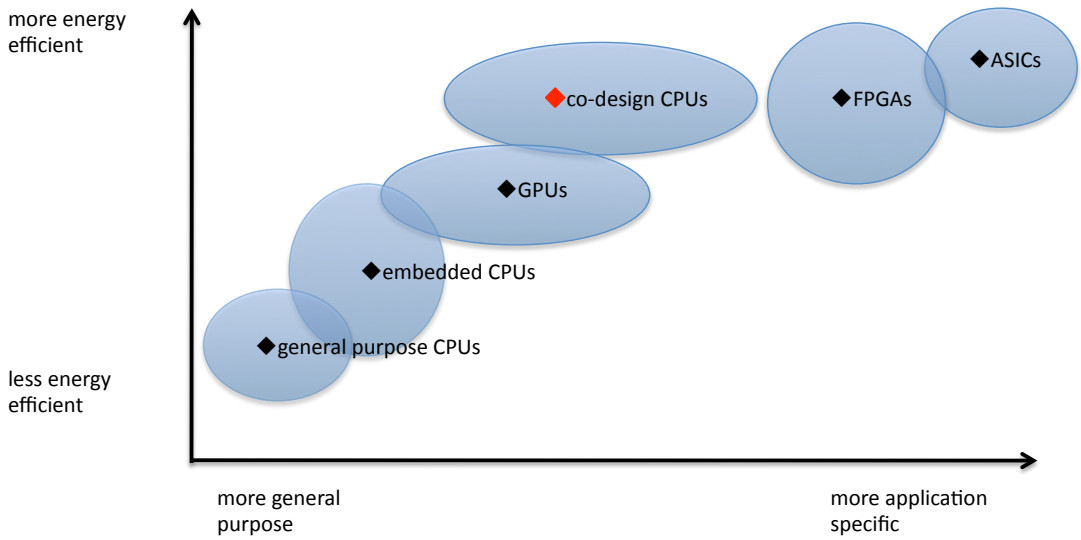


Figure 10.1: This diagram draws energy efficiency (y-axis) over application specialization (x-axis) for common on-market architecture designs.

As other architectures, Green Wave suffers from additional overhead added by scatter-gather memory accesses through writing to and from MPI buffers. A future study might explore the benefit from scatter-gather units within the DMA engine. This way inter-node communication could be completely overlapped with computation and additional energy efficiency gains could be achieved especially for kernels where communication takes a large portion of total runtime. A second study should evaluate the possible benefits of a 2-way SIMD unit as it promises great efficiency enhancement for TTI kernels. Based on the instruction performance ceiling up to 60% performance gain is possible and would directly lead to improved energy efficiency. Future work

should trade off gained performance against increased programming complexity and energy consumption of this SIMD design.
The Green Wave design choice depends on which kernel version is suppose to run on it most of the time. If the focus is more on subsurface structure with isotropic or VTI like properties the ISO design could be a good trade-off since ISO and VTI show similar requirements. If the target are more complicated subsurface structures that require accurate modeling of anisotropic properties the TTI design should make the race. It provides best floating-point performance and still keeps some capacity in its local stores to be suited for future, even more complex and flop-heavy wave-propagation kernels.
To bring the Green Wave design closer to an actual production design, additional in-depth studies have to take place for all parts of the architectural design. Even though market-strength Tensilica tools provide pre-verified RTL that can be mapped on FPGAs for verification, timing, reliability and fault tolerance have to be analyzed further. For a complete cluster setup this thesis provides a rough estimate only. Many more issues have to be considered to realize a production system - examples are cooling and facility. All these issues need to be exploited by several teams of engineers and computer scientists. Overall, this thesis introduced a promising novel approach how future supercomputer architectures can achieve required performance by maintaining a feasible power consumption and generality.

References

[1] Saman Amarasinghe. ExaScale Software Study: Software Challenges in Extreme Scale Systems, September 2009.

[2] G. Amdahl. The validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of AFIPS Spring Joint Computer Conference*, 1967.

[3] K. Asanovic, R. Bodik, B. Catanzaro, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS, University of California, Berkeley, 2006.

[4] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, New York, NY, USA, 2006. ACM.

[5] Edip Baysal, Dan Kosloff, and John Sherwood. Reverse Time Migration. In *Geophysics*, volume 48, 1983.

[6] Scott Beamer. Designing Multi-socket Systems Using Silicon Photonics. In *23rd International Conference on Supercomputing (ICS-09)*, 2009.

[7] J. Bee Bednar. Two-way wave equation migration: Overkill or Necessity. Technical report, Core Laboratories, 2003.

[8] A. J. Berkhout. Grand Challenges for Geophysics, a seismic vision of the future. In *SEG Houston 2009 International Exposition and Annual Meeting*. SEG, 2009.

[9] Biondo L. Biondi. *3D Seismic Imaging*. Society of Exploration Geophysics, 2006.

[10] R. Philip Bording and Christopher L. Liner. Theory of 2.5-D reverse time migration. In *Ann. Internat. Mtg: Soc. of Expl. Geophys.*, volume 64th, pages 692–694. University of Tulsa, 1994.

[11] Cadence Inc. Denali DDR3 memory controller IP. Whitepaper, April 2011.

[12] Peter Ming-Chien Chen, Garth A. Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two Papers on RAIDs. Technical Report UCB/CSD-88-479, EECS Department, University of California, Berkeley, Dec 1988.

[13] Chunlei Chu and Paul L. Stoffa. Acoustic Anisotropic Wave Modeling Using Normalized Pseudo-Laplacian. In *SEG Denver 2010 Annual Meeting*, 2010.

[14] Robert Clapp. Reverse time migration with random boundaries. In *SEG Houston 2009 International Exposition and Annual Meeting*. SEG, 2009.

[15] Robert G. Clapp, Haohuan Fu, and Olav Lindtjorn. Selecting the right hardware for reverse time migration. *The Leading Edge*, 29(1), 2010.

[16] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20 – 24, New York, NY, USA, 1970. ACM.

[17] G.C. Cohen. *Higher-order numerical methods for transient wave equations*. Scientific computation. Springer, 2002.

[18] Convey. http://www.conveycomputer.com/, 2012.

[19] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, et al. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *Proceedings SC '08*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[20] John D. Davis, Charles P. Thacker, and Chen Chang. BEE3: Revitalizing Computer Architecture Research. Technical report, Microsoft Research, 2009.

[21] D. Donofrio, L. Oliker, J. Shalf, M. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin. Energy-Efficient Computing for Extreme-Scale Science. In *IEEE Computer*, 2009.

[22] David Donofrio, Leonid Oliker, John Shalf, Michael F. Wehner, Chris Rowen, Jens Krueger, Shoaib Kamil, and Marghoob Mohiyuddin. Energy-Efficient Computing for Extreme-Scale Science. *Computer*, 42(11):62–71, November 2009.

[23] X. Du, R.P. Fletcher, and P.J. Fowler. A New Pseudo-acoustic Wave Equation for VTI Media. In *EAGE*, 2008.

[24] James P. Durbano, Fernando E.Ortiz, John R. Humphrey, Mark S. Mirotznik, and Dennis W.Prather. Hardware Implementation of a Three Dimensional Finite-Difference Time-Domain Algorithm. In *IEEE Antennas and Wireless Propagation Letters*, volume 2, 2003.

[25] Eric Dussaud, William Symes, Paul Williamson, Larent Lemaistre, Paul Singer, Bertrand Denel, and Adam Cherrett. Computational strategies for reverse-time migration. In *SEG Las Vegas 2008 Annual Meeting*, Las Vegas NV USA, 2008. Total.

[26] Roger Espasa and James E. Smith. Vector Architectures: Past, Present and Future. In *ICS'98 Melbourne Austria*, 1998.

[27] Darren Foltinek, Daniel Eaton, Jeff Mahovsky, Peyman Moghaddam, and Ray McGarry. Industrial-Scale Reverse Time Migration on GPU Hardware. In *SEG Houston International Exposition*, 2009.

[28] Fraunhofer ITWM. GPI - http://www.gpi-site.com, Dezember 2011.

[29] Fraunhofer ITWM. HPC TOOLS - http://www.itwm.fraunhofer.de/abteilungen/hpc/hpc-tools.html, Dezember 2011.

[30] Haohuan Fu, Robert Clapp, Oskar Mencer, and Oliver Pell. Accelerating 3D Convolution using Streaming Architectures on FPGAs. In *SEG*, 2009.

[31] Haohuan Fu, William Osborne, Robert G. Clapp, Oskar Mencer, and Wayne Luk. Accelerating Seismic Computations Using Customized Number Representations on FPGAs. *EURASIP Journal on Embedded Systems*, 2008.

[32] G.Fairweather and A.R.Mitchell. A High Accuracy Alternating Direction Method for the Wave Equation. *J.Inst. Maths Applics*, 1:309–316, 1965.

[33] Samuel H. Gray, Carl Notfors, and Norman Bleistein. Imaging using multi-arrivals: Gaussian beams or multi-arrival Kirchhoff? In *SEG International Exposition and 72nd Annual Meeting*, 2002.

[34] Chuan He, Mi Lu, and Chuanwen Sun. Accelerating Seismic Migration Using FPGA-based Coprocessor Platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[35] Gilbert Hendry. Analysis of Photonic Networks for a Chip Multiprocessor Using Scientific Applications. In *ACM/IEEE International Symposium on Networks-on-Chip*, 2009.

[36] Gilbert Hendry, Johnnie Chan, Shoaib Kamil, Lenny Oliker, John Shalf, et al. Silicon Nanophotonic Network-on-Chip Using TDM Arbitration. *High-Performance Interconnects, Symposium on*, 0:88–95, 2010.

[37] Gilbert Hendry, Shoaib Kamil, and Aleksandr Biberman. Analysis of photonic networks for a chip multiprocessor using scientific applications. In *NOCS*, pages 104–113, 2009.

[38] N. R. Hill. Prestack Gaussian-beam depth migration. In *Geophysics*, volume 66, pages 1240–1250, 2001.

[39] IBM. *Cell Broadband Engine Programming Handbook.*

[40] Ilyasov. *Handbook of GeoMathematics*, volume 1. Springer, 2010.

[41] Maxim Ilyasov. *Helmholtz Wavelets on Non-smooth Regions and Their Application to Seismic Data Processing.* PhD thesis, University of Kaiserslautern, 2010.

[42] Innovative Computing Laboratory (ICL), University of Tennessee. PAPI - http://icl.cs.utk.edu/papi/.

[43] Intel. Intel Cilk Plus - http://software.intel.com/en-us/articles/intel-cilk-plus/, July 2011.

[44] William J.Clamp and Philippe Thierry. Trends for high-performance scientific computing. *The Leading Edge*, 29(1):44–47, January 2010.

[45] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and Explicit Optimizations for Stencil Computations. In *ACM Proceedings 2006*, Berkeley,CA USA, October 2006. ACM.

[46] Peter Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, September 2008.

[47] Tom Krazit. http://www.pcworld.com/article/118165/intel-shelves-plans-for-4ghz-p4.html, 2004.

[48] Mark LaPedus. Intel develops 0.07-micron transistor for 10-GHz processors by 2005. www.eetimes.com, December 2000.

[49] Lawrence Berkeley National Laborator, UC Berkeley. Berkeley UPC - Unified Parallel C, July 2011.

[50] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[51] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *International Symposium on Computer Architecture*, 2007.

[52] Wei Liu et al. Anisotropic Reverse-Time Migration Using Co-Processors. In *SEG Houston 2009 International Exposition and Annual Meeting*. SEG, 2009.

[53] Yang Liu and Mrinal K Sen. Advanced finite-difference methods for seismic modeling. *GEOHORIZONS*, December 2009.

[54] Maxeler. www.maxeler.com, 2012.

[55] Paulius Micikevičius. 3D Finite Difference Computation on GPUs using CUDA. Technical report, Nvidia, 2009.

[56] Paulius Micikevičius. Multi-GPU Programming for Finite Difference Codes on Regular Grids, January 2012.

[57] Micron Inc. Calculating Memory System Power for DDR3, June 2010.

[58] Microsoft. BEE3 - Microsoft Research - Revitalizing Computer Architecture Research, 2010.

[59] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[60] E. Motuk, R. Woods, and S. Bilbao. Implementation of finite difference schemes for the wave equation on FPGA. Technical report, University of Belfast, 2005.

[61] Carl Notfors, Yi Xie, and Sam Gray. Gaussian Beam migration: a viable alternative to Kirchhoff ? In *AESC2006*, 2006.

[62] NVidia Corporation. http://developer.nvidia.com/gpudirect.

[63] NVidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, NVidia, 2010.

[64] OpenMP Architecture Review Board. The OpenMP® API specification for parallel programming - http://openmp.org/, July 2011.

[65] Francisco Ortigosa. Seismic Imaging is in the mind and at the fingertips: The future of Seismic Imaging Business. In *SEG*, 2008.

[66] Francisco Ortigosa, Mauricio Araya-Polo, Felix Rubio, Mauricio Hanzich, Raul de la Cruz, and Jose Maria Cela. Evaluation of 3D RTM on HPC Platforms. In Barcelona Supercomputing Center, editor, *SEG Las Vegas 2008 Annual Meeting*. SEG, 2008.

[67] Jairo Panetta et al. Computational Characteristics of Production Seismic Migration and its Performance on Novel Processor Architectures. In *SBAC-PAD*, pages 11–18, 2007.

[68] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD Conference*, pages 109–116, 1988.

[69] Hanspeter Pfister and Arie Kaufman. Cube-4 - A Scalable Architecture for Real-Time Volume Rendering. In *Symposium on Volume Visualization*, 1996.

[70] Suhas Phadke, Dheeraj Bhardwaj, and Sudhakar Yerneni. 3D Seismic Modeling in a Message Passing Environment. In *In proceedings of 3rd Conference and Exposition on Petroleum Geophysics*, 2000.

[71] QLogic. *QLogic 12200 IB Switch.* QLogic Corporation, 26650 Aliso Viejo Parkway, Aliso Viejo, CA.

[72] QLogic. *QLogic QLE7340 Single-Port 40 Gbps QDR InfiniBand Host Channel Adapter*, 2011.

[73] J. Fowler R. Fletcher, X. Du. Pure P-wave Propagators Versus Pseudo-Acoustic Propagators. In *EAGE*, 2010.

[74] Xiang Du Robin Fletcher and Paul J. Fowler. Stabilizing acoustic reverse-time migration in TTI media. In *SEG Houston International Exposition*, 2009.

[75] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[76] Samarskii. *Theorie der Differenzenverfahren.* Akademische Verlagsgesellschaft Geest Portig K.G., 1984.

[77] Samarskii and Nikolaev. *Numerical Methods for Grid Equations*, volume 2. Birkhäuser Verlag, 1989.

[78] Samarskii and Nikolaev. *Numerical Methods for Grid Equations*, volume 1. Birkhäuser Verlag, 1989.

[79] Sandia National Laboratories. Structural Simulation Toolkit (SST), 2011.

[80] Paul Sava. Introduction to this special section: High-performance computing. *The Leading Edge*, 29(1):42–43, 2010.

[81] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept/Oct 2000.

[82] Sushant Sharma, Chung hsing Hsu, and Wu chun Feng. Making a case for a Green500 list. In *In Proc. of the Workshop on High-Performance, Power-Aware Computing*, 2006.

[83] Sushant Sharma, Chung-Hsing Hsu, and Wu chun Feng. http://www.green500.org.

[84] David E. Shaw, Martin M. Deneroff, et al. Anton, a special-purpose machine for molecular dynamics simulation. *Commun. ACM*, 51:91–97, July 2008.

[85] R. Sheriff and L. Geldart. Exploration Seismology. *Cambridge University Press*, 1995.

[86] Timothy Sherwood, George Varghese, and Brad Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, June 2003.

[87] Silicon Creations Inc. Si Creations Programmable PLL IP product. Whitepaper, April 2008.

[88] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The SOLOMON computer. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, AFIPS '62 (Fall), pages 97–107, New York, NY, USA, 1962. ACM.

[89] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press Cambridge, 1995.

[90] Robert Soubaras and Yu Zhang. Two-step explicit marching method for reverse time migration. In *SEG Las Vegas 2008 Annual Meeting*. CGGVeritas, 2008.

[91] E.L. Stoll. *Encyclopedia of computer science and engineering (2nd ed.)*. Van Nostrand Reinhold Company Inc., 1983.

[92] Yonghe Sun. 3-D prestack Kirchhoff beam migration for depth imaging. In *Geophysics*, volume 65, pages 1592–1603, 2000.

[93] Herb Sutter. The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 2005.

[94] William Symes. Reverse time migration with optimal checkpointing. In *SEG 2007 Annual Meeting*. Department of Computational and Applied Mathematics, Rice University, 2007.

[95] Tensilica Inc. http://www.tensilica.com - Tensilica: Customizable Processor Cores for the Dataplane.

[96] Tensilica Inc. Xtensa Architecture and Performance. Whitepaper, October 2005. http://www.tensilica.com/pdf/xtensa-arch-white-paper.pdf.

[97] Tensilica Inc. TIE - The Fast Path to High Performance Embedded SOC Processing. Whitepaper, April 2009.

[98] Leon Thomsen. Weak elastic anisotropy. In *Geophysics*, volume 51, pages 1954–1966, 1986.

[99] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[100] Top500.org. http://www.top500.org/list/2007/11/100, November 2007.

[101] Top500.org. http://www.top500.org, September 2011.

[102] Baerbel M. Traub. *Anisotropic Parameter Estimation from PP and PS Waves in 4-Component Data.* PhD thesis, University of Karlsruhe, 1999.

[103] Peter N. Vabishevich. *Computational Methods of the Mathematical Physics. Inverse and Control Problems.* Vuzovskaya kniga, 2009.

[104] Roelof Versteeg. The Marmousi experience: Velocity model determination on a synthetic complex data set. *The Leading Edge*, 13(9):927–936, 1994.

[105] D. Wang, B.Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, 2005.

[106] Zhijing Wang. Seismic anisotropy in sedimentary rocks. In *Annual meeting of the Society of Exploration Geophysicists*, 2001.

[107] M. Wehner, L. Oliker, and J. Shalf. Green Flash: Designing an energy efficient climate supercomputer. *IEEE Spectrum*, 2009.

[108] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific Computing Kernels on the Cell Processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.

[109] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, UC Berkeley, 2008.

[110] www.streambench.org. The STREAM Benchmark: Computer Memory Bandwidth, 2011.

[111] Sudhakar Yerneni, Suhas Phadke, Dheeraj Bhardwaj, Subrata Chakraborty, and Richa Rastogi. Imaging subsurface geology with seismic migration on a computing cluster. *Current Science*, 88(3), February 2005.

[112] Öz Yilmaz. *Seismic Data Analysis*. Society of Exploration Geophysics, 2001.

Glossary

(R)DMA Remote Direct Memory Access

ADI Alternation Direction Implicit

AMD Advanced Micro Devices

ASIC Application Specific Integrated Circuit

AVX Advanced Vector Extensions

BEE Berkeley Emulation Engine

CACTI HP integrated cache and memory access time, cycle time, area, leakage, and dynamic power model.

CBEA Cell Broadband Engine Architecture

CDP Common-Depthpoint

CISC Complex Instruction Set Computer

CMP Common-Midpoint

CoDEx Co-Design for Exascale

COTS Commercial off-the-Shelf

CPU Central Processing Unit

CSE Common Subexpression Elimination

CUDA Compute Unified Device Architecture

DARPA Defense Advanced Research Projects Agency

DDR Double Data Rate

DFG Deutsche Forschungsgesellschaft

DIMM Dual Inline Memory Module

DoE Department of Energy

DRAM Dynamic Random Access Memory

FD(M) Finite Difference (Method)

FMA Fused Multiply-Add Instruction

FPGA Field Programmable Gate Array

FSB Front Side Bus

GB Gigabyte(10^9)

GDDR Graphics Double Data Rate

Gflops Giga floating-point operations (10^9)

GPI Global Address Space Programming Interface

GPU Graphic Processing Unit

HCA Host Channel Adapter

HDD Hard Disk Drive

HDL Hardware Description Language

HPC High Performance Computing

HT HyperTransport

I/O Input/Output to external mass-storage devices

IB Infiniband

IBM International Business Machines (company)

IP Intellectual Property

ISA Instruction Set Architecture

ISO Shortcut for the isotropic wave equation kernel

ISS Tensilica Instruction Set Simulator

ITWM Fraunhofer Institute for Industrial Mathematics

KB Kilobyte (10^3)

LBNL Lawrence Berkeley National Lab

LLNL Lawrence Livermore National Lab

LS Local Store

MADD Multiply-Add Instruction

MB Megabyte(10^6)

MHz Megahertz

MIC Many Integrated Core Architecture (Intel)

MIMD Multiple Instruction Multiple Data

MMX Multimedia Extension

MPI Message Passing Interface

MW Megawatt

NERSC National Energy Research Scientific Center in Oakland, California

NIC Network Interface

NMO Normal Moveout Correction

NoC Network-on-Chip

NUMA Non-Uniform Memory Architecture

OBC Ocean Bottom Cable

OBS Ocean Bottom Seismogram

OS Operating System

P2P Peer-to-Peer

PCI Peripheral Component Interconnect

PCIe PCI express

PDE Partial Differential Equation

PGAS Partitioned Global Address Space

PPE Power Processor Element

QDR Quad Data Rate

QPI Quick Path Interconnect

RAID Redundant Array of Inexpensive Disks

RAMP Research Accelerator for Multiple Processors

RISC Reduced Instruction Set Computer

RTL Register Transfer Logic

RTM Reverse Time Migration

SEG-Y An international standard how seismic data for seismic data

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SM Streaming Multiprocessor

SMP Symmetric Multiprocessor Architecture

SMT Simultaneous Multithreading

SNL Sandia National Lab

SoC System-on-a-Chip

SPE Synergistic Processor Element

SRAM Static Random Access Memory

SSD Solid State Drive

SSE Streaming SIMD Extensions

STI Sony, Toshiba, IBM

TB Terabyte(10^{12})

TCO Total Costs of Ownership

TI Transverse Isotropy

TIE Tensilica Instruction Extension

TLB Translation Lookaside Buffer

TTI Tilted Transverse Isotropy

UPC Unified Parallel C

VLIW Very Long Instruction Word

VTI Vertical Transverse Isotropy

XPG Tensilica Xtensa Processor Generator Toolchain

XTSC Xtensa SystemC