Alma Mater Studiorum - University of Bologna

Advanced Research Center on Electronic Systems for Information and Communication Technologies E. De Castro

Ph.D. in Information Technology

Cycle XXIII

ING-INF/05

HETEROGENEOUS MULTICORE SYSTEMS FOR SIGNAL PROCESSING

Ph.D. Thesis by:
Florian Ries

Ph.D. Coordinator:
Chiar.mo Prof. Ing. Claudio Fiegna

Ph.D. Tutors:
Chiar.mo Prof. Ing. Roberto Guerrieri

Chiar.ma Prof.ssa Ing. Eleonora Franchi

Submission date: 15.03.2011

Hereby I testify that I did this Ph.D. Thesis independently and without accessing unauthorized help. Apart from the ones mentioned in this document, no further resources have been used.

Bologna, 15.02.2011

Florian Ries

Abstract

This thesis explores the capabilities of heterogeneous multi-core systems, based on multiple Graphics Processing Units (GPUs) in a standard desktop framework. Multi-GPU accelerated desk side computers are an appealing alternative to other high performance computing (HPC) systems: being composed of commodity hardware components fabricated in large quantities, their price-performance ratio is unparalleled in the world of high performance computing. Essentially bringing “supercomputing to the masses”, this opens up new possibilities for application fields where investing in HPC resources had been considered unfeasible before. One of these is the field of bioelectrical imaging, a class of medical imaging technologies that occupy a low-cost niche next to million-dollar systems like functional Magnetic Resonance Imaging (fMRI). In the scope of this work, several computational challenges encountered in bioelectrical imaging are tackled with this new kind of computing resource, striving to help these methods approach their true potential.

Specifically, the following main contributions were made: Firstly, a novel dual-GPU implementation of parallel triangular matrix inversion (TMI) is presented, addressing an crucial kernel in computation of multi-mesh head models of encephalographic (EEG) source localization. This includes not only a highly efficient implementation of the routine itself achieving excellent speedups versus an optimized CPU implementation, but also a novel GPU-friendly compressed storage scheme for triangular matrices.

Secondly, a scalable multi-GPU solver for non-hermitian linear systems was implemented. It is integrated into a simulation environment for electrical impedance tomography (EIT) that requires frequent solution of complex systems with millions of unknowns, a task that this solution can perform within seconds. In terms of computational throughput, it outperforms not only an highly optimized multi-CPU reference, but related GPU-based work as well.

Finally, a GPU-accelerated graphical EEG real-time source localization software was implemented. Thanks to acceleration, it can meet real-time requirements in unpreceeded anatomical detail running more complex localization algorithms. Additionally, a novel implementation to extract anatomical priors from static Magnetic Resonance (MR) scansions has been included.

Acknowledgements

It is a pleasure to thank those who made this thesis possible; most of all, I am heartily thankful to my supervisor, Professor Roberto Guerrieri, who always had an open door to provide guidance and advice. I am also grateful to my parents and my girlfriend for their continuous support, and last not least I would like to thank Antonio Deledda, without whom I would probably never have gone to Bologna in the first place.

Contents

List of Figures

List of Tables

1 Introduction

No one will argue that multicore processing is the future of computing. But the inevitable paradigm shift also brought new challenges, both to hardware designers and application programmers. Parallel applications have very different needs towards the processing system. When designing the architecture around a particular target problem, excellent results can be obtained, but the ability to efficiently process different types of problems should be the foremost objective in designing a computing system. However, so far no "one size fits all" architecture has been found, and many are convinced that there will never be any.

In the mainstream, the quest for a universal parallel computer lead to multicore CPUs, which preserve the highest versatility. They devote a lot of silicon space on non-arithmetic logic, and excel at control-intensive tasks. On the other hand, they perform poorly on tasks that require a lot of arithmetic, but little or no flow control. Of course, just scaling up the number of these devices does work to improve performance, and is a common approach in traditional supercomputing. However, the control capabilities are wasted on many compute-intensive parallel problems and with them money and energy: often a more adequate architecture, like a streaming vector processor, could have achieved the same performance using substantially less silicon space.

This dilemma has led to increasing popularity of heterogeneous approaches. Heterogeneity in parallel processing allows exploiting the raw computational power of massively parallel vector processors while maintaining the flow control and random access capabilities of general purpose processors by combining architectures of both types. Although more challenging to program, heterogeneous systems usually beat homogeneous alternatives in terms of price/performance ratio, as well as in terms of relative energy consumption.

Of this class of systems, the multicore desktop machine accelerated by one or multiple Graphics Processing Units (GPUs) is a particularly promising base architecture, and has received a lot of attention in the last years. Originally designed purely as accelerators for graphics applications, GPUs quickly evolved into high-performance parallel coprocessors that can greatly speed up all kinds of applications, while the cost of the hardware upgrade remains modest. Recent developments show a trend to pack increasing numbers of GPUs into the same system, essentially creating small-scale supercomputers in the frame of a desktop machine. This thesis explores the capabilities and limitations of these systems, on the basis of their performance on selected real-world problems.

As a target application, this thesis focuses on computational challenges that limit the capabilities of bioelectrical technologies in medical imaging. These methods rely on electrical sensors to gather information about characteristics or activity of the body part under examination by measuring differences in electrical quantities between these electrodes. This work concentrates on two particular technologies: electroencephalography (EEG), which measures potentials on the scalp to detect electrical activity in the brain and the electro impedance tomography (EIT), which actively injects small currents between pairs of electrodes to gather information from the measured impedances. As opposed to other imaging

methods like functional magnet resonance imaging (fMRI) or positron emission tomography (PET), these bioelectrical methods have a distinct benefit: the equipment is very inexpensive by comparison. Work in this field therefore promises to improve medical standards worldwide including rather ill-developed regions, rather than just in a few high-tech hospitals. Unfortunately, bioelectrical methods are still outperformed by competing technologies, because they produce data that does not allow a similarly straightforward interpretation. Aside from the challenges of electrode design, retrieving meaningful information from bioelectrical measurements is first and foremost a computational problem. Starting from discrete measurements on the scalp, the propagation of the electrical quantities needs to be traced back to the regions of interest, which are often remote from the measurement points. Given the complexity of live tissue as a volume conductor, this requires an elevated amount of data elaboration.

Constrained by computing resources, contemporary solutions often use oversimplified models, and the lack of anatomical detail limits the capabilities of bioelectrical technologies. It is the ambition of this thesis to push these limits, but without thwarting the economical benefits by introducing specialized supercomputers. Instead, the architectures used in this work promise to be much more affordable, but still powerful enough to pave the way to new levels of quality in bioelectrical imaging.

This work is structured as follows: Chapter 2 covers the heterogeneous multicore systems employed in this work, starting with a short introduction on multicore, parallel computing and the heterogeneous approach. After describing these basics, the concept of heterogeneous CPU/GPU systems is described in detail, comprising system setup, GPU architectures, software hierarchy and the GPU programming model.

Chapter 3 provides a brief outline of the field of bioelectrical imaging as far as it concerns this work, with emphasis on the EEG and EIT technologies, the principles of brain functional localization and the relation of forward and inverse problem. This chapter is meant to introduce the background of the applications that will be described in the ensuing chapters, highlighting prominent computational problems in this field, where application of the previously described architectures promises significant advantages.

The ensuing three chapters then cover the major contributions of this work, addressing computational challenges in bioelectrical imaging that where highlighted in the previous chapter. Chapters 4 and 5 treat very general operations as produced by (but not restricted to) the EEG and EIT forward problems, while Chapter 6 is more specific to real-time EEG source imaging, comprising solution of the inverse problem.

More precisely, Chapter 4 presents a dual-GPU accelerated parallel triangular matrix inversion routine, addressing a predominant computational bottleneck in the EEG forward problem, while Chapter 5 presents a multi-GPU accelerated solver for non-hermitian linear system forming an integral part in an EIT simulation environment. Finally, Chapter 6 introduces cudaEEG, a GPU-accelerated EEG source localization software. This implementation explores not only different algorithms to solve the inverse problem, but also several methods to extract anatomical priors from MRI. Moreover, it uses the GPU both for

computation and in-place graphics rendering of the results, greatly alleviating traffic on the system infrastructure.

The results obtained in this work are discussed in Chapter 7, while Chapter 8 concludes this work.

2 Heterogeneous multicore systems

The scope of this chapter is to provide a clear outline of the hardware architecture and software framework on which this work is based on, starting with an general introduction to parallel and heterogeneous processing. From there, the focus will be put on important details of the particular class of systems concerned in this work, namely multi-GPU accelerated desktop machines.

2.1 The Multicore Revolution

Starting with invention of the integrated circuit in 1958 and shortly thereafter of the first microprocessor in 1968 [46], computing has been a rapidly evolving field. A central factor of this development has been the progress in system integration density, i.e. miniaturizing transistors to put more on them on a smaller area, while maintaining or even reducing manufacturing costs. So far, the historic trend in integration density has confirmed Gordon Moore's famous prediction from 1965, saying that this number would double every two years [47]. The law holds for over half a century now and is expected to end not before 2015 [48]. The historic trend with extrapolation to the close future is depicted in Figure 1.

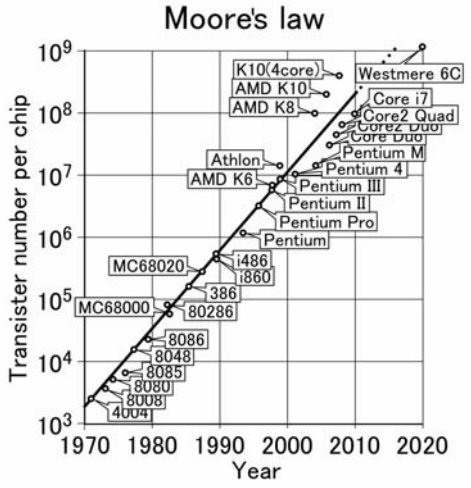


Figure 1: Transistor count per chip, for common microprocessors over time (source: Wikicommons).

There is a close correlation between integration density and computing power, since smaller feature sizes allow faster clock rates, which in turn are linearly connected to instruction throughput. For many years, frequency scaling has been the predominant way of increasing performance [49]. It was so effective that there was no motivation to abandon the traditional serial processor approach, around which the whole software world has established itself so conveniently. However, about ten years ago, the frequency boom started to slow down as it was facing increasingly rigid physical constraints. The predominant obstacle is power density, which is linearly related to frequency. With standard cooling, power density in modern chips is already dangerously close to what silicon can withstand. Pushing the limit further would require more powerful cooling techniques, which are economically impracticable, or switch to new materials, which are currently not within sight.

At the same time, processor manufacturers strived to augment performance by other means which depended on increasing the transistor count alone, while conceding the inflation of clock rate and even taking a small step backwards. The result were elaborate pipelines, huge hardware controlled caches and complex logic for instruction-level parallelism [56][55]. But not only does it prove increasingly difficult to get additional performance gain from these techniques, they also greatly increase the chip's die size. At a certain point, the gain no longer justifies the increase development cost and rejects in manufacturing.

In the end, chip manufacturers accepted that the only room left to grow is multi-core parallelism. To date, roadmaps from all major vendors actively embrace this paradigm shift, and there is throughout consensus that multi-core is the future of computing [59]. Present general purpose processors typically integrate two to eight cores [53][57], while projections suggest that future technologies will allow single-chip integration of many more cores, up to hundreds or thousands [54].

2.2 Parallel computing

Meanwhile, the mainstream software world, used to decades of serial computing, has been a bit slower in catching up with the new trend. Rather than to the real limits of parallel computing, this is owed to a certain amount of inertia: parallel programs are more difficult to write, and programmers tend to have less experience in this field. But in fact, many applications do offer a lot of concurrency. What is more, the real world itself is massively parallel. For that reason, parallel computing has always been way more advanced in scientific applications than in other fields [61].

The theoretical argument for parallelism is very compelling: multiply the number of processors and you multiply performance. Practically, however, this promise is almost impossible to fulfill, as parallel processing is subject to several inherent limiting factors. The first is hardware independent and directly connected to the application itself: according to Amdahl's law, the maximal speedup from parallel execution is inherently limited by the fraction of the program that is parallelizable, following the formula

$$S = 1 / ((1 - P) + P / N)$$

where S is the speedup, P the fraction of parallelizable code and N the number of processing elements [58]. The speedup scaling with the number of processors according to Amdahl's law is visualized in Figure 2.

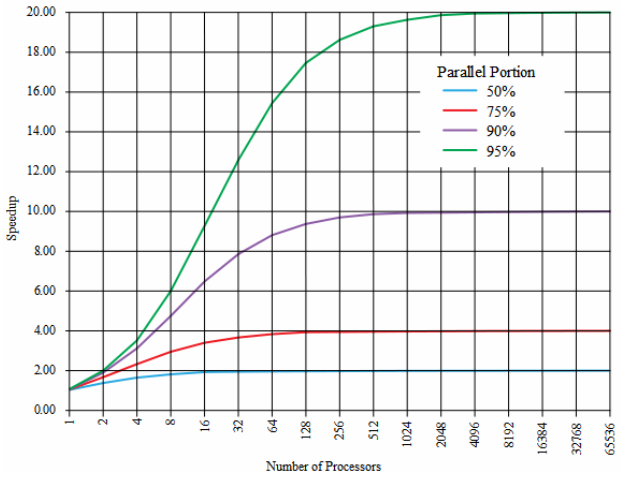


Figure 2: Amdahl's law of parallel computing (source: Wikicommons).

Note that this assumes that the parallel sections offer at least as many independent instruction streams as there are arithmetic units. Few application offer optimal performance scaling with the number of processors; this usually implies that the instruction streams are very independent and require very little or no interaction. Applications that meet these conditions are commonly referred to as “embarrassingly parallel”.

However, most applications do require at least a minimum of communication between the processing elements, which leads to the second limiting factor in parallel computing. Data traffic and synchronization between processing nodes can be quite expensive, depending on the underlying system and especially the memory architecture. Parallel computer memory architectures are generally differentiated between shared and distributed memory. The two models can also be combined into hybrid solutions, as shown in Figure 3.

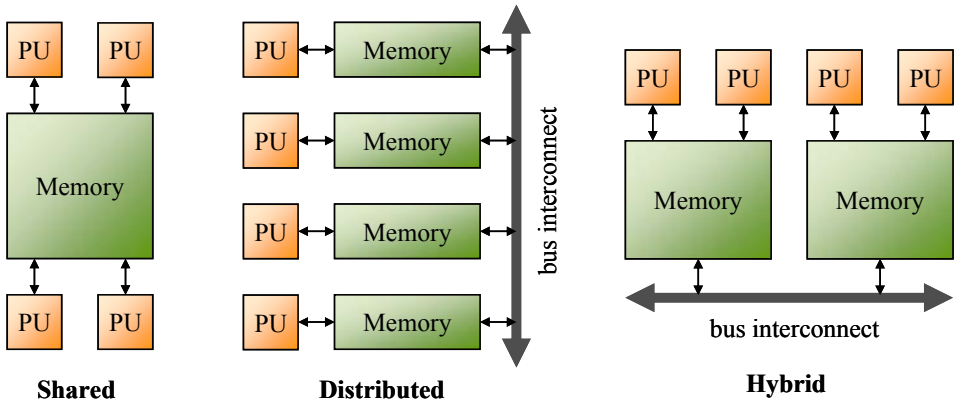


Figure 3: Parallel computer memory architectures.

Both models have their advantages and disadvantages. Shared memory provides the quickest communication between processing units, and multiple processing streams might share one instance of the same base data. On the downside, shared memory does not scale well with the number of processing units attached to it, as data traffic and competition for resources can scale up geometrically with this number. It also raises the need to make shared memory larger. Larger shared memory is harder to put close to the processing units, which results in

either higher design and production cost or slower memory access; both put severe limits on the feasibility of increasing the number of processors.

In the distributed memory model, each processor has its own local memory, which is connected to other instances via some communication infrastructure. This ensures fast local memory access and stable design and production costs in scaling, but all communication has to pass through the interconnect network, which will become quickly the bottleneck for most applications. What is more, when processors are working on the same base data, each of them needs to be supplied with its own copy, unnecessarily increasing memory consumption and I/O. Which solution performs better is highly application-specific, for that reason the most successful parallel computers employ a hybrid mix of shared and distributed memory architectures.

The considerations about memory architecture already indicate that there is no "one size fits all" solution. Applications can display different forms of parallelizability, and accordingly they have different demands towards the processing system. This also applies to design of the arithmetic processing arrays. According to Flynn's Taxonomy [63], computer architectures can be classified into four sub-groups:

- Single Instruction Single Data (SISD)
- Multiple Instruction Single Data (MISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Multiple Data (MIMD)

The general principles of all four are displayed in Figure 4:

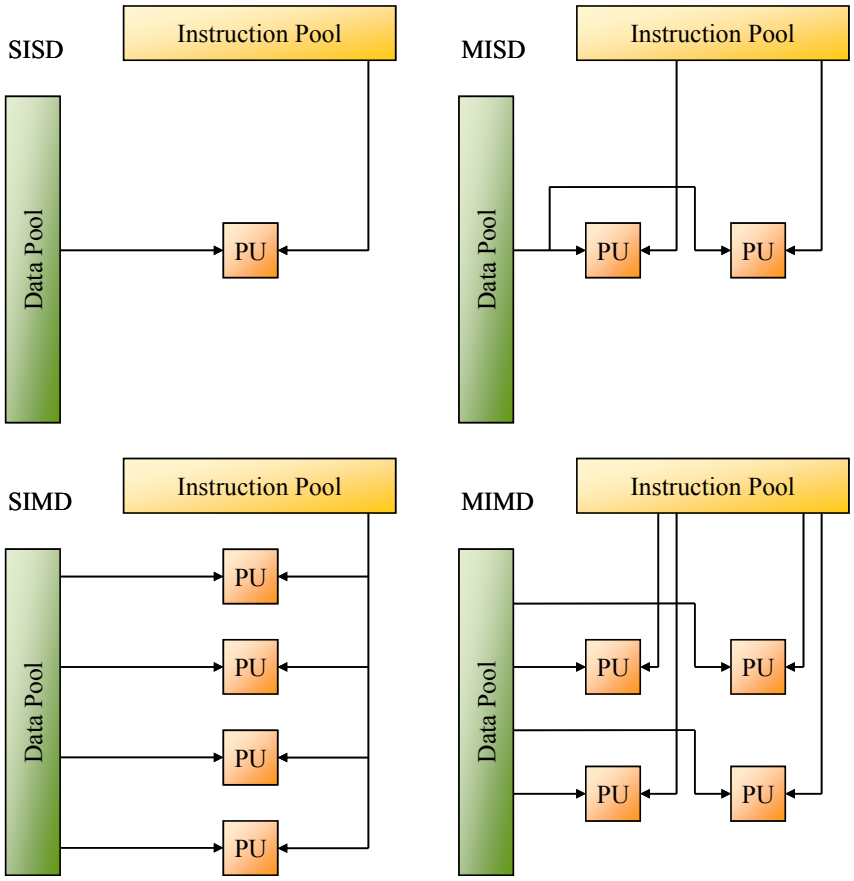


Figure 4: Flynn's Taxonomy of computer architectures with one or more Processing Units (PU) (source: Wikicommons).

SISD describes the legacy serial processor and MISD is a very rare case with few actual example architectures. The classes of interest in parallel programming are SIMD and MIMD. Multi-core general purpose processors, for example, fall into the MIMD category. They are especially appealing if one wants to perform parallel execution on a task level, i.e. by having the cores concurrently process independent serial sub-programs (s. Figure 5).

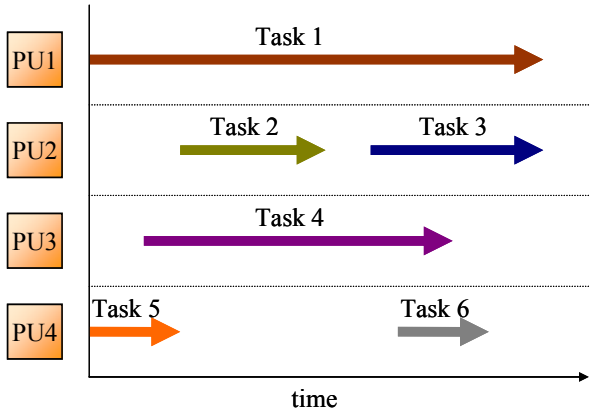


Figure 5: Task parallelism in a MIMD machine.

Each processing unit follows its own execution path, which might perform be very different operations. Classical challenges in task-parallel programming include prioritizing access to resources and finding optimal, load-balanced schedules for parallel execution.

Task-parallel processing on MIMD architectures is a very useful approach to control different, if possible independent parts of a system, like running office application in a desktop PC or handling different peripheral devices. The number of concurrent processes in the aforementioned applications rarely exceeds a few dozens, so there is usually no point in having more than a couple of execution units. However, when the goal is data intensive processing, more often than not the problem can be spit into a large number of concurrent instruction streams which perform the same basic operations on different data sets (s. Figure 6). Applications of this type comprise graphics processing, linear algebra and many simulation environments.

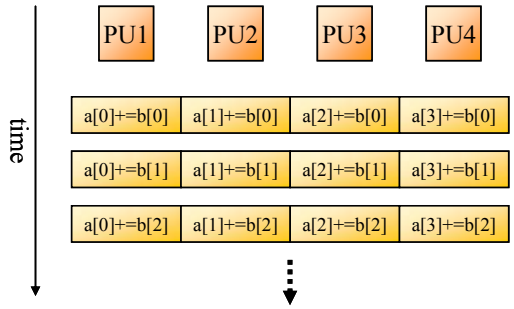


Figure 6: Parallel execution of a SIMD program.

The capabilities of general purpose multicore processors to handle multiple execution paths are basically wasted on such applications; on the other hand, there are often thousands of parallel streams with little or no divergence. This obvious mismatch led to the development of SIMD-type architectures which often operate in streaming [64], both in the form of massively parallel vector processors [65] and SIMD instruction set extensions for CPUs [66]. These architectures are not limited by instruction rate issues like conventional processors, and can efficiently pipeline operations on large streams of data. Note that MIMD architectures still can process SIMD code, although not as efficiently as could be, while streaming SIMD architectures are restricted to a small subset of problems. For that reason, pure SIMD processors are often designed as application specific co-processors.

2.3 The heterogeneous approach

Due to the high application dependency of parallel performance, no universal multicore solution has been found so far, and many are convinced that there will never be any. One might consider the closest thing to this being modern multi-core CPUs as they offer the highest versatility. But exactly this versatility constraints their design to dedicate the major part of their transistors on non-computational parts like logic and cache, leaving little resources to actual number-crunching. In many modern supercomputers, where neither money nor energy are on a very strict budget, high performance is obtained by simply scaling up the number of these chips to the hundreds and thousands and clustering them together using a high-speed interconnect.

In most environments, however, price-performance and/or power-performance ratios are of primary concern. Of course, if it is exactly known what the application looks like, the architecture can be built around it with excellent results. However, the ability to process

different applications should be considered the primary purpose of computer architecture design. For that reason, heterogeneous architectures that combine traditional general-purpose processors with application-specific accelerator cores have become a popular alternative. Accelerators are designed to maximize throughput for a specific compute-intensive task (e.g. a large-scale SIMD operation), given a certain transistor and power budget. All control logic not immediately necessary for this task is cut down. This makes them incapable of exerting any control of their own. The controlling processor provides input data and instructions, and then collects the results from the accelerators output. Figure 7 shows the general design of a system employing multiple accelerators, controlled by a versatile main processor.

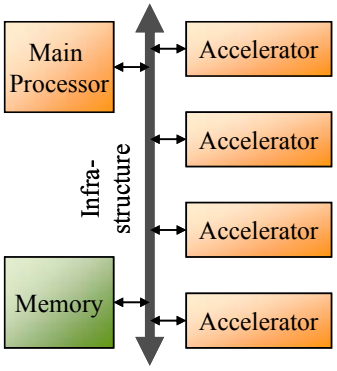


Figure 7: Heterogeneous system with accelerators, controlled by a main processor.

Recent trends show that accelerators are becoming extremely successful in parallel computing. Some designs, like modern cell-phone processors, take the heterogeneous approach very far by integrating a highly specific accelerator for all major operations within the chips functionality scope [68]. But by far the most common approach is to combine one (or very few) general purpose processors with a large, uniform, often massively parallel array of programmable throughput-oriented co-processors [51]. Accelerators may be on-chip like in the IBM Cell Broadband Engine (BE) [69] and many System-on-Chips (SoC), or off-chip, like Field Programmable Gate Arrays (FPGA) and Graphics Processing Units (GPU). As an first example of typical heterogeneous design architecture, Figure 8 presents a simplified architectural overview over the Cell BE.

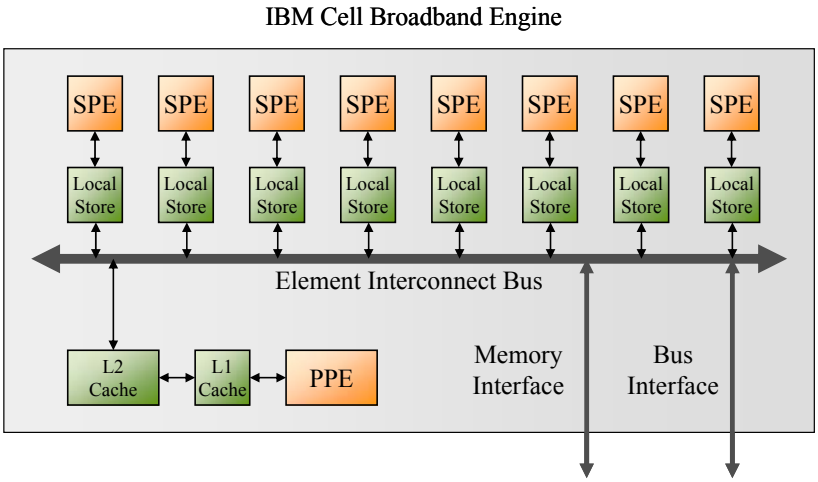


Figure 8: The IBM Cell Broadband Engine with eight Synergistic Processing Elements (SPE), controlled by a single Power Processor Element (PPE).

The Cell BE is used in many multimedia applications [70]. It received attention as central processor of the PlayStation 3 video game console [71] and as a basic building block in the IBM Roadrunner supercomputer [72].

FPGAs are just an example for reconfigurable devices being used as computational accelerators [74][75], but it is the only accelerator of that type that has a significant market share. FPGAs are arrays of logic gates that can be hardware-programmed to resemble any logic circuit. In this way, they can be configured to devise one or more highly specific accelerator units, which might be very efficient for these tasks. A system using a FPGA as accelerator is depicted in Figure 9.

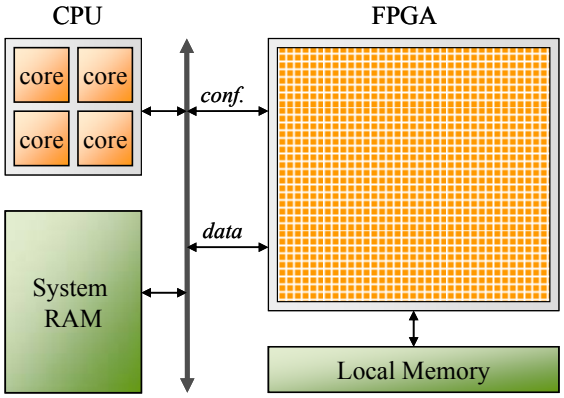


Figure 9: FPGA accelerator with dedicated local memory. The device is connected to the host system via a configuration and a data interface.

FPGAs are very versatile but hard to program, which is usually done using hardware description languages. They are rarely used for heavy floating point arithmetic, but excellent results have been reported for pattern matching [77], encryption [76] and signal processing applications [78]. Leading vendors of FPGA accelerator boards are Nallatech, DRC and Pico, while Xilinx and Altera are leaders in FPGA design.

However, the success stories of the Cell BE and FPGA pale compared to that of the GPU, which some call the “king of accelerators”. GPUs are massively parallel coprocessors that

excel at heavy, SIMD-like floating point arithmetic. They are integrated on a graphics card with their own memory. The graphics card is connected to the host system, traditionally a standard PC main board, via a high-speed peripheral interconnect port. Figure 10 shows the general architecture of a GPU-accelerated PC system.

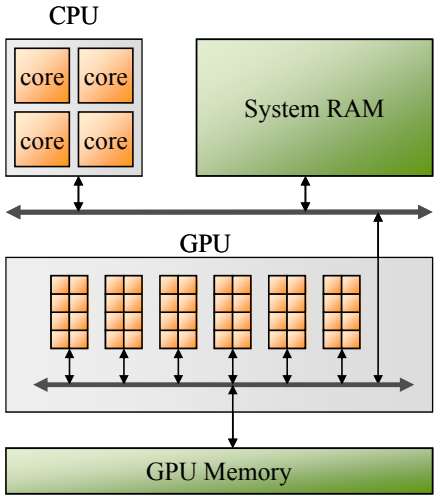


Figure 10: GPU in a standard PC environment.

Presently, there are only two relevant GPU vendors: ATI (now part of AMD) and NVIDIA. In this work, only GPUs from NVIDIA were used, which currently lead the market of computational accelerators [80].

In High Performance Computing (HPC), heterogeneous systems with accelerators are beginning to displace traditional supercomputers, and many think that accelerators will pave the road to exascale computing [79]. Particularly GPUs are enjoying resounding success in this development: at the time of writing, three of the five strongest supercomputers on earth were using these devices.

The remarkable floating point capabilities of GPUs paired with their affordable price make GPUs the perfect choice for our target application field (s. Chapter 3): all implementations presented in this thesis are based on desktop machines accelerated by one or more GPUs. The following Chapter 2.4 will provide a detailed overview over the GPU, its history, its employment in scientific applications and its general architecture. The Chapters 2.5 and 2.6 will discuss the challenge of multi-GPU integration and the programming framework of heterogeneous multi-CPU/multi-GPU systems.

2.4 The Graphics Processing Unit

The GPU as computational accelerator is of central importance in this thesis. Therefore, this additional chapter is devoted to discuss the history, applications and architectural details of these devices.

2.4.1 History

The GPU was originally devised in the 1990s as a hardware accelerator for 3D applications. Graphics rendering, e.g. computing the color values of all pixels on the screen from a 3D

scene description, involves a lot of compute intensive, massively parallel low precision floating point operations, with very moderate need for data caching and flow control. This is quite the opposite of what a traditional CPU is good at. When graphics on desktop machines started to become of economic interest, particularly in the form of video games, these graphics accelerators were introduced to satisfy the market's demands for better, fancier visuals.

As input, the graphics card receives only a description of the 3D scene, and then uses the GPU to render it and store the result in a frame buffer, from which the screen content is being updated. The principle is outlined in Figure 11.

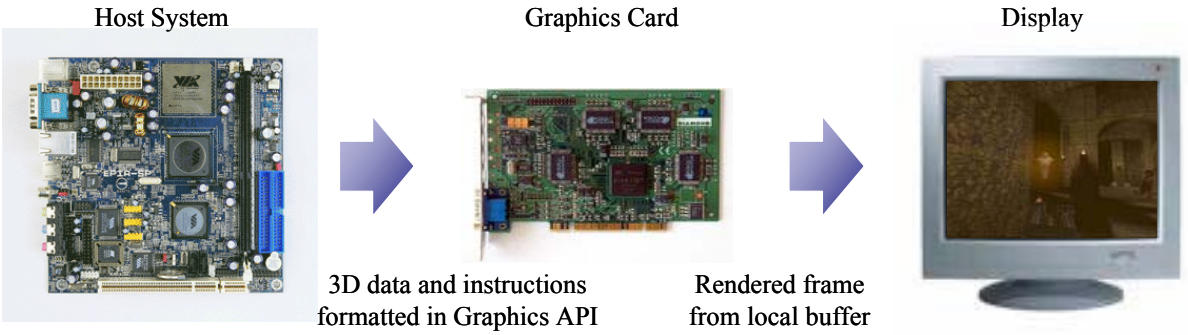


Figure 11: A description of the scene is passed to the graphics card in form of a graphics Application Programming Interface (API). The screen image is rendered by the GPU and stored in the local on-board frame buffer, from which the display is updated.

Graphics rendering consists of a straight sequence of few well-defined steps commonly referred to as the rendering pipeline:

1. Vertex processing: 3D coordinates (called vertices) from the scene description are translated into 2D coordinates on the screen. These vertices define triangulated surfaces which assemble to polygons (three-dimensional objects).
2. Rasterization: The triangles are scan-converted to fragments of the physical pixel grid of the screen
3. Fragment processing: The color value of all pixels in all fragments is computed taking into account lighting, texture etc.
4. Composition: The fragments are assembled together to the final image.

Soon, the boom of the video game industry piled economic pressure on GPU development, leading to an unparalleled evolution of these devices. Initially, GPUs were little more than hardwired ASICs (Application Specific Integrated Circuits) implementing the rendering pipeline. Around the year 2000, parts of the pipeline were replaced by programmable processor arrays (s. Figure 12). Now small programs (called "shaders") could be executed to modify vertex and pixel data, allowing more spectacular graphics effects.

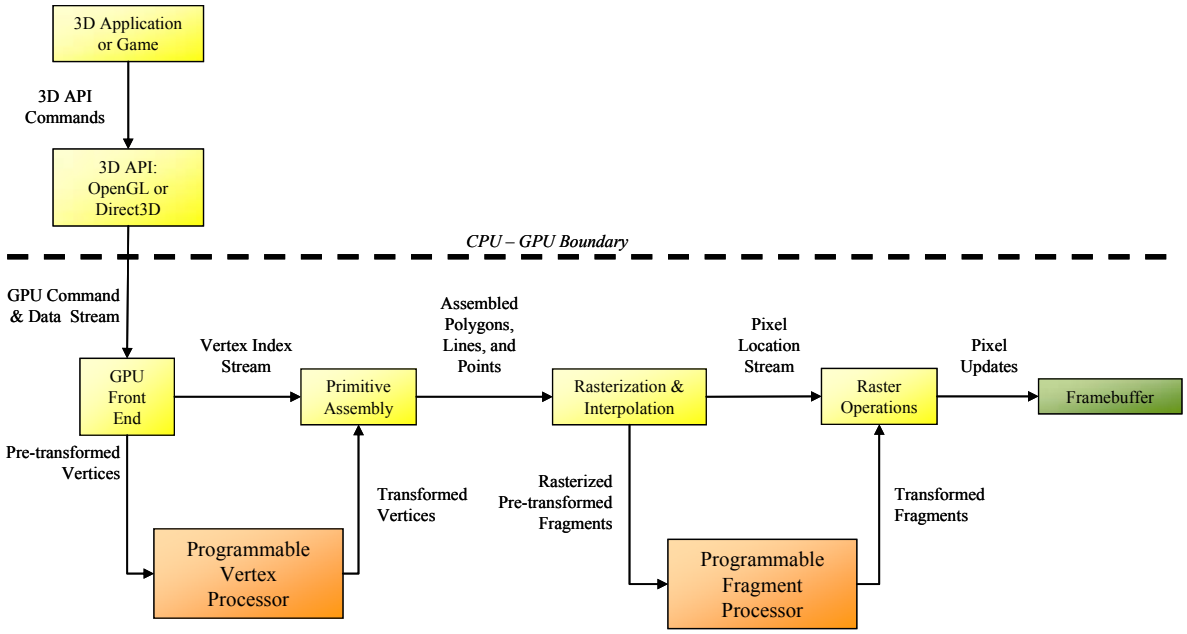


Figure 12: Legacy GPU with programmable vertex and fragment (pixel) processor arrays.

As of 2006, the processor arrays for vertices and pixels were merged into a unified shader model, abolishing the rigid rendering pipeline of earlier generations. The motivation behind this was to avoid bottlenecks through display of very vertex or very pixel intensive scenes, and to once again broaden the range of graphics effects than can be generated. However, this design decision had consequences reaching far beyond of the original scope. It basically turned the GPU into a high-throughput programmable many-core processing unit, and suddenly multiplied the devices capabilities for non-graphics applications as well.

With increasing programmability of GPU arithmetic, more and more people tried to program them to do other things than graphics (s Chapter 2.4.2); this is when the buzzword GPGPU (General Purpose GPU) was coined. Although the trend was started by independent tinkerers, GPU vendors soon embraced the new concept seeing a chance to broaden the target audience of their products. This led not only to the release of GPGPU-friendly application programming interfaces like ATI's CTM (Close to Metal, now called "Stream") [86] and NVIDIA's CUDA (Compute Unified Device Architecture) [87]. It also motivated inclusion of hardware features that do not make any sense for graphics rendering. These include, in chronological order:

- Faster read-back from GPU memory
- Double precision support
- Fully IEEE-compliant floating point arithmetic
- ECC error correction
- Hardware-controlled caching

Today, GPUs are ubiquitous, cheap and powerful. In terms of raw floating point operations per second (FLOPs), they have long left serial computers behind, with newer devices entering

the teraflop range; the evolution of both architectures in terms of GFLOP/s (billion floating point operation per second) is shown in Figure 13.

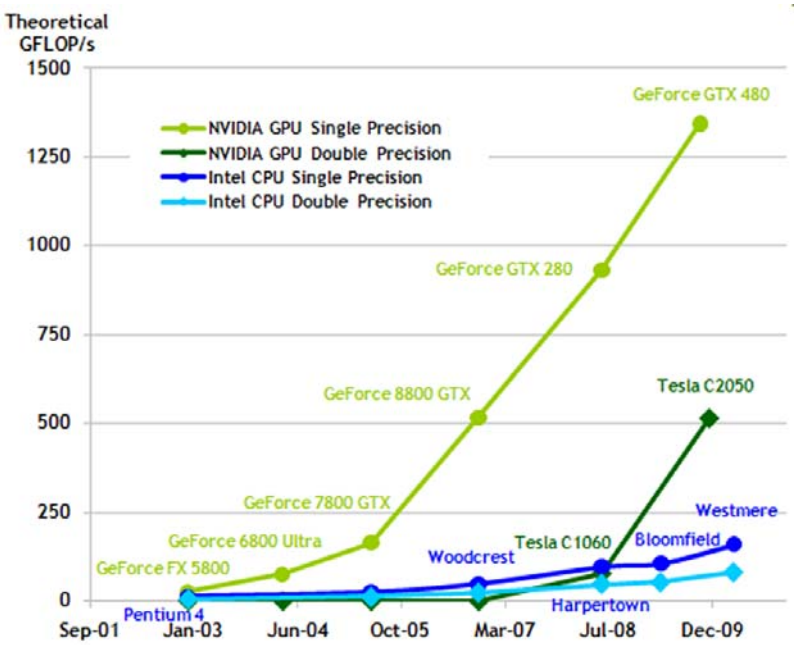


Figure 13: Evolution of Intel CPUs and NVIDIA GPUs in terms of computational throughput. Source: NVIDIA.

Given the present state of the art in graphics hardware, it almost seems out of place to longer speak of graphics accelerators that can be also used for computation. Rather, these devices are computational accelerators that can also be used for graphics, with capabilities far exceeding their original purpose. This goes so far that there are graphics cards that do not even have a graphics output, like the NVIDIA Tesla series [85].

2.4.2 GPU in scientific applications

First attempts to use GPUs for non-graphics applications were made even before introduction of the unified shader model [90]. By casting both input data and computational instructions in the format of a graphics API, the fragment processors of early GPUs could be "tricked" into performing simple parallel algorithms like simulation of cloud or fluid dynamics [81][84], band system solvers [82] and basic linear algebra operators [83][89].

While these implementations already produced encouraging results, the programming was too cumbersome to allow widespread application of GPGPU, and the graphics-tuned architecture of these GPUs severely limited performance for most applications. However, the introduction of unified shaders and computation-oriented software design kits initiated a downright "gold rush" in the GPGPU field, with many new applications reporting speedups over CPUs in the order of tens to hundred times. Basically, any application involving highly parallel arithmetic-intensive floating point computations can greatly benefit from GPU acceleration. Examples thereof cover

- Dense linear algebra [93][94][95]
- Sparse linear system solvers [96][97][98][99][100]

- Computational fluid and molecular dynamics [103][104][105][106]
- Bioinformatics [107][108][109][110]
- Image processing [111][112][113]
- Signal processing [114][115][116]

just to name a few. Recently, considerable effort has also been made to include GPU acceleration in popular scientific software tools like Matlab (AccelerEyes Jacket [119] and GPU support for the Parallel Computing Toolbox [120]), Mathematica [121] and "R" [122].

Concerning related work to the contributions of this thesis, particular attention should be paid to the state of the art in GPU-accelerated dense linear algebra and sparse linear system solvers. What regards the dense triangular matrix inversion routine presented in Chapter 4, related work on factorization methods exists [166][167], but my contribution was the first to solve the inversion problem with GPU acceleration.

The non-hermitian system solver presented in Chapter 5 belongs to the category of GPU-accelerated sparse linear system solvers. The central relevance of this discipline in the field of HPC has led to a plethora of related implementations, making analysis of - and comparison to - the state of the art much a more extensive issue. In-depth discussion of related work will be provided in Chapter 5.2, while comparative analysis in Chapters 5.5.4 and 5.5.5 will demonstrate superiority over existing solutions.

Chapter 6 presents a very application-specific implementation containing parts from image and signal processing, but to the best of my knowledge no other GPU-accelerated implementation is similar enough to be considered as related work.

2.4.3 Architectural overview

The GPU is a massively parallel computational unit designed for maximum throughput, as required by graphics rendering. Compared to general purpose processors, most of the logic is devoted to arithmetic units rather than caching and flow control. All GPUs used in this work are based on the unified shader model (s. Chapter 2.4.1). A large part of the architecture consists of SIMD-type arrays of stream processors (Stream Multiprocessors, SMs), which are packed together in Texture Processing Clusters (TPCs). The chip has access to the host system's north bridge via a high-speed PCI express interface, as well as to a dedicated off-chip DRAM memory on the graphics board (s. Chapter 2.5.1). There is some control logic to distribute the work load of vertex and pixel shader programs to the TPCs, in case the GPU is actually used for graphics rendering. Computational applications are handled by an additional work distribution scheduler. The general GPU architecture of the NVIDIA Series 8, 9 and 200 is displayed in Figure 14.

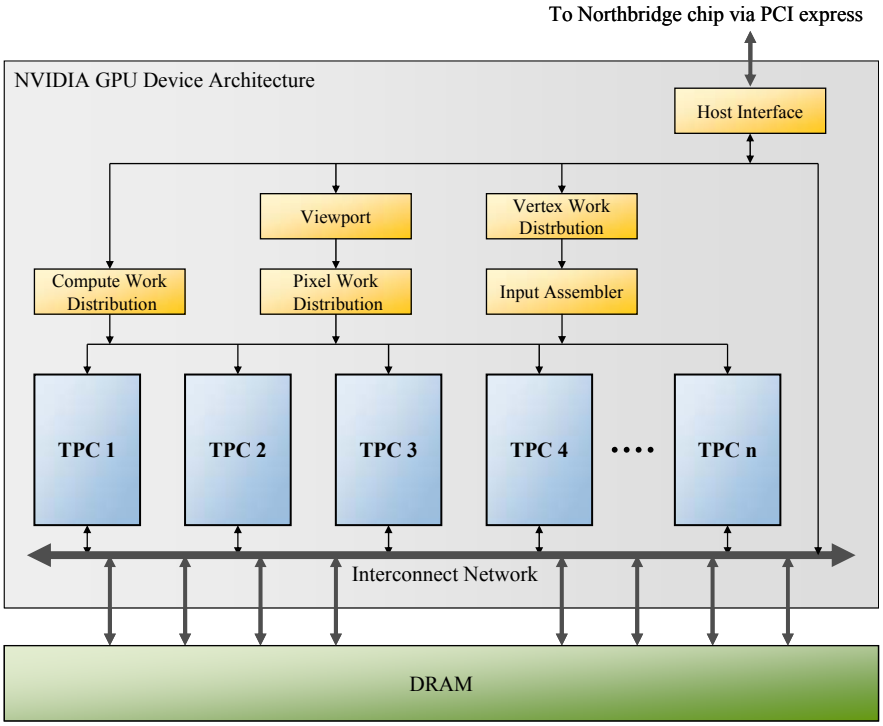


Figure 14: GPU general architecture.

The interface to the off-chip DRAM is an obvious bottleneck, so considerable resources are devoted to maximize bandwidth. Depending on the GPU model, 256-512 parallel lanes go out to the memory banks. Memory itself is based on the Graphics Double Data Rate (GDDR) design, which is tailored to GPUs requirements.

Number and architecture of the TPCs depend on the individual GPU model: in GeForce series 8 and 9 GPUs, each TPC integrates two stream multiprocessors, each of which contains eight 32-bit floating point stream processors (SP) and two Special Function Units (SFU). Next to a small instruction and constant cache, each SM shares 16kB of fast shared memory. SPs are optimized for single precision floating point arithmetic; integer and logic operations are possible but may take more clock cycles. Additional control logic includes a Texture Unit (TU) and "Multi-Thread Instruction Fetch & Issue" logic (MTIFI). In the series 200, there are three per TPC, and each SM is additionally equipped with one double precision unit (DP). Each cluster has a shared Geometry Control (GC) and Stream Multiprocessor Control (SMC) [85]. Both TPC variants are shown in Figure 14.

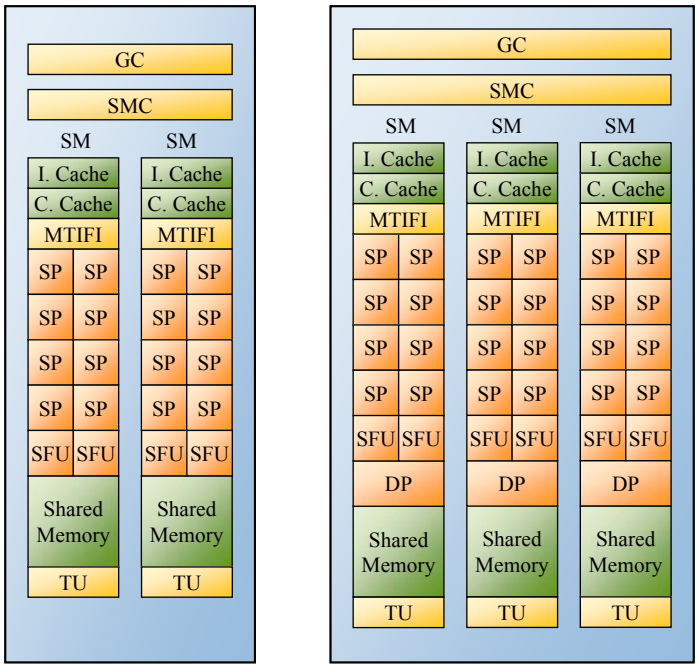


Figure 15: Texture Processing Cluster (TPC) variants of the NVIDIA GeForce series 8 and 9 (left) and the NVIDIA GeForce series 200 (right).

This general architecture applies to most of the devices used in this work. The latest generation of NVIDIA GPUs, however, employs a completely revised GPU design. The architecture of the GeForce 400 series (codename: Fermi) is straightforwardly focused on computation rather than graphics rendering. The TPC clustering is abandoned; instead the chip is covered by a uniform array of much larger Stream Multiprocessors. A Fermi SM contains 32 Stream Processors and four Special Function Units. The SPs are different too: they perform fully IEEE-compliant floating point arithmetic and can issue one integer operation per cycle. There are no double precision units, rather are joint pairs of two SPs used to perform double precision. This raises the double precision throughput considerably. The shared memory is much larger with 64kB, and has hardware controlled caching. One might also note the lack of texture units and other hardwired graphics-legacy logic, a clear statement of the intended primary use of these devices. Work load distribution is now handled by a single uniform scheduler called “GigaThreads” [124]. An overview of the Fermi architecture is displayed in Figure 16. Further details remain undisclosed by NVIDIA.

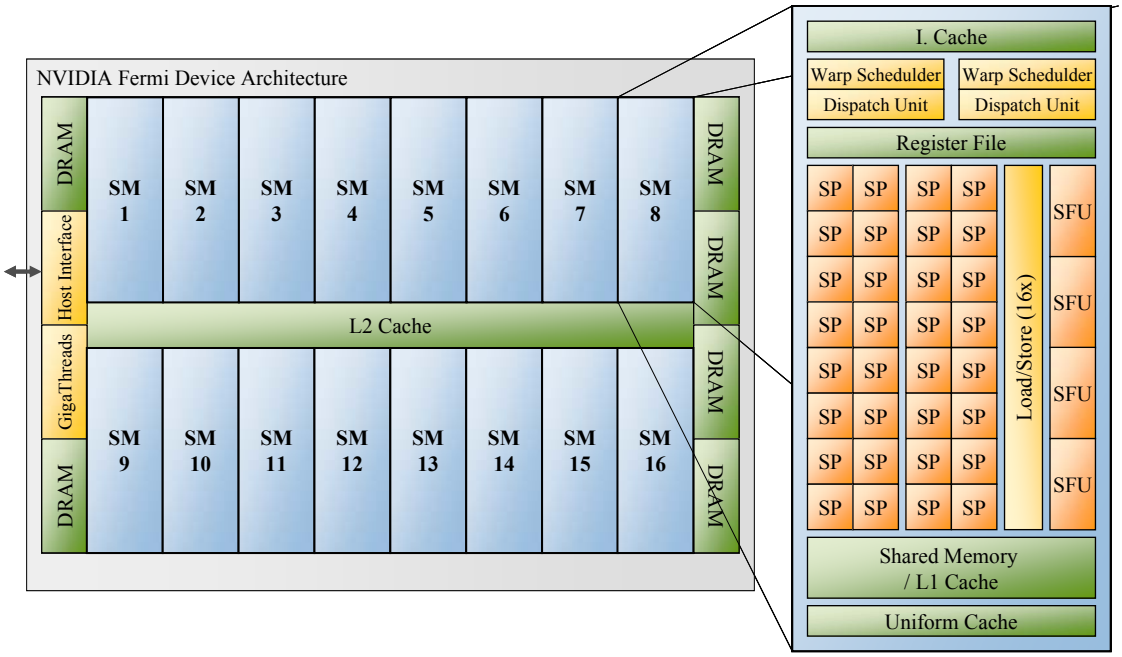


Figure 16: General architecture of the NVIDIA series 400 (Fermi) GPU.

Configurations of the GPUs used in this work are listed in Table 1. SP and DP stand for single and double precision, respectively. The GTX 295 has two identical GPUs integrated on the same graphics board, each with its own DRAM but sharing the same PCI express port to the host.

GPU Model	# TPCs	# SMs	DRAM (MB)	SP throughput (GFLOP/s)	DP throughput (GFLOP/s)	Bandwidth (GB/s)	TDP (watts)
8800 GTS	6	12	384	346	-	63	143
8800 GT	7	14	512	504	-	58	105
GTX 280	10	30	1024	933	78	142	236
GTX 295	2×10	2×30	2×896	2×894	2×75	2×113	237
GTX 480	-	15	1536	1345	672	177	250

Table 1: Parameters of GPU models used in this work.

Throughput is measured in billion floating point instructions per second. The values indicate the theoretical maximum of the device, calculated by pretending that all computational units of the corresponding level of precision are continuously busy with fused multiply-add operations. TDP is a common acronym for “thermal design power”, and represents the maximum thermal power that the processor’s cooling system is required to dissipate. All GPUs used in this work can survive by air cooling despite the discrepancies in computing power: TDP is heavily influenced by which manufacturing process is used. We see Moore’s law at work here.

2.5 Multi-CPU/multi-GPU systems

In recent development of GPGPU applications, a trend to pack multiple GPUs into the same system can be observed [128]. This is not part of the GPU's original design concept, so assembly of a multi-GPU system introduces additional challenges, which are discussed in the following.

2.5.1 GPU interfacing

Bandwidth can be considered the "gravity" of modern computer systems: handling of data traffic between key components ultimately dictates system performance. This is especially true for parallel systems. For this reason, we begin this chapter with an introduction to the GPU interfaces before moving on to the general system architecture.

When GPUs entered the mass market and started to grow in power, it soon became apparent that the traditional Peripheral Component Interconnect (PCI) [125] was way too slow to feed the graphics card's hunger for input data. In 1997 a new interface type, the Accelerated Graphics Port (AGP), was introduced und soon included in almost all successive main boards. AGP was connected directly to the main board's north bridge and provided a dedicated pathway between the slot and the processor, as well as prioritized access to system RAM. Integration of the GPU in this classic PC architecture is shown in Figure 17.

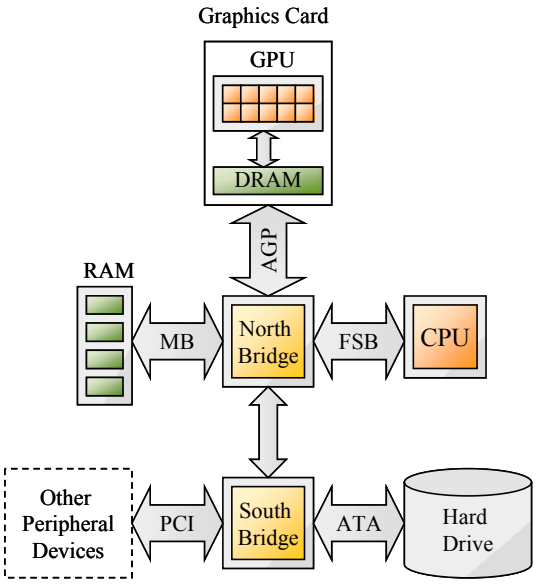


Figure 17: GPU in classic PC architecture. MB = Memory Bus, FSB = Front Side Bus, ATA = Advanced Technology Attachment.

Apart from the lack of bus contention that would occur on the PCI bus, the direct connection also allowed higher clock speeds [126]. AGP achieved speeds of up to 2 GB/s in upload, but only 256 MB/s in read-back, which strongly limited early GPGPU applications.

Starting from 2004, AGP was gradually replaced by the new PCI express interface, which abandons the arbitrated bus structure of conventional PCI in favor of a switched point-to-point connection between devices. Each participant has a direct link to the switch, which consist of 1-16 parallel lanes. Depending on the configuration one speaks of 1x, 2x, 4x, 8x or

16x PCIe. Each lane is 2-bit wide, allowing simultaneous transfer in both directions. The PCIe principle in contrast to PCI bus sharing is visualized in Figure 18.

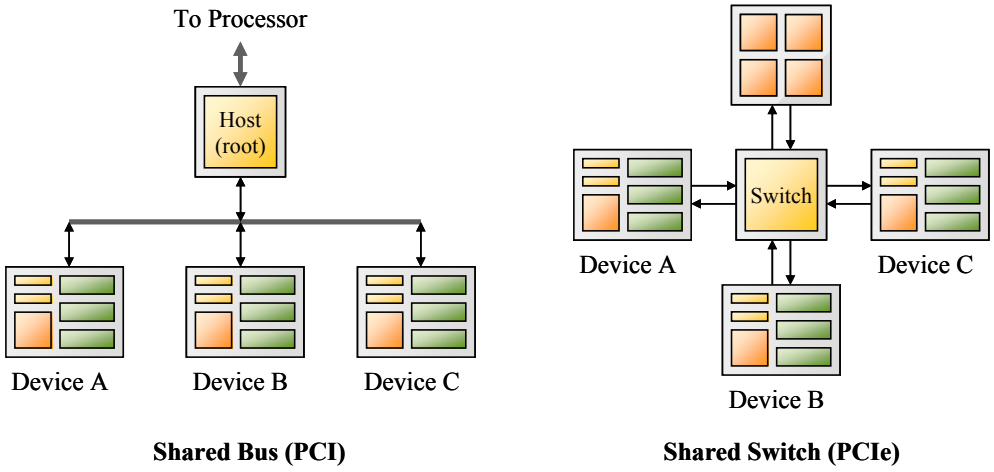


Figure 18: Conventional PCI and PCI express interconnect principles.

The PCI express technology has recently advanced to the third generation. However, few main boards and peripherals are PCIe 3.0 compliant yet. At the time of writing, 16x PCIe 2.0 is the standard interface for modern graphics cards, which will also apply for at least the upcoming generation of GPUs. Table 2 shows a comparison of interface "speeds and feeds" for past and present GPU-host interfaces.

Interface	Upstream (MB/s)	Downstream (MB/s)	2-way?
PCI 32-bit	133	133	no
PCI 64-bit	512	256	no
AGP	266	256	no
AGP x2	533	256	no
AGP x4	1066	256	no
AGP x8	2133	256	no
PCIe 1.0 x16	4096	4096	yes
PCIe 2.0 x16	8192	8192	yes

Table 2: Interface speeds in standard PC architectures.

We see that the upstream bandwidth roughly doubles with every generation, an evolution that is in tune with the increase in GPU computing power (s. Figure 13). We can also observe that read-back speed was stagnant until the paradigm shift of PCIe [127].

2.5.2 System infrastructure

When setting up a multi-GPU system, the primary concern must be data feeding of the computational units. This means that the full 16x PCI express bandwidth must be available to all graphics cards. Unfortunately, in the traditional role of the GPU, there was little need to handle more than one graphics card at the same time. After all, the user had rarely use for more than one display. Hence, most conventional main board featured only one high-speed

interface like AGP or 16x PCIe. However, the revolution of GPGPU made it appealing to distribute work load on multiple GPUs packed in the same system [128]. Main boards providing full speed for two PCIe slots are becoming increasingly common, with high end solutions supporting up to four.

Advancing the traditional PC architecture from Figure 17, we end up with the general system architecture for a desktop machine integrating multiple GPUs on a shared PCI express interconnect, which is displayed in Figure 19.

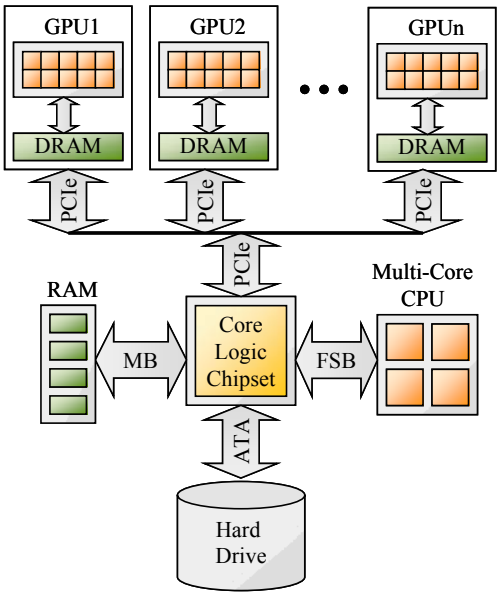


Figure 19: General system architecture of a heterogeneous multi-CPU/multiGPU system.

Fast host-device interfacing is even more important as inter-GPU communication is not as efficient as it could be. At the present state of the art, no direct point-to-point connections can be established for data transfers between graphics cards. Instead, data must be downloaded from the device memory of the first GPU to host RAM, and then uploaded from host RAM to the device memory of the second GPU. The path of data movement is shown in Figure 20.

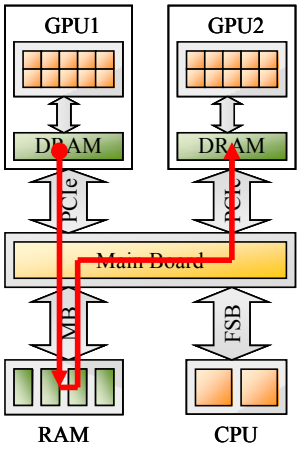


Figure 20: Data exchange between two GPUs.

This problem persists for quite some time now and was not addressed yet by GPU vendors despite remarkable advancements in other aspects. Probably the graphics legacy handling of

the GPU-host interface makes very hard to synchronize two devices efficiently enough to enable a fast direct transfer.

In the context of multi-GPU communication, one should also mention the efforts both primary GPU vendors made in distributing the work load of graphics rendering over multiple GPUs: NVIDIA's Scalable Link Interface (SLI) [128] and ATI/AMD's Crossfire [130]. These special interfaces fuse 2-4 GPUs on a low driver level: depending on the variant, the GPUs render different interlaced frames or parts of the screen. The interfaces initially required an additional physical connector between graphics cards; only recently, the technologies could be extended to also work across the PCIe bus. Unfortunately, neither SLI nor Crossfire is supported by current GPGPU drivers and software design kits.

There are also graphics cards like the NVIDIA GTX 295 that integrate two GPUs on the same board, along with one dedicated DRAM per GPU. This has the advantage that more GPUs can be packed on a single main board; however, the GPU pairs are competing for the same PCI express interface. Moreover, no additional advantage can be drawn from the physical proximity of the GPUs on the same graphics board: swapping data between the device memories still requires buffering on the host RAM, an absurd inefficiency that is owed to the original intention that the two GPUs should be used for graphics in SLI mode.

2.5.3 CPUs

The CPUs primary use is handling and control of the GPUs, which is done via continuous synchronization loops between both processing units. From this results the most important constraint concerning the CPU part: there should be at least one dedicated CPU core per GPU to avoid conflicts and unnecessary delays. Most contemporary CPUs have two or four cores; at the moment only high-end models have more. But it is usually much cheaper to just put two quad-core CPUs on the main board.

The gross of computational load will usually be left to the GPUs so the throughput of the CPUs is of secondary concern. Admittedly, some applications have reported relevant performance gains from using the CPU in parallel to the GPU, like factorization methods where the performance gap between both architectures is not so large [93]. In most cases, however, the CPU's role will rather be reduced to that of the "steering" of the heterogeneous system, as humorously allegorized in Figure 21.

Figure 21: Humorous allegory of the CPU's role in a heterogeneous CPU-GPU system (source: intelsinsides.com).

While slightly exaggerating, the parody does not omit the fact that the CPU is still of primary importance in the system, although not for heavy-duty computational work. Furthermore, it is also plausible that faster CPUs reduce the overhead of CPU-GPU synchronization, although I am not aware of any study quantifying this effect.

2.5.4 Power and cooling

GPUs are power hungry and have long left the range where the graphics card could be supplied via the bus interface. A high-end graphics card typically requires external power supply in the range of 110-270 watt [131]. When multiple graphics cards are packed on the same main board, the power requirements soon rise to the order of kilowatts. Desktop power supplies that powerful are available as commodity hardware, although they are more expensive than the weaker run-off-the-mill products.

Another issue comes from the fact that all that power ultimately translates into heat: we introduced values of typical TDPs in Table 1. GPUs are equipped with large heat sinks and strong fans (s. Figure 22), but the physical proximity of multiple devices might raise concerns about heat dissipation. However, experience shows that air cooling elements suffice even in a multi-GPU system, provided that the case allows enough circulation.

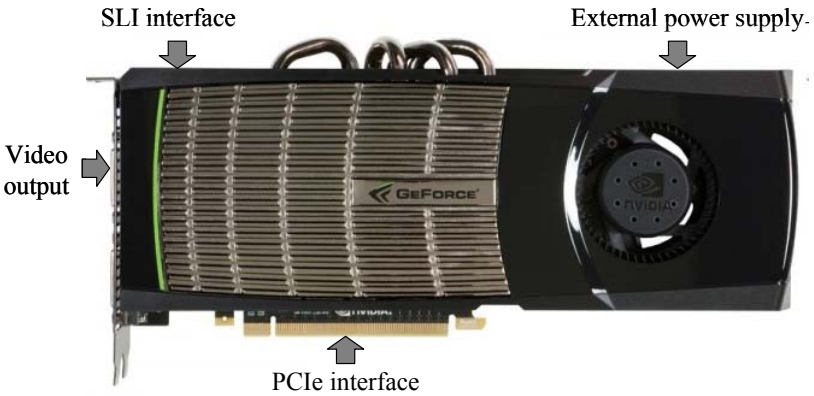


Figure 22: GTX 480 graphics card with cooling element.

2.5.5 Desktop supercomputer "cuba"

As an example setup for a heterogeneous multi-CPU/multi-GPU system, this chapter presents the current configuration of our experimental platform, the desktop supercomputer "cuba". It was used as target platform for the implementation presented in Chapter 5. An overview of the system architecture is shown in Figure 23.

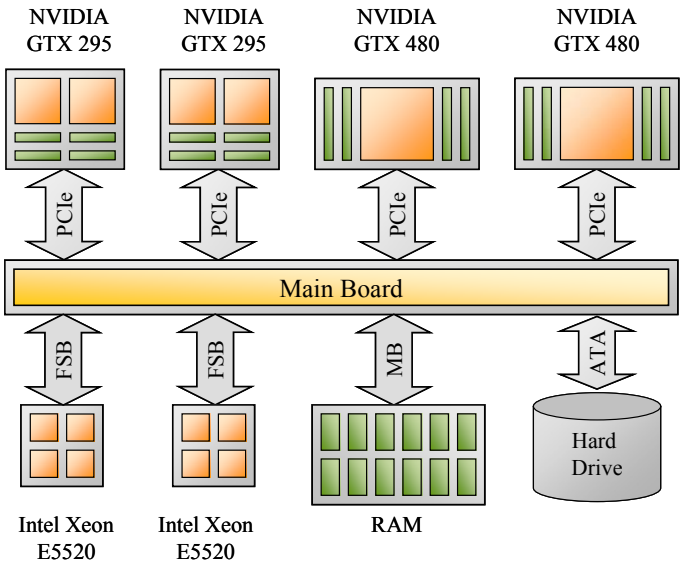


Figure 23: System overview of "cuba"

The system configuration is as follows:

- Two Intel Xeon E5520 CPUs @ 2.27 GHz
- Two NVIDIA GTX 295 graphics cards
- Two NVIDIA GTX 480 graphics cards
- 24 GB RAM
- 4 kW power supply

In total, the system comprises eight CPU cores and a total of six GPUs of two different architectures (s. Chapter 2.4.3). Summarizing the theoretical throughput of the GPUs alone, the system is capable of

- 6266 GFLOP/s in single precision
- 1644 GFLOP/s in double precision

The overall system cost is around 10K€, which is remarkably affordable considering the computational throughput.

2.6 Programming environment

A system like the one presented in Chapter 2.5.5 has remarkable computational power for it's cost and energy budget, put exploiting these resources is not a trivial task. GPUs have very specific aptitudes, and the programming model was created assuming single-GPU usage. This chapter will outline the software architecture for programming a heterogeneous Multi-CPU/Multi-GPU system.

2.6.1 Software architecture

To fully exploit the potential of a Multi-GPU system, only two additional library dependencies are required: The CUDA run-time and driver libraries [87] and OpenMP [132]. CUDA provides the application programming interface to the GPU, while OpenMP handles CPU multithreading. Both will be further described in Chapters 2.6.2 and 2.6.3. The Software hierarchy is shown in Figure 24.

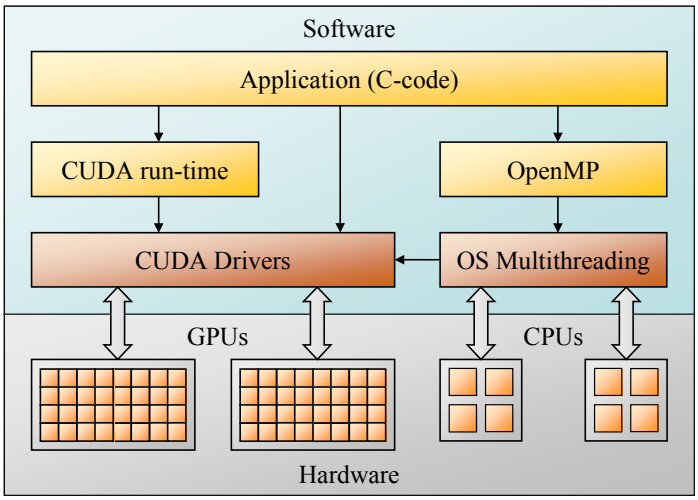


Figure 24: Software hierarchy and interaction with hardware. Both the CUDA driver interface and the multithreading depend on the operating system.

As mentioned in Chapter 2.5.3, each GPU needs to be tied to a dedicated CPU core for handling and control. This is why CPU multithreading is required in the first place. Other solutions exist, but OpenMP is lightweight, convenient and portable, thus perfectly fitting our needs. It will be shortly described in Chapter 2.6.2.

A typical multi-GPU application flow proceeds as follows. The application starts as a single master thread. Then, one CPU thread per GPU is spawned; given enough resources each thread will run on a different CPU core. Within each thread, a *CUDA context* for each of the GPUs is created. This is like creating a handle to the GPU: it sets up the synchronization loop and initializes the runtime library. In the following, all CUDA API calls will refer to the selected GPU: memory pointers will point to the GPU's own device memory and kernels will be executed on the corresponding GPU. Host RAM is shared among CPU threads and with them also the CUDA contexts; it can (and in fact *must*, s. Chapter 2.5.2) be used as a buffer to exchange data between GPUs.

2.6.2 OpenMP

OpenMP is a multithreading standard based on a shared memory principle. In the code, parallel regions are defined via preprocessor directives, upon arrival on which execution is forked into multiple threads. These process the region on different cores (as far as those are available). At the end of the region, threads are joined again with the master thread. A simple example is shown in Figure 25.

```
float a[4],b[4],c[4];
int i;
#pragma omp parallel (private i)
{
  i = omp_get_thread_num();
  c[i] = a[i] + b [i];
}
printf("c = %f %f %f %f\n",
        c[0],c[1],c[2],c[3]);
```

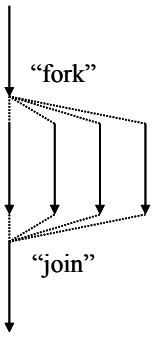


Figure 25: Parallel processing with OpenMP.

The best way to synchronize GPUs is to join OpenMP threads. It would also be possible to synchronize via semaphores, but my experiments have shown that this creates significantly more overhead. Note however that this is only possible because OpenMP threads are not actually destroyed when joining; this would also destroy the CUDA context. Instead, they become dormant and are reactivated during the next forking.

The OpenMP standard actually provides a lot more features than indicated in Figure 25, however no more than the most basic directives is requires for our purposes [132]. It is, however, restricted to shared memory systems. For distributed system like multi-GPU clusters, a multithreading API that supports network communication is required. For these frameworks, the Message Passing Interface (MPI) is a popular solution [133].

2.6.3 CUDA

The term CUDA (Compute Unified Device Architecture) refers to the engine in NVIDIA GPUs that handles computing work load instead of graphics, as well as to the corresponding GPGPU programming interface. It was introduced by NVIDIA in November 2006 and allows direct programming of the GPU in a C-based environment. At the time of writing, the latest CUDA version was 3.2 [87].

2.6.3.1 GPU programming model

The GPU is still a passive co-processor; all control is exerted by the CPU side. A typical application flow proceeds as follows: the CPU uploads input data to GPU memory. The parallel GPU kernel code is then transferred and launched on the device. The code executes asynchronously; to verify completion the status needs to be polled. After completion, the CPU downloads the processed data sets to host memory for output or further usage. The principle is shown in Figure 26.

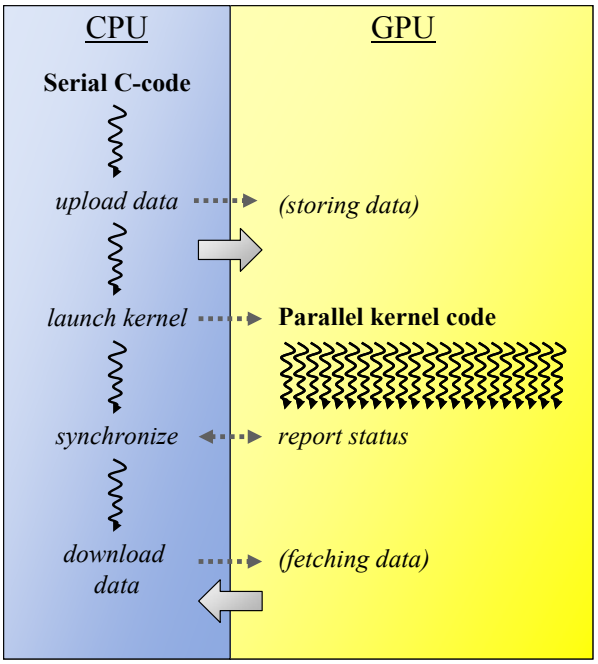


Figure 26: Heterogeneous programming of a CPU/GPU bundle using CUDA.

The GPU is programmed in a so-called Single Instruction Multiple Thread (SIMT) style. This means that while there is a single instruction stream followed by all computational units, it is not entirely restricted to global processing of the same arithmetic, as opposed to SIMD. Some divergence is possible without implications on performance; to which extent will be described in the following.

As shown in Chapter 2.4.3, the GPU architecture is organized in two levels of parallelism: a parallel array of stream multiprocessors, which in turn are parallel processing units. The same two-level structure reflects also in the programming model. The parallel thread pool needs to be evenly divided into a one- or two-dimensional grid of *thread blocks*. These blocks can be up to three-dimensional, which can make address generation more convenient in some applications.

During the work distribution phase, i.e. after kernel launch and before actual execution, the thread blocks are equally distributed to the stream multiprocessors (s. Figure 27).

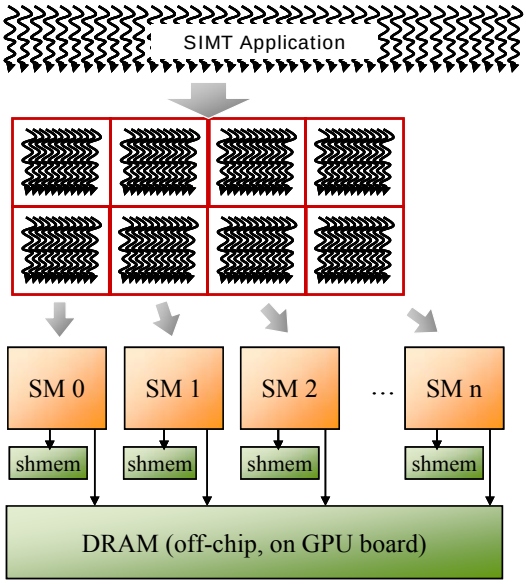


Figure 27: GPU thread clustering and dispatch.

Blocks are then processed by the stream multiprocessors as isolated, distributed sub-problems. This proceeds until all blocks are finished, which also concludes kernel execution. Until this, no interaction or synchronization is possible between blocks. This applies even if they are assigned to the same stream multiprocessor, over which the application programmer has no control. Threads inside a block, on the other hand, can easily communicate via the on-chip shared memory of the stream multiprocessor and can synchronize using a low-overhead barrier.

The CUDA programming model is highly scalable in that it hides all work load distribution and scheduling from the application programmer. Understanding the process, however, is still important in order to produce efficient GPU kernel code. On a stream multiprocessor, threads are divided in groups of 32 identical instructions called *warps*. Only these warps are strictly SIMD; if threads inside a warp diverge, the execution paths are serialized. This effectively divides instruction throughput by the number of diverging paths. However, if two full warps diverge but are in themselves consistent, no performance is lost at all.

All warps on a stream multiprocessor, which might be part of different thread blocks, are executed on the arithmetic units following a prioritized schedule as outlined in Figure 28.

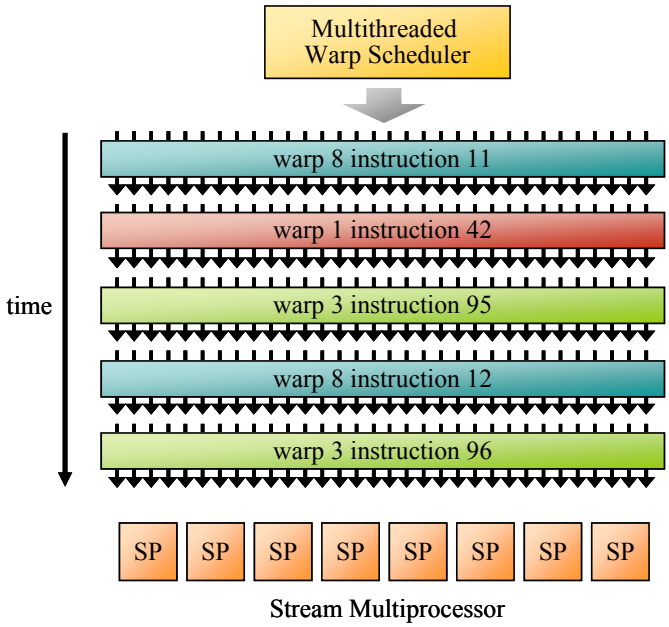


Figure 28 CUDA warp scheduling. In series 8, 9 and 200 GPUs, each of the eight Stream Processor (SP) processes four arithmetic instructions in streaming. In the Fermi architecture (series 400), 32 SPs process the warp completely in parallel.

The warp scheduler tries to line up as many warps from mutually independent instructions stream as possible in order to hide memory access latencies. A maximum of 24 warps can be scheduled at once, while at least 13 are required to fully hide these latencies. Latency-induced stalls are to be expected if there are too few active threads to reach this number. This can happen if blocks require too many resources, limiting the number of active blocks on the multiprocessor. It can also happen if blocks are too small: they should contain at least 32 threads to fill up a warp, but in addition there is a maximum of eight blocks per thread, which calls for at least 96 threads per block to maximize the number of active warps. And naturally, block size should always be a multiple of the warp size for best occupancy. There can be no more than 512 threads per block; 1024 for the latest GPU generation [87].

2.6.3.2 Memory model

Understanding the CUDA memory hierarchy is of crucial importance as nearly all GPU applications are inherently bandwidth limited. Since most of the chip is devoted to arithmetic units, there is only a small amount of fast on-chip memory (s. Chapter 2.4.3). Local data reusage is made even more difficult through the fact that there is no hardware-controlled cache hierarchy comparable to the designs that can be found in CPUs, with the exception of Fermi which has introduced a similar feature. The GPU device memory (GDDR3 or GDDR5 RAM, off-chip) has high theoretical bandwidth, but requires accesses to be very structured to be efficient – another legacy from the device's graphics processing history.

Next to shared memory, a file of 16k registers is shared among threads active on a multiprocessor. Each thread has exclusive access to local memory, which might be contained in the register file as long as enough registers are available. After that, local memory is

swapped out to the much slower off-chip memory. Local memory, wherever resident, is freed as soon as the thread finishes.

The next layer is shared memory. It is consistently shared among threads of the same block; resources allocated in shared memory are freed once the block is completely finished. This implies that all data in shared memory become inconsistent after the kernel finishes; they cannot be reused in a successive kernel.

All threads in a grid have full access to global memory. Only data in global memory remain consistent for the lifetime of the CUDA context, and thus is shared between kernels. The hierarchy is shown in Figure 29.

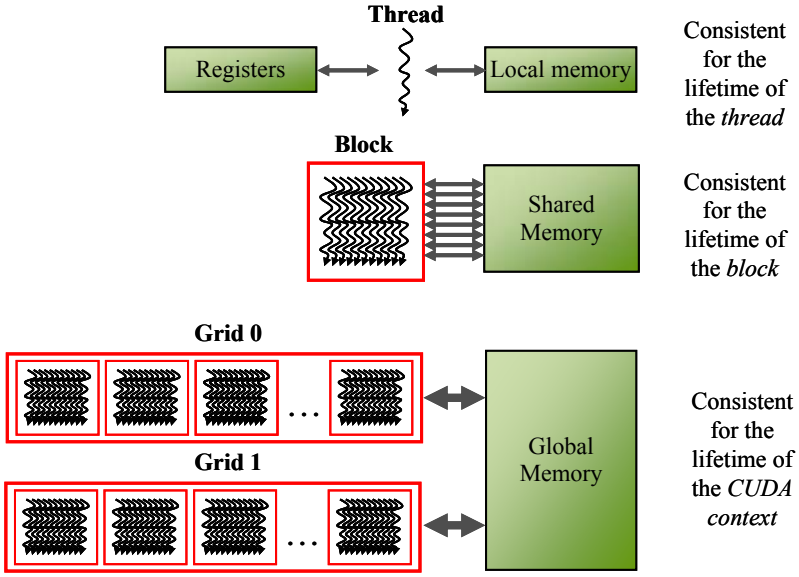


Figure 29: CUDA memory model. Local memory is actually part of global memory, but dedicated to a single thread.

Registers can be accessed in two clock cycles. The same is true for shared memory, provided that no access conflicts occur: to increase bandwidth, shared memory is divided into equally sized modules called banks. In the GeForce 8, 9 and 200 series, each 64 byte segment is accessed via 16 parallel banks. Hence, full parallel access is possible if threads from a half-warp access data with a 32-bit stride. Violation of this rule results into bank conflicts, where conflicting accesses are serialized. A special case is if all threads of a half warp access the same bank; then data is scattered in a single turn. Examples of each case are shown in Figure 30. For the series 400 GPUs, the rules slightly change in that there are 32 banks and accesses are assigned in full warps.

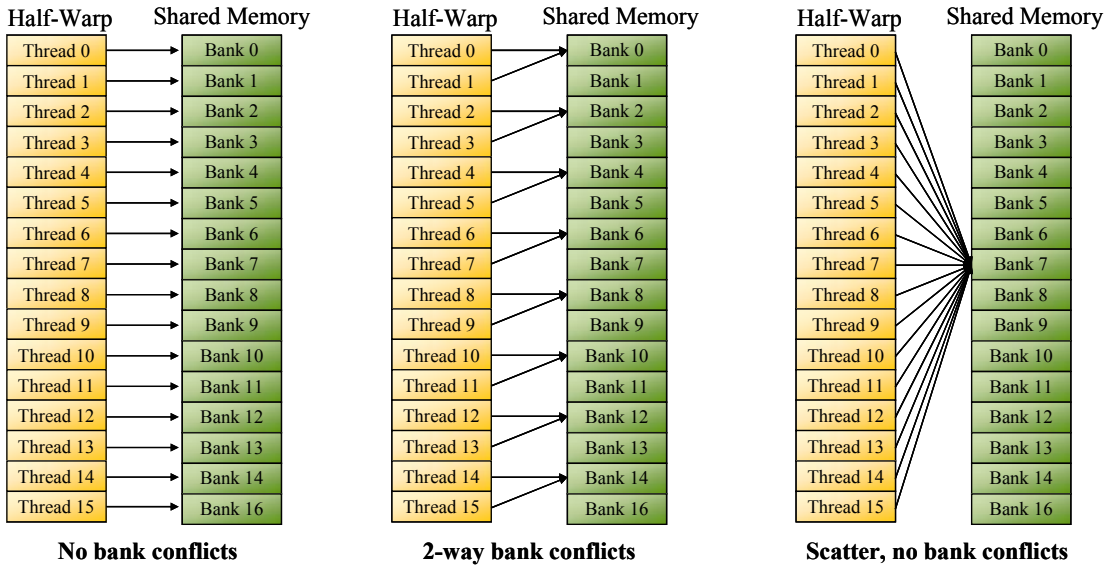


Figure 30: Shared memory access patterns.

Global memory has a latency of 400-600 cycles, which can usually be hidden by warp scheduling (s. Chapter 2.6.3.1). In addition, accessing patterns to global memory must be structured in a particular way to allow *coalescing* of the access, which means that data requested by a half-warp are transferred in a single transaction. Failing to coalesce accesses is penalized with performance drops of up to 90% [93], so this matter has a high priority.

The conditions for coalescing vary depending on the GPU generation, while older architectures are stricter concerning the access patterns. In series 8 and 9 GPUs, all accesses from a half-warp must target the same 128-byte segment and be ordered congruently to the thread index. Otherwise, all transfers are serialized. In the series 200 and newer, the ordering can be arbitrary, and there is always only one transfer per data segment that is accessed. And like for shared memory accesses, the series 400 effects one transaction per warp, not half-warp.

Finally, a small part of the global memory is reserved to constants and texture data, and is cached by the constant and texture caches. A constant cache is integrated into each stream multiprocessor, while there is a shared texture cache for each Texture Processing Cluster. Both caches are read-only.

2.6.3.3 Application Programming Interface

A GPU kernel is defined in a C function, which describes the arithmetic executed by all parallel streams, not unlike the parallel regions of OpenMP (s. Chapter 2.6.2). The grid and block configuration is appended as launching parameters to the argument list when calling the kernel function. The C standard is extended by a minimal set of additional keywords such as to identify functions as GPU kernels or to define a pointer as pointing to GPU memory. Inside GPU kernels, built-in variables can be accessed to identify the thread's position in the block and grid configuration. A set of API functions is provided to control GPU configuration, memory management and transfers between device and host. An example of kernel and host code is shown in Figure 31.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = blockDim.x*blockId.x + threadIdx.x;
        C[i] = A[i] + B[i];
}

// Main program
int main()
{
        ...
        // Calculate kernel configuration, block size is 32
        int Npad = 32*(N/32 + (N%32)?1:0);
        dim3 block_size(32,1,1);
        dim3 grid_size(Npad/32,1,1);

        // Allocate memory on device board
        cudaMalloc(A,Npad*sizeof(float));
        cudaMalloc(B,Npad*sizeof(float));
        cudaMalloc(C,Npad*sizeof(float));

        // Upload input
        cudaMemcpy(A,A_host,N*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(A,A_host,N*sizeof(float),cudaMemcpyHostToDevice);

        // Kernel invocation with Npad threads
        VecAdd<<<grid_size,block_size>>>(A, B, C);

        // Wait until kernel is finished
        cudaThreadSynchronize();

        // Download output
        cudaMemcpy(C_host,C,N*sizeof(float),cudaMemcpyDeviceToHost);
        ...
}
```

Figure 31: Example code in CUDA C defining and launching a simple vector addition kernel.

This is a simple implementation of GPU-accelerated vector addition, following the typical flow mentioned in Chapter 2.6.3.1. Note that the kernel can be scaled to any one-dimensional grid and block configuration. The example also indicates that there is often a need to artificially increase problem size to fit into the grid segmentation, especially since block size should be a multiple of warp size. Idem, allocated GPU memory must be adapted to this new problem size using a technique called “padding”. Conveniently, this often automatically produces the alignment conditions for coalesced memory accesses, at least for the majority of structured problems.

2.6.3.4 Compiler chain

Source code containing CUDA keywords, API calls or kernels must be compiled with NVIDIA’s NVCC compiler driver [161]. Its basic work flow consists in separating CPU and kernel code and compiling the latter into PTX code (a kind of assembly form [160]) or directly into a *cubin* object (binary format).

Note that PTX (Parallel Thread eXecution) is a virtual Instruction Set Architecture (ISA) providing a layer of abstraction for the various CUDA GPU architectures. On a physical level, PTX code is then translated to real device machine code for these targets. The GPU machine code is merged again with the remaining host code, which is then compiled by invoking a C compiler like *gcc* [159].

The compiled host code can also be linked with objects produced by other compilers. This is necessary to include code written in C++ or FORTRAN, as to include libraries providing interfaces in these formats. Prominent examples thereof are OpenMP [132], MPI [123] or LAPACK [139]. A flow chart of the CUDA compile chain is shown in Figure 32.

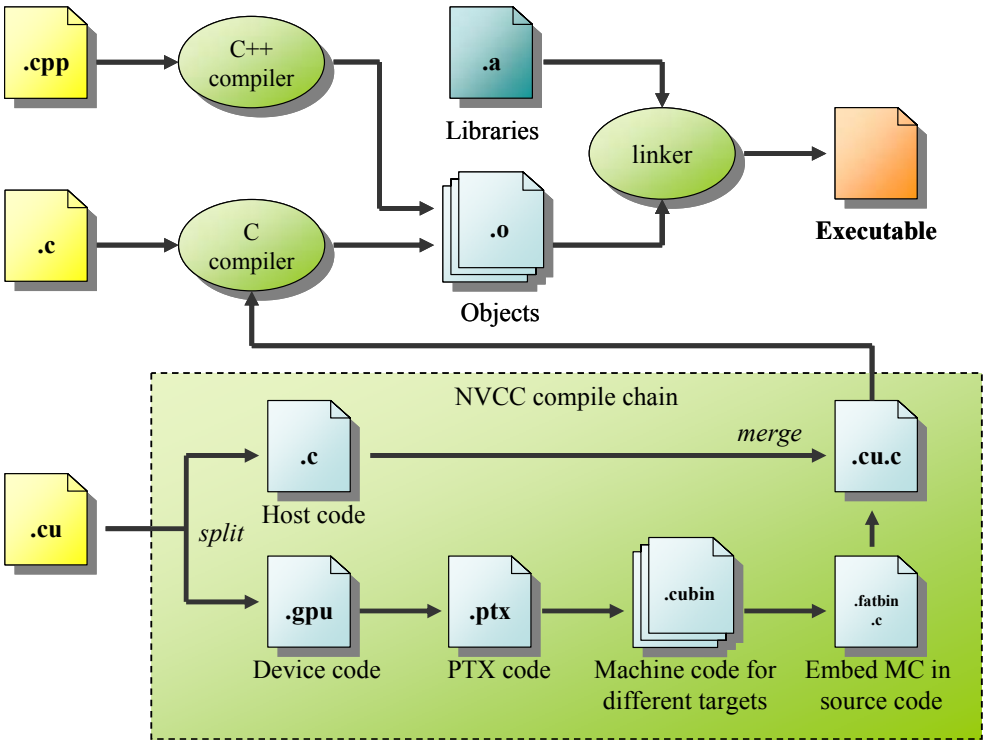


Figure 32: CUDA compile chain. MC = machine code.

The chart is slightly simplified in that some intermediate stages are left out. A description of the full, detailed procedure can be found in [161].

3 Target application: bioelectrical imaging

Since the main contributions of this thesis are motivated by applications in bioelectrical imaging, this chapter was included to give a quick overview over the field as far as it concerns the subsequently presented implementations. However, the focus of this work is on parallel processing on heterogeneous multi-core systems, so this chapter is condensed to a short and selective review. For a more comprehensive introduction to the field as a whole I recommend [183] for further reading. More detailed information about electroencephalography can be found in [5], while the concept of source imaging is exhaustedly treated by [13], [14] and [17]. What concerns Electrical Impedance Tomography, a good and recent review can be found in [9].

3.1 Electroencephalography

Electroencephalography uses an array of passive electrodes to record potential differences on the scalp. These originate from the firing of neurons in the cerebral cortex, the part of the brain regarded to be responsible for a wide array of higher level brain functions like awareness, memory, perception, though, language and conscientiousness.

Neurons are the basic building blocks that process and transmit signal inside the cortex. This is done by an alternating chain of chemical and electrical signals. Ion pumps on the neuron's cell membrane create an unequal potential distribution. A signal is represented by a temporal depolarization that travels along the neuron. This entails compensatory currents to flow the outside of the neuron, which create an electric field throughout the head, which in turn ultimately creates potential differences on the scalp. There are limits to the type of activity that can be detected by EEG, though. Single neurons are far too weak to create a notable signal, the electrodes are large and remote by comparison and the skullcap in between creates a powerful shielding effect. Detectable currents are created only by patches of pyramidal cells aligned perpendicularly to the cortical surface that are simultaneously active. Even then the measured potential differences are quite small, in the order of microvolts (μV) [7]. Figure 33 shows a detail of typical EEG recordings: four channels over approximately five seconds.

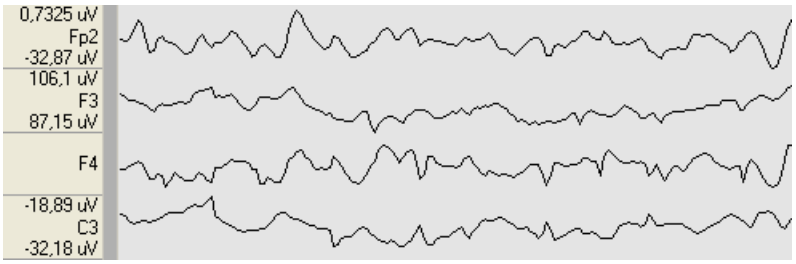


Figure 33: EEG traces

An EEG cap typically consists of 19 to 256 electrodes. In clinical diagnostics there are some standardized montages that define different sets of channels with their precise positions on the scalp. An example for a common montage scheme is given in Figure 34.

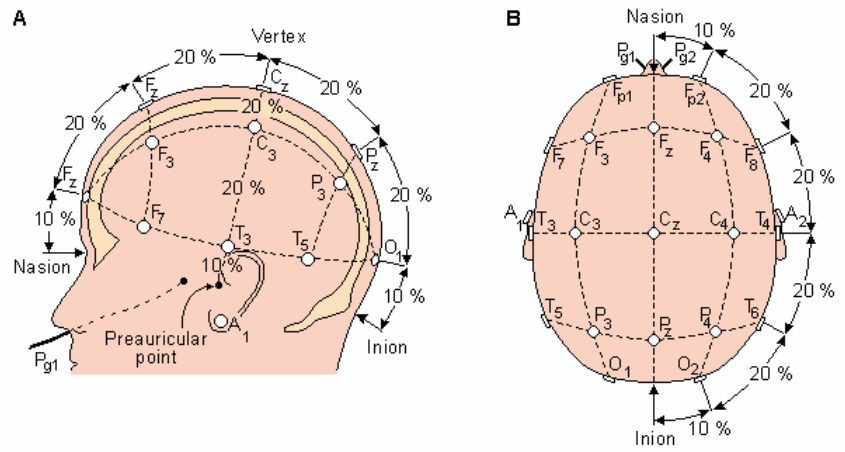


Figure 34: EEG scalp montage according to the common 10/20 system from a side-faced (A) and a top-down view (B). (source: [6])

Compared to other imaging methods, EEG has a high temporal resolution. Most EEG systems sample at a rate of around 240 Hz. In standard clinical studies, the main frequencies of interest range up to approximately 30 Hz. The spectral composition of the signals can convey additional information. In classical EEG analysis, rhythmic activity is considered to be divided into the following bands:

- Delta rhythm: 1-4 Hz
- Theta rhythm: 4-7 Hz
- Alpha rhythm: 8-12 Hz
- Beta rhythm: 12-30 Hz

Frequency components above these bands are usually considered to be artifactual, i.e. from an unwanted source not related to neuronal activity, like noise and muscle movements.

Routine EEG has a wide range of applications. A major one is to aid diagnosis of epilepsy, where abnormal patterns such as sharp waves and spikes can be seen; other clinical applications include diagnosis of coma and brain disorders. In research, EEG plays an important role in cognitive sciences and psychophysiology. Certain parts of the cortex can roughly be mapped to certain brain functions, like the visual cortex and the motor cortex, so signals picked up from an EEG channel close to these regions can give insight into the brain's principal activity or indicate a focal brain disorder [5].

EEG alone however does not allow precise anatomical localization of activity; this requires further signal processing in the framework of a neural source localization method, the principles of which will be outlined in Chapter 3.2.

3.2 EEG source imaging

In traditional EEG diagnostics, the raw recordings are visually inspected by the doctor looking for anomalies in the waveforms. This gives at best a very crude estimate in localizing the regions of activity. However, within certain limits, the scalp potential measurements from

EEG can also be used to trace back the electric field distribution in the head and localize the neuronal sources thereof in the cortex.

3.2.1 Applications

EEG source imaging has been used in pre-surgical analysis before being gradually replaced by anatomical imaging methods like fMRI. However, as opposed to these alternatives, EEG source imaging provides the only direct measurements of source activity, combined with an excellent temporal resolution. It is therefore of particular interest in understanding the generation and propagation of this activity [156]. These capabilities are often exploited for more precise localization of epileptic discharges, where visual interpretation can at best provide an estimate of which cortical lobe is affected [171].

A newer application is neurofeedback, which gives the user insight into his own brain activity mapping while the underlying sensor data is being recorded. This enables a direct feedback loop, where the user can actively try to influence his activity patterns. Some therapies are based on training the user in recognizing and suppressing certain patterns related to their neuropathological condition. Examples thereof include treatment of Attention Deficit Hyperactivity Disorder (ADHD) [171], substance abuse [172], depression and anxiety [173], and more. Though still at an experimental stage, first results of neurofeedback based therapy are promising.

Rather than for direct visualization, the output of EEG source imaging can also be forwarded to automatic interpretation. The results thereof can then be used to drive control of a computer of machine interface. Being under research for 40 years now, these Brain Computer Interfaces (BCI) and Brain Machine Interfaces (BMI) have already reached some maturity [174], with first commercial products entering the market [175].

3.2.2 Forward and inverse problem

The topic of source localization can be classified into two parts, the forward and the inverse problem, which will both be introduced in the following two sub-chapters. The scope of the forward problem is to construct an electrical model of the head, containing a distribution of current sources in the parts of the brain that are able to produce electrical activity (i.e. the grey matter, s. Chapter 3.1). The forward problem is solved when the impact of each current source on each electrode is known. It needs to be solved once per subject and electrode cap [14].

The result is a highly underdetermined system of forward equations. Obtaining an anatomically reasonable resolution of this system is called the inverse problem. It needs to be solved for each sample vector of sensor measurements; the result is an instantaneous distribution of source activity in the brain, i.e. the amplitudes of the current sources in the electrical model [17]. The electrical model should be as anatomically precise as possible as its quality also constraints the quality of the source localization.

3.2.2.1 Forward problem

To localize the neural current source generators of EEG, the first question that needs to be asked is: how would a single given current source in the brain affect the potential distribution on the scalp? This is called the forward problem of EEG source imaging. Starting from the quasi-static Maxwell equations, this basically breaks down to solving the equation

$$\nabla \cdot (\sigma \nabla \varphi) = \nabla \cdot j \qquad (1)$$

Where σ is the conductivity tensor, φ is the potential distribution and j is the current density vector field in the medium, in this case the head. Quasi-stationarity is a fair assumption considering the low frequency range of EEG generators (s. Chapter 3.1).

There is only a limited amount of potential measurements, so we can transfer φ into a discrete vector $\boldsymbol{\Phi}$ of length N_E containing all instantaneous sensor measurements, with N_E being the number of electrodes. For technical feasibility, also the current sources must be discretized. Picking individual neurons for this is unfeasible, instead, a macroscopic current dipole model is commonly chosen to segment the cortical grey matter into an uniform array of discrete current sources, called "voxels". The result is a vector $\boldsymbol{J} = (J_1, J_2, J_3, \ldots J_{Nv})$, with J_i being the 3-dimensional current density vector at voxel i and N_V being the number of voxels.

All in all, the solution to the (discretized) forward problem is a linear map $\boldsymbol{K}$: $\boldsymbol{R}^{N_V} \times \boldsymbol{R}^3 \rightarrow \boldsymbol{R}^{N_E}$, mapping current densities to scalp potentials such that

$$\boldsymbol{J} = \boldsymbol{K} \cdot \boldsymbol{\Phi} + c\boldsymbol{1} \qquad (2)$$

$\boldsymbol{K}$ is commonly referred to as the electric lead field. The summand $c\boldsymbol{1}$ is added due to the fact the electric potential is defined only up to an arbitrary constant; c is a scalar and $\boldsymbol{1}$ denotes a vector of ones. The linearity holds due to the superposition principle.

Solution of the forward problem (i.e. obtaining the coefficients of $\boldsymbol{K}$) strongly depends on the volumetric conductivity model of the head (s. Chapter 3.2.3) and can be very challenging if the conductivity model contains some anatomical detail [14].

3.2.2.2 Inverse problem

The lead field maps brain activity patterns to sensor waveforms, but what we actually want in source imaging is a reverse of this operation: given a vector of scalp measurements $\boldsymbol{\Phi}$ in a system with parameters $\boldsymbol{K}$, how can we obtain the underlying brain activity patterns $\boldsymbol{J}$? This is referred to as the inverse problem. It is highly underdetermined since there are far more unknowns than information ($3 \cdot N_V >> N_E$), so the solution will always be an estimate.

There are manifold ways to tackle underdetermined problems [15], but due to the linearity of the forward equations, it is appealing to address the EEG inverse problem with linear estimators. In fact, many solutions are based on the least squares of errors, such that

$$\| \boldsymbol{\Phi} - \boldsymbol{K} \cdot \boldsymbol{J} \|_2$$

is minimized. In the simplest form this yields the Moore-Penrose pseudoinverse [19]:

$$\boldsymbol{T} = \boldsymbol{K}^T \cdot (\boldsymbol{K} \cdot \boldsymbol{K}^T)^{-1} \qquad (3)$$

where $\boldsymbol{T}$: $\boldsymbol{R}^{N_E} \rightarrow \boldsymbol{R}^{N_V} \times \boldsymbol{R}^3$ denotes the resulting linear estimator, called the transition matrix. This approach however suffers from several limitations like instability, bias and low spatial resolution. More advanced methods add various techniques to improve this, some of which will be introduced in Chapter 3.2.5. Furthermore, one can try to limit the degrees of freedom by inferring additional anatomical information, as it will be outlined in Chapter 3.2.4. There are also approaches that are not based on linear estimators, but discussing those would go beyond the scope of this work [17].

3.2.3 Head models

As mentioned in Chapter 3.2.2.1, solving the forward problem for anatomically correct head models poses a great challenge: from an electrical point of view, the head is a highly inhomogeneous and anisotropic medium. As a result, head models in practical use are strongly simplified. This is very limiting, as the inverse solution is always just as good as the model it is based on [16].

First head models were based on concentric spherical shells to define up to four internally homogeneous compartments of scalp, skull, cerebrospinal fluid (CSF) and brain [22]. Spherical models can be calculated analytically, however, the head does not come in these convenient shapes, so anatomical correctness is rather poor [20].

While, some forming can be applied to morph the spherical model more into the shape of a head, spherical models were soon replaced by multi-shell models with more accurate compartment boundaries based on triangulated surface meshes. An example is shown in Figure 35.

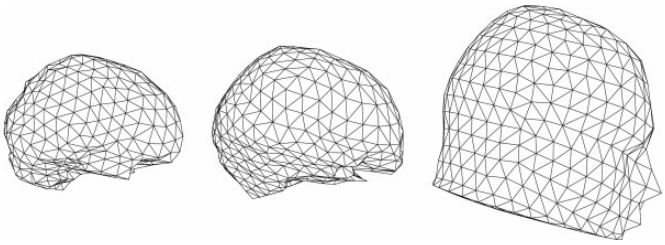


Figure 35: Triangulated surface meshes of brain, skull and scalp (source : [21]).

Inside the compartments, the volume conductor is still assumed to be homogeneous and isotropic. The forward problem for a surface mesh model needs to be computed by numerical methods, where the Boundary Element Method (BEM) is particularly appealing [21]. Surface mesh models still define the state of the art for practical implementations, with many commercial solutions relying on it [23][43]. The BEM solution to the forward problem required inversion of a large matrix, whose rank depends on the resolution of these meshes. Contemporarily, the cubic complexity of this operation restricts this approach to rather poor anatomical detail.

In experimental setups, more detailed head models are currently subject of study. These include for example segmentations based on the Finite Element Method (FEM), the Finite Difference Method (FDM) [26] and the Finite Volume Method (FVM) [27]. All have the advantage that they allow inclusion of tissue anisotropy, along with a much finer level of

inhomogeneity [24]. Discretization can be either adaptive or regular, like the example given in Figure 37.

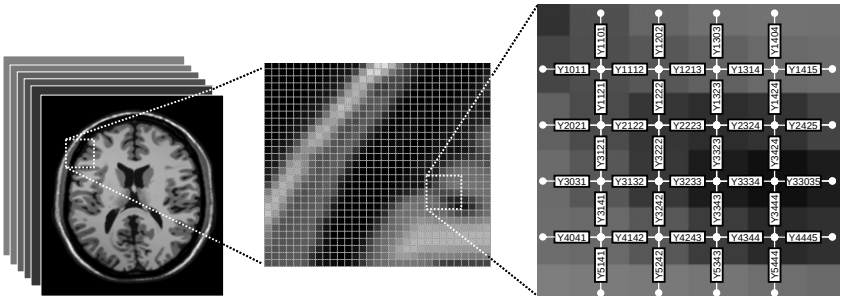


Figure 36: Fine-grained regular FVM-based discretization of the head conductor space

These models lead to very large sparse linear systems that need to be solved in order to obtain the EEG forward equations. They are usually addressed by iterative sparse system solvers [14]. However, the sheer magnitude of these systems (a MRI-based FVM discretization results in a network of up to several million nodes) make computation of these models very time consuming, impeding their application in contemporary practice.

3.2.4 Retrieving anatomical information

Anatomical correctness of head models limited not only by computational feasibility, but by the level of trustworthiness of the anatomical information itself. The same applies to anatomical priors in the inverse problem. The primary sources of information are head scansion from static Magnetic Resonance Imaging (MRI), typically a 3D black-and-white image of ~1 mm resolution. On this raw image, various image processing techniques can be applied to retrieve meshes of the brain, skull and scalp surfaces [29], segment the image into different neuro-anatomical structures [31] and label different tissue types [30] (s. Figure 37).

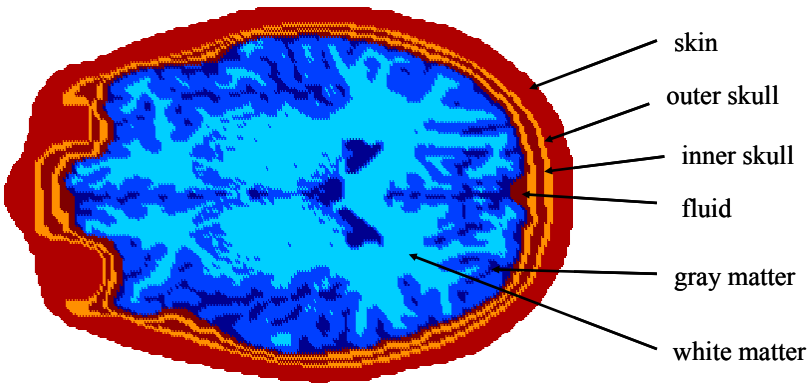


Figure 37: Segmentation of MRI voxel space into tissue compartments, as produced by the BrainSuite software [29].

Concerning the electrical properties of said tissue types, a plethora of experimental measurement data are available, both in vivo and in vitro [32][34][35]. Together, this information is all that is needed to formulate the forward problem.

Also the inverse problem can benefit from these data. Knowing what parts of the brain can be considered possible generators of source activity helps constraining the solution space, and

thus allows better estimates. Inferring other anatomical priors like cortical orientation has also been proposed [36].

3.2.5 Linear estimators for the inverse problem

As mentioned in Chapter 3.2.2.2, most solutions to the inverse problem are based on squared error minimization. However, numerous improvements have be made to cope with the disadvantages of the naïve pseudoinverse [40], some of which will be introduced in the following.

First of all, the error minimization problem could be ill-posed or singular, leading to a large number of possible solutions. This can be remedied by adding a regularization term to give some solutions prevalence over other, for example by preferring the solution with the lowest total energy. In this case, the minimization term becomes

$$\| \boldsymbol{\Phi} - \boldsymbol{K} \cdot \boldsymbol{J} \|_2 + \alpha \cdot \| \boldsymbol{J} \|_2$$

where α is called the regularization parameter. The resulting estimator

$$\boldsymbol{T} = \boldsymbol{K}^T \cdot (\boldsymbol{K} \cdot \boldsymbol{K}^T + \alpha \cdot \boldsymbol{I})^{-1} \tag{4}$$

is called the Minimum Norm Estimate (MNE) [37]. MNE approaches are a special case the Tikhonov regularization [38]. The regularization parameter α controls how strongly the total energy will be minimized in respect to the error. Picking an optimal value is a difficult problem itself to which several approaches exist, like the L-curve method [39].

The estimated current density vectors, given by

$$\boldsymbol{J}_{est} = \boldsymbol{T} \cdot \boldsymbol{\Phi}$$

with $\boldsymbol{J}_{est}$ being the vector of estimated currents of size $\boldsymbol{R}^{N_V} \times \boldsymbol{R}^3$ (s. Chapter 3.2.2).

Source localization of the MNE can be improved by applying a standardization procedure on estimated current density vectors. This requires an estimate of the solution's variance, which according to [43] can be derived as

$$\boldsymbol{S} = \boldsymbol{T} \cdot \boldsymbol{K}$$

This coincides with the resolution matrix [44] and is not invertible, hence the standardization is performed voxel per voxel to retrieve the standardized current density powers:

$$j_{l,std} = \boldsymbol{j}_{l,est}^T [\boldsymbol{S}_{ll}]^{-1} \boldsymbol{j}_{l,est}$$

where $\boldsymbol{j}_{l,est}$ contains the three components of is the *l-th* element of $\boldsymbol{J}_{est}$ and $\boldsymbol{S}_{ll}$ is the *l-th* diagonal sub-block of $\boldsymbol{S}$. This technique is applied in the Standardized low resolution brain electromagnetic tomography (sLORETA) method [43], which is able to localize single sources in noise-free environments with 100% accuracy.

Still, these methods suffer from poor spatial resolution. This problem is already inherent in EEG, and the MNE tends to make things worse. Several efforts have been made to cope with this limitation. For example, the focal underdetermined system solver (FOCUSS) tries to give some focal resolution on distributed source models by iteratively applying a weight to the columns of the lead field:

(5) $$\boldsymbol{T}_i = \boldsymbol{W}_i \cdot \boldsymbol{W}_i^T \cdot \boldsymbol{K}^T \cdot (\boldsymbol{K} \cdot \boldsymbol{W}_i \cdot \boldsymbol{W}_i^T \cdot \boldsymbol{K}^T + \alpha \cdot \boldsymbol{I})^{-1}$$

where $\boldsymbol{W}_i$ is a diagonal matrix based on the solution retrieved during the last iteration, normalized by the norm of the columns of $\boldsymbol{K}$:

$$\boldsymbol{W}_i = diag(\boldsymbol{J}_{est,i-1}) \cdot diag(\boldsymbol{K}^T \cdot \boldsymbol{K})^{-1} \qquad with\ \boldsymbol{J}_{est,i-1} = \boldsymbol{T}_{i-1} \cdot \boldsymbol{\Phi}$$

If continued long enough, FOCUSS converges to a set of concentrated sources equal to number to the number of electrodes [41].

Another way to use the previous solution for iterative refinement is shrinking the solution space: if a part of the cortex is found to contain relatively low activity (below a certain threshold), the corresponding columns of $\boldsymbol{K}$ are temporarily removed and the inverse estimator is re-calculated. Solutions found in a smaller solution space tend to exhibit better spatial resolution [42]. Advanced linear solvers typically use a combination of the above mentioned methods, like the Shrinking Standardized LORETA-FOCUSS (SSLOFO) [45]. While these methods are much harder to handle computationally when compared to simpler solutions, the source reconstruction is often of better quality.

Contemporary practice is limited to simple algorithms like WMN and sLORETA, with very low voxel resolutions of a few thousand discrete points. More advanced methods and higher voxel resolutions produce work loads where processing on normal workstations is still unfeasible.

3.3 Electrical Impedance Tomography

Electrical Impedance Tomography (EIT) is a relatively new bioelectrical method. Although it yields great potential to be a valuable addition to clinical diagnosis, the technical and computational difficulties of this technology have so far prevented it from being included in clinical routine.

Like EEG, EIT works with non-invasive measurements from skin-mounted electrodes, but instead of passively recording potential differences, currents are actively injected into the body, usually in patterns of several electrode pairs and at different frequencies. Meanwhile, the resulting potentials differences are measured (s. Figure 38). Based on these data, an impedance map of the target body can be estimated to get insight into internal tissue properties [12], e.g. to localize anatomical abnormalities.

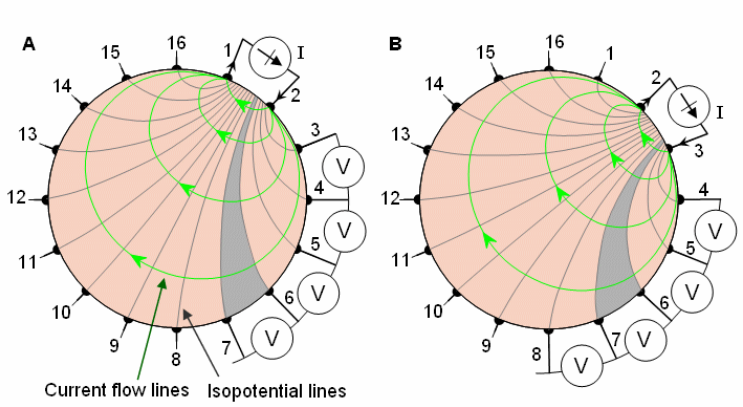


Figure 38: Application of current patterns through the target body and measurement of resulting surface potentials (source: [10]).

EIT has a wide range of applications including monitoring of pulmonary ventilation, breast tumor detection and the reconstruction of brain activity, all of which are still at an experimental stage. However, the first commercial clinical solutions for pulmonary ventilation monitoring are approaching market maturity. EIT-based brain analysis and monitoring currently subject to study, particularly to localize well defined regions of the brain like epileptic foci, local ischemia and lesions [8][9].

EIT might even be combined with EEG, considering that both technologies use different sources of information. It is also plausible to use conductivity measurement from EIT to refine volume conductor models in EEG source localization [11]. Given an intelligent electrode design, both might even share the same sensor montage.

3.4 Selecting computational problems

Bioelectrical imaging is a wide field of study, and this chapter could barely scratch the surface. However, this short introduction already demonstrates that there is a plethora of computational challenges with very different characteristics. Primarily, problems encountered in this field can classified into the following categories:

- Dense linear algebra
- Sparse systems
- Image processing
- Signal processing
- Data visualization

Each of these classes has different demands towards the computing platform, so focusing on a single, isolated problem would not suffice to evaluate the benefits that the heterogeneous CPU/GPU systems described in Chapter 2 can contribute to this field as a whole. Instead, this thesis presents a selection of applications covering all of the aforementioned categories. Furthermore, very different scales of CPU/GPU systems are employed, ranging from low-end single-GPU setups that might also fit into mobile systems, over average dual-GPU desktop systems, up to small-scale supercomputers featuring several CPUs and graphics cards.

In the next three chapters, the following computational problems are addressed. Chapter 4 presents an example of heavy, dense linear algebra with a large triangular matrix inversion problem emerging from EEG volume conductor modeling (s. Chapter 3.2.3, meshed head models). Chapter 5 addresses solution of large sparse, non-hermitian systems as part of an EIT simulation environment (s. Chapter 3.2.3, FVM-based head models and Chapter 3.3). Lastly, Chapter 6 covers problems encountered in the EEG inverse problem comprising real-time signal processing (s. Chapter 3.2.5), image processing of MRI scansions (s. Chapter 3.2.4) and graphical 3D visualization of large data sets.

Given the wide range of both applications and target systems treated in this work, conclusions can finally be drawn with sufficient validity. The results of the general evaluation will be discussed in Chapter 7.

4 Dual-GPU accelerated Triangular Matrix Inversion

This is the first in a series of three chapters presenting the major contributes of this thesis. They all address very different computational problems that where highlighted in Chapter 3, using different scales and configurations of the platforms described in Chapter 2. This part revolves around an example of dense linear algebra, addressed by a mid-sized dual-GPU platform.

The scope of this Chapter is to present an efficient implementation of Triangular Matrix Inversion on Graphics Processing Units. While TMI can be applied in several contexts, we will focus on its role in the inversion of dense matrices using the algorithmic steps shown in Figure 39.

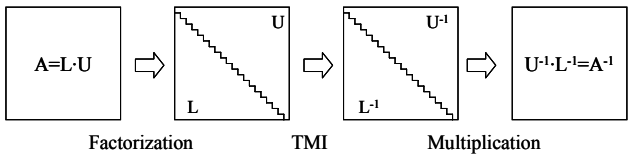


Figure 39: Basic work steps in dense non-symmetric matrix inversion.

The matrix is factorized using LU decomposition and the upper and lower triangular factors are stored in a shared array, then TMI is applied to the resulting triangular matrices. After this step, the inverse of the full matrix is obtained by the multiplication of the inverted triangular matrices.

Implementation of matrix factorization routines on GPUs has been examined in several contributions, most notably [93][94][166] and [167], and matrix multiplication is a standard operation. But in spite of being the heaviest part in the inversion work flow, TMI was neglected by prior work. This is mainly because the mere solution of a single linear system is rarely computed via the inverse of the matrix: matrix inversion has a cubic complexity, producing huge work load for larger problem sizes. However, there are several applications where the full matrix inversion is beneficial or inevitable. Apart from the bioelectrical problem that motivated this implementation (s. Chapter 4.1), typical applications for full matrix inversion include preliminary steps for optimization [168] and network coding [169]. This implementation fills the gap with a GPU-accelerated TMI routine, based on a recursive divide-and-conquer approach to exploit maximal parallelism. This application assumes a shared array LU matrix as it is commonly produced by matrix factorization methods as a starting point, and both upper and lower triangular matrix inversion has been implemented. The original implementation has been published in [1], on which later substantial improvements have been made like better multi-GPU performance and efficient memory usage. The final version has been accepted in [2].

4.1 Motivation and background

This implementation was motivated by a severe computational bottleneck in solving the EEG forward problem using a three-layer mesh model of the head (s. Chapters 3.2.2.1 and 3.2.3).

This problem was addressed using the methodology described in [21], which introduces an advanced approach based on the Boundary Element Method (BEM).

Most steps of this algorithm require negligible effort, so let us skip right forward to the aforementioned bottleneck. In the last step, the method produces the following equation defining the electric lead field $\boldsymbol{K}$:

$$\boldsymbol{K} = \boldsymbol{D} \cdot (\boldsymbol{I} - \boldsymbol{C})^{-1} \cdot \boldsymbol{G} \tag{6}$$

In this equation, $\boldsymbol{C}$ is a $N_T \times N_T$ matrix whose elements are determined by geometry of the meshes and the conductivities of the compartments, where N_T is the total number of triangles in all three meshes. Matrix $\boldsymbol{G}$: $\boldsymbol{R}^{N_V} \times \boldsymbol{R}^3 \rightarrow \boldsymbol{R}^{N_T}$ is some kind of preliminary lead field mapping the current density vectors from all N_V voxels to the centroids of all N_T triangles, pretending a homogeneous infinite conductor. Its elements can easily be computed analytically [14]. Matrix $\boldsymbol{D}$: $\boldsymbol{R}^{N_T} \rightarrow \boldsymbol{R}^{N_E}$ can be considered to be the opposite interface. It is a sparse matrix mapping potential from centroids of the outer mesh triangles to potentials at electrode positions, which can be done either by clinging to the nearest centroid or by interpolating between the nearest neighbors.

It is obvious that once the inverse $(\boldsymbol{I} - \boldsymbol{C})^{-1}$ in (6) has been computed and stored, the model can easily be adapted to arbitrary voxel and electrode configurations. This however turns out to be a considerable computational challenge: the meshes used in this scope as produced by the "*freesurfer"* software (a free software that allows extraction of surface meshes from MRI, among other features) have a detail level of over 3000 triangles each [163][164]. All three layers combined, we end up with a problem size of $N_T \approx 10000$.

Even when using an optimized multithreaded implementation, inverting a matrix of this dimension takes several minutes on a typical workstation, making GPU acceleration appealing. With this implementation, the computation time can be reduced to a few seconds.

4.2 Parallel Triangular Matrix Inversion algorithm

Triangular matrix inversion can offer substantial parallelism with the following divide-and-conquer approach. The triangular matrix (size *dim*) is partitioned into two triangular matrices and one square matrix, all three of half the size. Then, the resulting triangular sub-matrices can be partitioned in the same way. This is done recursively k times, until we are left with 2^k triangles of size $m = dim/2^k$ lined up along the diagonal. The result is a fractal segmentation of the triangular matrix. In what follows, k will be referred to as the segmentation depth. An example for a segmentation with $k = 3$ is shown in Figure 40.

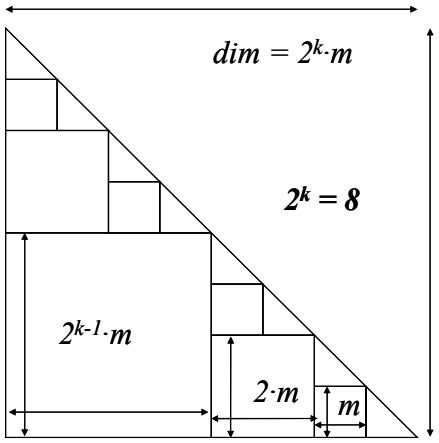


Figure 40: Fractal segmentation of a lower triangular matrix employing a segmentation depth of k = 3. The same procedure is followed for an upper triangular matrix.

Given that all triangles for a certain segmentation depth are already inverted, the inversion of the triangles of the next lower segmentation step can be completed by applying the operations shown in Figure 41 to calculate the square matrix wedged in between the inverted triangles:

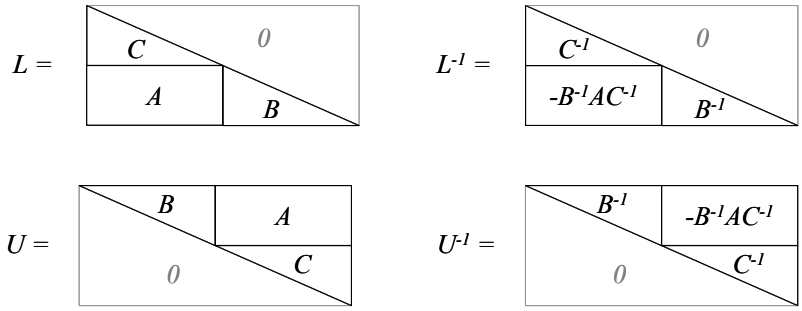


Figure 41: Recursive inversion of triangular matrices.

So starting from initial inversion of the 2^k diagonal triangular matrices of size m (referred to as "step 0"), the full matrix can be inverted in k steps. The computation order is visualized in Figure 42.

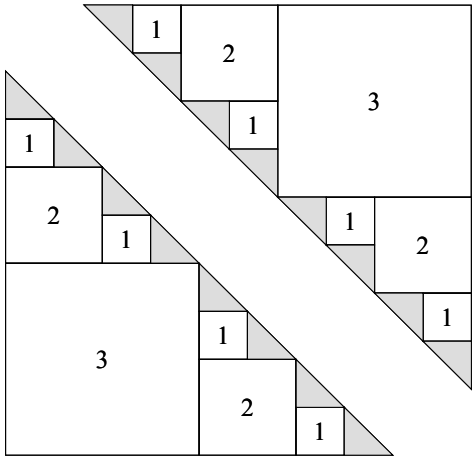


Figure 42: Computation order for recursive triangular matrix inversion, for both lower and upper triangular matrix inversion, keeping k = 3.

Each step i of the computation offers a parallelism of $2^{k-i} \cdot (m \cdot 2^{i-1})^2 = m^2 \cdot 2^{k+i-2}$ independent scalar streams, which is the number of square matrices (sub-blocks) to be calculated in step i, multiplied by the number of elements per sub-block.

This algorithm is called recursive triangular matrix inversion (RTMI). The method was adapted from [134], where RTMI was optimized for MIMD computers. A method that recursively partitions a triangular matrix was originally proposed in [135] and [136], while [137] was the first to use a divide-and-conquer method on partitioned matrices for matrix-matrix multiplication, from which a recursive method for full matrix inversion could be derived [138].

RTMI is optimal, which means that it requires no more arithmetic operations than the best known serial algorithm. Performing the complete inversion of a triangular matrix has a computational complexity of

$$dim^3/3 + dim^2/2 + O(dim)$$

which includes the above mentioned "step 0". When we consider only the recursive part of the algorithm, we end up with a computational complexity of

$$dim^3/3 + dim^2 \cdot (1 - 1/2^{k+1}) + O(dim)$$

for a recursive computation over k steps [134].

4.3 Implementation

4.3.1 GPU Kernels

RTMI is processed in k steps, where step $i = 1..k$ consists of the computation of 2^{k-i} square sub-blocks of size $m \cdot 2^{i-1}$. Each sub-block $j = 1..2^{k-i}$ is computed according to Figure 41 by performing two square-per-triangular matrix multiplications:

$$X_j = B_{i,j} \cdot A_{i,j}$$

$$A_{i,j} = X_j \cdot C_{i,j}$$

These can be implemented in efficient parallel GPU kernels. The parallelism is fully exploited, so $m^2 \cdot 2^{k+i-2}$ threads are launched in each step i. The kernels are slightly different for L and U inversion, so four different kernels are implemented. The precise structure is discussed in what follows. As stated in Chapter 2.6.3, we must segment the $2^{k-i} \times m \cdot 2^{i-1} \times m \cdot 2^{i-1}$ parallel instruction streams into a grid of thread blocks. A thread block size of 16×16 is chosen for several reasons. The most important issue is bandwidth maximization: the GPU performs all global memory accesses in half-warps, i.e. 16 threads with consecutive indices, where the first thread index is a multiple of 16. Having only 8-byte elements, data sections accessed in this way always cover a 128 byte segment. Furthermore, since all GPU memory is covered by 16×16 blocks, the segments are always aligned to multiples of 128 byte. This fulfills the conditions described in Chapter 2.6.3.2 for coalesced memory accesses, for which bandwidth to and from global memory is maximized. Still, the application would be greatly bandwidth limited without data re-usage. Instead of operating directly on global memory, data

operands are fetched in 16×16 blocks and processed on-chip, which greatly reduces off-chip accesses. Registers are used to accumulate data. This setup allows for up to three blocks to be active on one SMP, which corresponds to up to 24 independent warps (three blocks of 16×16 threads, divided by the GPU warp size 32), more than enough to hide most global memory access latencies through intelligent scheduling (s. Chapter 2.6.3.1). The grid of 16×16 block tiles must cover all 2^{k-i} square sub-blocks. Since grids in CUDA can have only two dimensions, we consider the first $ld(2^{k-i})$ bits of the block column index as sub-block index (*sbIdx*). An example of block tiling is shown in Figure 43.

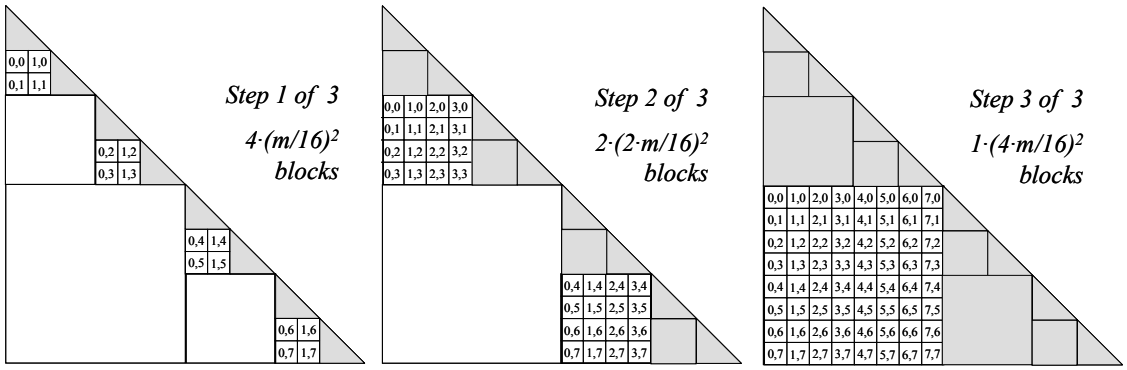


Figure 43: Block tiling and block indices in a lower triangular matrix for parameters m = 32 and k = 3. Regions highlighted in gray have already been inverted.

In summary, the launching parameters for both kernels of step $i = 1..2^k$ are:

- *block_size = (16,16,1)*
- *grid_size = ((m/16)·2^{i-1}, dim/32, 1)*

The kernels themselves implement the two sequential square-per-triangular blocked matrix products mentioned earlier, whereas all sub-blocks of the actual step are processed concurrently. Figure 44 and Figure 45 illustrate the kernels involved in L- and U-inversion, describing the processing of one isolated sub-block, in which one exemplary thread block is highlighted. Dashed arrows indicate the order in which operand blocks are fetched from global memory. The last 16×16 block is modified in shared memory by zeroing all trans-diagonal elements and, in cases of L-inversion, setting the diagonal elements to one.

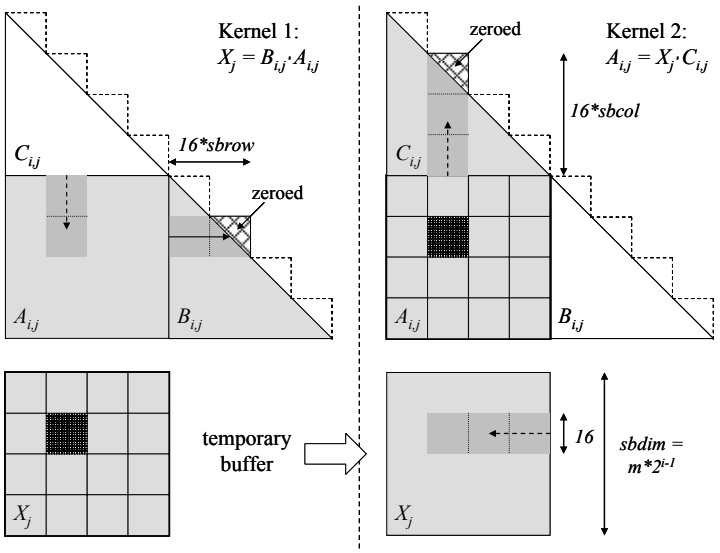


Figure 44: GPU kernels for recursive lower triangular matrix inversion, showing one of 2^{k-i} identical sub-blocks. In this example, the sub-block dimensions (*sbdim*) are 128, *sbrow* and *sbcol* denote the block row and column within this sub-block.

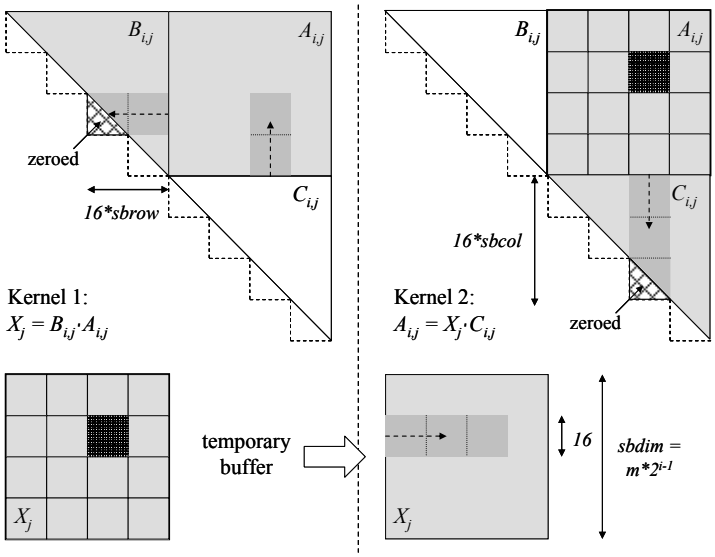


Figure 45: GPU kernels for recursive upper triangular matrix inversion, analogous to the example shown in Figure 44.

It is obvious from the figures that some blocks will take longer to finish than others; the number of loop iterations is dependent on the position of the 16×16 block inside the square sub-block. With the given grid layout, each SMP gets assigned a random mix of blocks with different execution length, but it is inevitable that some SMPs will finish their work load sooner than others, which results in idle processing resources towards the end of kernel execution. The impact of this is keener for kernels with fewer blocks, hence for smaller problem sizes. Another slow-down factor results from the bank alignment of shared memory, which is optimized for single-precision (4 byte) data elements. Accessing vectors of double precision (8 byte) elements always results in 2-way bank conflicts (s. Chapter 2.6.3.2).

4.3.2 Memory optimization

The maximum problem size that can be processed by the routine is restricted by the total amount of GPU device memory. With normal 2-dimensional data alignment, a single triangular matrix occupies twice as much memory as there are data elements. A densely packed data format, as used by many CPU-optimized libraries, is not applicable, as this would thwart the memory alignments and disrupt coalesced memory accesses.

However, since data are only accessed in blocks of 16×16 elements, a block-packed data format can preserve the coalesced memory accesses while providing nearly the same storage optimization. Data are uploaded in rectangular strips of 16 rows in height, which are stored next to each other in GPU memory. This block-row packed format is preferable to a block-column packed format for various reasons, among others to minimize the number of host-device and device-host transfers.

A small additional overhead is introduced because a dedicated memory transfer has to be issued for up- and downloading each block row, so the total transfer time is increased by *dim/16* times the transfer initialization time of 11 µs [93].

The distributions of L- and U-matrix data in host and GPU memory are shown in Figure 46 and Figure 47.

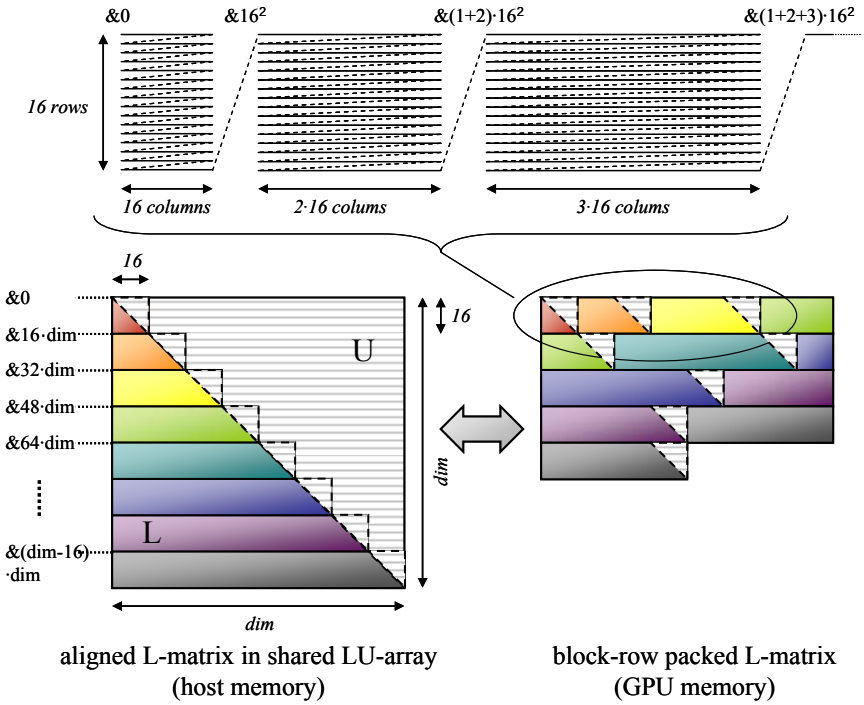


Figure 46: Block row packed storage of the L matrix.

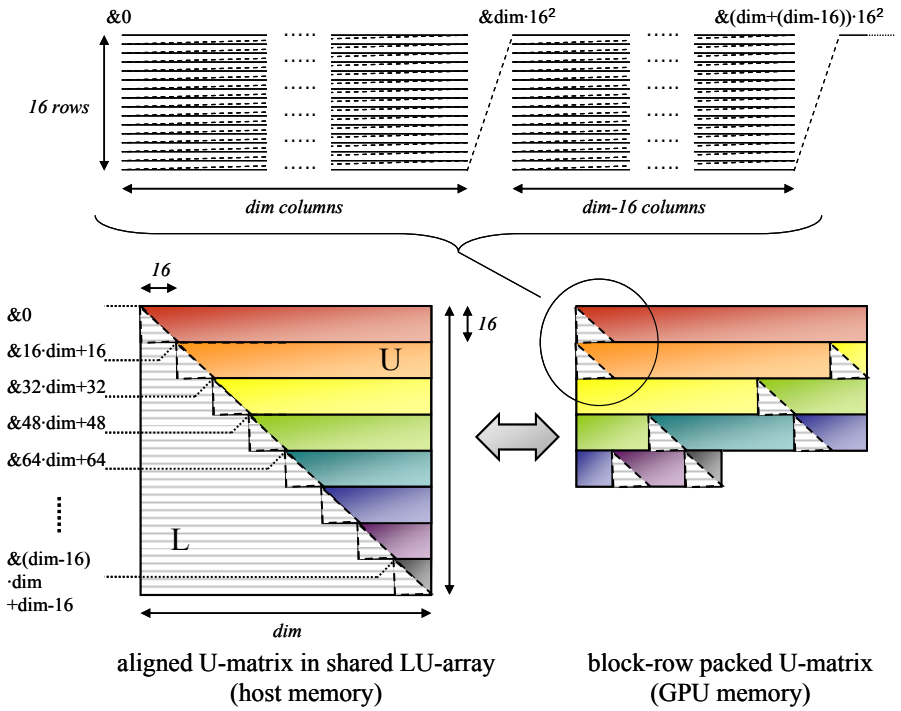


Figure 47: Block row packed storage of the U-matrix, analogous to L-matrix storage but starting with the longest block row.

We can derive that the total storage requirements on the GPU memory therefore amount to $(dim+16)\cdot dim/2$ elements. The block-packed data still contains some elements from across the diagonal, in total a number of $7\cdot dim$ elements.

Note that there is also the need for a buffer in GPU memory which temporarily holds data to prevent racing conditions. The buffer occupies up to $dim^2/4$ elements, an additional ~50% of storage occupation on the GPU.

4.3.3 Address generation

Since data are no longer aligned with the matrix dimensions, accessing a particular element in packed data storage involves an increased effort in address generation. Precisely the following parameters require more complex computation compared to aligned storage:

1. offset of the block row is now following a non-linear function based on block row index
2. pitch inside block row is no longer constant but linearly connected to block row index

Both compute differently for L and U matrices.

Inside the GPU kernels, every sub-block needs to access three sub-matrices to perform the operations described in Chapter 4.2: the square block *A* and the prior inverted triangles *B* and *C*. These blocks usually cover several block rows as shown in Figure 48:

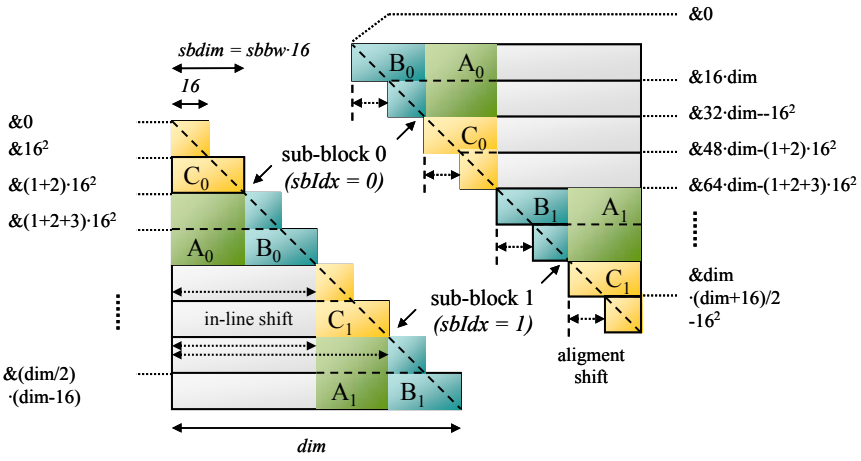


Figure 48: Sub-matrices *A*, *B* and *C* of two sub-blocks both in a L and U matrix. Addresses indicate the offset of the corresponding block row in block-packed storage. In this example, sub-block dimensions (*sbdim*) are 32, matrix dimensions (*dim*) are 128.

As for the block row offsets, the following formulas can be derived:

- $row\_offset_L = 16^2 \cdot (1+2+3+\ldots+r) = 16^2 \cdot (r+1) \cdot r/2$ *elements*
- $row\_offset_U = 16^2 \cdot ((dim/16+(dim/16-1)+\ldots+1) - (1+2+3+\ldots+(dim/16-r))) = 16^2 \cdot ((dim/16+1) \cdot (dim/16)/2 - (dim/16-r+1) \cdot (dim/16-r)/2)$ *elements*

where *r* denotes the index of the block row. The pitch inside a block row linearly depends on the block row index:

- $pitch_L = 16 \cdot (r+1)$
- $pitch_U = dim - (16 \cdot r)$

Given this, we can easily access any 16×16 block with block-coordinates (*r*,*c*) in packed storage:

- *local_block_r_c[y][x] = packed_storage[row_offset + y·pitch + c·16 + x]*

where *y* and *x* are the row and column coordinates of the elements inside the block.

So, to access the sub-matrices *A*, *B* and *C,* the rows inside those sub-blocks (sbr_A, sbr_B and sbr_C) need first to be mapped onto the global block rows (r_A, r_B and r_C). This calculation is based on the index (*sbIdx*, s. Figure 48) and block-width of the sub-block (*sbbw)*. The dimensions of all sub-matrices *A*, *B* and *C* in step *i* are:

- $sbdim = m \cdot 2^{i-1}$

Conclusively, *sbbw* amounts to the sub-block dimensions divided by the block width 16:

- $sbbw = m \cdot 2^{i-1}/16$

These variables allow a quite succinct formulation of how to calculate the global block rows r_A, r_B and r_C:

- L-matrix:
  - $r_A = sbr_A + (1+sbIdx) \cdot sbbw$

- $r_B = sbr_B + sbIdx \cdot sbbw$
- $r_C = sbr_C + sbIdx \cdot sbbw$

- U-matrix:
  - $r_A = sbr_A + sbIdx \cdot sbbw$
  - $r_B = sbr_B + sbIdx \cdot sbbw$
  - $r_C = sbr_C + (1+sbIdx) \cdot sbbw$

These allow us to calculate the aforementioned block row offset, but an additional in-line shift (s. Chapter 4.3.3) needs to be inserted to reach the correct block position. For the L-matrix, these are:

- $shift_A = 16 \cdot 2 \cdot sbIdx \cdot sbbw$
- $shift_B = 16 \cdot 2 \cdot sbIdx \cdot sbbw$
- $shift_C = 16 \cdot 2 \cdot sbIdx \cdot sbbw$.

For the U-matrix, the *B* and *C* matrices are lined up at the beginning of block rows, so no in-line shift is necessary. However, a negative alignment shift is included to account for the "cut-out" pieces in the triangular sub-matrices:

- $shift_A = 16 \cdot (2 \cdot (1+sbIdx) \cdot sbbw) - 16 \cdot sbr_A$
- $shift_B = - 16 \cdot sbr_B$
- $shift_C = - 16 \cdot sbr_C$

The shift basically maps the sub-block column *sbc* to the global block column *c*.

- $c = sbc + shift/16$

In summary, we obtain the base address of the beginning of the block row in any *A*, *B* or *C* sub-matrix by adding the shift to the global block row offset:

1. $addr\_offset = row\_offset + shift$

Note that both *addr_offset* and *pitch* depend only on the sub-block index *sbIdx* and the sub-block row *sbr*. Now we can obtain any 16×16 block in sub-block row *sbr* and sub-block column *sbc* using the pre-calculated *addr_offset* and *pitch*:

2. $local\_block\_sbr\_sbc[y][x] = packed\_storage[addr\_offset + y \cdot pitch + sbc \cdot 16 + x]$

Running through the whole addressing procedure whenever an element in device memory is accessed would produce an unacceptable overhead, particularly as the GPU has only limited capabilities when performing integer operations [85]. This is avoided by storing as much pre-calculated information in the GPU constant memory as possible. Threads can retrieve data from the constant memory at both high bandwidth and low latency, especially when all threads in a warp are accessing the same piece of data.

For each computation step *i*, we prepare and upload three look-up tables (LUT) containing the address offset (*addr_offse*t) and address pitch (*pitch*) for every sub-block row of *A*, *B* and *C* in every sub-block.

Given the offset and pitch from the LUT, only a minor effort is required to complete the address generation of any element with the individual addressing inside the block row. This reduces the run-time address generation overhead to a level comparable to that usually produced when using a non-packed format.

4.3.4 Allocation flow

The input data are prepared in the form of a shared LU array in a shared buffer in host memory. This buffer can be allocated in page-locked memory, which greatly increases transfer speed to and from GPU memory from ~1.0 GB/s to ~3.3 GB/s over a 16x PCIe 2.0 port. However, the amount of memory that can be allocated that way is limited by the kernel's memory region. In our system, we have to switch to pageable buffers for matrix dimensions greater than 8192. With matrix dimensions that large, the host-device transfers constitute a relatively small amount of the total execution time (<7%), so the resulting decline in performance is acceptable.

Once the input buffer is prepared and the CUDA runtime is initialized for both GPUs, two CPU threads are forked. One processes the lower and one the upper triangular matrix, which comprises the following work steps:

1. Invert the diagonal sub-blocks of the lower/upper triangular matrix in host memory.
2. Create a buffer of size $(dim+16)\cdot dim/2 + dim^2/4$ in GPU device memory to store the triangular matrix and temporary working data. Also create a page-locked write combining buffer in host memory to prepare the address LUTs in for upload to the GPU constant memory.
3. Upload the lower/upper triangular matrix to the buffer in block-packed format, as described in Chapter 4.3.2
4. Until the end of recursion is reached, calculate the address LUTs according to Chapter 4.3.3, upload them to GPU constant cache, and run the L-/U- RTMI kernels described in Chapter 4.3.1.
5. Download the now fully inverted lower/upper triangular matrix from block-packed format to the shared array buffer.
6. Free GPU resources and LUT buffer and join threads.

The data and work flow of this procedure are visualized in Figure 49.

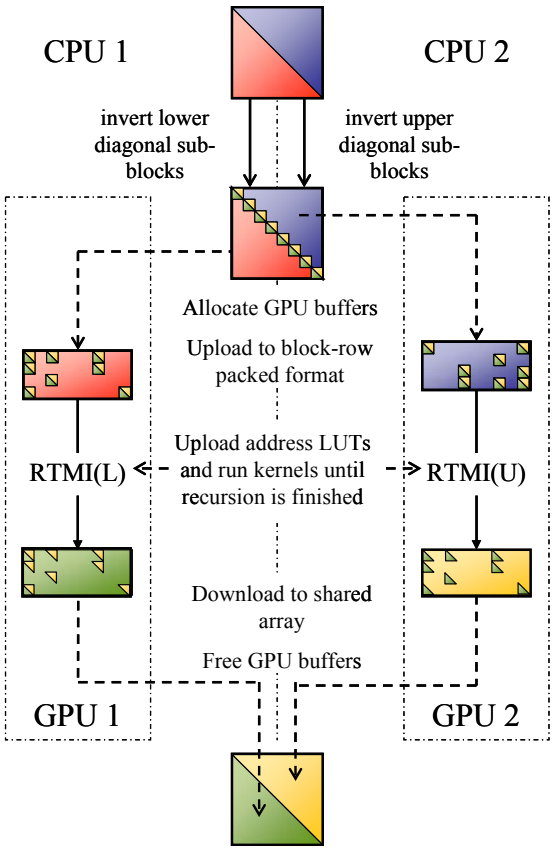


Figure 49: Application data and work flow. Red and blue are the L and U-mtatrices while green and yellow represent their inverted counterparts.

4.4 Limitations and generalizations

Due to the restrictions introduced by both the algorithm and the implementation, the inversion routine is directly applicable only to matrices of dimensions $dim = m \cdot 2^k$, where k and m are positive integers and m is additionally a multiple of *16*. If, for example, we require a $k \geq 5$, our applicable matrix dimensions are bound to be a multiple of *512*. Some dimensions allow higher values of k, and the highest applicable value should be applied for best performance.

The maximal value of *dim* is restricted by the on-board memory size of the graphics card. The GTX 295 has a total of 1.792GB, divided evenly over both GPUs. With the temporary data buffer, the maximum matrix dimensions on both boards amount to 11776.

To adapt any non-aligned matrix so as to be processed by the routine, the matrix needs to be "padded" to artificially increase the dimensions to an eligible value. The dimensions of a square matrix can be inflated by appending the unity matrix of size *dimpad* - *dim* in the lower right or upper left corner (where *dimpad* is an eligible matrix size) and filling up the rest with zeros. The inverted padded matrix then contains the inverse of the initial matrix, as depicted in Figure 50.

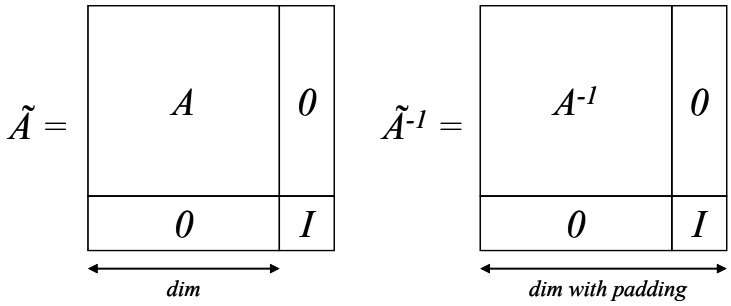


Figure 50: Insertion of data padding to increase the matrix dimensions up to a size eligible for the inversion routine. *I* refers to the unity matrix.

Data padding is the method of choice when the matrix size is only short of a few elements. The overhead grows polynomially with the number of padding rows and columns, according to the numerical complexity of the TMI.

When the actual matrix dimension is only slightly above an eligible size, it may be more efficient to "cut out" a smaller matrix of the relevant size and apply the routine to it, as shown in Figure 51. Using the same rules described as in Chapter 4.2, the remaining rows and/or columns are updated on the host side using the CPU.

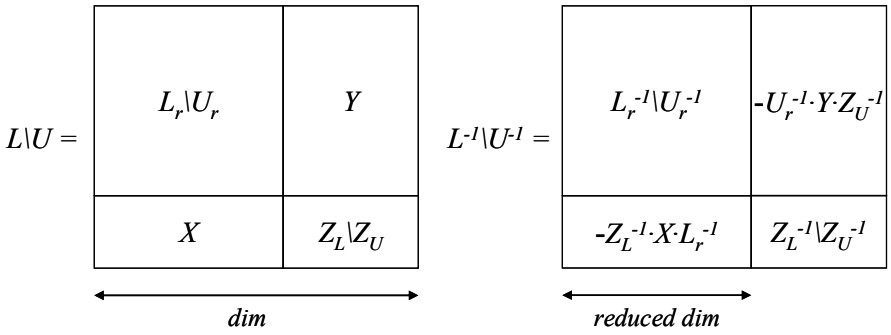


Figure 51: Size reduction to decrease the matrix dimensions down to a size eligible for the matrix inversion routine. This refers to a shared LU array. The *Z* sub-block is also a shared array of this type.

This turns out only to perform better than the padding method for a minority of configurations where the size simply needs to be reduced by a relatively small amount of rows and columns. Otherwise, data padding is preferred.

4.5 Benchmarking

We benchmarked our application on a contemporary PC desktop system, equipped with a commercial dual-GPU NVIDIA graphics card. The configuration of the benchmarking platform is as follows:

- Intel Core2 6300 CPU @ 1.86 GHz
- NVIDIA GTX 295 graphics card
- 4 GB RAM

The performance results were compared its performance to that of a CPU-only reference. This reference application is based on the free FORTRAN library of LAPACK, which offers highly optimized serial implementations of common linear algebra algorithms [139]. To further exploit the capabilities of modern CPUs, we use the LAPACK routines in concert with OpenMP so as simultaneously to invert the lower and upper triangular matrix. The reference application was run on an Intel Core2 quad Q6600, clocked at 2.4 GHz, in a system with 4 GB of RAM; a much faster model than in the GPU benchmarking platform.

We start by comparing the raw execution times of both our application and the reference. The measurements include all set-up and clean-up times save runtime initialization. To be precise, the application benchmarks include the following operations:

- Thread forking
- Memory allocation
- Data upload
- Pre-calculation of the diagonal sub-blocks
- Address LUT calculation and upload
- RTMI kernels
- Data download
- Memory release
- Thread joining

We present benchmarks for all problem sizes that comply with a minimal segmentation depth of $k = 5$, higher values of k being applied where possible.

The scaling of execution times is visualized in Figure 52.

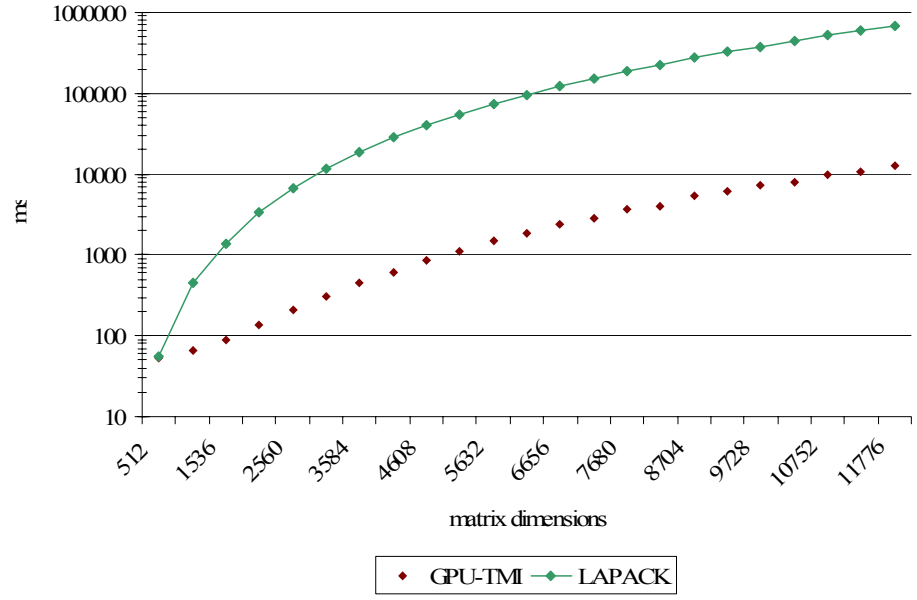


Figure 52: Execution times of this application and the LAPACK based dual-CPU reference in milliseconds (double precision).

We obtain a speedup of up to 57x compared to the LAPACK based dual-CPU reference application.

Rather than the raw execution times, the throughput in terms of floating point operations per second is often considered a more expressive measure for the quality of dense linear algebra implementations as it gives insight in how well the hardware is utilized. We calculate this from the minimum amount of floating point operations required to process the given problem size, divided by the total execution time. The scaling of the throughput is visualized in Figure 53.

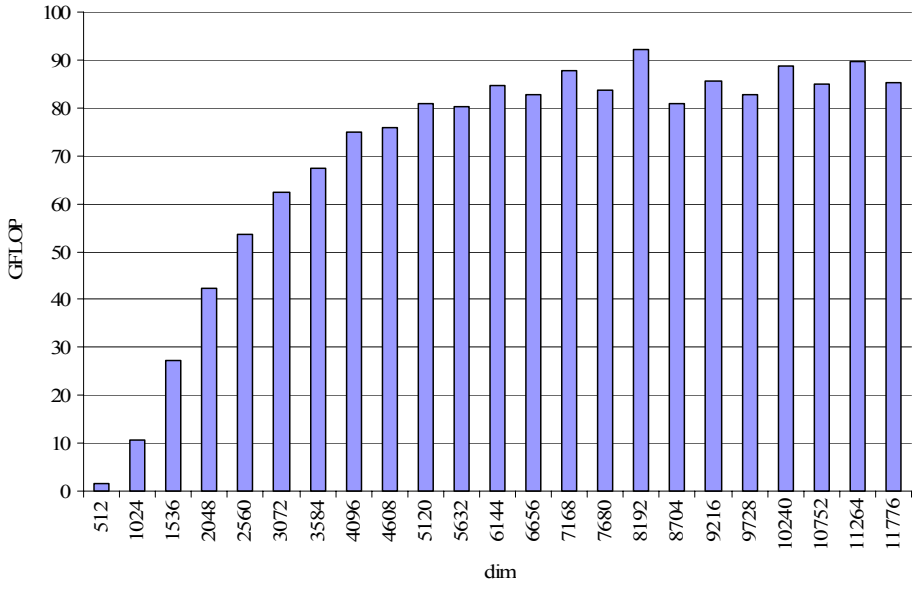


Figure 53: Computational throughput of our application in GFLOPS (double precision).

As can be seen, a problem size of *dim = 5120* or greater is required for the application to reach its full potential of over 80 GFLOPS, whereas a peak performance of up to 92 GFLOPS is achieved. Furthermore, it can be observed that problem sizes that allow a higher segmentation depth (parameter *k*) are generally processed slightly more efficiently. This accounts for the "bumpy" shape of Figure 53 in the saturation region.

The memory requirements per GPU device memory for any given problem size of both the new version using block-packed storage and the old version using naively aligned storage are displayed in Figure 54.

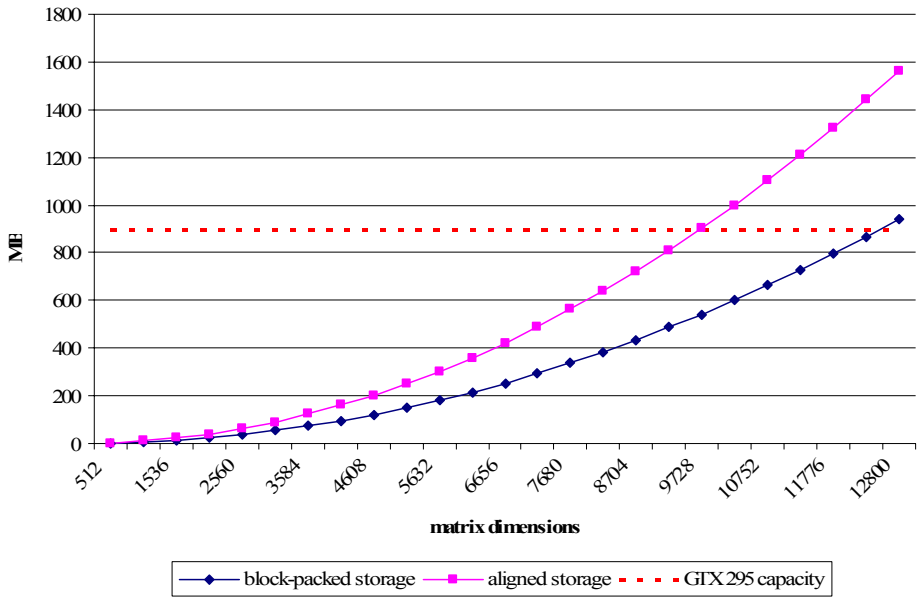


Figure 54: Memory requirements per GPU device of the TMI routine using memory of block-packed storage (this version) and naively aligned storage (old version [9]) in megabytes. Red line indicates the memory capacity of the NVIDIA GTX295 (896 MB per GPU).

We see that the memory savings due to block-packed storage are around 40% for all problem sizes.

4.6 Discussion

When measuring the performance of our application, we exclusively focused on double precision computations, which are usually required by scientific applications. To put the results from our benchmarks into perspective, it is important to note that double precision support was introduced in the GeForce 200 series GPU as an add-on, with the bulk of the arithmetic still focused on single precision calculations (s. Chapter 2.4.3). However, we resisted the temptation to present a single precision benchmark for an application that typically requires double precision, especially for the relatively large problem sizes discussed in this work.

To evaluate the quality of our implementation, a comparison with the theoretical peak throughput of the hardware is helpful. The two GPUs of the GTX 295 add their theoretical peak throughput up to 149 GFLOPS. A peak performance of 92 GFLOPS is obtained by our

application, though as stated in Chapter 4.5, this number is based on execution times that include all kind of overhead from various sources. On top of this, part of the computation, namely the initial "step 0", is performed on the much weaker CPU. When we only consider kernel execution times and the work load of the recursive part of the algorithm, we can gauge the throughput of the isolated GPU kernels as being up to 70% of the theoretical peak, a satisfying result considering the inevitable slow-down factors described in Chapter 4.3.1.

The TMI implementation is subject to a few limitations, introduced from both the RTMI algorithm and the hardware, that need to be addressed for the routine to be of any practical value. Only input data that constitute a problem size aligned to $2^k \cdot m$, with k being positive and m being a positive multiple of 16, can be processed without prior modification. Otherwise, the input array has to be altered according to Chapter 4.4. Not only does the "padding" data reorganization create some minor offset of its own (≤ 1% of total computation time), it also inflates the problem size by up to hundreds of elements. Moreover, the higher the desired segmentation depth, the more the problem size needs to be artificially increased. The complexity grows according to the 3rd order polynomial mentioned in Chapter 4.2. While higher segmentation depth increases the share of computation processed on the GPU and therefore also overall performance, cutting back on this parameter decreases the intervals between aligned problem sizes, and potentially the amount of necessary padding. In the benchmarks presented in Chapter 4.5 we took a minimum segmentation depth of $k \geq 5$, so over 96% of the triangular matrices are inverted on the GPUs, at the cost of having intervals as large as 512 elements between eligible problem sizes. For these intervals, the *worst case* increment in computation time amounts to up to ~15% for large problem sizes ($dim \approx 11000$), up to ~23% for medium problem sizes ($dim \approx 7200$), but up to ~60% for small problem sizes ($dim \approx 3100$), where the 512 interval makes a large fraction of the problem size. There is always a tradeoff between the benefits of higher segmentation depth and the overhead due to padding. In general, the artificial increase in problem size should be kept to a small percentage of the actual problem size by reducing the segmentation depth accordingly.

5 Multi-GPU accelerated complex Bi-Conjugate Gradient solver

This chapter documents the second of the three major contributions of this thesis: a numerical solver for large non-hermitian linear systems, accelerated by multiple GPUs. In contrast to Chapter 4, it addresses a typical problem from sparse linear algebra. It is implemented on the desktop supercomputer described in Chapter 2.5.5. Both the GTX 295 and the GTX 480 were used.

In scientific computing, physical scenarios are often described in the form of partial differential equations (PDEs), analytical solution of which is rarely feasible. Such problems are commonly reformulated by discretization of the domain using a finite element, finite difference or finite volume approach, which produces a sparse linear system to be solved.

The resulting system can be very large which inflates complexity of a direct solution, making iterative methods appealing. Among these methods, Krylov subspace methods account for some of the most prominent solvers. Recently, tapping the capabilities of cheap graphics hardware to accelerate linear solvers has become increasingly popular; in particular, notable effort has been devoted to the Conjugate Gradient method (CG) [98][140][141]. CG is widely used to solve positive-definite symmetric systems, as produced by elliptic PDE problems.

In complex number space, the linear system needs to be hermitian for CG to work. Unfortunately, real-world problems still tend to produce symmetric (hence non-hermitian) systems, so a complex CG solver has little practical value. Hence, the complex Bi-Conjugate Gradient (BiCG) method [28] is appealing, in that it does not incur such limitations. Among the methods we tested, it showed the fastest convergence behavior for our specific problem, which emerges from the field of medical imaging.

In this chapter, I present a linear solver for structured non-hermitian systems, based on the Bi-Conjugate Gradient method and implemented on a multi-GPU system. This implementation combines several established optimization techniques, but also exploits the specific structure of the algorithm in new ways to further push the limits and outperform more straightforward implementations. I present benchmarking results of both our multi-GPU accelerated solver and an equally optimized reference relying solely on multi-core CPUs. Where possible, I also try to relate our results to similar work from the literature. Finally, an analytical model to predict the limits of multi-GPU scalability is included.

5.1 Motivation and background

This implementation was created as part of a software framework for EIT forward problem simulation [4]. This framework includes a complete flow to simulate injected current flows through the head tissue and electrode contacts, respecting as much anatomical detail as possible. Not only does it use a very high resolution (~1 mm^3), it also includes tissue anisotropy and uses a realistic electrode model [162]. An example of what is produced by such a simulation is shown in Figure 55.

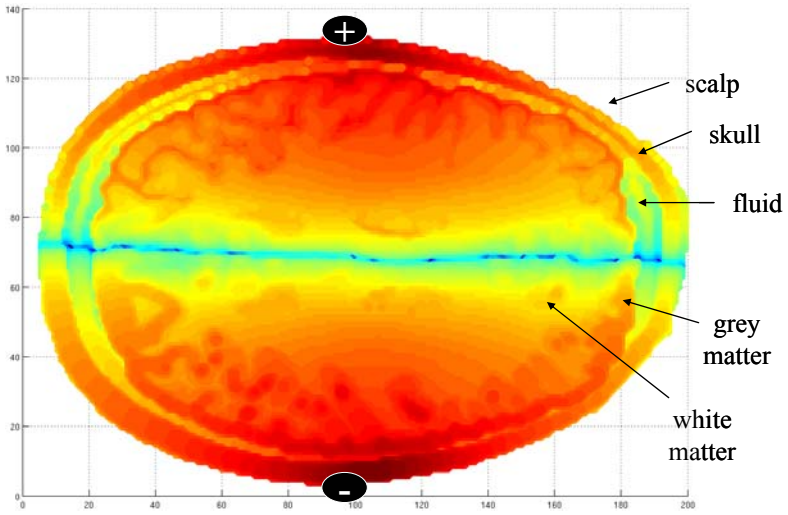


Figure 55: Mapping of potentials after current injection, as produced by the EIT forward problem simulation framework. The color map refers to a logarithmic scale.

The core of the simulation framework is based on a high-resolution anisotropic head model (s. Chapter 3.2.3). The basic concept of this model is a representation of the subject's head and electrode cap as a large regular impedance network, which is obtained by discretizing the Poisson equation

$$\nabla(\varepsilon \cdot \nabla\varphi) = 0 \tag{7}$$

with Neumann boundary condition $\varepsilon \cdot \nabla\varphi \cdot n = j_k$ under electrode k and $\varepsilon \cdot \nabla\varphi \cdot n = 0$ elsewhere on the scalp (s . Chapter 3.2.2.1) . Discretization of (7) is done using the Finite Volume approach described in [27]. But unlike other bio-electrical imaging methods, EIT gathers information from measurements at different frequencies, so the permittivity ε is complex and likewise the solution to the Poisson equation [33].

The main source of individual anatomical information is a Magnetic Resonance Image (MRI) of the subject. A MRI image consists in a 3D image volume of several million voxels, which forms the base of our regular FVM grid. Based on associated dielectric parameters from additional segmentation software and measurement tables, a regular impedance network is built, aligned to the MRI voxels [27].

For a typical MRI, the network contains several million nodes. Similar to nodal circuit analysis, the volume conductor model can be described as a linear system

$$\boldsymbol{A} \cdot \boldsymbol{\varphi} = \boldsymbol{i} \tag{8}$$

where $\boldsymbol{A} \in C^{n \times n}$ is the admittance matrix, $\boldsymbol{\varphi} \in C^n$ is the potentials at each node and $\boldsymbol{i} \in C^n$ is the corresponding currents derived from the boundary conditions.

Simulating current flows in the head model implies repeated solution of said linear system on large sets of input data. For problem sizes with millions of nodes, solution of such systems can take several minutes on conventional desktop systems. The following chapters will show that with multi-GPU acceleration, this process can be sped up substantially, making simulation at this level of detail feasible in the first place [3].

5.2 Related work

Some contributions are available that discuss the general problem of solving non-hermitian systems using parallel architectures, notably [142] and [143], but without presenting GPU-accelerated implementations. On the other hand, there is other work concerning the GPU implementation of sparse linear solvers [140][141] and even the Bi-Conjugate Gradient method [144], but none of these implementation supports complex coefficients.

Only recently, the inclusion of GPU-acceleration into the PETSc framework has been initiated, which indeed features a BiCG solver supporting complex numbers. The implementation however is still at a preliminary stage [145] and no performance analysis has been published yet.

Articles discussing multi-GPU distribution of linear solvers are also still sparse. The most relevant one is [147] which treats domain distribution of a regular grid, however it lacks an analytical discussion of multi-GPU scalability like it is presented in our work.

Leaving the field of complete solver implementations, one might argue that BiCG is composed mostly of basic subroutines, of which there are optimized GPU-implementations available, like the NVIDIA-supported CUBLAS library [148]. Furthermore, free routines are available for sparse matrix-vector products, including banded matrices [146]. I will discuss the limits of a straightforward subroutine-based implementation in Chapter 5.5.5.

5.3 The complex bi-conjugate gradient method

The following chapter will introduce the algorithm applied to solve the non-hermitian linear systems produced by the EIT forward problem environment (s. Chapter 5.1), namely the complex bi-conjugate gradient method [28].

Consider a linear system of the form

$$\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b} \tag{9}$$

where **A** is the sparse coefficient matrix, **x** is the vector of unknowns and **b** is the right hand side vector. Provided that **A** is positive definite, the linear system (9) can be solved by BiCG via the following steps:

Choose $\boldsymbol{x}_0$ (usually **0** if no better initial guess is available)

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\cdot\boldsymbol{x}_0$

Choose $\boldsymbol{r'}_0 = \boldsymbol{r}_0$

while $(|\boldsymbol{r}_i| > \varepsilon\cdot|\boldsymbol{r}_0|)$

$\boldsymbol{z}_i = \boldsymbol{M}\cdot\boldsymbol{r}_{i-1}$

$\boldsymbol{z'}_i = \boldsymbol{M}^*\cdot\boldsymbol{r'}_{i-1}$

$\rho_i = \boldsymbol{z}_i^*\cdot\boldsymbol{r'}_{i-1}$

$\boldsymbol{p}_i = \boldsymbol{z}_i + (\rho_i/\rho_{i-1})\cdot\boldsymbol{p}_{i-1}$ // $\boldsymbol{p}_1 = \boldsymbol{z}_1$ *on first iteration*

$\boldsymbol{p'}_i = \boldsymbol{z'}_i + (\rho_i/\rho_{i-1})^*\cdot\boldsymbol{p'}_{i-1}$ // $\boldsymbol{p'}_1 = \boldsymbol{z'}_1$

$$\boldsymbol{q}_i = \boldsymbol{A}\cdot\boldsymbol{p}_i$$

$$\boldsymbol{q'}_i = \boldsymbol{A}^*\cdot\boldsymbol{p'}_i$$

$$\alpha_i = \rho_i / \boldsymbol{p'}^*_i\cdot\boldsymbol{q}_i$$

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha_i\cdot\boldsymbol{p}_i$$

$$\boldsymbol{r}_i = \boldsymbol{r}_{i-1} - \alpha_i\cdot\boldsymbol{q}_i$$

$$\boldsymbol{r'}_i = \boldsymbol{r'}_{i-1} - \alpha_i^*\cdot\boldsymbol{q'}_i$$

where i indicates the iteration index. $\boldsymbol{M}$ is the preconditioner which is not necessarily implemented as a matrix-vector multiplication, it can also constitute a two-phase solution of incomplete triangular factors. Preconditioning actually substitutes the linear system (9) with

$$\boldsymbol{M}\cdot\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{M}\cdot\boldsymbol{b}.$$

Provided that $\boldsymbol{M}\cdot\boldsymbol{A}$ has a smaller spectral radius than $\boldsymbol{A}$, introduction of the preconditioner reduces the number of iterations to convergence. However, the preconditioning step introduces an additional work load at each iteration, so it must be light enough not to undo the advantage gained from quicker convergence. In this work, only simple Jacobi preconditioning was used: the linear systems produced by the target application are very well-conditioned, thus it outperformed more complex procedures like [147] for our specific application while still converging to acceptably precise solutions (s. Chapter 5.5.7).

5.4 Implementation

5.4.1 Computational kernels

As described in Chapter 2.6, implementation was done using the CUDA drivers and software design kit for NVIDIA GPUs [85]. CPU multithreading was controlled using the free Linux library of OpenMP [132]. A dedicated CPU thread is spawned using OpenMP to handle each GPU. Synchronization between the data streams is necessary on several occasions to ensure data consistency. Also, some minor parts of the algorithm are best left to the CPU. Within these constraints we strive to split computation into as few GPU kernels as possible. This saves kernel launching overhead and helps maximize GPU occupancy and data locality. Mapping of the algorithm from Chapter 5.3 into computational kernels on both CPU and GPU, along with corresponding data I/O, is outlined in Figure 56.

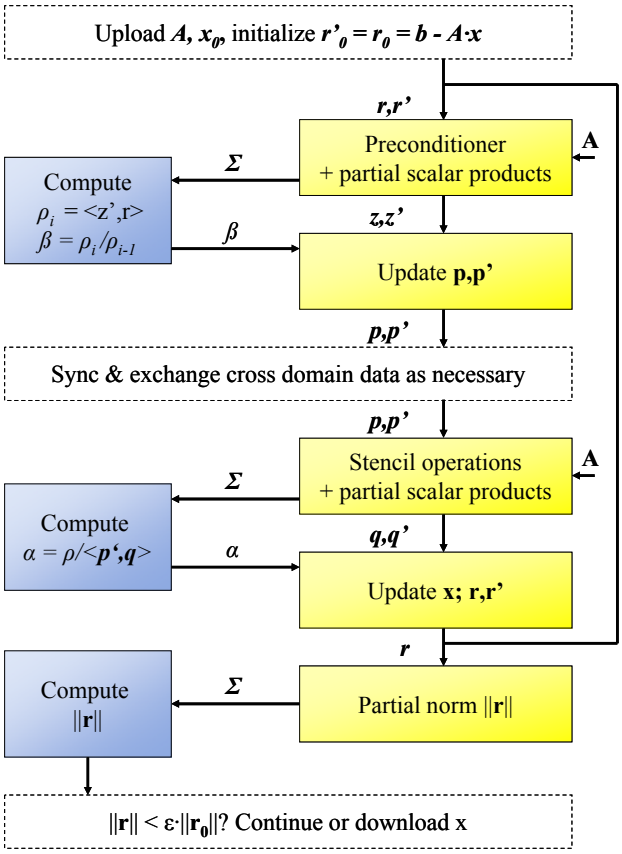


Figure 56: Outline of the computation flow of our complex BiCG solver in a CPU/GPU system. Dark grey kernels (left) run on CPU, light grey kernels (right) run on GPU. Arrows between left and right indicate memory transfers between RAM and GPU device memories.

Details about these kernels are described in what follows.

5.4.1.1 Stencil operations

The term "stencil operation" refers to a linear mapping that sums data from both a local node and its defined neighbors, covered by the so-called "stencil" (s. Figure 57).

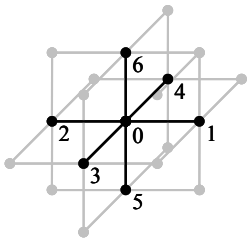


Figure 57: Node with its 6 neighbors in a regular grid, also called a 7-point stencil.

This can be described as a sparse matrix-vector product (spMV), and in the case of a BiCG it is requested by the operations

$$q_i = A \cdot p_i$$

$$q'_i = A^* \cdot p'_i$$

Sparse matrices are usually stored in a compressed format to reduce memory usage by skipping as many zero elements as possible. Efficiency of this storage depends on the

regularity patterns of the matrix, and not all formats are equally fit for GPU processing [146]. For regular grids however, we can apply the GPU-friendly diagonal format, which is visualized in Figure 58.

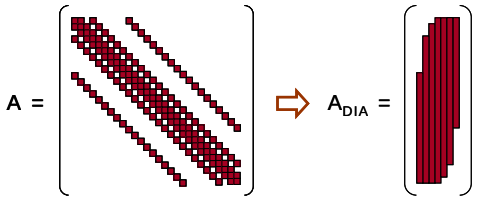


Figure 58: Diagonal format for sparse banded matrices.

Address offsets between neighboring nodes are pre-calculated and stored in the GPU constant cache to save the effort of calculating them at run-time.

The regularity of our grid permits us furthermore to apply the techniques proposed in [149], where data are processed in parallel inside a horizontal plane which is moving downwards along the z-axis, as visualized in Figure 59.

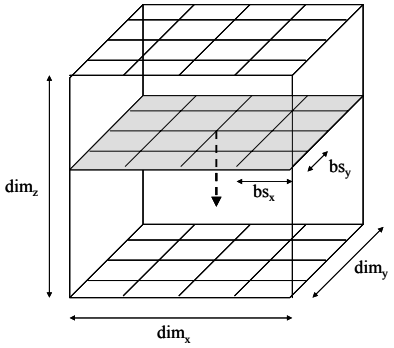


Figure 59: Processing a 3D grid using a sliding horizontal plane of parallel threads (highlighted in grey) , tiled into blocks of $bs_x \times bs_y$ elements.

The plane is composed of a grid of square thread block tiles, where tiling into $bs_x \times bs_y = 16 \times 16$ blocks was found to offer the highest throughput. This approach enables efficient use of caching, as each thread block experiences a large amount of data redundancy.

5.4.1.2 Preconditioning

The preconditioning step comprises the operations

$$\boldsymbol{z}_i = \boldsymbol{M} \cdot \boldsymbol{r}_{i-1}$$

$$\boldsymbol{z'}_i = \boldsymbol{M}^* \cdot \boldsymbol{r'}_{i-1}$$

For the simple Jacobi preconditioning used in this work, the coefficients of M are defined as:

$$m_{ij} = 1/a_{ij} \text{ if } i==j, \ 0 \text{ otherwise}$$

The operations are again processed using the technique presented in Figure 59.

5.4.1.3 Scalar products and residual norm

Due to their communication intensive nature and their high data-to-arithmetic ratio, scalar products are potential bottlenecks in this algorithm. For both the preconditioning and the stencil operations, a part of the ensuing scalar product is merged into the same kernel, reusing the operands (**z*·r'** for the preconditioning step, **p'*·q** for the stencil operation) while still in on-chip memory. The partial scalar products cover the $bs_x \times bs_y \times dim_z$ elements of the same thread block. These sub-products are written back to vector $\boldsymbol{\Sigma}$ of length $(dim_x/bs_x) \times (dim_y/bs_y)$, which is downloaded to host memory and reduced to the final value on CPU. This heterogeneous approach leaves only the massively parallel part to the GPU and reduces communication between nodes to the part inside the same thread block, where it is cheap thanks to the on-chip shared memory. Furthermore it saves additional kernel calls and comprised off-chip memory accesses.

The global norm of the residual is computed analogously, but need not to be computed at every iteration; overshooting the instant of convergence by a few iterations is usually well worth the reduction in computation cost. It is processed in the same way as the scalar products.

5.4.1.4 Update functions

All update functions are naively parallel vector operations. The updates for solution and residual

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha_i \cdot \boldsymbol{p}_i; \quad \boldsymbol{r}_i = \boldsymbol{r}_{i-1} - \alpha_i \cdot \boldsymbol{q}_i; \quad \boldsymbol{r'}_i = \boldsymbol{r'}_{i-1} - \alpha_i^* \cdot \boldsymbol{q'}_i$$

are unified into a single kernel to improve occupancy of the GPU. Likewise are the functions

$$\boldsymbol{p}_i = \boldsymbol{z}_i + (\rho_i/\rho_{i-1}) \cdot \boldsymbol{p}_{i-1}; \quad \boldsymbol{p'}_i = \boldsymbol{z'}_i + (\rho_i/\rho_{i-1})^* \cdot \boldsymbol{p'}_{i-1}$$

5.4.2 Domain distribution

Multi-GPU accelerated desktop systems are based on a complex hierarchy of distributed and shared memory, which makes it challenging to fully utilize the available processing power. In general, applications that are load-unbalanced, bandwidth bound or communication-intensive tend to perform poorly on such systems. Hence, load balancing and data traffic optimization are imperative for good performance.

The grid is segmented into equal compartments which are divided among the GPU boards. The solver is then run simultaneously on all GPUs for the corresponding fraction of the problem size. For best performance, data should be decomposed in such a way that the compartments cover coherent memory sections. In a regular grid, this corresponds to horizontal slabs similar to the approach chosen by [147]. Apart from this being easier to set up, data on "seam planes" are also coherent in memory and can be swapped between GPUs in two high-speed memory transfers: a single download from device 1 to host followed by a single upload from host to device 2 (currently, direct device-to-device transfers are not supported). Additional transfers as required by other decompositions would be penalized by the transfer initialization overhead [93].

This data-swapping is required during stencil operations, as part of the input operands lie beyond the domain assigned to the respective GPU. Thus a redundant layer has to be added to each GPU domain in order to hold the said cross-domain elements. Data exchange between the different memory domains is visualized in Figure 60.

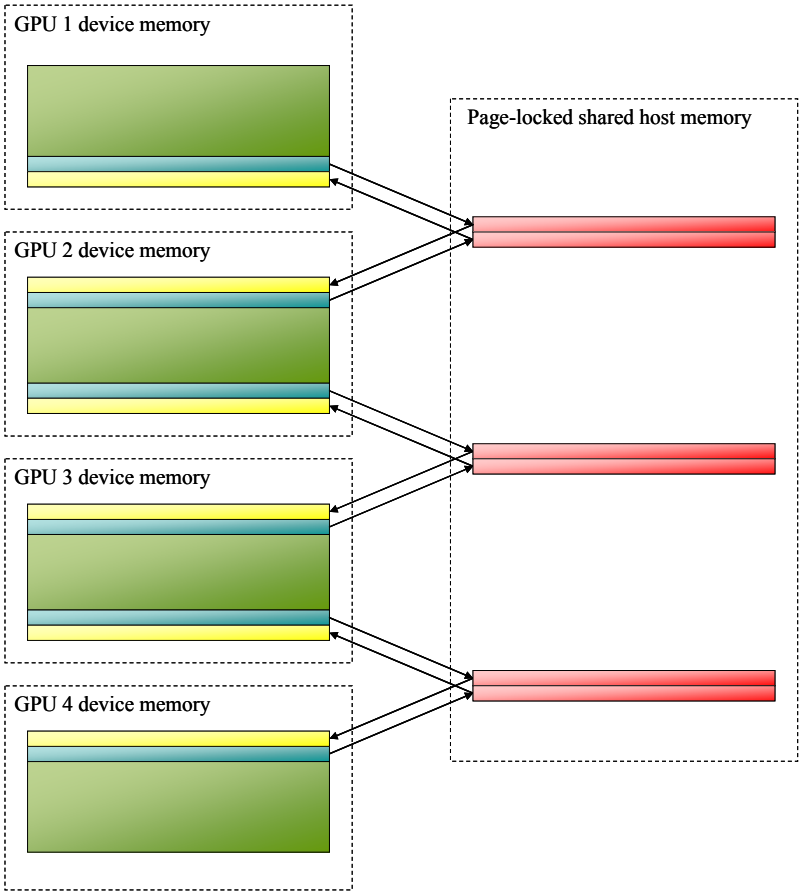


Figure 60: Data-swapping via page-locked shared host memory. Yellow sections in GPU memory indicate redundant layers.

The constant data exchange between GPU memory domains implies the risk of severely impeding performance. Fortunately, all the transfers indicated in Figure 60 can be overlapped with computation, so stalling can be avoided. This is done by segmenting some of the kernels in such a way that first the elements that need to be transferred are calculated; then the remaining vector is processed while the transfer is under way. On the receiving end, the procedure is inverted: elements not needing cross data are processed first while cross-domain data are being updated from the host-side exchange buffer. With the problem sizes treated in this work, latencies through data swapping can in this way be almost completely hidden (s. Chapter 2.6.3.2).

5.4.3 Maximizing bandwidth and instruction throughput

BiCG basically introduces a “mirror image” to all the vectors and vector-to-vector operations in CG; these are independent from each other and could be performed concurrently, essentially doubling the parallelism exposed by the method compared to CG. However, BiCG for the problem sizes treated by our application already offer millions of independent streams,

more than enough to utilize a multi-GPU system to full capacity. Instead, we use the particular structure of BiCG to increase the amount of local data reusage.

The traditional CG is inherently bandwidth limited as the total count of arithmetic is rather light for sparse systems. For complex BiCG, however, the arithmetic load per node and iteration is about 8 times higher at only 4 times the I/O. With the bandwidth optimizations described earlier, we actually reach the point where data feeding is no longer the only limiting factor and instruction throughput optimization gains higher importance.

The GPU's arithmetic units reach the highest instruction throughput when performing floating point multiply-add instructions, so the arithmetic flow should be cast as much as possible in this kind of operation. The typical operation

$$\boldsymbol{a} = \boldsymbol{a} + \boldsymbol{b} \cdot \boldsymbol{c}$$

$$\boldsymbol{a'} = \boldsymbol{a'} + \boldsymbol{b}^* \cdot \boldsymbol{c'}$$

is processed by the following routine:

$$BIMAC(\boldsymbol{a}_b, \boldsymbol{b}, \boldsymbol{c}_b)$$

$$\{$$

$$a_x \mathrel{+}= b_x \cdot c_x$$

$$a_x \mathrel{+}= -b_y \cdot c_y$$

$$a_y \mathrel{+}= b_x \cdot c_y$$

$$a_y \mathrel{+}= b_y \cdot c_x$$

$$a_z \mathrel{+}= b_y \cdot c_w$$

$$a_z \mathrel{+}= b_x \cdot c_z$$

$$a_w \mathrel{+}= b_x \cdot c_w$$

$$a_w \mathrel{+}= -b_y \cdot c_z$$

$$\}$$

where $\boldsymbol{a}_b, \boldsymbol{b}, \boldsymbol{c}_b$ are coupled elements cast to GPU built-in vector data types:

- $\boldsymbol{a}_b = \{a_x, a_y, a_y, a_w\} = \{ Re(a), Im(a), Re(a'), Im(a') \}$
- $\boldsymbol{b} = \{b_x, b_y\} = \{ Re(b), Im(b) \}$
- $\boldsymbol{c}_b = \{c_x, c_y, c_y, c_w\} = \{ Re(c), Im(c), Re(c'), Im(c') \}$

This function only contains multiply-add instructions and two sign changes. It also increase data re-useage as $\boldsymbol{b}$ need only be loaded once.

5.5 Benchmarking

All performance analysis for the proposed solution is based on experimental measurements of execution times. This is the usual approach for GPU-based applications where the complexity and limited public documentation of the hardware make analytical elaboration extremely

difficult. Again following custom and practice, we compare the GPU-accelerated solution to a reference implementation on a contemporary desktop machine.

To make this comparison as fair as possible, the reference implementation must be reasonably optimized. Ours is based on lightweight OpenMP multithreading [132]. It is run on the same system as introduced in Chapter 2.5.5 using two quad-core CPUs with Streaming SIMD Extensions (SSE).

5.5.1 Single-GPU performance

This benchmark focuses on computing performance, so we exclude most problem-related factors by comparing processing throughput in terms of million nodes per second (Mnodes/s). We compare several problem sizes emerging from typical MRI resolutions, up to 192^3 nodes in the single precision and up to 128^3 nodes in the double precision benchmark. Unfortunately, larger setups exceed the memory capabilities of a single GPU. Comparative benchmarks for the single and double precision dual-CPU reference and single-GPU implementation are shown in Figure 61.

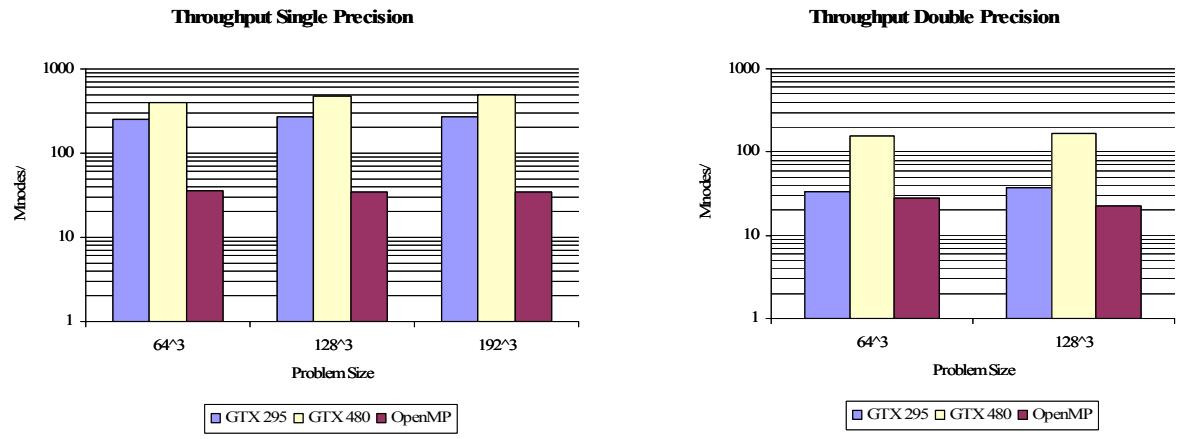


Figure 61: Single-GPU and dual-CPU solver throughput in terms of Mnodes/s. Analysis was done on cubic grids derived from common MRI resolutions.

For the benchmarks shown in Figure 61, the solver was accelerated using a single GPU of the GTX 296 and the GTX 480 type. For problem sizes in the order of millions of nodes, the GPU is "saturated", so thread launching overheads and load misbalancing between SMs no longer have any significant impact. For the larger problems that we tested, the GTX 295 outperforms the dual-CPU reference by a factor of up to 8x in single precision and by a factor of up to 1.7x in double precision. A single GTX 480 even achieves over 14x in single precision and up to 7.5x in double precision. In general, for larger problems, better GPU performance could be observed.

5.5.2 Multi-GPU performance

Next, we examine performance scaling with multi-GPU parallelism. Figure 62 shows the performance scaling for the addition of up to four GPUs.

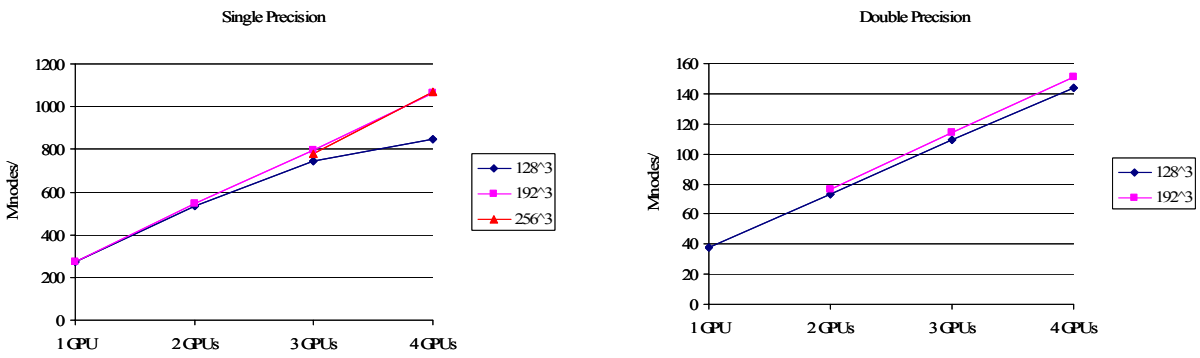


Figure 62: Performance scaling with number of GPUs, for both single and double precision and for different problem sizes.

The restructuring techniques described in Chapter 5.4.2 provide near loss-less multi-GPU scaling for almost all set-ups, with an overhead of just around 5% per added GPU for the largest tested problem size. Using all four GTX 295 GPUs we achieve total speedups vs. the dual-CPU reference of up to 31x in single precision and up to 7x in double precision. Using both GTX 480 GPUs, the speedup is 28x in single precision and 15x in double procision.

When speaking about performance scaling with parallelism, it might also be of interest how the CPU reference scales. A corresponding graph is shown in Figure 63 where the average throughput is measured in respect to OpenMP thread parallelism.

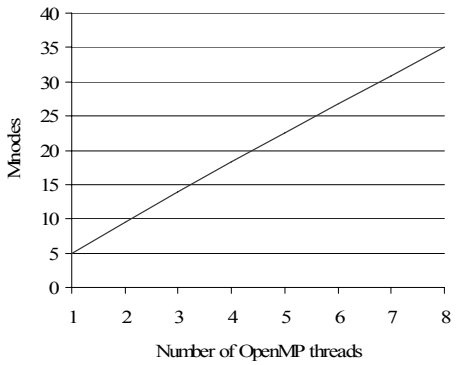


Figure 63: Performance scaling with the number of OpenMP threads of the dual-CPU reference application.

We see that on the way up from one to eight CPU cores, about 12% performance loss is observed compared to ideal scaling, more than what was lost in parallel multi-GPU execution.

5.5.3 Arithmetic throughput

For the reasons invoked in Chapter 5.5.1, we decided to measure performance of our solver in terms of million nodes per second. However, in linear algebra, especially for dense problems, it is also very common to measure the throughput of arithmetic operations in terms of billion floating point operations per second (GFLOP/s).

Fortunately, the structure of the algorithm allows a pretty straightforward conversion between both units: each iteration, 156 flops are required to process one node. Therefore, the

arithmetic throughput of a single GTX 295 amounts to around 42 GFLOP/s, while up to 167 GFLOP/s can be achieved using all four GPUs. A single GTX 480 achieves over 62 GFLOP/s, and up to 125 GFLOP/s can be observed when using both GPUs. Single-GPU double precision throughputs amount to 5.8 and 21.7 GFLOP/s per GPU, scaling up accordingly to 22.8 GFLOP/s on four GTX 295, and 43.3 GFLOP/s on two GTX 480.

5.5.4 Comparison to other work

While no exact counterpart exists in literature, comparison to similar applications can at least give an idea of the efficiency of the implementation. For example, for the concurrent number cruncher [140], a speedup of only 6x is reported compared to one CPU, though they were using an 8800 GTX graphics card, with a GPU roughly half as powerful as ours. The multi-algorithm solver from [144] claims to outperform a mono-core CPU reference by 20-25x, also using a 8800 GTX GPU. Note that our reference fully uses two CPUs, four cores each.

Rather than speedups versus CPU, the arithmetic throughput (s. Chapter 5.5.3) can be considered more unbiased for comparison. The CGS and BiCGStab solvers from [150] achieve peak throughputs of 16 and 18 GFLOP/s, the banded sparse matrix product from [146] achieved up to 36 GFLOP/s with caching, both using a slightly more powerful GPU model (a GTX 280) than ours. The benchmarks of [141] speak of 11.6 GFLOP/s on a single 8800 GTS, a device roughly 40% as powerful as our GTX 295, so putting this into perspective one might expect at most 30 GFLOP/s on a comparable GPU.

Looking at the performance reports of these solutions and taking into account the hardware they were using, we can summarize that our solver runs significantly more efficiently than related implementations.

5.5.5 Comparison to CUBLAS/CUSP

As mentioned in Chapter 5.2, there are optimized library routines for basic linear algebra operations, most notably the vendor-supported CUBLAS library that comes with the CUDA software design kit [87]. Moreover, there are optimized implementations for banded matrix-vector products [146], made public in the open-source CUSP (for “CUDA Sparse”) library. A simple single-GPU BiCG solver can be implemented depending only on these libraries. However, just putting together optimized subroutines does not yet result in an optimized program. Complex BiCG offers many ways to save overhead and data traffic by fusing computation on a low level (s. Chapters 5.4 and 5.4.3), and a considerable performance boost can be obtained by doing so. In fact, we observed a speedup of around 40% when comparing our solver to a simple CUBLAS/CUSP reference implementation. The gap was slightly stronger for smaller problem sizes than for larger ones, since the impact of having unnecessary launch overheads and synchronization decreases with problem size.

5.5.6 Profiling

As mentioned earlier in Chapter 5.4.1, some parts of the algorithm perform more efficiently than others. I used the vendor-supplied profiling tool, the NVIDIA Visual profiler, to gain

insight into the bottlenecks of this implementation. Figure 64 shows an analysis of the time share of each GPU kernel on the total execution time, kernel launching overhead comprised.

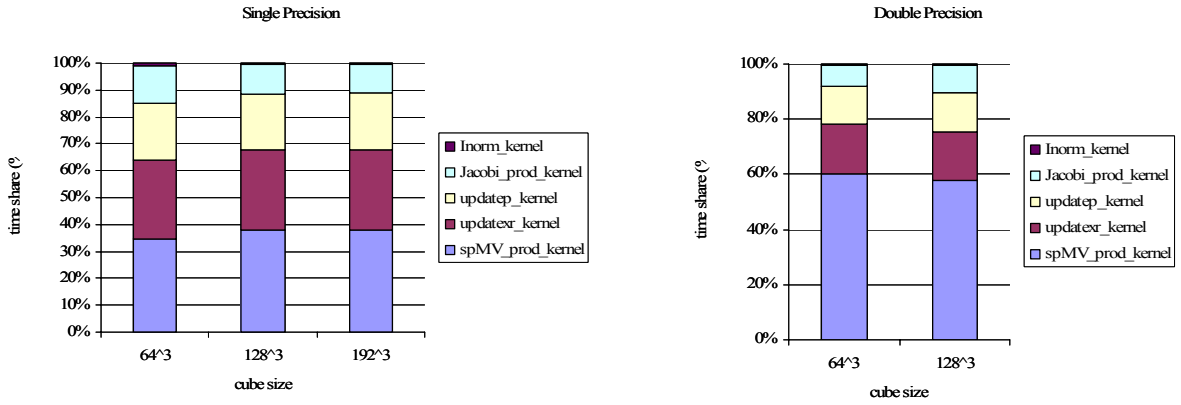


Figure 64: Time share of each GPU kernel on the total execution time, for different problem sizes and both single and double precision.

Unsurprisingly, the kernel covering the sparse matrix-vector multiplications and ensuing partial scalar product (*spMV_prod_kernel*, Chapters 5.4.1.1 and 5.4.1.3) takes the largest time share of all kernels, especially in double precision: after all, it covers the largest part of the arithmetic. What is notable, however, is the relatively large time share of the update functions (Chapter 5.4.1.4), considering that they are both computationally low-cost and naïvely parallel. One reason for this is that they are too light to run efficiently on the GPU, causing kernel launching overhead and off-chip memory latencies to take a disproportionate share of the total execution time. Also, these kernels are strongly I/O limited. Details will be discussed in Chapter 5.6.

The kernel covering preconditioner and ensuing partial scalar product (*Jabobi_prod_kernel*, Chapter 5.4.1.2) has only a minor impact with a mere 10% approx. of the total execution time. This can also serve as an estimate of the overall impact scalar products have in this implementation, as the kernel does little else than this.

5.5.7 Convergence behavior

As expected, single precision performance is way higher on the GTX 295 than double precision performance. However, in the view of the well-known draw-backs of single precision in scientific applications [151], this benchmark would not be complete without analysis of convergence behavior for both implementations. Figure 65 traces the development of the residual and the error over iteration count, for a real-world problem emerging from the application field introduced in Chapter 5.1.

$||\mathbf{b}-\mathbf{A}\cdot\mathbf{x}_i||$ is the l_2 norm of the residual and $||\mathbf{x}_i-\mathbf{x}_r||$ is the l_2 norm of the deviation of the found solution $\mathbf{x}$ to the (in this experiment known) correct solution $\mathbf{x}_r$, at the time of iteration i.

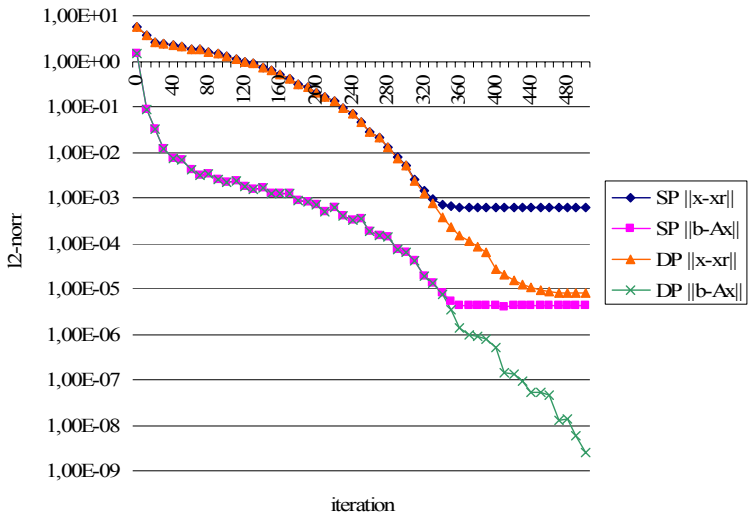


Figure 65: Convergence behavior for solution of a real-world mid-sized problem (4.8 million nodes), using both single (SP) and double precision (DP).

The convergence rate is nearly identical down to a certain level, where single precision is no longer able to approach the right solution. Our single precision implementation was able to find solutions for a tolerance $\varepsilon = ||r_i|| / ||r_0||$ as low as 10^{-5}, which should be enough for most applications. The double precision implementation, on the other hand, managed to approach the right solution down to an ε as low as 10^{-8}.

There is a simple explanation for the stagnation in single precision: cancellation. As the algorithm approaches the right solution, the addend to x in the update kernel becomes so small that the mantissas no longer overlap; hence nothing is added at all.

Note that the GTX 295 GPUs used in this work have an approximately 6x higher throughput in single precision than in double precision, owed to the fact that there are eight times more single precision units than double precision units (s. Chapter 2.4.3). If lower tolerances are required, it is worthwhile computing an intermediate solution for $\varepsilon = 10^{-5}$, followed by a refinement step in double precision. For the example in question, a tolerance of $\varepsilon = 10^{-8}$ can be matched with this mixed approach by performing 330 iterations in single precision, followed by 240 iterations in double precision. This saves ~40% computation time compared to the direct approach performing all 470 iterations in full double precision.

5.6 Discussion

The results presented in this work give rise to several questions that will be addressed in what follows. First, let us discuss the bottlenecks in this application. In single precision, the theoretical maximum throughput of both GPU models is very high compared to the off-chip memory bandwidth. For example, to fully load all single precision units of a GTX 295, it would take a ratio of 16 operations per word of DRAM I/O. With the small on-chip storage, it is almost impossible to become compute-limited with any realistic linear algebra application, as opposed to double precision where the ratio is around 5.4. For the GTX 480, the ratio is

about 15 for both precisions. These values can be calculated from the bandwidth and throughput of the devices [93].

Wherever possible, I merged the operations of BiCG to optimize on-chip data reusage. As a weak spot in the flow there remain the update kernels (Chapter 5.4.1.4), scaled vector additions that do not leave room for such optimizations and thus are highly bandwidth limited (~0.75 operations per word of DRAM I/O). In fact, in Chapter 5.5.6 we found that they accounted for a disproportionate amount of the total execution time. The share is lower in double precision where bandwidth limitation is generally less of an issue.

Another important aspect is multi-GPU scaling, which is directly related to the investment payoffs of hardware upgrades. Ideally, performance should grow proportionally with the number of GPUs. From one to two GPUs, the speedup was near ideal with >1.98x. We were using up to four GPUs and experienced speedups from 3.12x to 3.90x depending on the problem size, with larger problems gaining greater benefits from using more GPUs.

There are two limiting factors to performance scaling with parallelism. The first is the application itself. For the largest part of the algorithm, the problem sizes treated in this work offer enough parallelism to easily fill the capacities of several GPUs but there are also serial parts and scalar products, which according to Amdahl's law inhibit ideal performance scaling (s. Chapter 2.2).

The second limiting factor is hardware. Stencil operations require communication between adjacent nodes. Such data exchange is cheap inside the same thread block: threads running on the same stream multiprocessor can take advantage of the on-chip shared memory. Communication between blocks enforces passing data through off-chip DRAM which entails long access delays (~400-600 clock cycles), and launching a new kernel is the only mean of inter-block synchronization.

Communication between different GPUs, however, is much more expensive. Data need to be downloaded via the PCIe 2.0 interface to host RAM, and then uploaded to the other device. No direct GPU-to-GPU communication is available at the moment. The domain distribution presented in Chapter 5.4.2 aims to optimize this bottleneck, though the host RAM still needs to serially buffer

$$(N_{GPU}\cdot 2-2)\cdot dim_x \cdot dim_y$$

data elements every iteration. Host-device transfer speeds are partially dependent on the main board chipset; in our system, the PCIe 2.0 interface can transfer at a maximum rate of ~3.3 GB/s via page-locked memory, as revealed by the bandwidth test sample application [87]. This allows us to calculate the minimum time to upload or download all cross-GPU data to the host buffer for various problem sizes, with respect to the number of GPUs, which is shown in Figure 66.

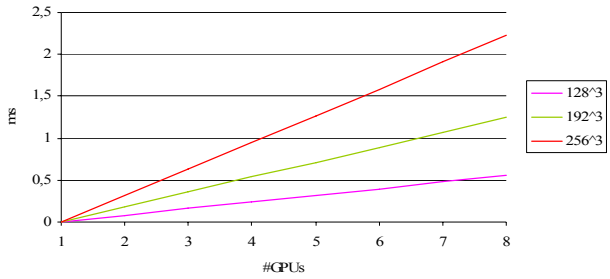


Figure 66: Minimum one-way transfer times (milliseconds) to or from host buffer, for different cube sizes and up to eight GPUs (single precision)

As described in Chapter 5.4.2, we segment the surrounding kernels to allow overlapping of these transfers with GPU computation. The ensuing kernels overlapping with upload are fairly large; however, the computation part which can be run concurrently with data download accounts for only approximately 15% of the total GPU computation time, so let us focus on this part as the limiting factor for multi-GPU scaling. These computation times are outlined in Figure 67.

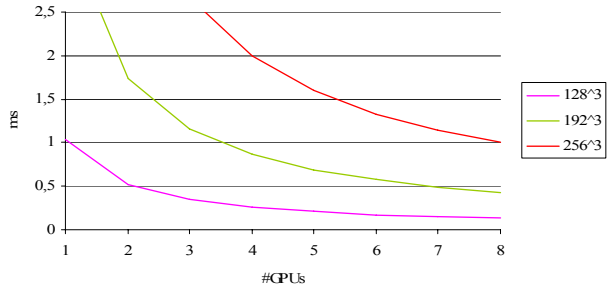


Figure 67: Approximate computation time in ms of that can be overlapped with GPU data download to host buffer (single precision)

Comparing both graphs, we notice that for the problem sizes and numbers of GPUs tested in this work the computation times are always long enough to cover the transfer times, with the exception of 128^3 nodes on 4 GPUs where the margin gets dangerously small. And indeed it appears from Figure 62 that we get a degraded speedup for this particular configuration. Note that this analysis assumes ideal conditions and ignores run-time and driver overheads, so having a safety margin of a few hundred microseconds (which is the order of these overheads) is advisable.

Let us extended this analysis to up to eight GPUs to explore the limits of multi-GPU scaling. According to this extrapolation, near-lossless scaling would still be feasible for six GPUs processing large problems ($\geq 256^3$ nodes). For smaller problems or more GPUs, some amount stalling is inevitable. If it were possible to transfer data directly from GPU to GPU without passing the host RAM, transfer times would no longer grow linearly with the number of GPUs, allowing efficient multi-GPU parallelism on a much larger scale.

Note also that for double precision, the transfer time doubles but the computation time of the corresponding kernel increases by over 4.5x. Taking these factors into account, efficient scaling should be possible for up to eight GPUs, if the problem spans at least 256^3 nodes.

Another strong field for improvement is preconditioning. Modern iterative solvers are rarely used without a preconditioner, which can greatly reduce the time to convergence as well as the numerical stability. The most prominent examples are based on incomplete factorization and solution of the sparse triangular systems. However, while very effective on serial computers, efficient implementation of these preconditioners on GPUs has proven extremely difficult. GPU-based iterative solvers are usually based on very simple solvers with high data locality, but at the time of writing, no solution was found to bring significant improvement in net computation time over the Jacobi preconditioner, probably because our base problem is not ill-conditioned enough for more complex solutions to pay off.

6 cudaEEG: real-time 3D source localization software

This chapter describes the third major contribute of this thesis, the cudaEEG software. In contrast to the solutions presented in Chapters 4 and 5, this implementation targets the EEG inverse problem, which was introduced in Chapter 3.2.2.2. In a nutshell, cudaEEG allows real-time localization and 3D-visualization of neuronal sources from EEG data, using the GPU for both data elaboration and in-place graphics rendering of the results. It is targeted at weaker GPUs, like older models in lower-end workstations or the GPUs in mobile computers.

6.1 sLORETA inverse estimator

The cudaEEG software is based on the sLORETA linear estimator (4) for EEG source imaging [43]; the algorithm is described in Chapter 3.2.2.2. Implementation is done in two parts:

1. Construction of the inverse estimator, i.e. the transition matrix $\boldsymbol{T}$ and the standardization matrix $\boldsymbol{V}$. This needs to be done *once* per target subject and electrode configuration.
2. Using the estimator to map electrode recordings to estimated source distributions in the brain ($\hat{\boldsymbol{J}} \leftarrow \boldsymbol{\Phi}$). This needs to be performed for *every time instant*. Hence, it is only this step that underlies real-time constraints.

Part one is relatively light rarely taking more than a minute. GPU acceleration is still worthwhile in respect to future frameworks where the lead field is adjusted dynamically and because the basic building blocks can be reused for other linear estimators. The following operations need to be implemented:

$$\boldsymbol{T} = \boldsymbol{K}^T \cdot (\boldsymbol{K} \cdot \boldsymbol{K}^T + \alpha \cdot \boldsymbol{I})^{-1}$$

and

$$\boldsymbol{V} = \textit{diagonal concatenation of all } [\boldsymbol{S}_{ll}]^{-1},$$

where $\boldsymbol{S}_{ll}$ are the N_V diagonal 3×3 sub-blocks of $\boldsymbol{S} = \boldsymbol{T} \cdot \boldsymbol{K}$.

The GPU implementation has two bottlenecks. One is the matrix-matrix product $\boldsymbol{K} \cdot \boldsymbol{K}^T$. It suffers from the fact that the matrix $\boldsymbol{K}$ is "long and thin" spanning $3 \cdot N_V \times N_E$ elements where $N_V >> N_E$: there are at most a few hundred electrodes, but there can be hundred thousands of voxels. Hence, the product consists of few but very long scalar products. Naïve implementation thereof offers only N_E^2 parallel streams of length $3 \cdot N_V$; however, this ratio can be improved by introducing a two-stage parallel reduction to the scalar products. The other bottleneck is the matrix inversion. With $N_E \times N_E$ elements, the matrix is much too small to make GPU-accelerated inversion profitable. For these small magnitudes, the fastest solution was found to outsource the operation to the CPU. The work flow of the implementation is displayed in Figure 68.

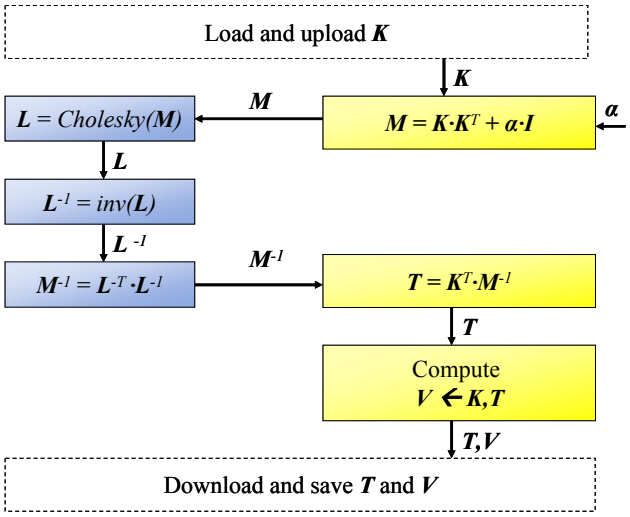


Figure 68: sLORETA work flow for computation of the inverse estimator. The blue parts (left) are processed on the CPU while the yellow parts (right) are handled by the GPU.

What regards the real-time part, the sensor array measurement $\boldsymbol{\varphi}(t)$ of each time frame needs to be mapped to the estimated current densities $\boldsymbol{J}_{est}(t)$ by

$$\boldsymbol{J}_{est}(t) = \boldsymbol{T}\cdot\boldsymbol{\Phi}(t)$$

and then standardized to the current density powers $\hat{\boldsymbol{J}}(t)$ by

$$j_{l,std}(t) = \boldsymbol{j}_{l,est}(t)^T \cdot \boldsymbol{V}_l \cdot \boldsymbol{j}_{l,est}(t)$$

for all $j = 1...N_V$. This is the result that is later mapped to the output.

Being based on matrix-vector products, this part is rather straightforward to implement. However, data reusage is limited and the arithmetic-to-data ratio is not high enough to overcome the bandwidth limitation to which GPU applications are extremely prone to. Hence, data are cast into a format that allows optimal bandwidth utilization according to Chapter 2.6.3.2: firstly, the matrix rows are "padded" to align the length of each row to a multiple of the warp size of 32, as shown in Figure 69.

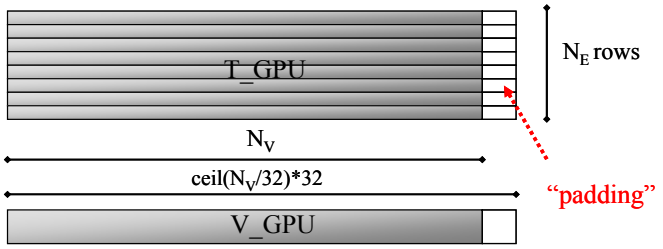


Figure 69: Data padding.

Secondly, the 3-element vectors are cast to 128-bit aligned data structures: not only are 128-bit transfers slightly faster than 32-bit transfers [93]. In combination with the padding technique, this also ensures that the memory accesses of each half-warp always aligned to 128-byte data segment. Ignoring this rule can result in a drop in memory access speed of up to 90%, so the benefits more than make up for the 25% increase in data size [93]. The alignment of the T_{ij} and V_j elements is shown in Figure 70.

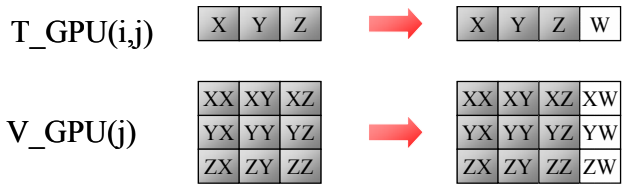


Figure 70: 128-bit alignment of T_{ij} and V_j data elements.

The application is embarrassingly parallel (s. Chapter 2.2), so performance scales roughly linearly with the product of N_E and N_V. But even with very high voxel resolutions, real-time requirements were easily matched. Our heaviest configuration with 59 electrodes and 195391 voxels (discretization of grey matter at a resolution of ~1.4 mm) still performed at over 200 time frames per second on an 8800 GTS GPU – including 3D visualization. The speedup versus a 1.86 GHz Intel dual-core CPU was around 15x. For fluent real-time visualization, a frame rate of 30 is usually considered the minimum; for more than 60 the viewer hardly observes any further improvements.

6.2 Graphical user interface

Since the source reconstruction is done in GPU memory, only the sensor array vector needs to be uploaded every time frame. This alleviates traffic on the GPU-host interface by up to three orders of magnitude.

For visualization, the standardized current density powers are mapped to a color scale. The 3D visualization also needs voxel positions and the originating MRI image to allow anatomical association of source activity. Visualization is based OpenGL, an open and portable graphics rendering API [153]. The dependencies and data movements are shown in Figure 71.

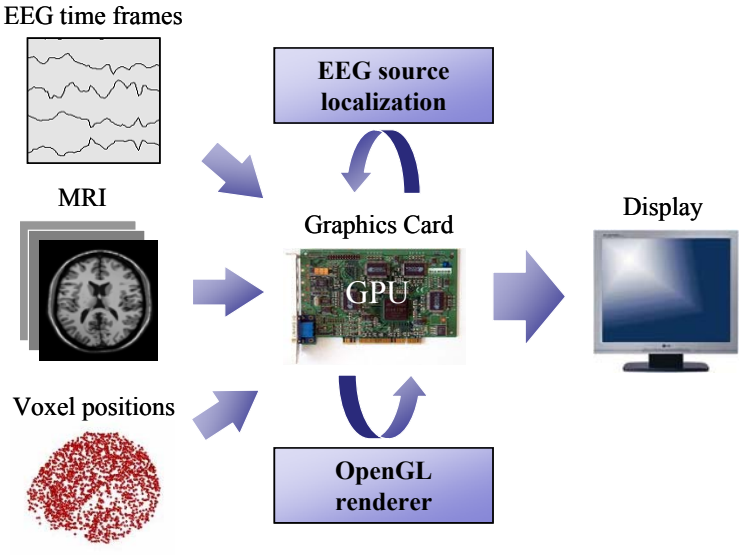


Figure 71: cudaEEG software dependencies and data movements.

The 3D visualization is based on a point-based rendering technique. This means that every voxel is rendered as a point, as opposed to interpolating triangulated surfaces. It is very popular in biomedical data visualization [154]. Anatomical structures are mapped on the

image by directly applying the raw MRI image as a 3D texture. This approach dynamically adapts to any MRI and voxel discretization without any additional pre-processing.

The software opens two windows: one containing the 3D view of the source estimation mapped on the brain model, the other providing a 2D interface to examine and navigate the EEG recordings. When in free navigation mode, the 2D interface can be used to select the time instant to be processed by sLORETA. The graphics output of cudaEEG is shown in Figure 72.

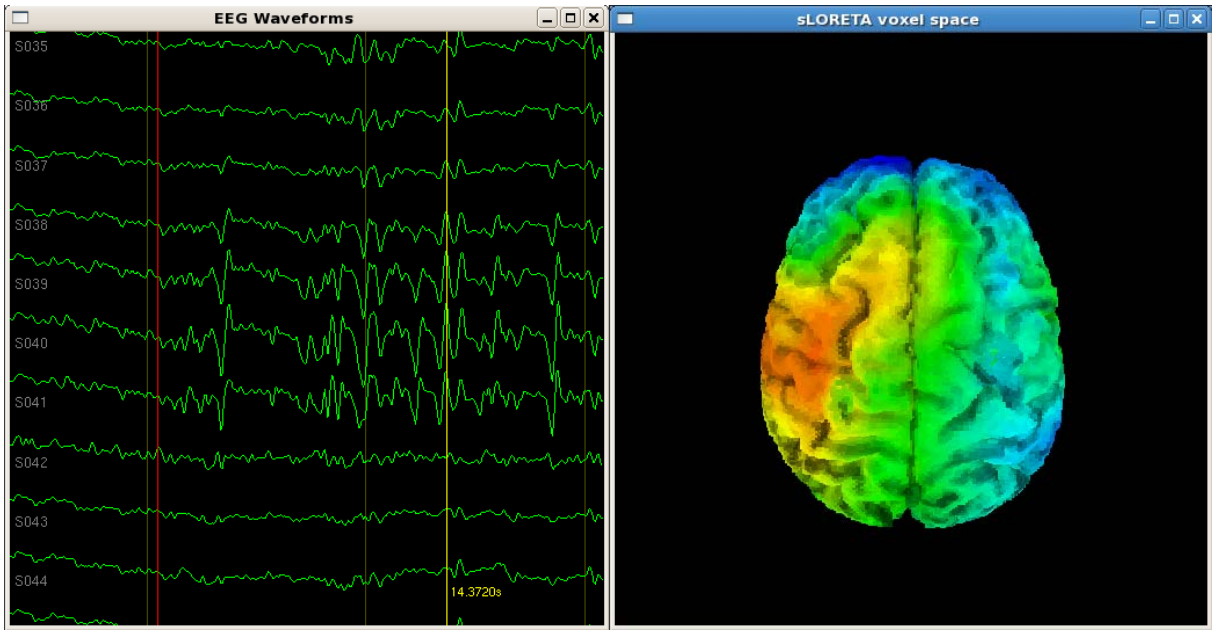


Figure 72: cudaEEG graphical user interface output.

The 3D view allows rotation of the brain model and examination of single slices. The 2D view supports zooming and panning through the EEG recordings. Slow-motion animation is also possible. A pop-up menu allows easy configuration.

6.3 Shrinking standardized LORETA-FOCUSS

The sLORETA linear estimator is an established solution in EEG source imaging because it produces decent results at a moderate computational cost. It has its weak points, though: sLORETA is often criticized for producing blurred images that make it hard to distinguish close focal points. This issue is inherent in l_2-norm based estimators. As described in Chapter 3.2.5, some iterative refinement methods have been developed to remedy these drawbacks without falling back to another form of error-term minimization. However, this requires the estimator to be recomputed at run time, which multiplies the computational cost. With sLORETA running much faster on the GPU than actually required for real-time visualization, the feasibility of a more complex linear estimator was explored, namely the shrinking standardized LORETA-FOCUSS (SSLOFO) [45] which was also introduced in Chapter 3.2.5.

SSLOFO iteratively updates a weighting matrix based on the solution found in the previous iteration:

$$W_i = diag(J_{est,i-1}) \cdot P \quad with\ J_{est,i-1} = T_{i-1} \cdot \Phi(t)\ and\ P = diag(K^T \cdot K)^{-1}$$

A sLORETA solution is used as initialization. Using this weighting matrix, the transition matrix is recomputed using a Weighted Minimum Norm (WMN) approach and applied to obtain an updated solution. This solution is smoothed to avoid trapping in local minima; in the same turn voxels showing low activity under a certain threshold are removed from the solution space and will no longer be considered in construction of the next estimator. This loop is repeated until an arbitrary stopping criteria is matched, which [45] suggests to be

1. no change to the previous iteration
2. solution is less sparse than in the previous iteration
3. any focal point exceeds a user-defined threshold

None of these are feasible under real-time constraints, as the duration of processing one time frame would be undetermined. Instead, a fixed iteration count was used in the implementation.

6.3.1 Implementation

What regards the implementation, the subroutines from Chapter 6.1 could be reused with small modifications. Smoothing is applied via a simple median filter, and the shrinking is done by setting the corresponding weight of the excluded voxel to zero. This efficiently removes the point from solution space without introducing any divergence in the parallel data flow. As threshold, 5% of the global peak was chosen. The heterogeneous implementation of the algorithm is displayed in Figure 73.

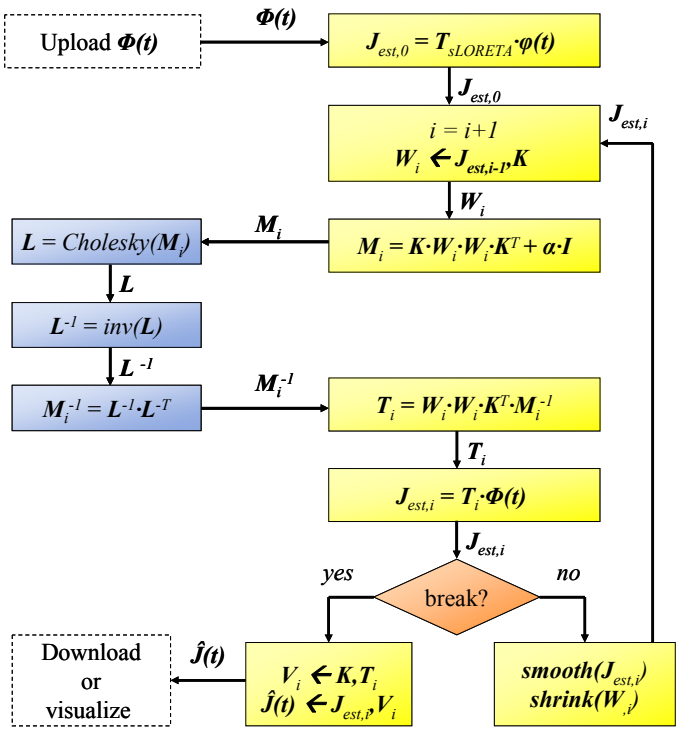


Figure 73: Heterogeneous implementation of the SSLOFO algorithm. Blue steps are processed on the CPU; yellow tasks are handled by GPU.

Note that the initialization phase where ***K*** is uploaded and the constant ***P*** is computed is not included in this diagram.

6.3.2 Performance

SSLOFO is much heavier than sLORETA, so a closer performance evaluation is necessary to see under which conditions real-time processing is still feasible. The implementation was benchmarked on a system using a NVIDIA 8800 GT graphics card and an Intel Core2 6300 CPU. Configurations using voxel resolutions between ~4000 to ~32000 were measured, in combination with electrode caps of 32, 64 and 128 electrodes. Computation times for these configurations per SSLOFO iteration are shown in Figure 74.

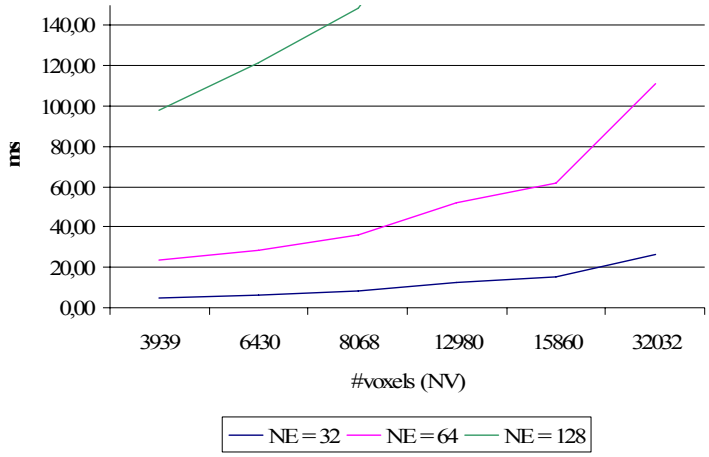


Figure 74: Computation time in milliseconds per SSLOFO iteration.

Assuming that at least two iterations are required to substantially increase sparsity of the solution, we see that only the 32 electrode configuration is fast enough to provide a frame rate suitable for real-time visualization. For 64 electrodes, only very low voxel resolutions can be processed barely fast enough on this GPU. For 128 electrodes the solver is way to slow even for low resolutions. Profiling of the application has revealed that the CPU-based inversion part quickly becomes a bottleneck with increasing electrode count, as its complexity grows cubic while the complexity of the GPU-parts only grow quadratic with this number. For 64 electrodes, the inversion already accounts for 55% of the total computation time, and it becomes even more dominant for higher values.

Compared to a serial CPU-only implementation, the observed speedup was around 20x for most configurations and up to 30x in the best case, while larger problem sizes also yielded higher speedups. Still, SSLOFO is not fast enough to process the high resolutions used in cudaEEG and is at the moment not integrated in the software.

6.4 Deriving neuron orientation from MRI

The low spatial resolution of EEG source imaging is owed to the fact that the inverse problem is extremely underdetermined: a low number of signals is used to compute a spatial source distribution of thousands of discreet points. Increasing electrode count slightly alleviates the

problem, but also increases system cost and the effort of applying the electrode cap. There are also physical limits on how many electrodes can be mounted on the head; when more than 70% of the scalp surface is covered, the interference starts making the measurements increasingly defective. The state-of-the-art maximum is about 300 electrodes, still a drop in a bucket considering the many degrees of freedom in the inverse problem [155].

To improve quality of the inverse solution, it is therefore crucial to infer as much anatomical a priori information as possible. As outlined in Chapter 3.2.4, some of these can be obtained from the subject's MRI image. We already use the MRI to constraint the solution space to the cortical grey matter, which alone is capable of generating electrical source activity.

If also the orientation of each voxel were known, the degrees of freedom could be instantly reduced by 2/3, leading to a great improvement in spatial resolution. The problem is getting a reliable estimate for said orientations. However, while neurons can be cross-wired unpredictably within the brain, the sources that can be detected by EEG are exclusively pyramidal cells directed orthogonally to the cortical surface, which according to [157] accounts for ~85% off all neuron in the cortical grey matter. It is therefore feasible to derive the voxel orientation from the orientation of the cortical surface around it. This too can be obtained from MRI in a successive processing step as displayed in Figure 75.

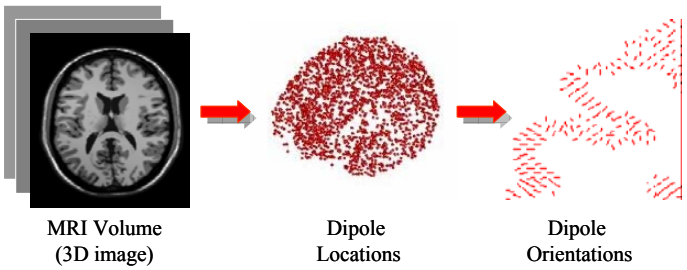


Figure 75: Deriving anatomical priors from MRI.

There are two alternative methods to obtain these data from MRI, which will be described in the following.

6.4.1 Derivation from MRI luminosity gradient

One method is to directly retrieve these orientations from the luminosity gradient at the point of the voxel inside MRI, as suggested by [36]. This approach is based on the observation that the layers of grey matter have slightly different luminosity levels in MRI, and there is an even greater gradient at the transition to other tissue types. Provided good resolution and low noise, the gradient can be used to form voxel orientations perpendicular to the cortical surface as shown in Figure 76:

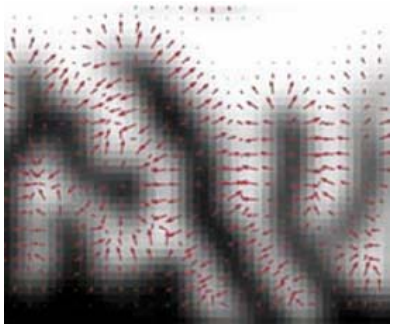


Figure 76: Luminosity gradient in cortical grey matter regions of MRI.

However, even a small amount of noise can have a devastating impact if only the gradients between two pixels are considered. This is a common problem in image processing where computation of resilient gradient maps is integral part of edge detection algorithms. It is often addressed by gradient averaging operators like the Frei-Chen operator [158]. Extending the principle to three dimension yields the following 3×3×3 operators for retrieval of averaged gradients in the x-, y, and z-direction (s. Figure 77).

$$\left(\begin{array}{ccc|ccc|ccc} -1/\sqrt{2} & 0 & 1/\sqrt{2} & -1 & 0 & 1 & -1/\sqrt{2} & 0 & 1/\sqrt{2} \\ -1 & 0 & 1 & -\sqrt{2} & 0 & \sqrt{2} & -1 & 0 & 1 \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} & -1 & 0 & 1 & -1/\sqrt{2} & 0 & 1/\sqrt{2} \end{array}\right)$$

X-direction

$$\left(\begin{array}{ccc|ccc|ccc} 1/\sqrt{2} & 1 & 1/\sqrt{2} & 1 & \sqrt{2} & 1 & 1/\sqrt{2} & 1 & 1/\sqrt{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/\sqrt{2} & -1 & -1/\sqrt{2} & -1 & -\sqrt{2} & -1 & -1/\sqrt{2} & -1 & -1/\sqrt{2} \end{array}\right)$$

Y-direction

$$\left(\begin{array}{ccc|ccc|ccc} -1/\sqrt{2} & -1 & -1/\sqrt{2} & 0 & 0 & 0 & 1/\sqrt{2} & 1 & 1/\sqrt{2} \\ -1 & -\sqrt{2} & -1 & 0 & 0 & 0 & 1 & \sqrt{2} & 1 \\ -1/\sqrt{2} & -1 & -1/\sqrt{2} & 0 & 0 & 0 & 1/\sqrt{2} & 1 & 1/\sqrt{2} \end{array}\right)$$

Z-direction

Figure 77: 3×3×3 Frei-Chen operators for 3D gradient averaging.

These operators are applied in each voxel position to retrieve the 3-element orientation vector, which is then rescaled to unity length. Being an image processing method, this procedure maps intuitively on the GPU; the processing time is in the range of a fraction of a second.

The image gradient method requires a certain quality of the MRI image. In low-resolution images, it can happen that some characteristics are only one or two pixels wide. In that case, the method fares poorly, producing a large number of erroneous priors. To cope with these limitations, an alternative method was devised, which is described in Chapter 6.4.2.

6.4.2 Derivation from pial surface mesh

One of the solutions to the EEG forward problem that were discussed in Chapter 3.2.3 involved deriving triangulated mesh models of scalp, skull and brain, from which a multi-

layered head model was constructed. As mentioned earlier, meshes were obtained from MRI using a third-party software called *freesurfer* [163][164].

The innermost of those three meshes, the brain mesh, describes the surface of the pia mater. Precisely enclosing the voxel space, this mesh can be reused for deriving the voxel orientations. This is done in the following way: each voxel scans through the surface triangles to find the triangle closest to itself. This triangle's normal vector is applied as the voxels orientation. The principle is shown in Figure 78.

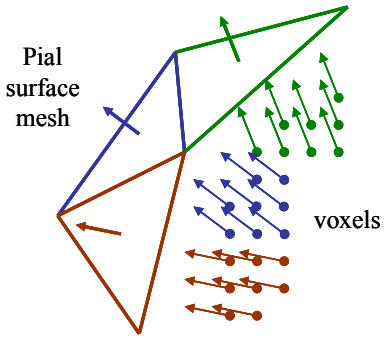


Figure 78: Deriving voxel orientations from pial mesh surface.

The result is a very smooth orientation map for all voxels. Again, the procedure is embarrassingly parallel, containing simple geometric operations. The GPU implementation runs just as fast as the image gradient variant.

To the best of my knowledge, no other solution exists that uses exactly this approach. However, [165] use an approach where they directly retrieve the voxels from the centroids of each surface mesh triangle, along with the normal vector as its orientation. This solution has the disadvantage that it does not result in a uniform discretization of grey matter as solution space.

6.4.3 "Fuzzy" orientation priors

Knowing the orientations has a great effect on constraining the solution space; however, there is also a drawback. The orientation of a current dipole has a strong effect on which electrodes are significantly influenced through the lead field. Thus, even small deviations from the correct orientation result in large errors in the model.

Given the sensitivity of the model to these parameters, it would be risky to apply them as strong priors, i.e. removing other orientations altogether from the solution space. A much better approach is to apply them as weak priors, where solutions defying the estimated orientation are derated, but still incorporated in the solution space.

This implementation proposes an approach where parts of the solution not complying to the orientation priors are attenuated, but not left out, thus putting "fuzzy" constraints on the solution space. This is done by calculating the transition matrix as a weighted minimum norm like in FOCUSS (s. Chapter 3.2.5).

(10) $$\boldsymbol{T}_{fuzzy} = \boldsymbol{W}\cdot \boldsymbol{W}^T\cdot\boldsymbol{K}^T\cdot(\boldsymbol{K}\cdot \boldsymbol{W}\cdot \boldsymbol{W}^T\cdot\boldsymbol{K}^T+\alpha\cdot\boldsymbol{I})^{-1}$$

such that $\boldsymbol{W}$ penalizes deviating orientations by a large percentage. Note that in this case, $\boldsymbol{W}$ is not a diagonal matrix like in FOCUSS and SSLOFO, but a matrix of 3×3 diagonal sub-blocks:

$$\boldsymbol{W} = \textit{diagonal concatenation of all } \boldsymbol{W}_{ll}, \textit{ with } l = 1..N_V$$

The N_V sub-blocks are calculated by

$$\boldsymbol{W}_{ll} = (\boldsymbol{r}_l, \boldsymbol{a}_l, \boldsymbol{b}_l)^T$$

where $\boldsymbol{r} = (r_{l,x}, r_{l,y}, r_{l,z})$ is the orientation vector of voxel l. The other two rows of each of 3×3 diagonal sub-blocks are defined as

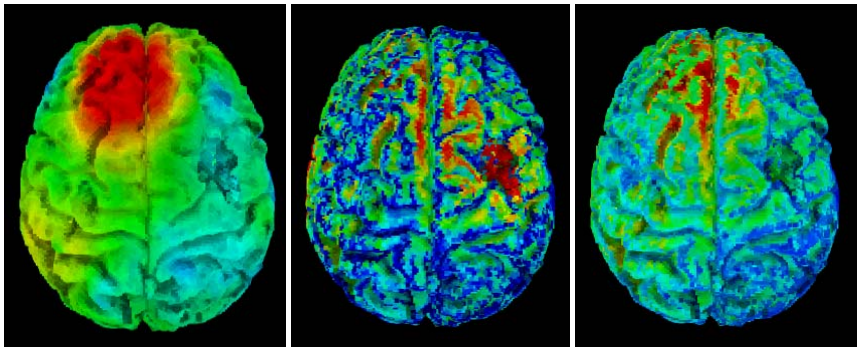
$$\boldsymbol{a}_l = (a_{l,x}, a_{l,y}, a_{l,z}) = \gamma \cdot (r_{l,x},\ 0,\ -r_{l,x}^2 / r_{l,z})$$

$$\boldsymbol{b}_l = (b_{l,x}, b_{l,y}, b_{l,z}) = \gamma \cdot (r_{l,y} \cdot a_{l,z},\ r_{l,z} \cdot a_{l,x} - r_{l,x} \cdot a_{l,z},\ -a_{l,x} \cdot r_{l,y})$$

The scalar γ denotes the attenuation factor for deviating solutions. Note that after this transformation, the 3-element vectors of each voxel are no longer aligned to the common Cartesian grid. Anyway, those vectors are standardized to current density powers in the end, so this alignment change does not affect the final solution.

Again, application of the weighting is embarrassingly parallel on a per-voxel basis, so the computational effort hardly increases in respect to normal sLORETA. The transition matrix needs only be computed once, so for the real-time part, this enhancement is even for free.

Compared to the default solution with unknown orientation, introduction of these priors leads to a spatially much sparser and "sharper" solution. Forcing orientations however also makes the solution very prone to errors. A comparison between results for unknown, forced and fuzzy orientation priors is shown in Figure 79.



Unknown Orientations *blurry* | Forced Orientations *error-prone* | Fuzzy Orientations

Figure 79: Impact of fixed and fuzzy orientation priors compared to unknown orientations. The penalization factor in this example is 70%.

All three images display source estimations for the same input vector. The first is blurred but at least localizes the center of activity correctly. The second is very sparse, but the example also betrays the error sensitivity of strong priors: the small dark patch on the right hemisphere is actually a part of the skull that was erroneously included into the solution space during preprocessing (s. Chapter 3.2.4). Still the center of activity is localized inside this artifact, debunking the solution as clearly wrong. In contrast, allowing some fuzziness in the priors

preserves much of the spatial resolution, but proves to be much more resilient in the face of this error.

A related method is presented in [165], where loose orientation constraints are introduced in l_2 and l_1 minimum norm solutions. Comparison of the methods is however difficult as they do not compute spatial distributions but map source activity on cortical surface patches.

7 General discussion

The scope of this thesis was to explore the capabilities – and limitations – of heterogeneous multicore systems, in the application framework of bioelectrical imaging. It quickly turned out that the PC architecture with one or multiple GPU accelerators yield the most fitting characteristic as a target computing platform: Composed of commodity hardware components, GPU-accelerated desktop machines are cheap, ubiquitous, highly configurable and easy to upgrade. Affordability is of primary concern for application in bioelectrical imaging, which pitches itself as a low-cost alternative to expensive anatomical imaging systems like fMRI. The other side of the deal is whether these systems can service the needs of the target application. GPUs excel at structured problems based on heavy floating point arithmetic, which to a large extent applies to all computational challenges encountered in this thesis. Given these characteristic, excellent results could be expected from GPU acceleration in bioelectrical imaging. The following discussion will now address the questions to which extent these expectations could be matched, what the limiting factors are and how these limits could be further pushed in future system designs and implementations.

7.1 Summary of the contributions and results

The main contributions of this thesis can be summarized in three principal parts, which address selected computational challenges that occur in bioelectrical imaging. Based on the individual demands of the task, different scales of CPU/GPU systems have been employed. The set-ups range from low-end systems with a single, weaker GPU that might as well fit into an average mobile system, over dual-GPU desktop workstations, up to desktop supercomputers with multiple CPUs and several high-end graphics cards.

Chapter 4 presented a dual-GPU based triangular matrix inversion routine, motivated by a rank 10000 matrix inversion problem encountered in EEG volume conductor modeling using a mesh-based multilayer head model. A double-precision throughput of 92 GFLOP/s is achieved using a GTX 295 dual-GPU graphics card, which is a high percentage of the device's theoretical peak. It outperformed a multithreaded LAPACK reference by a factor of 57x and uses an efficient novel data storage scheme for triangular matrix formats on GPUs.

Chapter 5 presented a multi-GPU accelerated solver for large non-hermitian linear systems. It is integral part of an EIT forward simulation environment, which requires repeated solution of banded systems of millions of complex variables as produced by high-resolution discretization of an anisotropic, frequency-sensitive head model. The complex BiCG based solver yielded single precision throughput of 42 and 62 GFLOP/s on a single GTX 295 and GTX 480 GPU respectively, and near-optimal performance scaling could be extrapolated for up to six GPUs. In double precision, single-GPU throughputs were 5.8 and 21.7 GFLOP/s per GPU, where analysis suggests that near lossless performance scaling should be possible for up to eight GPUs. Even a single GPU outperformed an optimized reference running on two quad-core CPUs, and speedups of over 30x could be observed in benchmarks using four GPUs.

Finally, Chapter 6 presented a 3D EEG source imaging software. This part revolves around the EEG inverse problem and comprises GPU-accelerated calculation and real-time application of EEG source estimators. The implementation is special in that it uses the GPU for both local computation and immediate 3D visualization of the results, greatly alleviating traffic on the system infrastructure. Both the lightweight sLORETA and the heavy SSLOFO estimators have been implemented, yielding speedups of 15x and up to 30x versus a corresponding CPU reference. In addition, a GPU based routine for derivation of neuron orientation priors has been developed, deriving these orientations alternatively from MRI luminosity gradients or pial surface meshes. For introduction of these priors into the inverse estimator, a method has been suggested that places "fuzzy" constraints on the solution space, which proved much more error resilient than fixed orientations.

7.2 On performance analysis

Looking at the results, one can safely state that the expectations put into the heterogeneous processing approach with GPUs have not been disappointed. Still, there are some common caveats that must not be ignored in this discussion.

Most importantly, one might want to point out the pitfalls of performance analysis. Some implementations report sensational speedups through GPU-acceleration, as a look in the "CUDA community showcase" reveals [181]. Table 3 shows a selection of implementations claiming speedups of ~500x and greater.

Application	Speedup	Reference
GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking	600x	[176]
Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units: Application to Analysis of Doppler Exoplanet Searches	600x	[177]
Massively Parallel Population-Based Monte Carlo Methods	500x	[178]
Accelerating numerical solution of Stochastic Differential Equations with CUDA	650x	[179]
Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units	496x	[180]

Table 3: Selection of applications claiming very high speedups through GPU acceleration.

How can these applications achieve over 500x speedup over CPUs, considering that the raw computational power of comparable generations of CPUs and GPUs does not diverge by more than one order of magnitude? Unfortunately, performance analysis is a weak spot in many examples of GPU-related work; often fine-tuned multi-GPU implementations are compared to un-optimized, serial reference code. This has become so common that some studies are exclusively devoted on "debunking the 100x GPU vs. CPU myth" [91][92].

A comparative performance analysis should hold strong in face of the following questions:

1. Are the capabilities of the CPU reasonably exploited? All modern CPUs have at least two cores, with high-end models integrating up to six, so straight-forward serial code rarely makes a fair comparison. Of course, code that is multithreaded or even uses the

CPU's SIMD extensions is harder to write, which directly leads to the second question:

2. Is the programming effort put into the reference implementation remotely comparable to that put into the GPU-accelerated version? This is particularly interesting in the context of commercial software development, where development times ultimately determine whether the project is worthwhile or not.
3. Is the scale and cost of the reference system comparable to that of the GPU-accelerated system? This is a bit tricky: each GPU needs at least one CPU-core for control, so introducing GPU acceleration will always increase system cost. However, upgrading a system with a high-end graphics card is generally cheaper than replacing the CPU with a high end substitute.

Many authors' reluctance to put too much effort into the reference application, which might be used just for this one benchmark, is understandable. An elegant way to get rid of this burden is using third-party optimized library routines, if available. However, even if the above mentioned terms are followed exemplary, there remains the notion of comparing apples and oranges when relating GPU-applications over speedup values. If the application is based solely on floating point arithmetic, it is therefore popular to measure the throughput in GFLOP/s: at least this expresses performance on a fixed scale. This is not perfect, though: on the one hand, the number is hardware dependent; to get a real idea of implementation quality, one needs to take into account the benchmarking device's capabilities, which might differ considerably (s. Table 1). On the other hand, whether these FLOPs are put to good use depends on the algorithm. Some parallel algorithms buy parallelizability through increase of total work load. Parallel prefix algorithms are a good example of this, but also iterative solvers where the choice of the preconditioner influences both convergence rate and parallel performance.

In this thesis, I tried to make performance evaluation as fair as possible. The TMI routine is benchmarked against a multithreaded LAPACK reference, a highly optimized collection of linear algebra routines. Application performance on one dual-GPU graphics card is compared to that on a dual-core CPU, so the scales are comparable. All pre-processing and address generation is included in the computation times, whereas the algorithm itself is optimal, meaning all computed FLOPs are actually required to solve the problem. Still, performance analysis yielded high speedup and throughput, which indicates that the GPU implementation is in fact highly efficient.

For the non-hermitian linear system solver, a fine-tuned multithreaded reference application using both OpenMP and streaming SIMD extensions was created. Just like the GPU implementation, it exploits the special structure of complex BiCG to improve data locality, and was found to outperform a competing alternative based on library subroutines (not included in the benchmarks). Hence, it is safe to say that the optimization effort is comparable. The reference runs on eight CPU cores, which is more than appropriate for comparison with one to four GPUs. The only remaining caveat might be that only Jacobi preconditioning was used, and that more advanced preconditioners tend to perform much

better on CPUs than on GPUs. The problem at hand was relatively well-conditioned, so more complex preconditioners did not pay off. However, it should be noted that incomplete factorization methods, which define the state-of-the-art in CPU-based preconditioners, are known to yield net speedups of up to 60% in solution of some problems. Yet, even when assuming the aforementioned conditions, a preconditioner does not reach the speedup obtained by even just a single GPU.

The case of cudaEEG is particular in that it does not implement an isolated computational routine. The effect from moving the signal processing from the CPU to the GPU was measured, but in fact both implementations use the GPU for graphics rendering. Being a graphics application, performance was measured in frame rates rather than GFLOP/s, which would be less meaningful in such a context. What concerns the derivation of neuron orientation priors, the focus is on functionality; the corresponding GPU kernels are very light and would also be feasible on a modern CPU, but the procedure is data-intensive, and having the complete flow on the GPU alleviates resource usage and traffic on the host system.

7.3 Limiting factors

GPU vendors advertise their products with high numbers of peak throughput; even low-end models are alleged to achieve several hundred GFLOP/s of computing power (s. Table 1). The numbers are obtained by pretending that all arithmetic units are perpetually busy with fused multiply-add operation (which count as two FLOPs, but require only one operation on GPUs). Of course, these assumptions are far from realistic and real applications, which even if reasonably optimized usually perform way below these levels.

GPU-accelerated implementations are subject to several limiting factor that are responsible for the large gap between actual application performance and the theoretical capabilities of the hardware. Some are inherent to parallel computing, as outlined in Chapter 2.2 (Amdahl's law): whenever an algorithm contains steps that cannot be fully parallelized, speedup versus serial processing stagnates at a certain level and can not be further improved by parallel scaling. This applies to the non-hermitian system solver in that it contains inner vector products and other scalar operations, but also to cudaEEG in that the graphics API setup is handled by serial code.

One might also want to extend Amdahl's law to the whole application, beyond the computational part. On a heterogeneous platform, programs typically start in a serial thread on the controlling processor. This thread handles data and memory setup and prepares work dispatch to the accelerators. After processing, results need to be moved back to the output. For a heterogeneous multi-CPU/multi-GPU application, non-computational overheads are listed in Table 4. Durations are very variable; the values indicate what is typically experienced on an average desktop machine.

Operation	Parallelism	Duration
OpenMP thread handling	1	~10 µs
Initialize GPUs (*)	1 - 8	~200 µs
Upload data to GPUs (*)	1 - 8	up to 300 ms

Kernel setup and sync	1 - 8	~12 µs
Kernel execution	10^3 - 10^8	50 µs - 5 s
Download results from GPUs (*)	1 - 8	up to 300 ms

Table 4: Non-computational overheads in relation to actual computation time. Asterisks indicate that the operation is usually only performed once throughout the application.

Taking these factors in consideration, even embarrassingly parallel applications are subject to the scaling limitations described by Amdahl's law. The overhead is negligible for large data sets, but can be compromising for small problems. For that reason, performance analysis usually measures different problem sizes. Expectedly, all implementations presented in this work yield better speedups for larger data sets.

A related limitation is introduced by the scalable programming model. A GPU kernel remains active until all blocks are finished, but the number of blocks is rarely aligned with the number of Stream Multiprocessors. More often than not, the array of SMs is only half occupied with the last row of blocks, which creates some overhead through processor idling. Three examples are shown in Figure 80. Keep in mind that most GPUs have more than four SMs (12 - 30 for the architectures used in this work), making the case in Problem 2 even more unlikely.

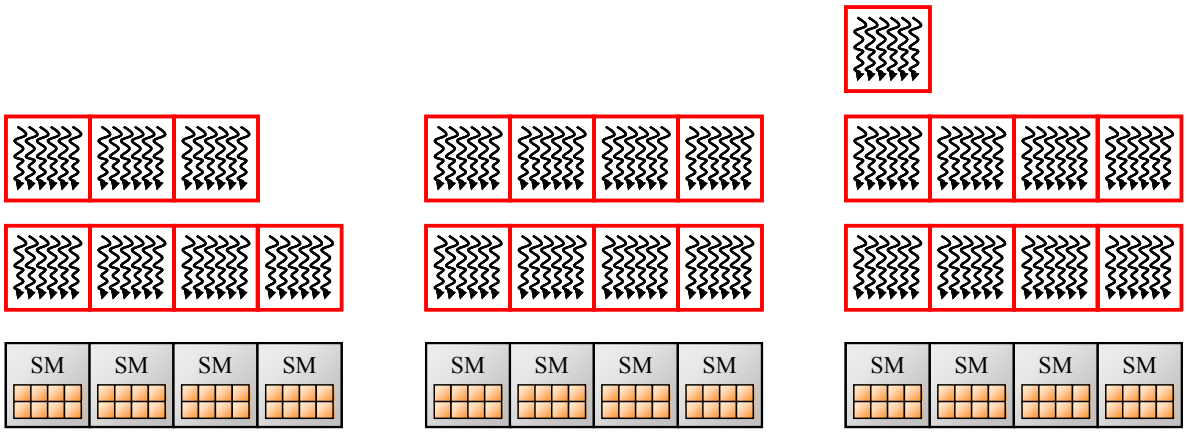


Figure 80: Performance of three configurations of the same kernel, on the same architecture with four Stream Multiprocessors. Block size is 256 in both cases.

Although Problem 3 is only 12.5% larger than Problem 2, it takes 50% more computation time because the blocks are less favorably aligned with the number of SMs. This can create considerable overhead, especially in the range of small problem sizes. The effect accounts for the "bumpy" trend lines when measuring GFLOP/s for different problem size, as in Figure 53 of the TMI performance analysis.

The issues discussed so far are owed to the heterogeneous platform and are still independent from what the actual computation looks like. But of course, arithmetic throughput and data feeding ultimately dictate application performance. As long as warps are held strictly in SIMT style, arithmetic throughput is usually of minor concern. Full 32-bit integer multiplication is

split in multiple operations on GPUs up to series 200, which primarily results in elevated address generation overhead compared to CPUs [87]. For the implementations in this work, address generation overhead is alleviated as much as possible through pre-calculation on the CPU and upload of ready-made LUTs to the GPU constant cache. This is of particular benefit in case of the TMI, where the complex addressing patterns to compressed storage would promise significant overhead. Using the heterogeneous address generation approach outlined in Chapter 4.3.3, this overhead could be reduced to an amount comparable to what would be experienced accessing normal aligned storage. What concerns computational arithmetic, the instructions usually fit very well in fused multiply-add calculations, which promise the highest throughput on GPUs. In this work, only the arithmetic of the complex BiCG had to be restructured to optimize instruction streams. However, only the TMI implementation has been found to be compute limited, a rare case among GPU implementations. This is due to the fact that the algorithm is structured, dense, allows a large amount of data reusage, and is run in double precision on a device that is still focused on single precision arithmetic. Consequently, the computational throughput is a high percentage of the theoretical peak, where the difference to said peak is owed to the aforementioned overheads.

For the vast majority of applications, however, data feeding is the predominant overhead. The way from host RAM to the GPU registers is long, and passes several levels and interfaces. Starting from the top, there is the shared on-chip memory, which grants almost immediate access of the addressing stride falls into different banks without conflicts. It also makes communication and synchronization inside a block very cheap. Further down is off-chip GDDR memory on the graphics board, which is shared among blocks. It potentially offers a very high bandwidth, as long as the access patterns are in tune with the requirements of the memory controller. But even then, it is usually unable to perpetually feed all the computational resources: in most GPUs, every data word that passes the interface must be reused over ten times to avoid the memory wall, rarely achievable considering that the small size of on-chip memory strongly limits data reusage. On top of all, there is a large latency when accessing off-chip memory, which, even when concealed by overlapping computation, strikes at least once at the beginning.

The worst bottleneck is the PCIe interface between host RAM and GPU memory. It is the only memory shared between multiple GPUs, making communication among different devices extremely expensive. During computation, transfers over this interface can sometimes be overlapped with GPU kernels, if the latter is large enough. However, traffic through host memory increases with the number of GPUs, which ultimately limits performance scaling with GPU parallelism. The effect was analyzed in Chapter 5.6 for the non-hermitian system solver, which does require constant data exchange between devices.

7.4 Perspectives

In the last years, there has been an outright boom in the already rapidly evolving field of GPU accelerators. The period of this thesis alone covers three major generations of NVIDIA GPUs, each introducing new features that are of interest for computing applications. While the first CUDA compatible devices required very structured addressing patterns and were restricted to

single precision, ensuing generations not only relaxed restrictions on memory access, but also introduced increasingly better double precision support, error correction and caching. Op top of that, each generation was about twice as powerful as the last.

Evolution of GPU-accelerated applications goes hand in hand with that of GPU hardware. The TMI routine presented in this work was rendered possible only with introduction of double precision support, while the non-hermitian system solver greatly benefits from the capabilities of the Fermi architecture. Consequently, future applications are also directed by the capabilities of upcoming GPU generations.

The next series of NVIDIA GPUs, the series 500, will still be a modified variant of the series 400; it promises to offer better performance at a stable price, but the new features are rather directed to graphics processing. It would be more interesting how the next large evolutionary step looks like, but little is disclosed to the public so far. The only safe assumption is that there will be again more performance for less money, and less power. As for new features, one might hope for improved communication between GPUs, i.e. by allowing point-to-point communication between GPU interfaces without passing host RAM. The PCIe standard allows this (s. Chapter 2.5.1), so the problem must be in the present model of CPU-GPU coupling. Perhaps future CUDA drivers will be more multithreaded, and allow inherently synchronized interaction with multiple devices. An alternative would be to fuse GPU memory spaces and computing arrays on a low level, making multiple devices visible to the application programmer as a single large entity.

Generally, attention should be paid to the evolution of software design kits. In this work, a combination of OpenMP and CUDA was used to program heterogeneous multi-CPU/multi-GPU systems. However, there are also efforts towards a unified programming environment for heterogeneous multicore systems. The most notable example is the OpenCL standard [182], which is managed by the non-profit consortium Khronos Group. OpenCL started out as GPGPU interface and shares much functionality with CUDA, but it aims to incorporate other architectures as well, like the Cell BE. At the time of writing, OpenCL did not offer any benefits over the setup used in this work, but its wider range of target platforms gives it the potential to develop into a more capable alternative.

For heterogeneous multicore systems in bioelectrical imaging, more computational capabilities (at stable expenses) open the door for more capable software. For example, modeling the bioelectrical properties of life tissue is a field that can almost infinitely be refined to be more realistic anatomically. But also data fusion from different technologies or even run-time refinement of anatomical models from instantaneous measurement data could be rendered feasible with sufficient computational power available.

8 Conclusion

The contributions of this thesis are very practical. They address well-defined computational challenges encountered in biomedical imaging and use affordable hardware upgrades to substantially reduce computation time spent on these problems. In their bioelectrical imaging environment they allow modeling and simulation in levels of anatomical detail that were considered unfeasible before (s. Chapters 4.1 and 5.1), and real-time EEG signal processing at very high resolution (s. Chapter 6).

Note that the cudaEEG software is the only part of this work that is specific to the field of bioelectrical imaging. The TMI and complex BiCG implementations on the other hand are in no way restricted to this. Both address common linear algebra problems with a wide range of possible applications. The TMI routine fills in an important gap completely left out by previous work on GPU acceleration, and the non-hermitian system solver targets a previously neglected class of problems while outperforming related implementations.

In summary it can be concluded that both the field of bioelectrical imaging and the field of general GPU-accelerated linear algebra benefit from this work.

Appendix A: References

[1] F. Ries, T. De Marco, M. Zivieri, R. Guerrieri, "Triangular matrix inversion on Graphics Processing Unit", *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pp.1-10, 2009.

[2] F. Ries, T. De Marco, R. Guerrieri, "Triangular Matrix inversion on Heterogeneous Multicore Systems", In PrePrints, *to appear in IEEE Transactions on Parallel and Distributed Systems (TPDS).*

[3] F. Ries, T. De Marco, R. Guerrieri, "Solving Large Non-Hermitian Linear Systems on Multi-GPU Accelerated Workstations", *submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS).*

[4] T. De Marco, F. Ries, M. Guermandi, R. Guerrieri, "EIT Forward Problem Parallel Simulation Environment with Anisotropic Tissue and Realistic Electrode Models", *submitted to IEEE Transactions on Biomedical Engineering (TBME).*

[5] E. Niedermeyer and F.L. da Silva, "Electroencephalography: Basic Principles, Clinical Applications, and Related Fields," Lippincot Williams & Wilkins, 2004.

[6] J. Malmivuo and R. Plonsey, "Bioelectromagnetism - Principles and Applications of Bioelectric and Biomagnetic Fields", *Oxford University Press*, New York, chapter 13.3, 1995.

[7] P.L. Nunez PL, R. Srinivasan, "Electric fields of the brain: The neurophysics of EEG", *Oxford University Press*, 1981.

[8] M. Cheney, D. Isaacson, and J. Newell, "Electrical impedance tomography," *SIAM review*, vol. 41, no. 1, pp. 85–101, 1999.

[9] R. H. Bayford, "Bioimpedence tomography (Electrical Impedence Tomographyerr)," *Annu. Rev. Biomed. Eng.*, vol.8, pp. 63-91,2006.

[10] J. Malmivuo and R. Plonsey, "Bioelectromagnetism - Principles and Applications of Bioelectric and Biomagnetic Fields", *Oxford University Press*, New York, 1995.

[11] T. Ferree, K. Eriksen, and D. Tucker, "Regional head tissue conductivity estimation for improved EEG analysis," Biomedical Engineering, IEEE Transactions on, vol. 47, no. 12, pp. 1584–1592, 2002.

[12] S. Goncalves, J. de Munck, J. Verbunt, F. Bijma, R. Heethaar, and F. Lopes da Silva, "In vivo measurement of the brain and skull resistivities using an EIT-based method and realistic models for the head," IEEE Transactions on Biomedical Engineering, vol. 50, no. 6, pp. 754–767, 2003.

[13] K. Wendel, O. Väisänen, J. Malmivuo, et al., "EEG/MEG Source Imaging: Methods, Challenges, and Open Issues*," Computational Intelligence and Neuroscience*, vol. 2009, Article ID 656092, 12 pages, 2009.

[14] H. Hallez, B. Vanrumste, R. Grech, J. Muscat, W. De Clercq, A. Vergult, Y. D'Asseler, K.P. Camilleri, S.G. Fabri, S. Van Huffel, and I. Lemahieu, "Review on solving the forward problem in EEG source analysis", J Neuroeng Rehabil. 4: 46, 2007.

[15] A. Tarantola, "Inverse Problem Theory and Methods for Model Parameter Estimation", *SIAM*, 2004.

[16] C. Ramon, P.H. Schimpf and J. Haueisen, "Influence of head models on EEG simulations and inverse source localizations", *Biomed Eng Online*, 5: 10, 2006.

[17] R. Grech, T. Cassar, J. Muscat, K.P. Camilleri, S.G. Fabri, M. Zervakis, P. Xanthopoulos, V. Sakkalis and B. Vanrumste, "Review on solving the inverse problem in EEG source analysis", *J Neuroeng Rehabil.,* 5: 25, 2008.

[18] C. Chen, M. Ju, Y. Sun and C.K. Lin, "Model analyses of visual biofeedback training for EEG-based brain-computer interface", *Journal of Computational Neuroscience*, Vol. 27, no. 3, pp. 357-368, 2009.

[19] R. Penrose, "A generalized inverse for matrices", *Proceedings of the Cambridge Philosophical Society*, 51, pp. 406–413, 1955.

[20] F. Vatta, F. Meneghini, F. Esposito, S. Mininel, and F. Di Salle, "Realistic and Spherical Head Modeling for EEG Forward Problem Solution: A Comparative Cortex-Based Analysis", *Computational Intelligence and Neuroscience*, 2010.

[21] M. Fuchs, R. Drenckhahn, H.-A. Wischmann, and M. Wagner, "An improved boundary element method for realistic volume-conductor modeling," *IEEE Transactions on Biomedical Engineering*, vol. 45, no. 8, pp. 980–997, 1998.

[22] P. Berg, M. Scherg, “A fast method for forward computation of multiple-shell spherical head models,” *Electroencephalography and Clinical Neurophysiology*, Volume 90, Issue 1, Pages 58-64, ISSN 0013-4694, Jan. 1994.

[23] W. Jiang "FieldTrip – the MATLAB toolbox for EEG/MEG analysis", Stone Studio, posted February 21, 2010.

[24] C. Wolters, A. Anwander, X. Tricoche, D. Weinstein, M. Koch, and R. MacLeod, “Influence of tissue conductivity anisotropy on EEG/MEG field and return current computation in a realistic head model: a simulation and visualization study using high-resolution finite element modeling,” *NeuroImage*, vol. 30, no. 3, pp. 813–826, 2006.

[25] D. Shattuck and R. Leahy, “BrainSuite: an automated cortical surface identification tool,” in *Medical Image Computing and Computer-Assisted Intervention–MICCAI*, Springer, 2000.

[26] J. Li, D. Yan, "Solving the EEG Forward Problem by FDM and FEM," *Proceedings of the 2nd International Conference on Biomedical Engineering and Informatics*, 2009.

[27] M. Cook and Z. Koles, “A high-resolution anisotropic finite-volume head model for eeg source analysis,” in Engineering in Medicine and Biology Society, 2006. EMBS ’06. 28th Annual International Conference of the IEEE, pp. 4536 –4539, Aug. 2006.

[28] R. Barrett, M. Berry, T.F. Chan, et al., “Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods”, SIAM, Philadelphia, pp. 19-20, 1994.

[29] D.W. Shattuck and R.M. Leahy, “BrainSuite: An Automated Cortical Surface Identification Tool,” *Medical Image Analysis*, 8(2):129-142, 2002.

[30] B. Fischl, D.H. Salat, E. Busa, M. Albert, M. Dieterich, C. Haselgrove, A. van der Kouwe, R. Killiany, D. Kennedy, S. Klaveness, A. Montillo, N. Makris, B. Rosen, A.M. Dale, “Whole brain segmentation: automated labeling of neuroanatomical structures in the human brain,” *Neuron* 33, 341-355, 2002.

[31] B. Fischl, D.H. Salat, A.J.W. van der Kouwe, N. Makris, F. Ségonne and A.M. Dale, “Sequence-Independent Segmentation of Magnetic Resonance Images,” *NeuroImage 23*:S69-S84, 2004.

[32] T. Ferree, K. Eriksen, and D. Tucker, “Regional head tissue conductivity estimation for improved EEG analysis,” Biomedical Engineering, IEEE Transactions on, vol. 47, no. 12, pp. 1584–1592, 2002.

[33] G. Dong, J. Zou, R. Bayford, X. Ma, S. Gao, W. Yan, and M. Ge, “The comparison between FVM and FEM for EIT forward problem,” *IEEE Transactions on Magnetics*, vol. 41, no. 5, 2005.

[34] S. Goncalves, J. de Munck, J. Verbunt, F. Bijma, R. Heethaar, and F. Lopes da Silva, “In vivo measurement of the brain and skull resistivities using an EIT-based method and realistic models for the head,” IEEE Transactions on Biomedical Engineering, vol. 50, no. 6, pp. 754–767, 2003.

[35] R. Sadleir and A. Argibay, “Modeling skull electrical properties,” Annals of Biomedical engineering, vol. 35, no. 10, pp. 1699–1712, 2007.

[36] C. Phillips, M. D. Rugg, K.J. Friston, “Anatomically Informed Basis Functions for EEG Source Localization: Combining Functional and Anatomical Constraints,” *NeuroImage*, Volume 16, Issue 3, Part 1, Pages 678-695, July 2002.

[37] N.G. Gençer and S.J. Williamson, “Characterization of Neural Sources with Bimodal Truncated SVD Pseudo-Inverse for EEG and MEG Measurements,” *IEEE Transactions on Biomedical Engineering*, 45(7):827-838, 1998.

[38] A.Tikhonov , “Solution of incorrectly formulated problems and the regularization method,” *Soviet Math Dokl,* pp. 1035–1038, 1963.

[39] N. Subramaniyam, O. Väisänen, K. Wendel and J. Malmivuo. “ Cortical potential imaging using L-curve and GCV method to choose the regularisation parameter“, *Nonlinear Biomed Phys,* 4 (Suppl 1): S4, 2010.

[40] R. Grech, T. Cassar, J. Muscat, K.P. Camilleri, S.G. Fabri, M. Zervakis, P. Xanthopoulus, V. Sakkalis and B. Vanrumste, “Review on solving inverse problem in EEG source analysis,” *Journal of Neuroengineering and Rehabilitation*, 5:25, 2008.

[41] I.F. Gorodnitsky and B.D. Rao, “Sparse Signal Reconstruction from Limited Data Using FOCUSS: A Re-weighted Minimum Norm Algorithm”, *IEEE Transactions on Signal Processing,* 45(3):600-615, 1997

[42] G. Xin, M. Xinshanand X. Yaoqin, “A new algorithm for EEG source reconstruction based on LORETA by contracting the source region,” *Progress in Natural Science*, 12(11):859-862, 2002.

[43] R.D. Pascual-Marqui, “Standardized low resolution brain electromagnetic tomography (sLORETA): technical details,” *Methods & Findings in Experimental & Clinical Pharmacology*, 24D: 5-12, 2002.

[44] G. Backus G and F. Gilbert, “The resolving power of gross earth data,” *Geophys*. J. R. Astr. Soc., 16, pp. 169-205, 1968.

[45] H. Liu, P.H. Schimpf, G. Dong, X. Gao, F. Yang, S. Gao, “Standardized Shrinking LORETA-FOCUSS (SSLOFO): A New Algorithm for Spatio-Temporal EEG Source Reconstruction,” *IEEE Transactions on Biomedical Engineering*, 52(10), pp.1681-1691, 2005.

[46] W. Aspray, "The Intel 4004 microprocessor: What constituted invention?," *Hist. Comput.* 19(3), pp. 4–15, 1997.

[47] G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[48] M. Kanellos, "New life for Moore's Law," *CNET News.com*, 2005.

[49] ITRS (International Technology Roadmap for Semiconductors), 2005 ed., http://public.itrs.net, 2005.

[50] S. Borkar, “Microarchitecture and Design Challenges for Gigascale Integration,” *Proc. Int’l Symp. Microarchitecture (MICRO)*, Dec.2004.

[51] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik and O. O. Storaasli, “State-of-the-Art in Heterogeneous Computing,“ *Scientific Programming*, 18(1), pp. 1—33, 2010.

[52] S. Cho and R.G. Melhem, "On the Interplay of Parallelization, Program Performance and Energy Consumption," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(3):342~353, Mar 2010.

[53] Intel, “A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors,” *Technology@Intel Magazine*, pp. 1-11, May 2005.

[54] K. Asanovi, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” *Technical Report UCB/EECS-2006-183*, Univ. of California, Berkeley, Dec. 2006.

[55] S. Cotofana, S. Vassiliadis, "On the Design Complexity of the Issue Logic of Superscalar Machines", EUROMICRO 1998, pp. 10277-10284, 1998.

[56] R. van der Pas, "Memory Hierarchy in Cache-Based Systems", Sun Microsystems, Santa Clara, California, Nov. 2002.

[57] S. Rusu, S.Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, S. Kottapalli and S. Vora, "A 45 nm 8-Core Enterprise Xeon¯ Processor," *IEEE Journal of Solid-State Circuits*, 45(1), pp. 7 - 14, Jan 2009.

[58] M.D. Hill, M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, 41:7, pp. 33-38, Jul, 2008.

[59] D. Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer*, 38(5), pp. 11-13, May, 2005

[60] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. Williams and K. Yelick, “Exascale computing study: Technology challenges in achieving exascale systems, “ *Technical report*, DARPA IPTO, 2008.

[61] M. Herlihy and V. Luchangco, "Distributed computing and the multicore revolution,", SIGACT News 39, 1, pp. 62-72, Mar 2008.

[62] N. Aggarwal, P. Ranganathan, N.P. Jouppi and J.E. Smith, "Isolation in Commodity Multicore Processors", *Computer*, vol. 40, no. 6, pp. 49-59, June 2007.

[63] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.* C-21: 948, 1972.

[64] U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, J.H. Ahn, P. Mattson and J.D. Owens, "Programmable stream processors," *IEEE Computer*, Aug 2003.

[65] J. Dongarra, H. Meuer, H. Simon and E. Strohmaier, "Recent trends in high performance computing," *The Birth of Numerical Analysis*, World Scientific, pp. 93-107, 2009.

[66] M. Hassaballah, S. Omran, and Y. Mahdy,"A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications," Computer Journal, vol.51, 6, pp. 630-649, Jan 2007.

[67] S. Murugesan, “Harnessing Green IT: Principles and Practices,” *IEEE IT Professional*, pp 24-33, Jan–Feb 2008.

[68] E. Miki, "Cell Phone Technology for Super 3G and Beyond", *Microprocessor Forum*, San Jose, CA, May 22, 2007.

[69] J.A. Kahl, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy, “Introduction to the Cell Multiprocessor,” *IBM Journal of Research and Development*, 49(4), 2005.

[70] S.K. Moore, "Winner: Multimedia Monster", *IEEE Spectrum*, Jan 2006.

[71] M. Linklater, "Optimizing Cell Core*", Game Developer Magazine*, pp. 15–18, Apr 2007.

[72] A. White, “Roadrunner: Science, Cell and a Petaflops/s”, *Fall Creek Falls Conference*, Los Alamos National Laboratory, Sep 2008.

[73] M. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani and M. Chiu, "Computing Models for FPGA-Based Accelerators", *Computing in Science and Engineering*, 10(6), Nov/Dec 2008.

[74] M. Kühnle, M. Hübner, J. Becker, A. Deledda, C. Mucci, F. Ries, A. M. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia, T. De Marco, and F. Campi, "An Interconnect Strategy for a Heterogeneous, Reconfigurable SoC", *Design&Test*, vol. 5, 25, pp. 442-451, Sep 2008.

[75] A. Deledda, C. Mucci, A. Vitkovski, P. Bonnot, A. Grasset, P. Millet, M. Kuehnle, F. Ries, M. Huebner, J. Becker, M. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia, F. Campi and T. DeMarco, "Design of a HW/SW communication infrastructure for a heterogeneous reconfigurable processor", *Conference on Design, Automation and Test in Europe* (Munich, Germany), pp. 1352-1357, Mar 2008.

[76] C. Shu, S. Kwon and K. Gaj, "FPGA accelerated tate pairing based cryptosystems over binary fields", *IEEE International Conference on Field Programmable Technology*, pp.173-180, Dec. 2006.

[77] C. R. Clark, D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'04), pp. 249-257, 2004.

[78] S.C. Chrepta, "FPGA-based Implementation of Signal Processing Systems (Woods, R. et al: 2008) [Book review]", *Signal Processing Magazine*, IEEE , vol.26, no.6, pp.206-207, Nov. 2009.

[79] A. Bland, “Towards Exascale”, *HPC projects*, Feb/Mar 2010.

[80] R. Weber, A. Gothandaraman, R.J. Hinde, G.D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study", *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, PrePrints, 2010.

[81] M.J. Harris , W.V. Baxter , T. Scheuermann , A. Lastra, “Simulation of cloud dynamics on graphics hardware”, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, San Diego, California, July 26-27, 2003.

[82] J. Bolz, I. Farmer, E. Grinspun and P. Schröder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid", *Proceedings of SIGGRAPH 2003*, 2003.

[83] J. Krüger and R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms”, *ACM Transactions on Graphics (TOG)*, vol .22, no .3, July 2003.

[84] W. Li, X. Wei and A. Kaufman, "Implementing lattice Boltzmann computation on graphics hardware ", *The Visual Computer*, Vol. 19, No. 7-8, pp. 444-456, 2004.

[85] E. Lindholm, J. Nickolls, S. Oberman. and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture*," IEEE Micro*, vol. 28, no. 2, pp. 39-55, 2008.

[86] AMD Corporation, “Close to Metal Technology Unleashes the Power of Stream Computing", AMD Press Release, Nov. 14, 2006.

[87] NVIDIA Corporation, “NVIDIA CUDA compute unified device architecture programming guide”, ver. 3.2, Oct. 2010.

[88] M. Fatica, P. LeGresley, I. Buck, J. Stone, J. Phillips, S. Morton, and P. Micikevicius, “High Performance Computing with CUDA”, *SC08*, 2008.

[89] A. Moravánszky, “Dense matrix algebra on the GPU. In ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0”, W*ordware Publishing*, pp. 352—380, 2002.

[90] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware*",* In *Eurographics 2005, State of the Art Reports*, pp. 21-51, Aug 2005.

[91] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, SChennupaty, P. Hammarlund, R. Singhal and P. Dubey, “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”, *SIGARCH Comput. Archit. News* 38, 3, 451-460, Jun 2010.

[92] R. Bordawekar, U. Bondhugula and R. Rao: “Believe It or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application*!”, Technical Report RC24982*, IBM Thomas J. Watson Research Center, Apr. 2010.

[93] V. Volkov and J. W. Demmel; "Benchmarking GPUs to tune dense linear algebra", *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA: IEEE Press, pp. 1-11, 2008.

[94] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo and E.S. Quintana-Ortí, “Solving Dense Linear Systems on Graphics Processors”, *Proceedings of the 14th international Euro-Par Conference on Parallel Processing - Lecture Notes In Computer Science*, vol. 5168, Springer-Verlag, Berlin, Heidelberg, pp. 739-748, 2008.

[95] S. Tomov, J. Dongarra, M. Baboulin, “Towards dense linear algebra for hybrid GPU accelerated manycore systems”, Parallel Computing, vol. 36, 5-6, *Parallel Matrix Algorithms and Applications*, pp. 232-240, June 2010.

[96] L. Buatois, G. Caumon, and B. Levy, “Concurrent number cruncher: a GPU implementation of a general sparse linear solver”, *Int. J. Parallel Emerg. Distrib. Syst.* 24, 3, pp. 205-223, Jun 2009.

[97] A. Gaikwad, I. Muni Toke,” Parallel Iterative Linear Solvers on GPU: A Financial Engineering Case”, Proceedings of the 18th International Conference on Parallel, Distributed and Network-Based Computing - PDP2010, IEEE CPS, 2010.

[98] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, “A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem“, to appear at the *18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP2010)*, 2010.

[99] A. Cevahir, A. Nukada, and S. Matsuoka, “Fast Conjugate Gradients with Multiple GPUs”, In *Proceedings of the 9th International Conference on Computational Science*: Part I, 2009.

[100] T. Jost, S. Contassot-Vivier, and S. Vialle, “An efficient multi-algorithms sparse linear solver for GPUs”, In *ParCo2009*, France Lyon, 2009.

[101] M. Mehri Dehnavi, D. Fernandez and D. Giannacopoulos: “Finite element sparse matrix vector multiplication on GPUs”. *IEEE Transactions on Magnetics*, vol. 46, no. 8, pp. 2982-2985, Aug 2010.

[102] N. Bell and M. Garland, “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors“. *in Proc. Supercomputing ’09,* Aug 2009.

[103] M. Geveler, D. Ribbrock, D. Göddeke and S. Turek, “Lattice-Boltzmann Simulation of the Shallow-Water Equations with Fluid-Structure Interaction on Multi- and Manycore Processors”, *Accepted in: Facing the Multicore Challenge*, Heidelberg, Germany, Mar. 2010.

[104] J. A. Anderson, Chris D. Lorenz, A. Travesset, “General purpose molecular dynamics simulations fully implemented on graphics processing units”, Journal of Computational Physics, Vol. 227, No. 10, 1, pp. 5342-5359, ISSN 0021-9991, May 2008.

[105] S. Hampton, P.K. Agarwal, S.R. Alam, P.S. Crozier, "Towards microsecond biological molecular dynamics simulations on hybrid processors", *2010 International Conference on High Performance Computing*, pp. 98-107, Jun 2010.

[106] T. Nagatake and T. Kunugi, "Application of GPU to computational multiphase fluid dynamics",*IOP Conf. Ser.: Mater. Sci. Eng.*, 10, 012024, 2010.

[107] L. Weiguo, B. Schmidt, G. Voss and W.Müller-Wittig, "Streaming Algorithms for Biological Sequence Alignment on GPUs", *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270-1281, 2007.

[108] C. Trapnell, M.C. Schatz, “ Optimizing data intensive GPGPU computations for DNA sequence alignment”, *Parallel Computing*, 2009.

[109] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases", In *IEEE International Workshop on High Performance Computational Biology* (HiCOMB 2009), 2009.

[110] M.C. Schatz, C. Trapnell, “Optimizing data intensive GPGPU computations for DNA sequence alignment Parallel Computing”, vol 35, pp. 429-440, 2009.

[111] J. Setoain, M. Prieto, C. Tenllado and F. Tirado, “GPU for Parallel On-Board Hyperspectral Image Processing”, *International Journal of High Performance Computing Applications*, vol. 22, no. 4, pp. 424-437, 2008.

[112] Y. Su and Z. Xu. “ Parallel implementation of wavelet-based image denoising on programmable PC-grade graphics hardware”, *Signal Process. 90*, 8, pp. 2396-2411, Aug 2010.

[113] N. Cornelis, L. Van Gool, "Fast Scale Invariant Feature Detection and Matching on Programmable Graphics Hardware", CVPR 2008 Workshop (June 27th), 2008.

[114] S.P. Mohanty, "GPU-CPU multi-core for real-time signal processing", *International Conference on Computers in Education*, 2009 Digest of Technical Papers International Conference on Consumer Electronics, pp. 1-2, 2009.

[115] O. Roy, I. Jovanovic, A. Hormati, R. Parhizkarand and M. Vetterli, “Sound speed estimation using wave-based ultrasound tomography: Theory and GPU implementation”, *in Proc. SPIE Medical Imaging*, 2010.

[116] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel and F. Tirado, "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting", *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299-310, 2008.

[117] B. Block, P. Virnau and T. Preis: “Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model”, *Computer Physics Communications,* 181:9, 1549-1556, Sep. 2010.

[118] Q. Fang and D.A. Boas, “Monte Carlo Simulation of Photon Migration in 3D Turbid Media Accelerated by Graphics Processing Units”, *Opt. Express*, vol. 17, issue 22, pp. 20178-20190, 2009.

[119] AccelerEyes, “MATLAB GPU Computing”, 2010.

[120] MatWorks Inc., “MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs”, 2010.

[121] A. Sims,"Mathematica and NVIDIA in Action: See Your GPU in a Whole Different Light", *Wolfram Blog*, Wolfram Research, Sep 14. 2010.

[122] J. Bruckner, “Enabling GPU Computing in the R statistical Environment”,

[123] V. Minden, B. Smith, and Matthew G. Knepley, “Preliminary Implementation of PETSc Using GPUs”, to appear in Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering, Springer, 2010.

[124] P.N. Glaskowsky, “NVIDIA’s Fermi: The first Complete GPU Computing Architecture”, NVIDIA Corporation, White Paper, Sep 2009.

[125] Intel Corporation, “Periferal Component Interconnect Specification”, Revision 3.0, 2002.

[126] Intel Corporation, “Accelerated Graphics Port Interface Specification”, Revision 3.0, 2002.

[127] Jon Stokes, “PCI Express: An Overview”, *arstechnica*, Jul. 2004.

[128] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space", IEEE International Symposium on Parallel&Distributed Processing, pp. 1-12, 2009.

[129] T. Kreiss, B. Töpelt, D. Schuhmann, "Performance Comparison Between Single Configurations And SLI Setups", *Tom's Hardware*, Dec 2005.

[130] B. Töpelt, "Benchmark Analysis - 65% More Performance Through Hybrid-Crossfire", Tom's Hardware, Mar 2008.

[131] T. Kreiss, “How Much Power does your Graphics Car Need?”, *Tom’s Hardware*, 21 Jan 2009.

[132] L. Dagum, and R. Menon, "OpenMP: an industry standard API for shared-memory programming", *Computational Science & Engineering*, IEEE , vol. 5, no. 1, pp. 46-55, 1998.

[133] W. Gropp, E. Lusk, Ewing and A. Skjellum, “Using MPI: portable parallel programming with the message-passing interface”, *Scientific And Engineering Computation Series*, MIT Press, *C*ambridge, MA, USA, 1994.

[134] W. Nasri Z. and Mahjoub, “Optimal parallelization of a recursive algorithm for triangular matrix inversion on MIMD computers”, Parallel Computing, vol. 27, no. 13, pp. 1767-1782, 2001.

[135] D. Heller, “A survey of parallel algorithms in numerical linear algebra”, SIAM Review, vol. 20, pp. 740-777, 1978.

[136] Y. Robert, “The Impact of Vector and Parallel Architectures on the Gaussian Elimination”, Halstead Press, New York, 1990.

[137] V. Strassen, “Gaussian elimination is not optimal”, Numerical Mathematics, vol. 13, pp. 354-356, 1969.

[138] S.M. Balle, P.C. Hansen and N. Higham, “A Strassen-type matrix inversion algorithm”, Advances in Parallel Algorithms, IOS Press, pp. 22-30, 1994.

[139] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney and D. Sorensen, "LAPACK Users' Guide, Third Edition”, Society for Industrial and Applied Mathematics, 1999.

[140] L. Buatois, G. Caumon, and B. Levy, “Concurrent number cruncher: a GPU implementation of a general sparse linear solver”, Int. J. Parallel Emerg. Distrib. Syst. 24, 3, pp. 205-223, Jun 2009.

[141] A. Cevahir, A. Nukada, and S. Matsuoka, “Fast Conjugate Gradients with Multiple GPUs”, In Proceedings of the 9th International Conference on Computational Science: Part I, 2009.

[142] P.Y. Aquilanti, S. Petiton, H. Calandra, “Parallel Auto-tuned GMRES Method to Solve Complex Non-Hermitian Linear Systems”, iWAPT: Fifth international Workshop on Automatic Performance Tuning, VECPAR’10, 2010.

[143] B. Carpentieri, Y.-F. Jing, T.-Z. Huang, Y. Duan, “A class of linear solvers built on the Biconjugate A-Orthonormalization Procedure for solving unsymmetric linear systems”, arXiv (United States), 2010

[144] T. Jost, S. Contassot-Vivier, and S. Vialle, “An efficient multi-algorithms sparse linear solver for GPUs”, In ParCo2009, France Lyon, 2009.

[145] V. Minden, B. Smith, and Matthew G. Knepley, “Preliminary Implementation of PETSc Using GPUs”, to appear in Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering, Springer, 2010.

[146] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA", NVIDIA Technical Report NVR-2008-004, Dec 2008.

[147] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, “A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem“, to appear at the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP2010), 2010.

[148] NVIDIA Corporation, “CUBLAS Library”, NVIDIA, ver. 2.2, 2008.

[149] P. Micikevicius, “3D finite difference computation on GPUs using CUDA”, *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (Washington, D.C., March 08 - 08, 2009). GPGPU-2, vol. 383. ACM, New York, NY, pp. 79-84, 2009.

[150] A. Gaikwad, I. Muni Toke,” Parallel Iterative Linear Solvers on GPU: A Financial Engineering Case”, *Proceedings of the 18th International Conference on Parallel, Distributed and Network-Based Computing* - PDP2010, IEEE CPS, 2010.

[151] D. Goddeke, R. Strzodka, "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid*", IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, 2010.

[152] M. Harris, “Optimizing Parallel Reduction in CUDA”, *NVIDIA Developer Technology*, NVIDIA Corporation, 2008.

[153] D. Shreiner, M. Woo, J. Neider, T. Davis, "OpenGL® Programming Guide: The Official Guide to Learning OpenGL®”, ver. 2.1, 6th ed., Addison-Wesley Professional, 2007.

[154] M. Sainz , R. Pajarola, "Point-based rendering techniques*", Computers & Graphics*, 2004.

[155] J.O. Ollikainen, M. Vauhkonen, P.A. Karjalainen, J.P Kaipio, "Effects of electrode properties on EEG measurements and a related inverse problem", *Med. Eng. Phys*., 22(8), pp.535-45, Oct. 2000.

[156] S. Baillet, L. Garnero, "A Bayesian approach to introducing anatomo-functional priors in the EEG/MEG inverse problem", *IEEE Transaction on Biomedical Engineering*, Vol. 44, 5, pp. 374-285, May 1997.

[157] R. Srinivasan, “Anatomical constraints on source models for high-resolution EEG and MEG derived from MRI”, *Technol. Cancer Res. Treat*., 5(4), pp. 389-399, Aug 2006.

[158] A.C. Bovik, "Handbook of image and video processing", Academic Press - Computers, p. 542, 2005.

[159] R.M. Stallman, “Using and porting the GNU compiler collection”, Free Software Foundation, Jun. 2001.

[160] NVIDIA Corporation, “PTX: Parallel Thread Execution ISA”, Ver 2.1, Apr. 2010.

[161] NVIDIA Corporation, “The CUDA compiler driver NVCC”, Ver, 3.2, Aug. 2010.

[162] G. Boverman, B. Kim, D. Isaacson, and J. Newell, “The complete electrode model for imaging and electrode contact compensation in electrical impedance tomography,” *in 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 3462–3465, 2007.

[163] A.M. Dale, B. Fischl abd M.I. Sereno,"Cortical Surface-Based Analysis I: Segmentation and Surface Reconstruction", *NeuroImage* 9(2), pp. 179-194, 1999.

[164] B. Fischl, A. Liu and A.M. Dale, "Automated Manifold Surgery: Constructing Geometrically Accurate and Topologically Correct Models of the Human Cerebral Cortex", *IEEE Transactions on Medical Imaging*, 20(1), pp. 70-80, 2001.

[165] F-H. Lin, J.W. Belliveau, A.M. Dale and M.S. Hämäläinen, “Distributed Current Estimates Using Cortical Orientation Constraints”, *Human Brain Mapping*, 27, pp. 1-13, 2006.

[166] N. Galoppo, N.K. Govindaraju, M. Henson and D.. Manocha, “LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware”, In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing*. IEEE Computer Society, Washington, DC, p.3, 2005.

[167] V. Volkov and J. W. Demmel, “LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs”, EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2008-49 May 13, 2008.

[168] M.S. Bazaraa, H.D. Sherali and C.M. Shetty, “Nonlinear Programming – Theory and Algorithms”, New York: Wiley, 3rd ed., 2006.

[169] N. Harvey, D. Lun and P. Maymounkov, “Methods for Efficient Network Coding”. In *44th Allerton Annual Conference*, 2006.

[170] M. Nuwer, "Brain Mapping and Quantitative EEG", *Encyclopedia of the Neurological Sciences*, Academic Press, NY, pp. 447-452, 2003.

[171] S. Butnik, "Neurofeedback in adolescents and adults with attention deficit hyperactivity disorder", *Journal of Clinical Psychology*, Vol. 61, 5, pp. 621-625, May 2005.

[172] D. Trudeau, "EEG Biofeedback for Addictive Disorders -- The State of the Art in 2004." *Journal of Adult Development*, Vol 12, Nos. 2/3, Aug. 2005.

[173] C. Hammond, "Neurofeedback Treatment of Depression and Anxiety." *Journal of Adult Development*, Vol 12, Nos. 2/3, Aug. 2005.

[174] E.C. Leuthardt, G. Schalk, J. Roland, A.Rouse, D.W Moran, "Evolution of brain-computer interfaces: going beyond classic motor physiology", *Neurosurg Focus*, 27:1E4, 2009.

[175] A. Luo and T.J. Sullivan, "A user-friendly SSVEP-based brain–computer interface using a time-domain classifier", *J. Neural Eng.* 7 026010, 2010.

[176] R.M Friborg, S. Hauberg and K. Erleben, "GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking", *At the CVGPU workshop at European Conference on Computer Vision (ECCV)* 2010.

[177] E.B. Ford, "Parallel algorithm for solving Kepler’s equation on Graphics Processing Units: Application to analysis of Doppler exoplanet searches ", *New Astronomy*, Vol. 14, 4, pp. 406-412, May 2009.

[178] A. Lee, C. Yau, M. Giles, A. Doucet and C. Holmes, "On the utility of graphics cards to perform massively parallel simulation with advanced Monte Carlo methods", To appear in: *Journal of Computational & Graphical Statistics*.

[179] M. Januszewski and M. Kostur, "Accelerating numerical solution of Stochastic Differential Equations with CUDA", *Computer Physics Communications*, Vol. 181, 1, pp. 183-188, Jan. 2010.

[180] E.H. Phillips, Y. Zhang, R.L Davis, J.D. Owens, "Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units", *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, Jan 2009.

[181] NVIDIA Corporation (2010), “CUDA Zone – CUDA Community Showcase”, retrieved 30 Jan. 2011.

[182] J.E. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems", *IEEE Design & Test*, Vol. 12, 3, pp. 66-73, May 2010.

[183] B. He, “Modeling & Imaging of Bioelectrical Activity: Principles and Applications (Bioelectrical Engineering)”, Springer, 1. ed., Apr. 2004.

Appendix B: Abbreviations

ADHD	Attention Deficit Hyperactivity Disorder
AGP	Accelerated Graphics Port
AMD	Advanced Micro Devices
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ATA	Advanced Technology Attachment
ATI	Array Technologies Incorporated
BCI	Brain Computer Interface
BE	Broadband Engine
BEM	Boundary Element Method
BiCG	Bi-Conjugate Gradient
BiCGStab	Bi-Conjugate Gradient Stabilized
BMI	Brain Machine Interface
CG	Conjugate Gradient
CGS	Conjugate Gradient Sqared
CPU	Central Processing Unit
CSF	Cerebrospinal Fuid
CTM	Close To Metal
CUDA	Compute Unified Device Architcture
DP	Double Precision
DRAM	Dynamic random access memory
EEG	Electroencephalygraphy
EIT	Electrical Impedance Tomography
FDM	Finite Difference Method
FEM	Finite Element Method
FLOP	Floating Point Operation
fMRI	functional Magnetic Resonance Imaging
FOCUSS	Focal Underdetermined System Solver
FPGA	Field Programmable Gate Array
FSB	Front Side Bus
FVM	Finite Volume method
GC	Geometry Control
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate
GFLOP/s	Billion Floating Point Operations per second
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
I/O	Input/Output
IBM	International Business Machines
IEEE	Institute of Electrical and Electronics Engineers
ITRS	International Technology Roadmap for Semiconductors
LORETA	Low Resolution Brain Electromagnetic Tomography

LUT	Look-Up Table
MB	Memory Bus
MC	Machine Code
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
MISD	Multiple Instruction Single Data
MNE	Minimum Norm Estimate
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MTIFI	Multi-Thread Instruction Fetch & Issue
OS	Operating System
PC	Personal Computer
PCI	Periferal Component Interconnect
PCIe	Periferal Component Interconnect express
PET	Positron Emission Tomography
PPE	Power Processor Element
PTX	Parallel Thread eXecution
PU	Processing Unit
RAM	Random Access Memory
RTMI	Recursive Triangular Matrix Inversion
SDK	Software Design Kit
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SLI	Scalable Link Interface
sLORETA	standardized Low Resolution Brain Electromagnetic Tomography
SM	Stream Multiprocessor
SMC	Stream Multiprocessor Control
SoC	System-on-Chip
SP	Stream Processor; Single Precision
SPE	Synergistic Processing Element
spMV	sparse Matrix-Vector product
SSLOFO	Shrinking Standardized LORETA-FOCUSS
TDP	Thermal Design Power
TMI	Triangular Matrix Inversion
TPC	Texture Processing Cluster
TU	Texture Unit
WMN	Weighted Minimum Norm