NTNU – Trondheim
Norwegian University of
Science and Technology

FDTD on Distributed Heterogeneous Multi-GPU Systems

Eirik Myklebost

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project will expand on previous work done by Eirik Myklebost on GPU behavior to include studying effect on systems with multiple GPUs.

We will primarily extend the work of Andreas Berg Skomedal, consisting of: A Yee-bench's FTDT method ported to CUDA and a heterogeneous scheduler, which divides work among CPU and GPU cores, or similar applications. Developing an efficient application for a variety of different GPUs, across different NVIDIA architectures, will be included.

This project may also include further development on the scheduler; making it work on systems with multiple GPUs.

Assignment given: 17. January 2014
Supervisor: Dr. Anne Cathrine Elster, IDI

Abstract

Finite-Difference Time-Domain (FDTD) is a popular technique for modeling computational electrodynamics, and is used within many research areas, such as the development of antennas, ultrasound imaging, and seismic wave propagation. Simulating large domains can however be very compute and memory demanding, which has motivated the use of cluster computing, and lately also the use of Graphical Processing Units (GPUs).

The previous work of Andreas Berg Skomedal's master thesis [1] from May 2013 includes a heterogeneous FDTD implementation, in the sense that it schedules domains between a CPU and a GPU on a single system. The implementation is a benchmarking code based on the Yee_bench [2] code by Ulf Andersson, and focuses on the performance of simulating many small individual FDTD domains.

This thesis introduces a new FDTD implementation based on the work by Skomedal and Andersson. The code is written in C++ and CUDA, and uses a decomposition approach as opposed to scheduling, which allows for larger domains to be divided among multiple execution units. It supports the use of both a CPU and several CUDA capable GPUs on a single system, in addition to multi-node execution through the use of the Message Passing Interface (MPI). A discussion of the differences between the CUDA capable GPU architectures, and how they affect the performance of the FDTD algorithm, is also included.

The results shows a performance increase of 66% when simulating large domains on two GPUs compared to a single GPU. Using the CPU in addition to one or two fast GPUs is shown to give a slight improvement, but the main advantage is the possibility to simulate larger domains. Results from multi-node executions is also included, but they refer to poor performance values, due to being severely limited by a 100 Mbit/s Ethernet.

The work of this thesis includes a working FDTD decomposition implementation, that can be executed on a cluster of heterogeneous systems with a multi-core CPU, and one or several CUDA capable GPUs. It is also written with the intention that it should be easily extendable to also work with non-CUDA capable GPUs. As with the previous work by Skomedal and Andersson, this implementation is only a benchmarking code, and is not suited for real world problems. It is instead intended to be used as a basis for future works, or as an example on how to do FDTD on a cluster of heterogeneous multi-GPU systems.

Acknowledgements

I wish to thank my advisor Dr. Anne C. Elster and my co-advisor Dr. Malik Khan, for their guidance and assistance during the planning and work related to my thesis. I would also like to extend my thanks to the members of the HPC-Lab at NTNU, for all their support and advices.

Finally, a special thank should be given to Andreas Berg Skomedal for all the work provided in his master thesis, which has been used as a basis for my work.

Eirik Myklebost, Trondheim, Norway, June 20, 2014.

Contents

List of Figures

List of Tables

Acronyms

ALU Arithmetic Logic Unit.

AMD Advanced Micro Devices.

API Application Programming Interface.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DP Double-Precision.

DRAM Dynamic Random-Access Memory.

ECC Error Correcting Code.

FDTD Finite-Difference Time-Domain.

FLOP FLoating Point Operations.

FLOPS FLoating-point Operations Per Second.

FPU Floating Point Unit.

FTDT Finite-Difference Time-Domain.

GPGPU General-Purpose computing on Graphical Processing Units.

GPU Graphical Processing Unit.

ILP Instruction Level Parallelism.

ISA Instruction Set Architecture.

MIMD Multiple Instruction Multiple Data.

MPI Message Passing Interface.

NUMA Non-Uniform Memory Access.

NVCC NVIDIA CUDA Compiler.

PDC Center for Parallel Computers.

PTX Parallel Thread Execution.

RAW Read-After-Write.

SFU Special Function Units.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Thread.

SM Streaming Multiprocessor.

SMX Next Generation Streaming Multiprocessor.

SP Streaming Processor.

TMU Texture Mapping Unit.

TPC Texture/Processor Cluster.

UML Unified Modeling Language.

Chapter 1

Introduction

The simulation of electromagnetic fields is an important part of many research areas, such as the development of antennas, ultrasound imaging, and seismic wave propagation. The Finite-Difference Time-Domain (FDTD) method used in this thesis is a very popular technique for modeling computational electrodynamics, and was first proposed in 1966 by Kane Yee [3]. The method can however be very compute and memory demanding when used to simulate large domains, and the capabilities of a single system can become inadequate. Implementations have therefore been developed to scatter the computational load and memory requirements among multiple systems, such as those described in the articles: *A Parallel FDTD Algorithm Based on Domain Decomposition Method Using the MPI Library* [4], and *Parallel FDTD Simulation Using NUMA Acceleration Technique* [5].

Due to the popularity of video games and demanding real-time graphics, development of Graphical Processing Units (GPUs) has been rapid and lead to them greatly exceeding the computational power of traditional Central Processing Units (CPUs). The introduction of programming models like the Compute Unified Device Architecture (CUDA), has made it convenient to utilize the processing power of modern GPUs for scientific computing. GPUs have already proven to be very efficient at accelerating electromagnetic simulations using FDTD, as shown in the article: *CUDA Based FDTD Implementation* [6].

The goal of this thesis is to develop a solution that can do FDTD simulations on a multi-node setup, where each node can utilize both a multi-core CPU and one or multiple GPUs. The work done by Andreas Berg Skomedal as part of his master thesis [1], will be used as a basis. His work is in turn based on the Yee_bench code developed by Ulf Andersson [2]. Skomedal's work comprises of a CUDA port of the Yee_bench code, and a scheduler that distributes individual FDTD domains among a multi-core CPU and a single GPU. His CPU version of the Yee_bench code has also been extended to use OpenMP, to utilizes multiple cores. A common constraint among the codes from Yee_bench, Skomedal's work and this thesis, is that they are only meant as benchmarking codes, and therefore lack several features required for practical FDTD simulations.

The code covered in this thesis is mainly written in C++, with the addition of CUDA for GPU executions, OpenMP for multi-threaded CPU execution, and the Message Passing Interface (MPI) for multi-node execution. The solution uses the concept of an abstract *Worker class* to control each execution unit, which can be either a CPU or a CUDA capable GPU. The workers runs simultaneously as POSIX threads, and allows for multiple GPUs to be used within a single system. The Worker class also allows for future exten-

sions, by developing new subclasses. Another thing that sets this solution apart from Skomedal's work, is that is uses a decomposition approach instead of scheduling.

The article: *GPU-Accelerated Parallel FDTD on Distributed Heterogeneous Platform* [7], was published during the writing of this thesis. It happens to cover a similar area of interest, and also uses similar techniques. The code developed as part of this thesis distinguish itself by utilizing multiple GPUs within each node. It is also written in C++ as opposed to Fortran, and emphasizes a modular design that is easily extendable.

Outline

The thesis is structured in the following way:

- **Chapter 2** presents background material covering CUDA, MPI, the FDTD method, the Yee_bench code, and Skomedal's work.
- **Chapter 3** describes the overall design and flow of the implementation.
- **Chapter 4** includes performance results from executing the code on a variety of software and hardware setups, as well as different program configurations.
- **Chapter 5** discusses the various results presented in Chapter 4. It will also discuss some of the known limitations of the implementation.
- **Chapter 6** concludes the findings of the thesis, and presents a selection of recommendations for future work.
- **Appendix A** contains a detailed specification for each of the generations of CUDA capable GPUs.
- **Appendix B** contains a user guide on how to compile and execute the code.
- **Appendix C** contains a complete Doxygen generated documentation of the entire code.

Chapter 2

Background

This chapter will give a brief overview of the history of GPUs related to CPUs, and what distinguishes them. It will provide an introduction to NVIDIA's Compute Unified Device Architecture (CUDA), some of its most used terms, and how it has evolved related to the hardware architecture. The mentioned topics are both based on and excerpts from the author's specialization project: *The Evolution and Current State of CUDA GPGPU* [8]. This chapter will also present the Message Passing Interface, the FDTD method, and Andreas Berg Skomedal's master thesis: *Heterogeneous FDTD for Seismic Processing* [1], which has been used as a basis for the work in this thesis.

2.1 Parallel computing on GPUs and CPUs

GPUs are originally designed to accelerate the rendering of complex real-time graphics, which involves doing the same computations on a lot of independent data before outputting to a frame buffer. Because of the nature of this work and in order to meet performance demands for real-time rendering, modern GPUs contains many cores that can operate on data in parallel. CPUs on the other hand have mainly been focused towards running sequential programs, and performance has steadily improved with each new generation by increasing the operating frequency. Over the last decade this has not been a feasible strategy due to three primary factors: The memory wall, the Instruction Level Parallelism (ILP) wall, and the power wall. These walls are combined known as the brick wall [9], and has motivated CPU manufacturers to look for other ways to improve performance, outside of just increasing operating frequency.

The main contributor to performance increments today is parallelism. Since the early 2000s, manufacturers like Intel, AMD, and others, have developed multi-core processors that implements multiple CPU cores into the same physical package. These cores are able to run in parallel and execute multiple instructions on different data at the same time. Each core is also able to execute instructions independent of the other cores, effectively making a multi-core CPU able to execute multiple programs simultaneously, which has previously only been seemingly done through context switching. This ability is why modern multi-core CPUs are referred to as Multiple Instruction, Multiple Data (MIMD) systems. Today, a regular consumer grade CPU usually contains between two and eight cores, while high-end CPUs targeted for servers can contain up to sixteen cores.

In contrast to multi-core CPUs, GPUs are in general known as Single Instruction, Multiple Data (SIMD) systems. This is because the cores on a GPU can generally only execute the same instructions, but on different data. However, newer GPU architectures

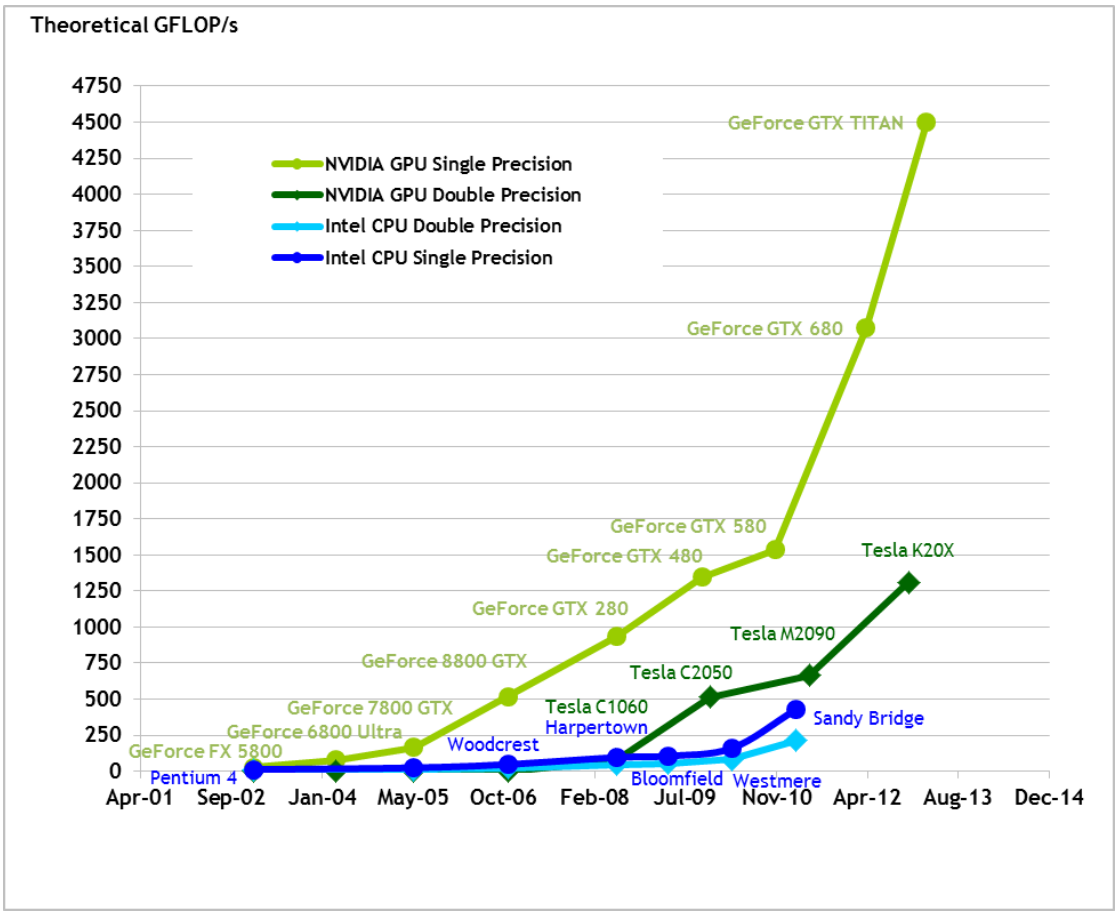

Figure 2.1: Floating-point operations per second development [10]

like NVIDIA's Kepler and Maxwell, does to some extent possess the ability to execute multiple different tasks simultaneously, which is further discussed in Section 2.2.3. Besides not being able to independently execute different instructions, GPU cores are also a lot simpler than CPU cores. CPU cores uses a lot of die space for optimizations that will benefit serial execution, like: Advanced instruction pipelining, multiple large cache hierarchies, and sophisticated branch predictions. Modern GPU architectures does incorporate some of these features, although in simpler forms.

The reduced complexity and size of a GPU core makes it possible to fit *a lot* more cores on GPUs compared to CPUs. The fastest Tesla GPU at time of writing: The NVIDIA Tesla K40, has as many as 2880 cores, with a combined peak of single-precision floating point performance of 4.29 Tera FLOPS. In order to keep the power and heat of a GPU within a reasonable limit, each GPU core operates at a significantly lower clock frequency compared to a CPU core. The Intel Core i7-4770K CPU operates at a clock between 3.5 and 3.9 GHz, depending on the load and temperature. In comparison, the cores on a NVIDIA Tesla K40 GPU, runs at a frequency between 745 and 875 MHz. Figure 2.1 shows how the theoretical performance of GPUs have developed compared to CPUs over the last decade.

Memory bandwidth is another important difference between CPUs and GPUs. Modern GPUs are equipped with Graphics Double Data-Rate (GDDR) 5 DRAM. GDDR DRAM is specially designed to be used as video memory, where high bandwidth is a requirement. It is very similar to the DDR2 and DDR3 DRAM used as system memory. The major difference is that GDDR DRAM runs at a higher frequency and thereby achieves higher

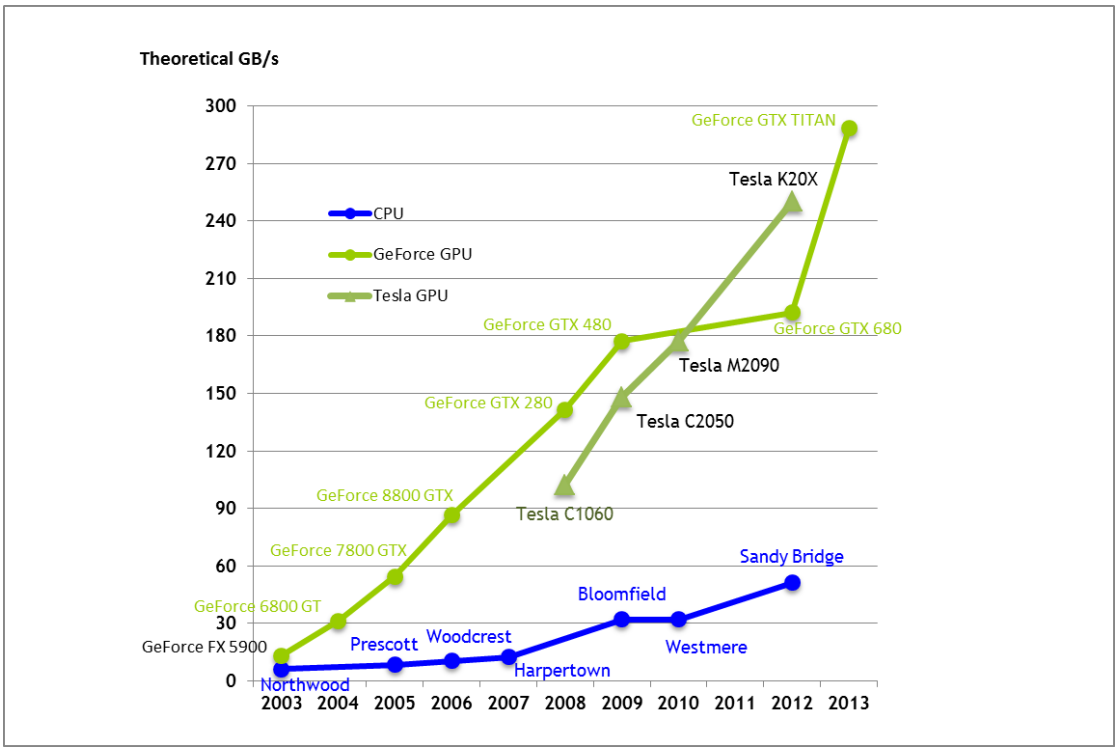

Figure 2.2: Memory bandwidth development [10]

bandwidth, it does however also have a slightly higher latency. Figure 2.2 shows how the theoretical bandwidth of GDDR DRAM has developed compared to regular DDR DRAM over the last decade.

2.2 Compute Unified Device Architecture (CUDA)

CUDA was released by NVIDIA in November 2006. It is a general purpose parallel computing platform and programming model, that enables developers to use C/C++ as high level programming languages. Doing general purpose computing on GPUs prior to CUDA was considered a much more tedious task, since one would have to rely on Application Programming Interfaces (APIs) primarily designed for rendering 2D and 3D graphics (like OpenGL), in order to interact with the GPU [11, p. 31]. The report: *Utilizing GPUs on Cluster Computers*, written by Leif Christian Larsen [12], gives a thorough evaluation on how GPUs can be used to offload computations in the pre-CUDA era.

CUDA is today the dominant proprietary framework for General-Purpose computing on Graphical Processing Units (GPGPU), and is only supported by NVIDIA GPUs of the G80 architecture or newer. The CUDA programming model consists of a host and one or more separate devices. The CPU is usually acting as a host by supplying the devices with work, and the devices are CUDA-capable GPUs.

Programs written in CUDA are compiled using the NVIDIA CUDA Compiler (NVCC) [13], into an Instruction Set Architecture (ISA) called Parallel Thread Execution (PTX) [14]. PTX is a machine-independent ISA that provides a common virtual machine model for all CUDA-capable GPUs. PTX instructions are further optimized and translated into native target-architecture instructions.

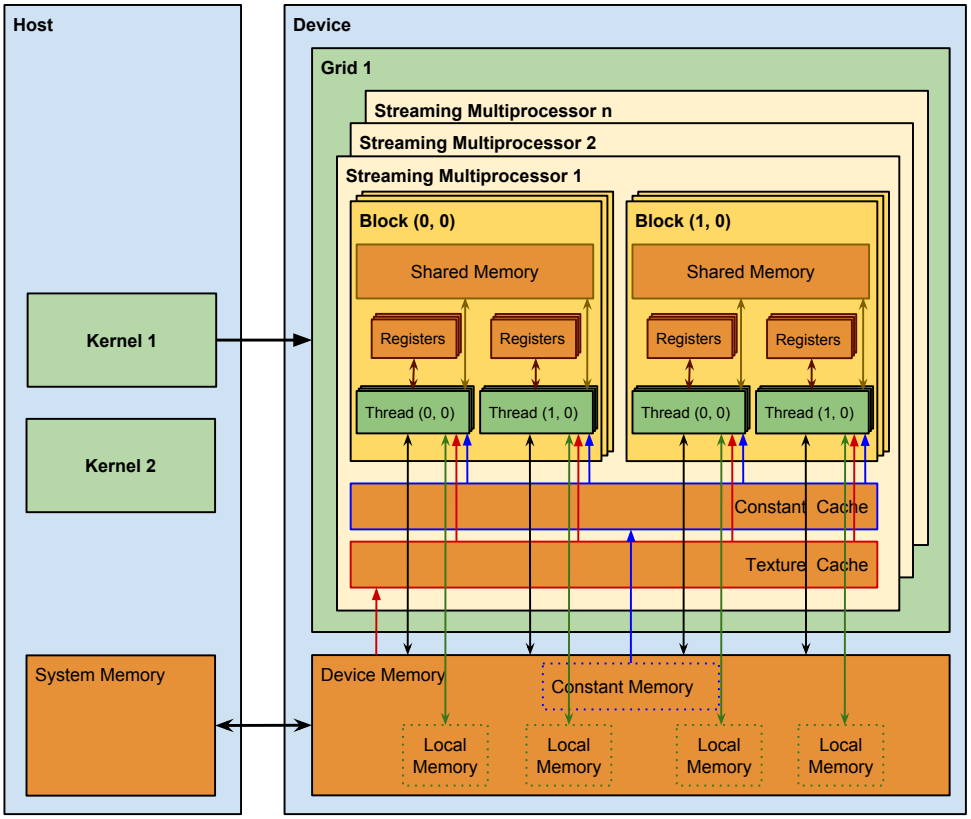

Figure 2.3: CUDA logical hierarchy [11, p. 47]

2.2.1 Technical terms

Following is a brief description of the terms used in CUDA programs and its relation to the hardware on CUDA supported GPUs. The specifications for the terms covered in this section are not constant across all the CUDA-capable GPUs. Newer GPUs might add new features or alter the specifications of old features. The *compute capability* version, is used to distinguish the architectures based on their features.

Grids and kernels

When data resides on the device memory, computations can be done by executing kernels on the device. Kernels are CUDA extensions that effectively are C functions that will execute on the device, instead of the host. A kernel function specifies the code to be executed by a number of threads. Threads in CUDA are organized into thread blocks and blocks are organized into grids. A kernel is executed as one grid of blocks of threads.

CUDA streams

A stream is a sequence of operations that is executed on a device in the order issued by the host. For example, the transfer of data from host to device, the launching of one or multiple kernels, and the transfer of data back to host. A default stream (Stream 0) is implicitly used in a CUDA program, and all data transfers and kernel executions will be done in serial. Using multiple streams allows for data transfers and kernel execution from different streams to be pipelined.

The amount of concurrency possibilities has changed with new architectures and compute capability versions. Compute capability 1.1 introduced concurrency by letting one transfer and one kernel execute concurrently. Compute capability 2.x expanded on this by letting one transfer per direction and multiple kernels execute concurrently [10, p. 32].

Blocks and streaming multiprocessors

CUDA-capable GPUs are organized into an array of Streaming Multiprocessors (SMs). Each SM executes one or multiple blocks of threads at a time. Threads inside the same block are able to cooperate by using the shared memory located as fast on-chip memory on the SM. Blocks are identified by their three-dimensional indexes within a grid of blocks.

Threads and CUDA cores

Each SM consists of many Streaming Processors (SPs), that executes in a Single Instruction, Multiple Thread (SIMT) fashion. SPs has since the Fermi architecture been named CUDA cores. CUDA cores executes threads, and threads are identified by their three-dimensional indexes within a block of threads. The thread and block indexes are used to differentiate the threads, and to calculate the addresses of the data each thread should do computations on.

Warps

In order for GPUs to achieve high latency tolerance, they use a concept called warps. Warps consists of 32-threads from the same block. Long-latency operations such as global memory accesses are hidden by scheduling the warps. While one warp is waiting for data from global memory, another warp can instead be selected for execution, and so on. Latency can therefore effectively be hidden by having enough warps to switch between. Because of warp scheduling, GPUs can get away with dedicating less chip area for cache and more for floating-point execution resources.

Occupancy and constraints

An important part of optimizing code for GPUs is to adjust the block size (number of threads per block) in order to achieve the best possible occupancy. As seen in Equation 2.1 [15, p. 44], occupancy is dependent on the number of active warps and the maximum number of active warps.

$$\text{Occupancy} = \frac{\text{Active Warps}}{\text{Maximum Active Warps}} \tag{2.1}$$

Occupancy is all about giving each SM enough warps to switch between in order to hide latency. For best possible occupancy, each SM should be running a number of threads equal to the warp size multiplied by the maximum number of active warps per SM [15, p. 44].

$$\text{Threads} = \text{Warp Size} \times \text{Maximum Active Warps} \tag{2.2}$$

Ideally, the block size should be of such a size so that when a SM starts executing a set of blocks, the ideal number of threads is running. This should be possible without surpassing the limit on blocks per SM and threads per block, or the limit on register and shared memory usage. If any of these limitations are surpassed, the CUDA runtime will have to limit the number of active blocks, thus reducing the occupancy.

Maximum Active Warps, number of blocks per SM, threads per blocks, registers per SM, and shared memory, are all limitations that differs between architectures. These specifications can be examined in the table included in Appendix A. Because the specifications and limitations can vary between systems, the block size should be set dynamically during execution for best occupancy and scalability.

2.2.2 Memory types

Even though GPUs have a potential for very high calculation throughput, it is limited by how fast data can be loaded from memory. This is why CUDA-capable GPUs implements different memory types, that differs in speed, size and scope. The memory types are the following: Global memory, Constant memory, Texture memory, Shared memory, Local memory and Registers. A complete comparison can be viewed in Table 2.1.

Table 2.1: CUDA memory types

	Location	**Cached**	**Access**	**Speed**	**Scope**
Global memory	Off-chip	Yes (Fermi)	Read/Write	Slow	All threads + host
Constant memory	Off-chip	Yes	Read	Fast	All threads + host
Texture memory	Off-chip	Yes	Read (Write on Fermi)	Slow	All threads + host
Shared memory	On-chip	No	Read/Write	Very fast	All threads in block
Local memory	Off-chip	Yes (Fermi)	Read/Write	Slow	One thread
Registers	On-chip	No	Read/Write	Very fast	One thread

Out of all of these memories, only registers and shared memory is actual physical memory located on each of the SMs. Global, constant, texture and local memory all resides on the device memory, but with different caches located on the SMs. The memory types and cache is illustrated in Figure 2.3.

Global memory

Global memory is the largest memory, accessible by all threads (hence the name global). It is also the slowest, with a latency of 400-800 cycles. Global memory originally did not have any caching, but this have been implemented in Fermi and newer architectures [16].

Constant memory

Constant memory is a small cached part of the device memory, limited to 64 KB for all currently available CUDA-capable GPUs. Access to constant memory is very fast on cache hits, but the real advantage is the potential reduction in throughput: When multiple threads in the same half-warp access the same address in constant memory, it only counts as one read instead of one read per thread as it otherwise would have been. Constant memory can however be slower than global memory when all the threads in a half-warp need data from different addresses, the various reads will then be serialized. The constant memory is read only (hence the name constant), and should only contain data that do not change during the execution.

Texture memory

Texture memory is another way of accessing data in device memory. It uses a texture cache that is optimized for either 1D, 2D or 3D spatial locality. Texture memory is designed for streaming fetches with a constant latency, and will therefore not give a lower latency on cache hits. Explicitly using texture memory on systems with L1 cache can therefore result in worse performance. Throughput demand will however be reduced on cache hits, and mapping data as textures can therefore be done to increase performance on kernels that are memory bandwidth bound. Texture memory is read-only, but GPUs with compute capability ¿ 2.0 allows for the use of *Surface memory*, which can also be written to.

The Kepler GK110 architecture with compute capability 3.5 lets the texture cache be accessed as a read-only data cache. This circumvents the somewhat unconventional way of utilizing the texture cache by mapping the data as textures.

Shared memory

Shared memory is a very fast on-chip memory located on each of the SMs. The shared memory of a SM is only accessible by the threads running on that SM. This is one of the reasons to why CUDA organizes threads in blocks. All threads running within a block is guaranteed to be running on the same SM, and therefore have access to that SM's shared memory. Shared memory is not automatically used, and data has to be explicitly moved from device memory onto shared memory. Tiling is an optimization technique often used, it simply works by dividing the data into smaller tiles that will fit in shared memory. The technique is useful when the same data is accessed multiple times by different threads. Moving data onto the shared memory requires one read, and consecutive reads will be much faster since the data then resides on the on-chip memory, this also reduces DRAM bandwidth usage.

Shared memory is physically divided among equally sized memory modules (banks) on the chip. Each memory bank can only service one memory access at a time. If multiple threads tries to access different memory locations in the same memory bank at the same time, a bank conflict will occur. Since a memory bank is limited to only serving one access at a time, the access will have to be serialized, which decreases performance. An exception is when multiple threads in a warp access the same memory location at the same time, since shared memory will broadcast the data, similar to constant memory.

Local memory and registers

Both local memory and registers has a scope that is local to the thread. Registers are extremely fast on-chip memory located on each SM, and generally have an access time of zero clock cycles per instruction. The latency can however differ on Read-After-Write (RAW) dependencies or on register memory bank conflicts. Latency due to RAW dependencies can be hidden by having enough warps for the SM to switch between, while bank conflicts are usually avoided by optimally schedule instructions. This is done by the compiler and hardware scheduler, and works best when the number of threads per block is a multiple of 64 [15, p. 41].

SMs do have a limit on the number of available registers per thread, and if this limit is exceeded the data will have to be put in local memory. This occurrence is called *register spilling* and is done automatically by the runtime. The local memory is located

on device memory like the global, texture and constant memory [15, p. 40]. This means that local memory is very slow and should generally be avoided by limiting the number of used registers to prevent register spilling. Local memory has however been cached on the Fermi architecture and successors [16].

There is also a limit on the total number of registers per SM, and if this limit is exceeded the runtime will reduce the number of active blocks. Both the maximum number of registers per thread and the maximum number of registers per SM can vary between compute capability versions. These values can be found in the table included in Appendix A.

2.2.3 Architectures

Prior to CUDA, NVIDIA GPUs utilized a graphical pipeline with different shader processors specialized for each stage. Figure 2.4 is an example of a simplified graphical pipeline with specialized vertex, geometry and pixel shaders. As graphics became more demanding more stages needed to be added and complex optimizations had to be done in every stage. However, the disadvantage of such a pipeline is that it becomes bottlenecked by its slowest stage. Frames in a video game might for example be very vertex shader-intensive, but less pixel shader-intensive. This will result in full utilization of the vertex shaders, but sub-par utilization of the pixel shaders due to them having to wait for work from the vertex shader stage. Switching to a unified shading architecture has eliminated this shortcoming.

The unified shading architecture replaces the specialized shaders with unified shaders. An unified shader core is a fully generalized scalar processor, capable of doing any of the shading operations in the graphical pipeline. The term unified shader core is used interchangeably with streaming processor and CUDA core. As demonstrated in Figure 2.5, the new architecture reduces the complexity of many physical stages by no longer executing the pipeline in a sequential flow. The pipeline is instead physically executed in a recirculating path, that revisits the unified shader cores multiple times, depending on the shading complexity. Unified shader cores coupled with a dynamic scheduler removes the previous problems with load balancing between specialized shaders. They are also by no means limited to graphical processing, making them very suitable for GPGPU.

Even though every NVIDIA GPU with unified shader cores supports CUDA, there has been some noteworthy changes to the hardware architecture since the introduction of CUDA. The significance of these changes are reflected in the compute capability version of the various GPUs. Some of these differences can have major impact on the performance and behavior of CUDA applications, and some features are only available on newer GPUs with higher compute capabilities. A good understanding of the architectures and their differences is therefore advantageous in order to write efficient CUDA programs. Following is a brief description of the major architectures. A summary of the specifications can also be viewed in the table included in Appendix A.

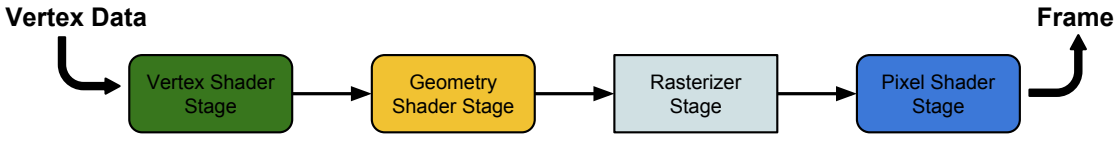

Figure 2.4: Graphical pipeline

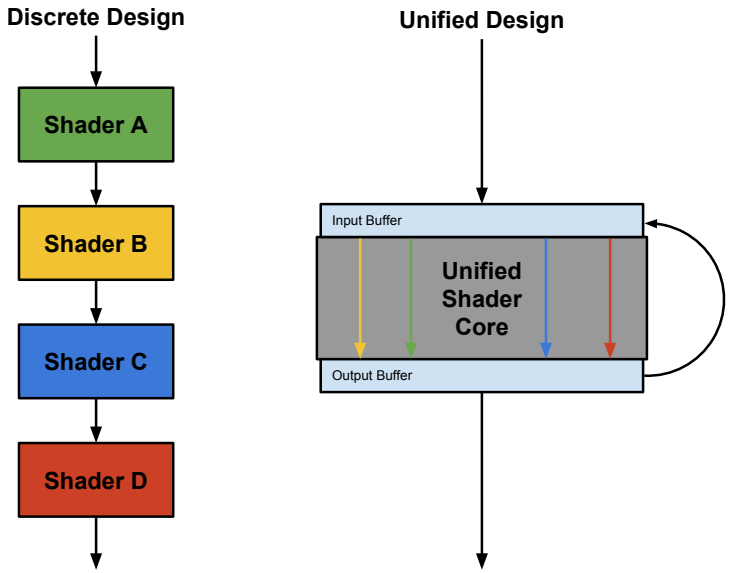

Figure 2.5: Unified design [17]

G80/GT200

The G80 and GT200 architectures uses a concept called Texture/Processor Cluster (TPC) that couples Texture Mapping Units (TMU) and Streaming Multiprocessors (SMs), as shown in Figure 2.6. The TMUs consists of texture address units and texture filtering units. Textures are cached in a fast 12 KB texture L1 cache in each TPC, and there is also a slower 256 KB L2 cache placed at the memory controller of each DRAM, both of these caches are read-only and only utilized for texture fetching [16, p. 15]. Every TPC also has one load/store unit that can fetch data while Streaming Processors (SP) do computations on already received data, this is what enables the GPU to have a high latency tolerance.

SMs in the G80 and GT200 architectures consists of eight SPs and two Special Function Units (SPU). The Special Function Units are processors made for transcendental operations, such as sines and cosines. Each SM also contains a small instruction cache, 8 KB constant memory cache, 16 KB of shared memory, and a MT issue unit that dispatches instructions to all the SPs and SFUs in the SM.

CUDA was developed to simplify the utilization of the computing power in GPUs designed with the new unified shading architecture, and the G80 architecture is hence the first GPU generation to support CUDA [17]. The GT200 architecture was released in June 2008 and is NVIDIA's second-generation unified shading and compute architecture. It is mainly an improvement on the G80, with no major architectural changes. GT200 has improved the performance over the G80 by increased the total number of SPs by almost the double. This is done by increasing the number of SMs per TPC from two to three, and the number of TPCs from eight to ten. The GT200 has also doubled the register memory size and improved memory access, instruction scheduling and clock rate compared to the G80 architecture. Arguably the most exciting new feature of GT200 regarding GPGPU is the addition of double-precision floating point support, which is a necessity for computations that requires high precision. [18]

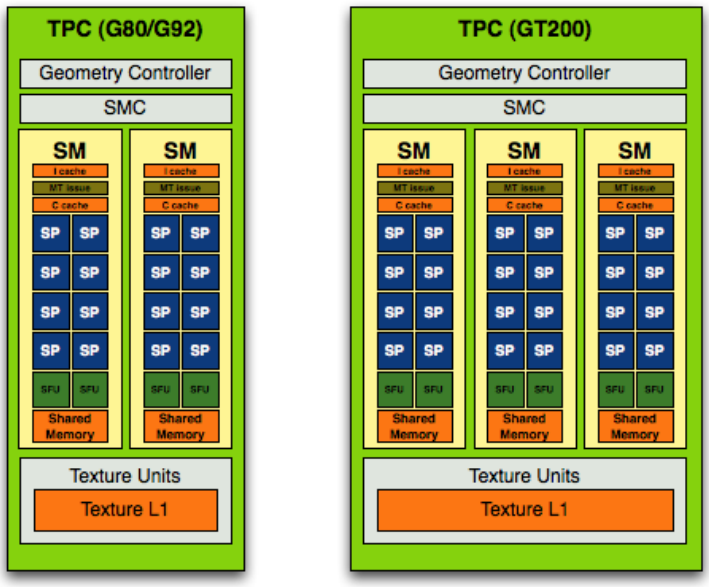

Figure 2.6: NVIDIA TPC
Source: www.anandtech.com/show/2549/2

Fermi

The Fermi architecture was launched with the GeForce 400 series in March 2010. It removed the concept of TPCs and instead introduced the 3rd generation of SMs, where each SM contains all the features of a TPC, as shown in Figure 2.7. Alongside the introduction of the new Fermi architecture some of the terms changed name: SPs are now known as CUDA cores and MT issue units are called Dispatch Units. The TMUs are now integrated in each SM and are called texture units or Tex, as seen in Figure 2.7. The new SMs have also been upgraded to implemented 16 load/store units instead of having one big per TPC, 32 CUDA cores compared to eight, and four SFUs compared to two.

While the G80 and GT200 SMs had 16 KB of shared memory, the new SMs have one pool of 64 KB configurable on-chip memory. This memory can be configured as either 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache. Every global memory access is cached in the new L1 cache, and it should therefore not be confused with the texture only L1 cache in the G80 and GT200 architectures. Applications that do not take advantage of shared memory can instead automatically benefit from having L1 cache. The texture memory still have a dedicated L1 cache of 12 KB, but this is now instead called texture cache. The L2 cache has been increased from 256 KB to 768 KB and now affects all accesses to the DRAM, which will benefit global and local memory accesses instead of being reserved for texture memory. The constant cache still remains unchanged at 8 KB.

As seen in Figure 2.7, the Fermi architecture has two sets of warp schedulers and dispatch units. This enables the SM to concurrently dispatch instructions from two warps into any two of the columns shown as green in Figure 2.7. One SM can for example execute up to 32 floating-point operations per clock by dispatching 16 instructions into each core-column.

CUDA cores consists of a fully pipelined integer Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU). In the Fermi architecture the FPU has been upgraded to

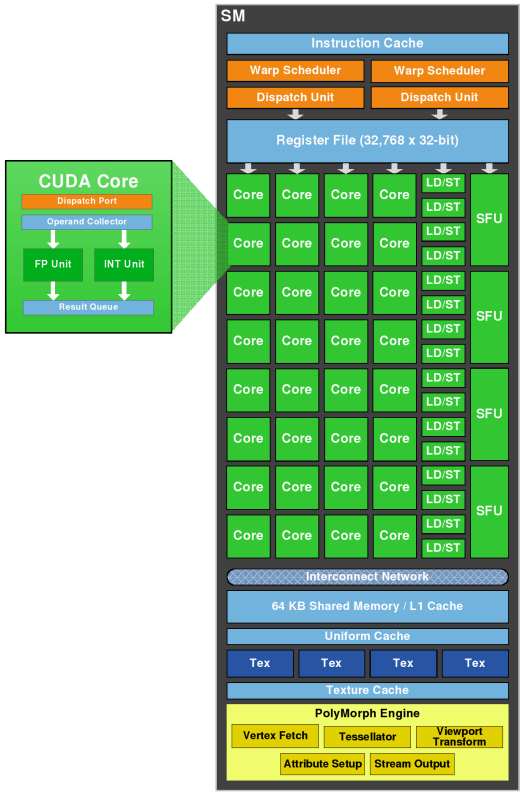

Figure 2.7: NVIDIA Fermi SM [16]

support the IEEE 754-2008 floating-point standard, from the 1998 standard used in G80 and GT200. The ALU has also been upgraded to fully support 32-bit precision for all instructions, an upgrade from the previous architectures that only supported 24-bit and had to simulate 32-bit precision using multiple instructions. 16 CUDA cores per SM is also optimized for 64-bit and extended precision operations. By being able to perform up to 16 double-precision operations per SM, the Fermi architecture has shown an improvement of up to 4.2x compared to the GT200 architecture in double-precision applications.

The Fermi architecture supports the PTX 2.0 ISA, which implements a unified address space that unifies the three address spaces of local, shared and global memory. This enables true C++ support, since pointer's target address can be found at compile time. Another new feature introduced with Fermi and compute capability 2.0 is *concurrent kernel execution*: Previous architectures only allowed for a single transfer and kernel execution from separate streams to be carried out simultaneously. Concurrent kernel execution enables up to 16 kernels from different streams to be executed in parallel. The Fermi architecture also adds a second copy engine, which makes it possible to transfer data between host and device in both directions concurrently.

Fermi GPUs also provides Error Correcting Code (ECC) support for memory on Tesla models, and an updated GigaThread engine. GigaThread is the name of the the global thread scheduler that distributes thread blocks among the SMs. The new GigaThread engine has faster context switching, block scheduling and the ability execute kernels concurrently. Kernels in previous architectures could only be executed sequentially, thereby limiting GPU utilization when executing multiple small kernels. [16]

Figure 2.8: NVIDIA Kepler GK110 SMX [19]

Kepler (GK104/GK110)

NVIDIA GTX 680 was released in March 2012 and was the first GPU based on the new Kepler architecture, codenamed GK104. While previous architectures mainly focused on increasing pure performance, the Kepler architecture also focuses excessively on energy efficiency.

The major change in the Kepler architecture is the Next Generation SM (SMX). The new SMX contains a lot more processing units than the previous Fermi SMs: Six times as many CUDA cores, and eight times as many SFUs and Load/Store Units. In order to supply the large number of processing units the number of warp schedulers and the register file size has been doubled compared to the Fermi SMs. The reason for the large increase in processing units is the removal of the shader clock. In older architectures the processing units operated at a shader clock at double the frequency of the core clock, an optimization that allowed the processing units to do two operations per core clock cycle. In the Kepler architecture everything runs at the same core clock frequency. Running the processing units at the lower clock rate combined with a simpler pipeline reduces power consumption drastically. The performance of the new Kepler architecture relays on having more processing units instead of running at a high clock rate, and consequently achieves better power efficiency than the Fermi architecture.

16 of the CUDA cores in the Fermi SMs were capable of doing Double-Precision (DP) operations. The new SMXs has instead included up to 64 special processing units for DP operations per SMX, in addition to the regular CUDA cores. These new DP units are only used for DP operations.

The configurable shared memory/L1 cache still remains at 64 KB, but can now also be

configured evenly as 32 KB for each. The L1 cache is now reserved only for local memory accesses, and global loads are only cached in the L2 cache, which has increased to 1536 KB, from 768 KB on the Fermi architecture.

The GK110 is the second generation of the Kepler architecture and introduces some new features. One new feature is called Dynamic Parallelism, which is intended to make the device more independent of the host. This is done by allowing kernels to launch new kernels and allocate the necessary resources, without interacting with the host. Another new feature introduced in GK110 is Hyper-Q, which increases the total number of connections (work queues) between host and device from one to 32. Every CUDA stream will now be managed within its own hardware work queue, thereby enabling streams to execute concurrently and multiple CPU-cores to launch work on the GPU simultaneously. The texture cache size of GK110 has also been increased from 12 KB to 48 KB, and made possible to be used as a read-only data cache. Previously, the texture cache could only be utilized by mapping the data as textures. [19, 20]

2.3 The Message Passing Interface (MPI)

MPI is a standard for message-passing in distributed-memory applications. It is not a library or an implementation, but rather a specification that defines the syntax and semantics needed for creating a message-passing implementation. There exists multiple implementations based on the MPI standard, the most known being MPICH and Open MPI. Even though the code developed in this thesis should work with both implementations, Open MPI will be used.

A detailed description on the use of MPI will not be provided here, the official Open MPI documentation at `www.open-mpi.org/doc`, can instead be sought out for more information. However, following is a short description of one of the MPI mechanisms relevant to the implementation: A *derived data type* is a mechanism provided by MPI that lets users define new data types. These new data types have the capability to explicitly pack both contiguous and noncontiguous data into a contiguous buffer during a send operation. Correspondingly, a contiguous buffer can be explicitly unpacked into noncontiguous locations during a receive operation. This makes it possible to combine multiple send or receive operations into one, thereby reducing the fixed overhead of sending and receiving a message. There are multiple ways to create a derived data type, but creating a *struct* is the most flexible way, as it can be created from a general set of data types, displacements and block sizes.

2.4 Finite-Difference Time-Domain (FDTD)

The FDTD method is a discretization of the time dependent Maxwell's equations. It was originally introduced by Kane Yee in 1966, and is today widely used for modeling computational electrodynamics, with use cases ranging from biomedical imaging/treatment to seismic wave propagation and earthquake motion [21].

In the FDTD method, both space and time are divided into individual segments. Space is segmented into box-shaped cells called Yee cells. Each Yee cell contains three (x, y, z) electric and three (x, y, z) magnetic field components: The electric fields are located on the edges and the magnetic fields are located on the faces, as demonstrated in Figure 2.9. A FDTD grid is a three-dimensional volume consisting of many cells. Time

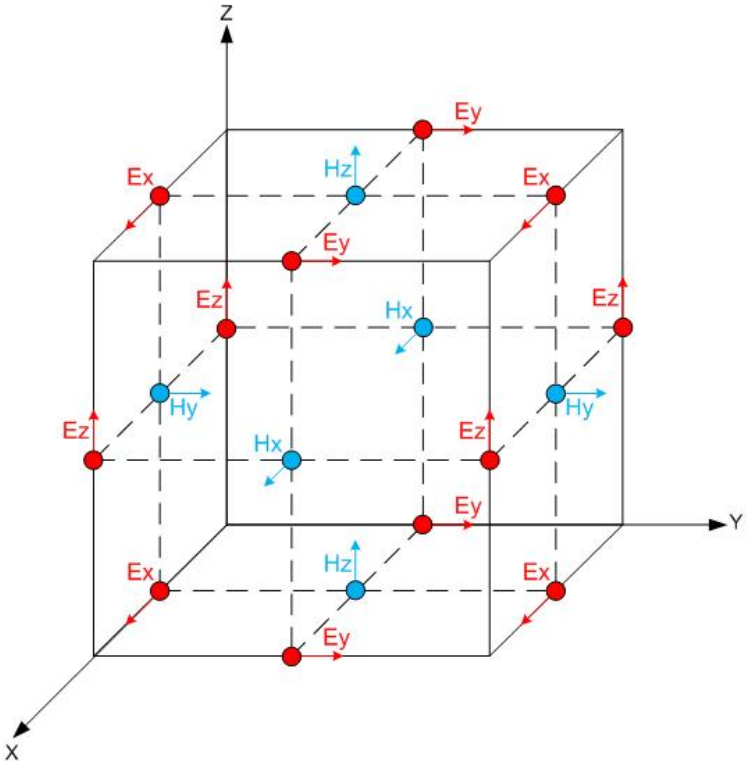

Figure 2.9: Yee Cell with electric and magnetic field components
Source: www.fdtd.wikispaces.com/The+Yee+Cell

is divided into small steps, where each time-step represents the time required for a field to travel from one cell to a neighboring cell. The fields are updated in a leapfrog manner, where the electrical field components of a space domain is updated at an instant in time, followed by an update of the magnetic fields in the next instant in time. Fields in the next time-step is only depended on the fields in the last time step. This process will repeat for the desired number of time steps, which ideally should be enough for the electromagnetic fields to reach a steady-state behavior.

Maxwell's equations in a source free region of space are the following:

$$\begin{aligned} \text{Gauss' law:} \quad & \nabla \cdot \vec{D} = 0 \\ \text{Gauss' magnetism law:} \quad & \nabla \cdot \vec{B} = 0 \\ \text{Faraday's law:} \quad & \nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \\ \text{Ampere's law:} \quad & \nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \vec{J}_c \end{aligned}$$

Where D is the Electric Flux Density, B is the Magnetic Flux Density, E is the Electric Field, t is the Time, H is the Magnetic Field, and J is the Electric Current Density

In materials with electric and magnetic properties that are field, direction and frequency independent. The electric and magnetic flux density, and the electric current density can be written as products of the electric and magnetic fields, and the constitu-

tive parameters (the material's physical constants):

$$\vec{D} = \varepsilon\vec{E}, \qquad \vec{B} = \mu\vec{H}, \qquad \vec{J_c} = \sigma\vec{E}$$

Where ε is the Permittivity, μ is the Permeability and σ is the Conductivity of a material

The Maxwell's equations for the Electric and Magnetic fields can thus be written:

$$\frac{\partial\vec{E}}{\partial t} = \frac{1}{\varepsilon}\nabla\times\vec{H} - \frac{\sigma}{\varepsilon}\vec{E}, \qquad \frac{\partial\vec{H}}{\partial t} = -\frac{1}{\mu}\nabla\times\vec{E}$$

Translated into Cartesian coordinates this results in six coupled partial differential equations in the six field components:

$$\frac{\partial\vec{E_x}}{\partial t} = \frac{1}{\varepsilon}(\frac{\partial\vec{H_z}}{\partial y} - \frac{\partial\vec{H_y}}{\partial z} - \sigma\vec{E_x}) \tag{2.4a}$$

$$\frac{\partial\vec{E_y}}{\partial t} = \frac{1}{\varepsilon}(\frac{\partial\vec{H_x}}{\partial z} - \frac{\partial\vec{H_z}}{\partial x} - \sigma\vec{E_y}) \tag{2.4b}$$

$$\frac{\partial\vec{E_z}}{\partial t} = \frac{1}{\varepsilon}(\frac{\partial\vec{H_y}}{\partial x} - \frac{\partial\vec{H_x}}{\partial y} - \sigma\vec{E_z}) \tag{2.4c}$$

$$\frac{\partial\vec{H_x}}{\partial t} = \frac{1}{\mu}(\frac{\partial\vec{E_y}}{\partial z} - \frac{\partial\vec{E_z}}{\partial y}) \tag{2.4d}$$

$$\frac{\partial\vec{H_y}}{\partial t} = \frac{1}{\mu}(\frac{\partial\vec{E_z}}{\partial x} - \frac{\partial\vec{E_x}}{\partial z}) \tag{2.4e}$$

$$\frac{\partial\vec{H_z}}{\partial t} = \frac{1}{\mu}(\frac{\partial\vec{E_x}}{\partial y} - \frac{\partial\vec{E_y}}{\partial x}) \tag{2.4f}$$

2.4.1 The Yee algorithm

A FDTD grid is a three-dimensional volume (a box) of size l_x, l_y, l_z. This grid is divided into N_x, N_y, N_z equal sized cells. Each cell is of size $\Delta x, \Delta y, \Delta z = l_x/N_x, l_y/N_y, l_z/N_z$. For convenience, let $u(x, y, z, t)$ denote any of the six field components. A point in the FDTD grid is represented by $(i\Delta x, j\Delta y, k\Delta z) = (i, j, k)$, and a time step is represented by $t = n\Delta t$. Then:

$$u(i\Delta x, j\Delta y, k\Delta x, n\Delta t) = u^n(i, j, k) \tag{2.5}$$

In the Yee algorithm, the derivations are approximated using central difference expressions. For example, the left hand side of Equation 2.4d can be written:

$$\frac{\partial\vec{H_x}}{\partial t} = \frac{H_x^{n+\frac{1}{2}}(i, j, k) - H^{n-\frac{1}{2}}(i, j, k)}{\Delta t}$$

Thus, using Notation 2.5 and central difference, Equation 2.4d can be written:
Where Equation 2.6b defines $H_x(i, j, k)$ at time step $n+\frac{1}{2}$, in terms of previously calculated values of H_x (at time step $n - \frac{1}{2}$), E_y and E_z (at time step n). Likewise, H_y and H_z at time step $n + \frac{1}{2}$, can be expressed similarly.

$$\frac{H_x^{n+\frac{1}{2}}(i,j,k) - H_x^{n-\frac{1}{2}}(i,j,k)}{\Delta t} = \frac{1}{\mu(i,j,k)} \left[\frac{E_y^n(i,j,k+\frac{1}{2}) - E_y^n(i,j,k-\frac{1}{2})}{\Delta z} - \frac{E_z^n(i,j+\frac{1}{2},k) - E_z^n(i,j-\frac{1}{2},k)}{\Delta y} \right] \tag{2.6a}$$

$$H_x^{n+\frac{1}{2}}(i,j,k) = H_x^{n-\frac{1}{2}}(i,j,k) + \frac{\Delta t}{\mu(i,j,k)} \left[\frac{E_y^n(i,j,k+\frac{1}{2}) - E_y^n(i,j,k-\frac{1}{2})}{\Delta z} - \frac{E_z^n(i,j+\frac{1}{2},k) - E_z^n(i,j-\frac{1}{2},k)}{\Delta y} \right] \tag{2.6b}$$

Where $\mu(i,j,k)$ denotes the Permeability at a grid point

The electrical field functions can also be expressed using Notation 2.5 and central difference. For instance, Equation 2.4a can be written:

$$\frac{E_x^{n+1}(i,j,k) - E_x^n(i,j,k)}{\Delta t} = \frac{1}{\varepsilon(i,j,k)} \frac{H_z^{n+\frac{1}{2}}(i,j+\frac{1}{2},k) - H_z^{n+\frac{1}{2}}(i,j-\frac{1}{2},k)}{\Delta y} - \frac{1}{\varepsilon(i,j,k)} \frac{H_y^{n+\frac{1}{2}}(i,j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i,j,k-\frac{1}{2})}{\Delta z} - \frac{\varepsilon(i,j,k)}{\sigma(i,j,k)} E_x^{n+\frac{1}{2}}(i,j,k) \tag{2.7}$$

However, Equation 2.7 requires electric field values at both full and half time steps. This can be avoided by approximating the half time step:

$$E_x^{n+\frac{1}{2}} \simeq \frac{E_x^{n+1} + E_x^n}{2}$$

For convenience, the following notations are also given:

$$C_{i,j,k} = \frac{1 - \frac{\sigma(i,j,k)\Delta t}{2\epsilon(i,j,k)}}{1 + \frac{\Delta t \sigma(i,j,k)}{2\epsilon(i,j,k)}}, \qquad D_{i,j,k} = \frac{\frac{\Delta t}{\epsilon(i,j,k)}}{1 + \frac{\sigma(i,j,k)\Delta t}{2\epsilon(i,j,k)}}$$

Where $\sigma(i,j,k)$ denotes the Conductivity and $\epsilon(i,j,k)$ denotes the Permittivity at a grid point

Then, expressing $E_x^{n+1}(i,j,k)$ in terms of previously calculated E field values at full time steps is possible by the equation:

$$E_x^{n+1}(i,j,k) = C_{i,j,k} E_x^n(i,j,k) - D_{i,j,k} \left[\frac{H_z^{n+\frac{1}{2}}(i,j+\frac{1}{2},k) - H_z^{n+\frac{1}{2}}(i,j-\frac{1}{2},k)}{\Delta y} + \frac{H_y^{n+\frac{1}{2}}(i,j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i,j,k-\frac{1}{2})}{\Delta z} \right] \tag{2.8}$$

As with the magnetic field expressions, E_y and E_z can also be expressed in a similar way to E_x.

The Yee algorithm computes the field components in a leap-frog arrangement, meaning that H and E field components are computed at alternating time steps. As seen from Equation 2.6b and 2.8, the magnetic field components are calculated at half time steps, while the electrical are calculated at full time steps. They are also displaced by half steps in space. There is no need to calculate every H and E field component in every Yee cell.

Given that the coordinates of the center of a cell in a FDTD grid is (i, j, k), the complete list of equations for the Yee algorithm is shown:

$$H_x^{n+\frac{1}{2}}(i, j+\tfrac{1}{2}, k+\tfrac{1}{2}) = H_x^{n-\frac{1}{2}}(i, j+\tfrac{1}{2}, k+\tfrac{1}{2}) + \frac{\Delta t}{\mu(i,j,k)}\left[\frac{E_y^n(i, j+\frac{1}{2}, k+1) - E_y^n(i, j+\frac{1}{2}, k)}{\Delta z} - \frac{E_z^n(i, j+1, k+\frac{1}{2}) - E_z^n(i, j, k+\frac{1}{2})}{\Delta y}\right] \tag{2.9a}$$

$$H_y^{n+\frac{1}{2}}(i+\tfrac{1}{2}, j, k+\tfrac{1}{2}) = H_y^{n-\frac{1}{2}}(i+\tfrac{1}{2}, j, k+\tfrac{1}{2}) + \frac{\Delta t}{\mu(i,j,k)}\left[\frac{E_z^n(i+1, j, k+\frac{1}{2}) - E_z^n(i+\frac{1}{2}, j, k+1)}{\Delta x} - \frac{E_x^n(i+\frac{1}{2}, j, k+1) - E_x^n(i+\frac{1}{2}, j, k)}{\Delta z}\right] \tag{2.9b}$$

$$H_z^{n+\frac{1}{2}}(i+\tfrac{1}{2}, j+\tfrac{1}{2}, k) = H_z^{n-\frac{1}{2}}(i+\tfrac{1}{2}, j+\tfrac{1}{2}, k) + \frac{\Delta t}{\mu(i,j,k)}\left[\frac{E_x^n(i+\frac{1}{2}, j+1, k) - E_x^n(i+\frac{1}{2}, j, k)}{\Delta y} - \frac{E_y^n(i+1, j+\frac{1}{2}, k) - E_y^n(i, j+\frac{1}{2}, k)}{\Delta x}\right] \tag{2.9c}$$

$$E_x^{n+1}(i+\tfrac{1}{2}, j, k) = C_{i,j,k}E_x^n(i+\tfrac{1}{2}, j, k) - D_{i,j,k}\left[\frac{H_y^{n+\frac{1}{2}}(i+\frac{1}{2}, j, k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i+\frac{1}{2}, j, k-\frac{1}{2})}{\Delta z} + \frac{H_z^{n+\frac{1}{2}}(i+\frac{1}{2}, j+\frac{1}{2}, k) - H_z^{n+\frac{1}{2}}(i+\frac{1}{2}, j-\frac{1}{2}, k)}{\Delta y}\right] \tag{2.9d}$$

$$E_y^{n+1}(i, j+\tfrac{1}{2}, k) = C_{i,j,k}E_y^n(i, j+\tfrac{1}{2}, k) - D_{i,j,k}\left[\frac{H_z^{n+\frac{1}{2}}(i+\frac{1}{2}, j+\frac{1}{2}, k) - H_z^{n+\frac{1}{2}}(i-\frac{1}{2}, j+\frac{1}{2}, k)}{\Delta x} + \frac{H_x^{n+\frac{1}{2}}(i, j+\frac{1}{2}, k+\frac{1}{2}) - H_x^{n+\frac{1}{2}}(i, j+\frac{1}{2}, k+\frac{1}{2})}{\Delta z}\right] \tag{2.9e}$$

$$E_z^{n+1}(i, j, k+\tfrac{1}{2}) = C_{i,j,k}E_z^n(i, j, k+\tfrac{1}{2}) - D_{i,j,k}\left[\frac{H_x^{n+\frac{1}{2}}(i, j+\frac{1}{2}, k+\frac{1}{2}) - H_x^{n+\frac{1}{2}}(i, j-\frac{1}{2}, k+\frac{1}{2})}{\Delta y} + \frac{H_y^{n+\frac{1}{2}}(i+\frac{1}{2}, j, k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i-\frac{1}{2}, j, k+\frac{1}{2})}{\Delta x}\right] \tag{2.9f}$$

2.5 Yee_bench - A PDC benchmark code

Yee_bench is a serial benchmarking code for the FDTD method, developed at the Center for Parallel Computers (PDC), and the Parallel and Scientific Computing Institute at the Royal Institute of Technology in Sweden. It is presented in the paper: *Yee_bench - A PDC benchmark code*, by Ulf Andersson, published in November 2002 [2].

Being a benchmarking code, the code is only meant to simulate the work needed for solving FDTD problems, and is therefore a simplification of the real world. Related to Equations 2.9a - 2.9f, the FDTD grid in the Yee_bench code is assumed to consist of a homogeneous media. This means that the ϵ (Permittivity) and the μ (Permeability) are constant, and that the σ (Conductivity) is zero across the entire grid. All the initial electromagnetic field values are for convenience set to zero. The electromagnetic field is excited with a point source (a dipole) in the center of the FDTD grid.

With a grid consisting of N_x, N_y, N_z cells, 64-bit precision, and no memory padding, the paper uses the equation: $24N^3 + 24(N+1)^3$, to calculate the number of bytes required to store the electromagnetic values of a task. This is important in order to know if a task will fit in memory. The paper presents Table 2.2, to show how the memory requirements grows with the problem size.

Table 2.2: Maximum problem size vs available memory for Yee_bench

128 Kbytes	2 Mbytes	4 Mbytes	1 Gbytes	2 Gbytes	4 Gbytes
N = 13	N = 34	N = 43	N = 281	N = 354	N = 446

The paper also presents a validation case in order to verify that the code produces the correct result: Given $N_x = N_y = N_z = 100$, $\Delta_x = \Delta_y = \Delta_z = 1$, and the number of time

steps $N_t = 200$, the sum of the E_z components should equal -0.0692134. This test is sufficient to verify the correctness and accuracy of the code. Because the E_z component values at the last time step is depended on all the electromagnetic field values calculated at the previous time steps.

In order to measure performance, the number of loads, stores, and arithmetic operations per cell, per time step is counted. It is presented in Table 2.3, from the Yee_bench paper.

Table 2.3: Floating-point and memory operations per cell and time step for Yee_bench

adds/subs	mults	stores	loads
24	12	6	20

The paper points out that because of the uneven ratio of multiplication to addition/-subtraction, the performance cannot be expected to reach more than 75% of peak, even with unlimited memory bandwidth. However, performance is both expected and proven to be bound by memory bandwidth.

2.6 Heterogeneous FTDT for seismic processing

Heterogeneous FTDT for Seismic Processing [1], is the name of the master thesis written by Andreas Berg Skomedal in 2013. This thesis uses Skomedal's thesis as a basis, and seeks to expand and improve upon his work. Skomedal's thesis focuses on implementing a heterogeneous scheduler that schedules FDTD tasks between a single CPU and a single GPU. It does so by having two different FDTD implementations (CPU and GPU) running as POSIX threads. Tasks are then scheduled among the CPU and a GPU, and executed concurrently.

Skomedal's work is in turn based on Yee_bench. The CPU version is a modified version of the Yee_bench code, while the GPU version is a CUDA port of the said code. The CPU version is improved by taking advantage of multi-core CPUs through the use of OpenMP directives.

2.6.1 Implementation

Yee_bench in CUDA

The GPU version of the FDTD method is a CUDA port of the CPU version supplied by Ulf Andersson. It was made by Skomedal as part of his specialization project during the fall of 2012 [22]. The GPU implementation allocates six arrays in device memory and initialize all the values to zero. The electromagnetic field values are calculated and updated by calling a set of kernels during every time step. Related to a FDTD grid with (N_x, N_y, N_z) cells, the kernels are launched with one thread per cell in the YZ-plane, which in turn iterates over the cells in the X-dimension. The block size is for best warp efficiency set to 16x16 (512 threads), and the grid size is set so to get at least one thread per element in the YZ-plane, excessive threads are terminated after launch. The following equation is used to calculate the grid dimensions, with respect to the FDTD domain size

and the block dimensions:

$$Block_x \times Block_y = 16 \times 16$$
$$Grid_x = Grid_y = \sqrt[2]{\frac{N^2}{Block_x \times Block_y}}$$

Where N is the length of the sides in a cubed FDTD domain

This CUDA implementation does not explicitly use the shared memory, it instead relays on implicit caching through the L1 cache, and is therefore configured to prefer L1 cache over shared memory. Skomedal explains this choice with the fact that since data fetches are not very systematic, the use of shared memory would not justify the increased complexity.

In order to allow for problems of non uniform sizes, the CUDA implementation has three "cleanup" kernels, each handling their own 2D-plane (yz, xz, xy). There is also a kernel that will set the point source, which is only executed by a single thread, and is therefore not optimal. However, Skomedal argues that this is more efficient than adding a conditional to one of the other kernels, or to transfer the required data back and forth to the host for execution.

In summary, the CUDA implementation works by iterating over the number of time steps. For each time step six kernels are launched in succession: Two update kernels (for magnetic and electric fields), three cleanup kernels (for each of the 2D-planes), and one kernel to set the point source. The electromagnetic field values will reside in device memory until all iterations are completed, before being transferred to host memory.

Heterogeneous Yee_bench scheduler

The Heterogeneous Scheduler is the main work in Skomedal's thesis. It is heterogeneous in the sense that it distributes work between both the CPU and the GPU. Work is defined as tasks, and each task is an independent FDTD problem, meaning that no data sharing or border management is done between tasks.

The Assignment Scheduler is a scheduler that divides sets of tasks among workers, in order to keep them busy for X seconds. A worker is a separate POSIX thread that uses either the CPU or the GPU to execute a task. An initial performance evaluation is done for all workers by letting them execute a dummy task. The performance (in FLOPS) of the workers is measured for every consecutive task, and the workers will only keep track of it's highest obtained performance. Each time a worker requests new tasks, the host will calculate the targeted FLOP count needed to keep it busy for X seconds:

$$F_k = S_k \times T_{k_{target}}$$

Where F is the targeted FLOP count, S is the performance in FLOPS,
T is the targeted busy time in seconds, and k is the current worker

The host uses the information about the targeted FLOP count to create a list of tasks. Tasks are added to this list until no more tasks can be added, without bypassing the targeted FLOP count for the task list:

$$F_{k_{actual}} = \sum_{i=nextavailable}^{F_{k_{actual}} > F_k} F_{task_i}$$

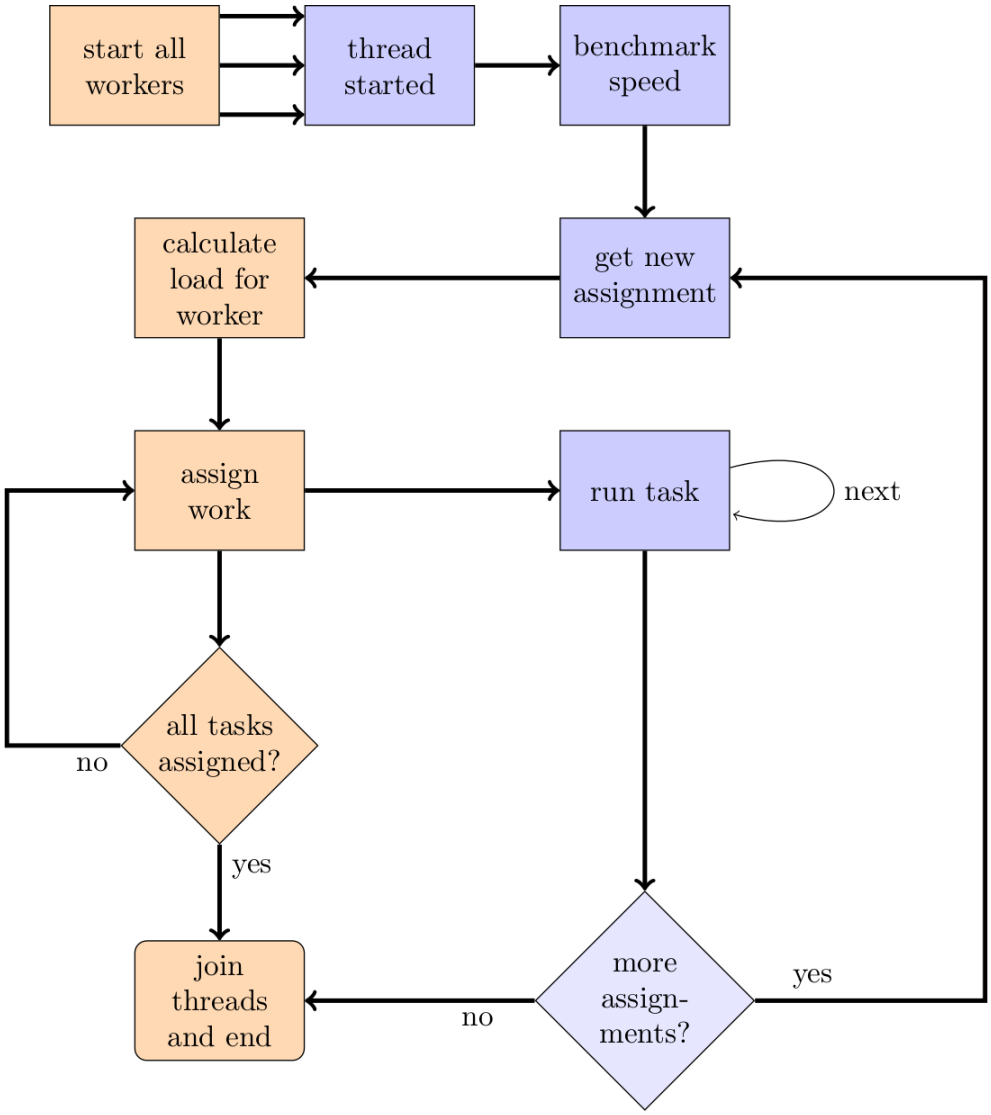

Figure 2.10: Assignment scheduler flow chart

Figure 2.10 is an excerpt from Skomedal's report, it demonstrates the flow of his Assignment Scheduler implementation.

Skomedal's work also includes another scheduler called the Greedy Scheduler, and three plotters called: Homogeneous Device Plotter, Heterogeneous Device Plotter, and Resource Plotter. The Greedy Scheduler is a simpler alternative to the Assignment Scheduler, and simply works by letting each worker fetch a new task when they have completed their current task. The plotters are benchmarks that also creates *.csv* files of performance plots, these files can later be used to create graphs. The Homogeneous Device Plotter is used to benchmark performance across different problem sizes on a single worker (CPU or GPU). The Heterogeneous Device Plotter emulates the Assignment Scheduler with multiple heterogeneous workers, and performance is benchmarked using a variety of problem sizes. The Resource Plotter is based on the Heterogeneous Device Plotter, and performance is benchmarked across different numbers of CPU and GPU threads.

2.6.2 Results

Skomedal's report presents a variety of results that will be summarized in this section. The results are reported to have been obtained using the execution platform listed in Table 2.4.

Table 2.4: Hardware and compiler configuration

Hardware and system	
CPU	Intel i5-3470 @ 3.2GHz Ivy Bridge
GPU	NVIDIA GeForce 480 GTX
GPU	NVIDIA Tesla K20
Memory	16GB 1333MHz
Motherboard	MSI Z77A G45
Operating system	Linux Mint 14, 3.5.0-23 64bit

Program	
Version	Yee_bench CUDA
GPU compiler	NVIDIA nvcc 5.0 V0.2.1221
CPU compiler	gcc 4.6.3
GPU driver	NVIDIA driver 304.88
Compiler flags	-O2

Table 2.5 presents the results from executing a task with the CPU code, using OpenMP directives and a varying number of cores. The results shows that using two cores almost doubles the performance, while going from two to three gives only a 22% increase, and three to four cores has almost a negligible increase in performance. Skomedal suggests that this is due to the memory being a limitation, he proves this by halving the frequency of the CPU cores and re-doing the tests. The new results indicates an almost linear speedup, with four cores giving a speedup of 3.59 compared to a single core.

Table 2.5: Performance for N = 150 on CPU

	1 CPU	2 CPU	3 CPU	4 CPU
MFlops	3417	6669	8139	8223
Speedup from 1 CPU	1.0	1.95	2.38	2.41

The performance of the GPU implementation running on a Fermi GTX 480 compared to the CPU implementation with different core configurations, is presented in Figure 2.11. The results show an improvement of almost 20x compared to execution on a single CPU core. A significant improvement, but still short of the full potential of the GPU. Skomedal provides and discusses a number of reasons to why the execution is not optimal, the most important reason being only reaching 46.6% global memory load efficiency due to uncoalesced memory accesses. On the other hand, both global write and warp execution efficiency is reported to perform close to optimal.

Tests were also done by executing multiple tasks on the same GPU, each on their own CUDA stream. Doing so allows for concurrency between kernel executions and memory transfers, and shows a slight improvement in performance. The tests shows that running two CUDA streams provides the best results.

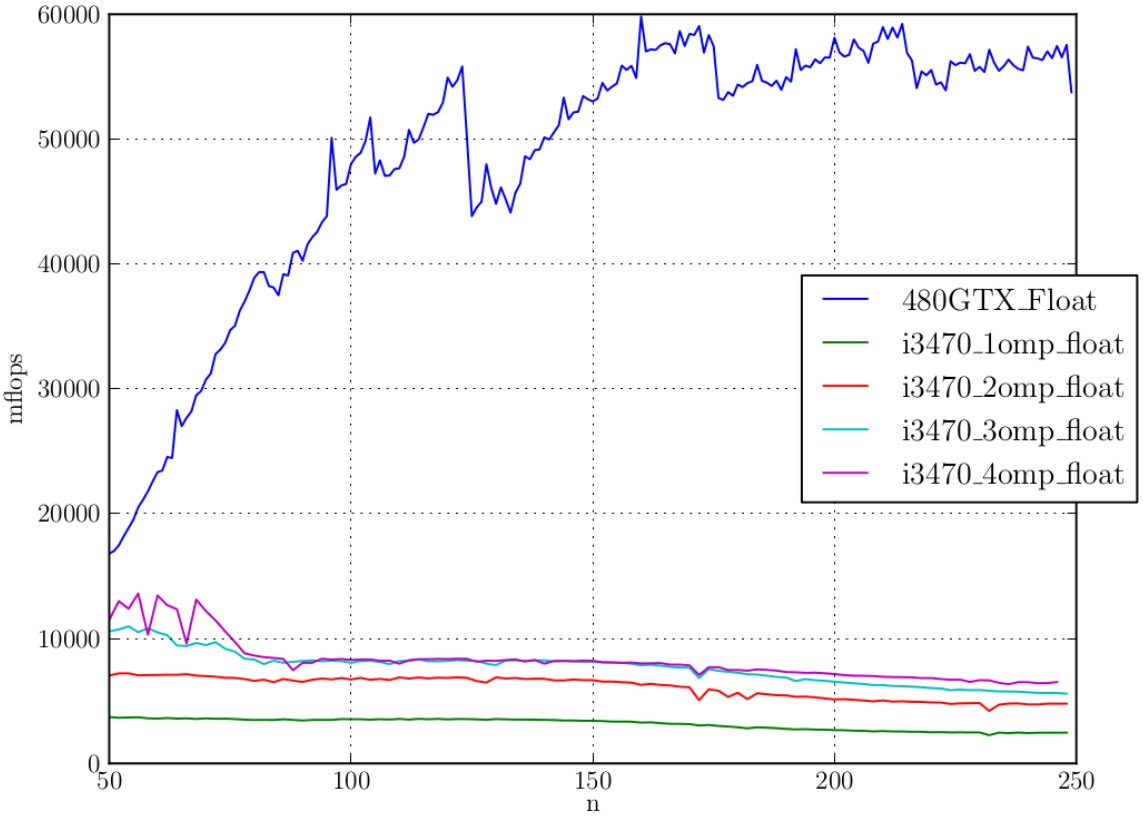

Figure 2.11: GPU speed, floating point precision for main loop

Table 2.6: Relative device performance

Problem size	CPUxGPU	CPU perf.	GPU perf.
	2x2	6200	2 x 25000
150	2x0	6450	0
	0x2	0	2 x 26500
	4x0	8130	0
	4x2	3170	2 x 26080
	2x2	5150	2 x 26700
200	2x0	5270	0
	0x2	0	2 x 27400
	4x0	7250	0
	4x2	3700	2 x 26630
	2x2	4400	2 x 24500
250	2x0	4427	0
	0x2	0	2 x 25200
	4x0	6527	0
	4x2	3720	2 x 24170

Performance has also been measured on the NVIDIA Tesla K20 GPU, which is built on the Kepler GK110 architecture. Even though the K20 is a lot better on paper, the test results actually shows a 5-10% decrease in performance. Skomedal assumes this is due to the architectural differences between Fermi and Kepler, and that the CUDA code needs to be adjusted in order to take advantage of the new architecture.

Table 2.6 gives a summary of the performance of the Heterogeneous Scheduler with a variety of CPU cores and GPU streams, across different problem sizes. The results shows that a combination of two CPU cores and two CUDA streams gives the best combined performance for all problem sizes.

2.6.3 Personal conclusions

Skomedal's implementation is designed to execute a set of small FDTD problems on a single system with one multi-core CPU and one CUDA capable GPU. The execution units does not collaborate with each other, and the implementation is therefore not optimal for larger problems, because the compute and/or the memory requirements can be too large for a single execution unit to handle.

The implementation requires individual FDTD problems in their entirety to be transferred to and from every execution unit as they are completed. This puts a lot of stress on the bus, a demand that will increase if more execution units are added. Execution and transfers could be done concurrently, but the benefit would most likely be small to non-existing, since the executions are already limited by memory bandwidth. In addition, the implementation would presumably not scale well to many-node systems with multiple execution units per node, since the host node would have to transfer FDTD domains to and from every slave node in order to keep all their execution units occupied.

As mentioned in Section 2.6.1, the CUDA implementation works by launching one thread per cell in the YZ-plane of a domain, and making each thread iterate over the cells in the X-dimension. Another approach could have been to launch one thread per cell in the domain, thus achieving the highest possible level of parallelism. An explanation as to why this approach was chosen is not included in Skomedal's thesis. However, the article: *CUDA Based FDTD Implementation* [6], by Veysel Demir and Atef Z. Elsherbeni, provides a comparison of these two approaches. The comparison shows that due to global memory reuse, the approach taken in Skomedal's implementation is in fact the most efficient approach. It will thus be retained in the new implementation.

Skomedal's test results shows the K20 GPU performing 5-10% worse than the GTX 480 GPU, even though it has more than 2.5 times the theoretical single-precision performance. Skomedal suggests that this might be due to differences between the Kepler and the Fermi architecture. As described in Section 2.2.3 and 2.2.3, the Kepler and Fermi architecture differs in a number of ways. How these architectural differences can affect performance in a negative way is discussed in Section 5.6.

Chapter 3

Implementation

This chapter will describe the overall design and flow of the implementation, the parallelization strategy, and a more comprehensive description of some of the complex parts. It will however not be explaining the implementation of the Yee algorithm itself in detail, since it is based on codes from previous works [1, 2], which already contains descriptions of said codes. The code implementing the Yee algorithm should however be self-explanatory when viewed in conjunction with Equations 2.9a-2.9f, and Section 3.5 will explain the exceptions related to those equations.

3.1 Overall design

The program is implemented using the C++ programming language. Some of the code and concepts are based on previous work [1]. However, the entire code has been reimplemented in order to better fit the new ideas and goals of this thesis.

This implementation uses a different approach from the previous implementation [1] due to the limitations mentioned in Section 2.6.3. Instead of having multiple execution units executing several individual FDTD domains, all the execution units will collaborate on a single FDTD domain before moving onto the next one. The new implementation also makes it possible to use multiple nodes, with each node using a CPU and/or multiple GPUs. The original implementation [1] was limited to using a single CPU and/or a single GPU on a single system. Dividing FDTD domains into sub-domains also allows for much larger domain sizes, since the electromagnetic field values will be split across multiple nodes and/or GPUs, each having their own memory. Table 2.2 on page 19 shows how the memory requirements grows with the domain size.

Another approach more similar to the original scheduling implementation [1] was initially considered, and to some degree also implemented [1]. This approach worked by scheduling *individual* domains to multiple nodes, where each node could have a CPU and one or multiple GPUs, each with multiple streams. However, this approach was scrapped since it did not scale well with many execution units and nodes. The host would have to distribute individual FDTD tasks to every execution unit on every node, while continuously also receiving completed sub-domains. It did also not perform well on large domains, and thus the decomposition approach was instead chosen.

A class diagram of the new implementation using the decomposition approach is shown in Figure 3.1. A complete documentation of all the classes, functions and variables is

[1] This implementation can still be retrieved from the GitHub repository associated with this thesis.

included in Appendix C.

Following is a short description of each of the classes shown in Figure 3.1:

Node: A class that handles all the MPI communications in the program. It's classified as a *utility* stereotype in the UML diagram, this means (in this context) that Node is an all static class that there should only exist one instance of per node.[2] The Node class also contains functions for executing a Task as either the host or as a slave node, this is done through one instance of the WorkerPool class.

Task: A container for all the data associated with a FDTD problem, including six arrays for the electromagnetic field values: One for both the electric and the magnetic fields in each of the directions (x,y,z). The electromagnetic field values are stored as six individual one-dimensional arrays: *hx, hy, hz, ex, ey, ez*, but could just as well be represented as six three-dimensional arrays. The reason for referencing the data as 1D-arrays opposed to 3D-arrays is because it is retained from the original Yee_bench implementation. The data layout in memory is the same whether 1D or 3D arrays are used.

Timer: A small helper class used to simplify timing.

WorkerPool: Serves as an abstraction when doing FDTD simulations across a set of Workers. Contains a pool of Workers and a set of functions corresponding to the public functions provided by the Worker class. Calling one of the WorkerPool's functions will subsequently call the corresponding function of all the Workers contained in the WorkerPool, then wait for the Workers to complete (synchronize) before returning to the caller.

Worker: An abstract class with virtual definitions for all the functions related to FDTD simulations. Provides the ability to create multiple sub-classes, which in turn can implement different versions of the compute intensive functions required to update the magnetic and electric field values of a FDTD domain. The Worker class also provides a set of public functions that will signal an internal POSIX thread, which in turn will execute a corresponding internal function. This is what makes it possible for multiple Workers to work in parallel.

CpuWorker: A class derived from the Worker class. Over-rides the functions related to FDTD simulations with methods that uses the CPU as an execution unit. Is also multi-threaded using OpenMP directives.

GpuWorker: Another class derived from the Worker class. Uses a NVIDIA GPU as an execution unit by implementing and calling CUDA kernels from the over-ridden functions. One GpuWorker relates to one GPU, and execution using multiple GPUs therefore requires multiple instances of the class.

[2]Node with a capital N is used when referring to the Node class, while node is used when referring to a physical node, such as in a cluster of nodes.

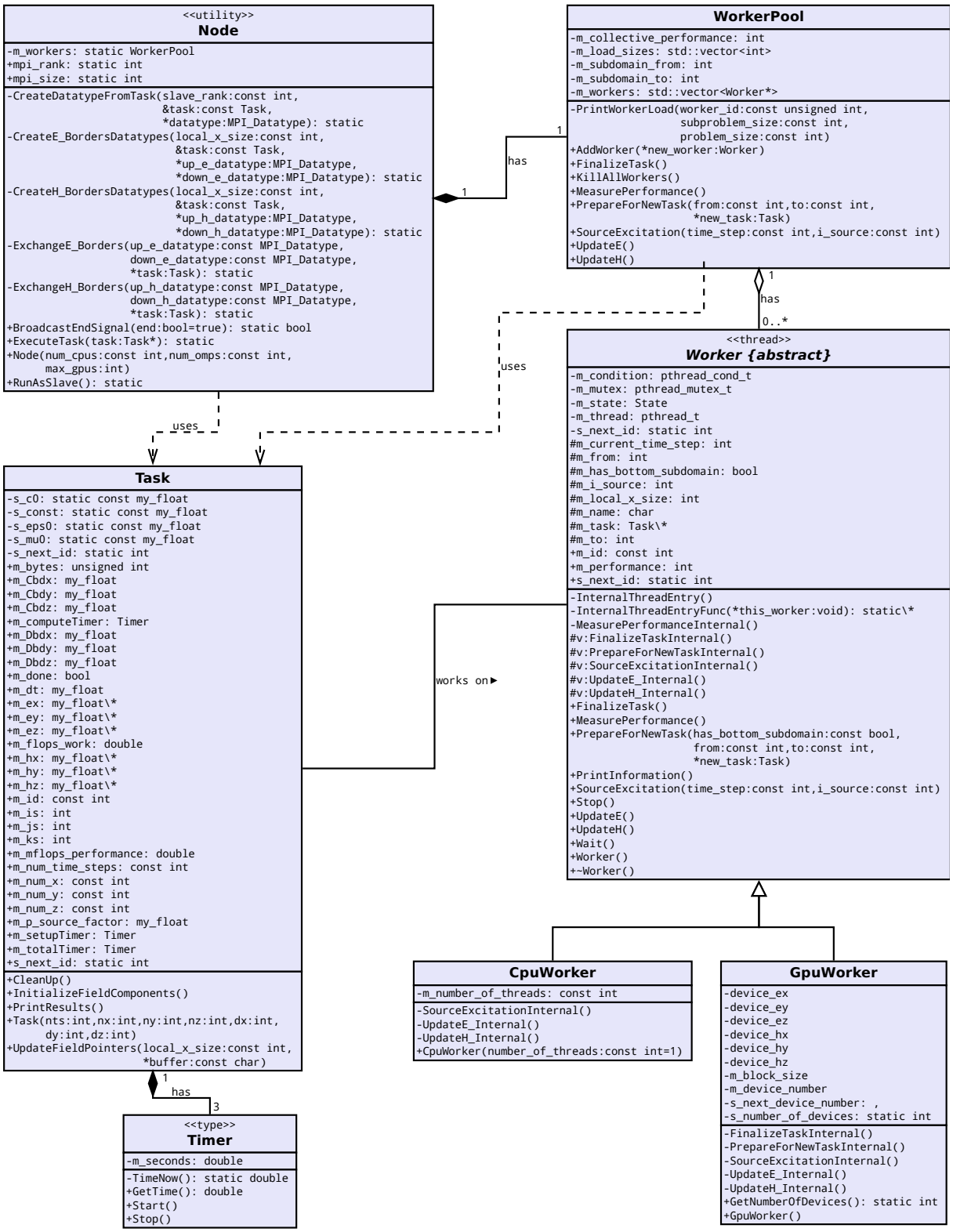

Figure 3.1: UML class diagram

3.2 Overall flow

A visual representation of the implementation's flow is shown in Figure 3.2. The flowchart does not map directly to the code and its functions, but rather the general flow of the implementation.

Following is a description of the various steps shown in Figure 3.2:

1. All the nodes starts by loading a *config* file that contains the user specified Worker configurations: This consists of the number of CpuWorkers and GpuWorkers, and the number of OpenMP threads used per CpuWorker.
2. Each Node will create a set of Workers according to the loaded configurations. A small predefined FDTD problem is executed individually on each of the created Workers in order to measure their performance.
3. The host Node generates one or multiple Tasks based on the FDTD problem configuration, which is also located in the *config* file.
4. The host Node picks the first Task from the generated Tasks, and distributes the FDTD domain evenly among the slave Nodes using MPI sends (the host Node will also get a sub-domain).
5. Each Node divides its sub-domain into even smaller sub-domains among its Workers. The size of a Worker's sub-domain is determined by its measured performance. Workers with dedicated memory will load its sub-domain into memory (e.g. GPUs).
6. Every Worker in every Node will execute its sub-domain in parallel:
   (a) Workers with dedicated memory loads its H borders from system memory.
   (b) All Workers updates the electrical field components based on the magnetic field components.
   (c) Workers with dedicated memory unloads its E borders into system memory.
   (d) Nodes will exchange their electrical field component borders with neighboring Nodes using MPI send/receive.
   (e) The Worker who owns the sub-domain containing the source point updates the source point.
   (f) Workers with dedicated memory loads its E borders from system memory.
   (g) All Workers updates the magnetic field components based on the electrical field components.
   (h) Workers with dedicated memory unloads its H borders into system memory.
   (i) Nodes will exchange their magnetic field component borders with neighboring Nodes using MPI send/receive.
   (j) Unless its the last time-step, the time-step is incremented and execution jumps to step *a*.
7. Each Node combines the sub-domains from all its Workers. This step is only relevant for Workers with dedicated memory, since they will have to copy their completed sub-domain into the system memory.

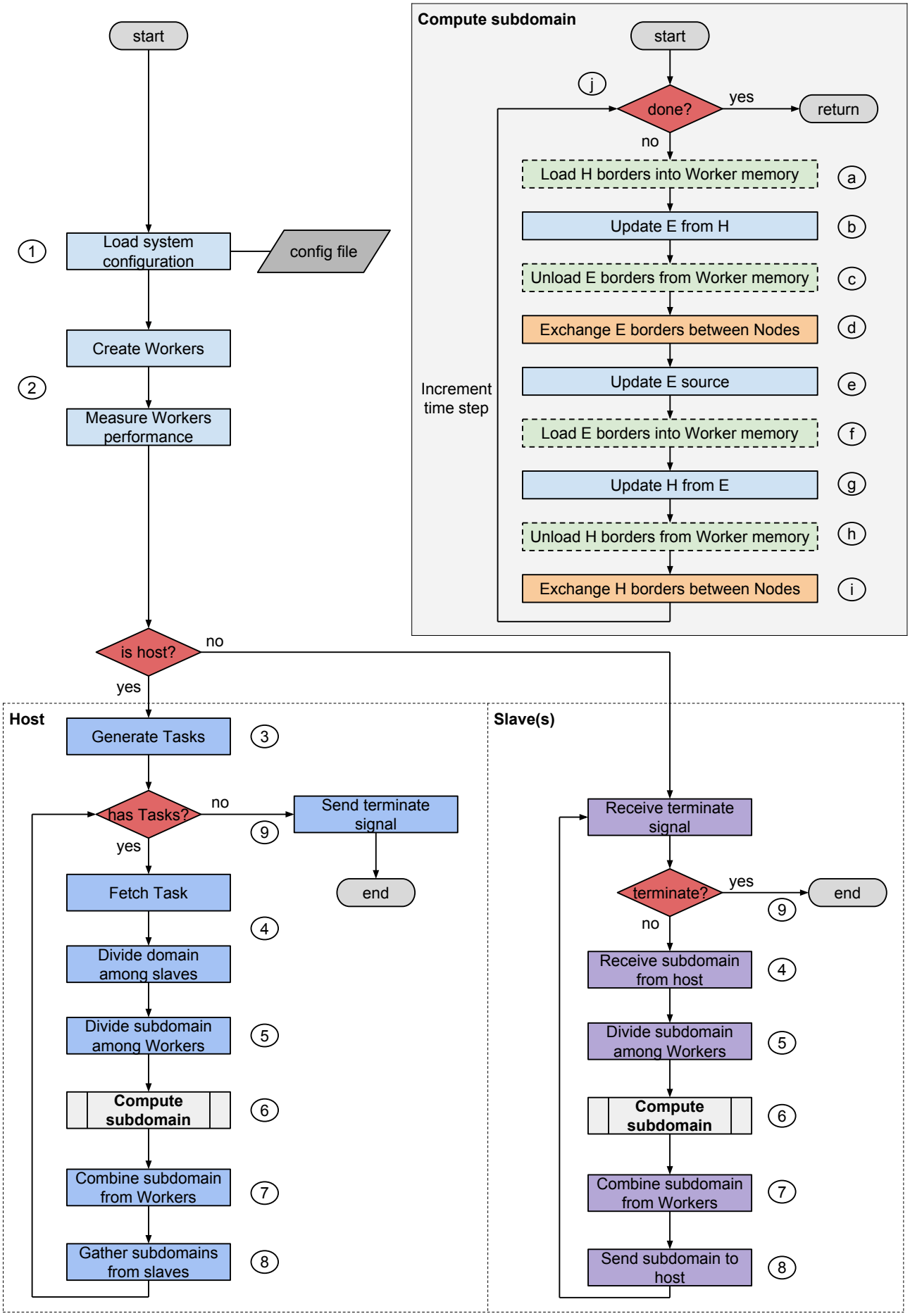

Figure 3.2: Flowchart

8. The host Node will gather the sub-domains from all the slave Nodes into a final solution using MPI receives.

9. If there are more Tasks, the host Node will fetch the next one and repeat from step *4*. Otherwise, the host Node will use MPI broadcast to signal all the slave Nodes to terminate.

3.3 Reimplementation in C++ and the use of libraries

The new implementation has been done in C++ rather than C, which was used in the original implementation [1]. The main reason for switching programming language was to get a modular and maintainable code through the use of classes. The ability to use inheritance also makes it possible to omit repeating code, which further improves the maintainability of the code. An example of the use of inheritance is the Worker class, which has been implemented as an abstract class with the CpuWorker and the GpuWorker as sub-classes. The use of an abstract class also makes it possible to extend the implementation with new Workers in the future (such as Workers based on OpenCL or OpenACC), without changing the overall structure of the code.

Even though the implementation is done in C++, it does not use the MPI C++ bindings. The reason for this is that the C++ bindings in Open MPI has been deprecated as of MPI-2.2. Choosing to use the C bindings will therefore be more future proof in case the code is ever used in future projects.

The use of libraries has in the interest of making the code understandable and future proof been limited to open and well proven standards, like the C++ standard library and POSIX threads. The use of C++11 features and/or the Boost library has been considered, but omitted, since it was determined that it would not make any notable improvements to the code, and instead only add unnecessary dependencies.

3.4 Parallelization strategy

Multiple Nodes and Workers are able to collaborate on one FDTD problem by splitting the domain into smaller sub-domains and exchange their borders during execution. This is also how it is done in previous FDTD implementations on distributed memory systems [4, 5, 7].

This implementation uses strip domain decomposition to divided the domain into sub-domains. Box decomposition has also considered, but the strip method was chosen for the following reasons:

- Simplicity
- Data contiguity
- Number of required border exchanges

Strip domain decomposition works by splitting a domain into multiple sub-domains along one of the axis, as shown in Figure 3.3. This makes it easy to split a domain of any size into any number of sub-domains.

Multidimensional arrays in C++ are stored contiguously in a row-major order in memory. Maintaining the contiguity in sub-domains is crucial for good cache utilization,

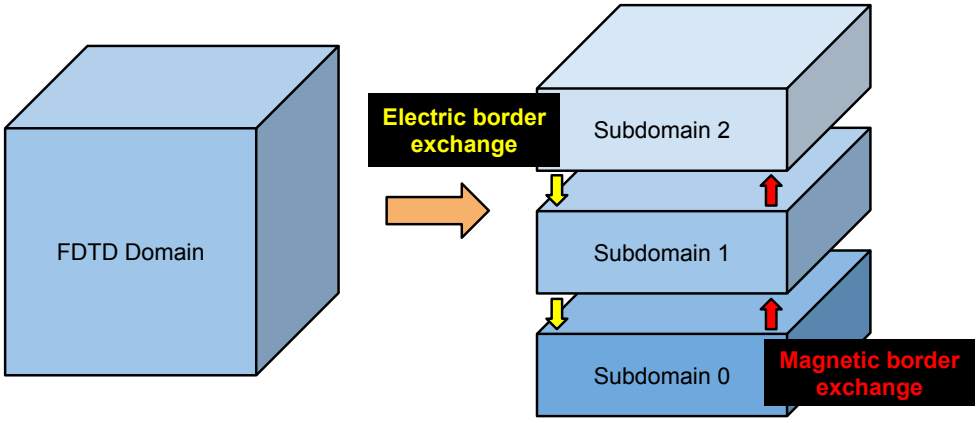

Figure 3.3: FDTD strip domain decomposition and border exchanges

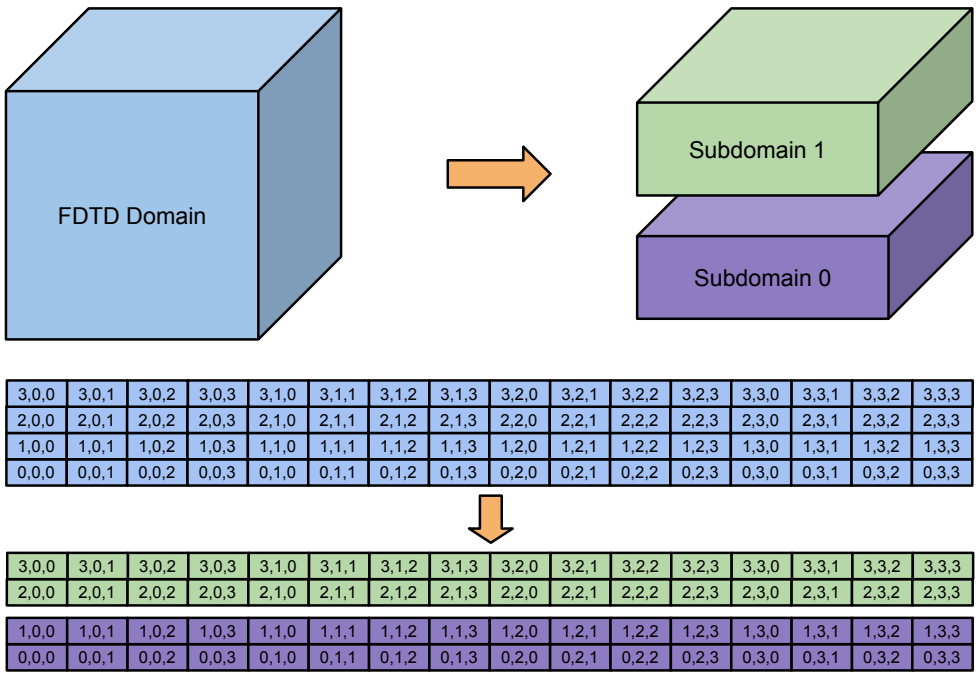

Figure 3.4: Data contiguity

since the cache utilization depends on memory reads to have spatial locality. Figure 3.4 shows how the cells are located in memory and how strip decomposition will maintain the contiguity. The cell indexes also shows that the domains will have to be split along the axis representing the highest dimension of the domains, in order to keep contiguity. In Equations 2.9a-2.9f the highest dimension will be represented by index i, which refers to the X dimension in the code.

As seen in Figure 3.3, sub-domains generated using strip decomposition have only (at most) two neighbors they need to exchange borders with. In comparison, sub-domains generated using box decomposition would have (at most) four neighbors (six if done in 3D). Even though the use of box decomposition would require more borders to be exchanged, the total amount of border-data needed to be transmitted is less. It therefore presents a potential improvement in performance for problems where border exchanges is a limiting factor. However, box decomposition is considered more complex, and has therefore been omitted in this implementation. It has instead been left as a potential task for future work, and is further discussed in Section 5.2.1.

3.5 Boundary handling

The FDTD equations requires the electromagnetic field values of neighboring cells when calculating the values of a Yee-cell. Cells along the domain boundaries will therefore have to be handled differently than the rest, because they do not have neighboring cells in all directions.

In Equations 2.9a-2.9f on page 19, the coordinates (x, y, z) are represented by the indexes (i, j, k). The equations for calculating the magnetic field values of a cell requires the electrical field values of the neighboring cells. This would result in a segmentation fault when calculating the magnetic values in a cell along one of the boundaries in the domain. Referencing un-allocated memory is avoided by using a *fixed boundary condition*, meaning that the electrical field arrays are allocated with a padding of constant cells in each of the dimensions: $X = N + 1, Y = N + 1, Z = N + 1$. This is sufficient, since the equations only requires the *next* cells in each of the dimensions.

The equations for calculating the electrical field values are opposite, in the sense that they require the magnetic field values of the *preceding* cells in each of the dimensions. Due to the way arrays are stored in memory (see Figure 3.4), calculating the electrical fields of a cell along one of the boundaries would not result in a segmentation fault, unless that cell is in the first plan ($i = 0$). It would however introduce an error source, since referencing a cell at $(i, -1, k)$ would be equivalent to referencing $(i - 1, N - 1, k)$, or a cell at $(i, j, -1)$, which would be equivalent to $(i, j - 1, N - 1)$.[3] Both segmentation faults and the error source is avoided by not calculating the electrical field values that requires accessing cells with a negative index. From Equation 2.9d, 2.9e and 2.9f, all the calculations with an index where either i, j or k equals 0 will have to be omitted, except for the following: $E_x(0, \tilde{j}, \tilde{k})$, $E_y(\tilde{i}, 0, \tilde{k})$ and $E_z(\tilde{i}, \tilde{j}, 0)$, where $\tilde{i}$, $\tilde{j}$ and $\tilde{k}$ can be in the range: 1 to $N - 1$.

[3] Assuming the domain is of size (N, N, N), where the indexes i, j and k ranges from 0 to N-1.

3.6 Nodes

In Figure 3.2, step 4, 6(d), 6(i) and 8 requires the nodes to communicate with each other. All communication between nodes is done using Open MPI by functions within the Node class. Step 4 of the flowchart scatters the domain across all the nodes, which is done in two stages:

During the first stage, the host node shallow copies[4] the Task instance to all the slave nodes by broadcasting it, so that all nodes contains a copy of the Task object in its memory. However, because it is only a shallow copy, the arrays containing the electromagnetic field values (*hx, hy, hz, ex, ey, ez*) will not be copied. Additionally, the pointers to these arrays will simply point to garbage on the slave nodes.

The next stage of step 4 is to divide the domain into sub-domains and distribute the electromagnetic field values among the slave nodes. Instead of having to do one send operation for each of the six arrays for all the slave nodes, a derived data type is created for each of the slave nodes. Listing 3.1 contains the function used to create such a derived data type from a Task, appropriately called *CreateDatatypeFromTask()*. The function is only called by the host node in a loop iterating over the number of slave nodes.

The CreateDatatypeFromTask() function will, if possible, divide the domain into equally sized sub-domains along the X-axis, as explained in Section 3.4. If the size of the domain in the X-dimension is not dividable by the number of nodes, the remainder will be added to the first nodes, starting with the node with rank 0 (the host node). The derived data types created from the aforementioned function are used to send one unique sub-domain to each of the slave nodes. Upon receiving the sub-domains, the slave nodes stores the received electromagnetic field values contiguously in a buffer. The array pointers in the slave nodes' copy of the Task object is then updated to point to the arrays stored in that buffer. The derived data types are kept by the host node until the FDTD simulation has completed (step 8 in Figure 3.2), they are then used when receiving the completed sub-domains from the slave nodes.

3.6.1 Inter-communication

During every time-step of a FDTD simulation, both the electrical and the magnetic field values will be updated. As mentioned in Section 3.5, updating a field value of a cell requires the field values of the surrounding cells. In order to update the cells along the borders[5] of a sub-domain, the borders of two adjacent sub-domains will have to be exchanged by the nodes owning them. This is done for both the electrical and the magnetic field values during step 6(d) and 6(i) of the flow in Figure 3.2.

Updating the electrical field values of a cell requires the magnetic field values of the preceding cells in each of the dimensions. Since the domain is divided using strip domain decomposition along the X-axis (as shown in Figure 3.3), only the preceding cell in the X-dimension is needed. Specifically, Equations 2.9d-2.9f on page 19 shows that only H_y and H_z from the preceding cell in the X-dimension is required. The same applies when updating the magnetic fields, except that it requires the E_y and E_z of the *next* cell in the

[4] A bitwise copy of an object. The process of shallow copying will copy all of the field values, even if the field value is a memory address (a pointer).

[5] Boundaries and borders are not equal in this context. A boundary describes the outer cells of the entire domain. A border describes the outer cells of a sub-domain where it is adjacent to another sub-domain.

Listing 3.1: Node::CreateDatatypeFromTask(...)

```
void Node::CreateDatatypeFromTask(
    const int slave_rank, const Task &task, MPI_Datatype *datatype)
{
    const int yz_offset_e = (task.m_num_y+1) * (task.m_num_z+1);
    const int yz_offset_h = task.m_num_y * task.m_num_z;

    int rank_x_size = task.m_num_x / mpi_size;
    int e_displacement = rank_x_size * yz_offset_e * slave_rank;
    int h_displacement = (rank_x_size-1) * yz_offset_h * slave_rank;

    /* Handles cases where work can NOT be evenly divided among Nodes */
    const int x_remainder = task.m_num_x % mpi_size;
    if (slave_rank < x_remainder)
    {
        rank_x_size++;
    }
    for (int rank = 0; rank < x_remainder && rank < slave_rank; rank++)
    {
        e_displacement += yz_offset_e;
        h_displacement += yz_offset_h;
    }

    const int e_size = (rank_x_size+1) * yz_offset_e * sizeof(my_float);
    const int h_size = (rank_x_size+1) * yz_offset_h * sizeof(my_float);

    const int number_of_blocks = 6;

    int block_lengths[number_of_blocks];
    block_lengths[0] = e_size;
    block_lengths[1] = e_size;
    block_lengths[2] = e_size;
    block_lengths[3] = h_size;
    block_lengths[4] = h_size;
    block_lengths[5] = h_size;

    MPI_Aint displacements[number_of_blocks];
    displacements[0] = (MPI_Aint)&task.m_ex[e_displacement] - (MPI_Aint)&task;
    displacements[1] = (MPI_Aint)&task.m_ey[e_displacement] - (MPI_Aint)&task;
    displacements[2] = (MPI_Aint)&task.m_ez[e_displacement] - (MPI_Aint)&task;
    displacements[3] = (MPI_Aint)&task.m_hx[h_displacement] - (MPI_Aint)&task;
    displacements[4] = (MPI_Aint)&task.m_hy[h_displacement] - (MPI_Aint)&task;
    displacements[5] = (MPI_Aint)&task.m_hz[h_displacement] - (MPI_Aint)&task;

    MPI_Datatype types[number_of_blocks];
    std::fill_n(types, number_of_blocks, MPI_BYTE);

    MPI_Type_create_struct(number_of_blocks, block_lengths,
        displacements, types, datatype);
    MPI_Type_commit(datatype);
}
```

X-dimension. The directions of both magnetic and electrical border exchanges are shown in Figure 3.3.

Derived data types are also used for border exchanges, to simplify the process of sending and receiving. Listing 3.2 shows the creation of data types for both the *up* and *down* magnetic borders of a sub-domain. Note that the displacement for the *low* border uses negative indexes for both the *hy* and *hz* arrays, this is because buffers for containing the borders has been allocated ahead of both *hy* and *hz* in memory. Thus making it possible to for example reference $hy(-1, j, k)$, which is needed when updating $ez(0, i, k)$.[6]

Listing 3.3 shows the function called by every node for every time-step when the magnetic borders needs to be exchanged. Every node will receive the upper-most magnetic border in the X-dimension from the preceding node, and send its upper-most border to the next node. Exceptions are the first and last node, since they will contain the first and last sub-domains, and therefore only need to either send or receive one border.

Corresponding functions to those included in Listing 3.2 and 3.3 also exists for handling the electrical borders. They are very similar to the included functions for the magnetic field borders and will therefore not be described here.

3.7 Workers

Each Node contains a set of Worker objects. Specifically, instances of either the *CpuWorker* or the *GpuWorker* sub-classes. The CpuWorker and GpuWorker implements their own versions of the internal functions: UpdateH_Internal(), UpdateE_Internal(), and SourceExcitationInternal(). The *internal* suffix emphasizes that these functions will be executed by the internal thread of the Worker object.

With the creation of every Worker object, a new thread will be created and associated with that object.[7] Calls to the Worker class' public functions will change a member state variable and signal the internal thread to wake up. The *InternalThreadEntry()* function included in Listing 3.4 shows the main function of a Worker's internal thread. The public functions of the Worker class will return immediately after signaling the internal thread, which is required, so that the main thread can signal the next Worker and thereby let multiple Workers run in parallel. The public *Wait()* function included in Listing 3.5 will instead wait for the internal thread to complete its current work and reach an idle state, a necessity for when Workers needs to synchronize between updates of the electromagnetic field values.

3.7.1 Intra-communcation

Workers within each Node that do not share memory will have to exchange borders for the same reasons they are exchanged between nodes. This applies to the GpuWorker sub-class, since it uses CUDA capable GPUs with their own memory.

During step 5 of the flow in Figure 3.2, the nodes' sub-domains is further divided into smaller sub-domains intended for the Workers. Each instance of the GpuWorker class will allocate enough memory for their sub-domain and border buffers, before loading the sub-domain into memory. The border buffers are needed for keeping copies of the

[6] The index of every sub-domain on every node will range from $(0, 0, 0)$, regardless of the starting index the sub-domain has in the complete domain.

[7] Having a thread linked to the objects lifetime omits the overhead of creating and destroying threads on the fly.

Listing 3.2: Node::CreateH_BordersDatatypes(...)

```
void Node::CreateH_BordersDatatypes(
    const int local_x_size, const Task &task,
    MPI_Datatype *up_h_datatype, MPI_Datatype *down_h_datatype)
{
    const int number_of_blocks = 2;
    int block_lengths[number_of_blocks];
    MPI_Aint displacements[number_of_blocks];
    MPI_Datatype types[number_of_blocks];
    std::fill_n(types, number_of_blocks, MPI_BYTE);

    const int yz_offset_h = task.m_num_y * task.m_num_z;
    const int h_size = yz_offset_h * sizeof(my_float);
    std::fill_n(block_lengths, number_of_blocks, h_size);

    displacements[0] =
        (MPI_Aint)&task.m_hy[(local_x_size-1)*yz_offset_h] - (MPI_Aint)&task;
    displacements[1] =
        (MPI_Aint)&task.m_hz[(local_x_size-1)*yz_offset_h] - (MPI_Aint)&task;

    MPI_Type_create_struct(number_of_blocks, block_lengths, displacements,
        types, up_h_datatype);
    MPI_Type_commit(up_h_datatype);

    displacements[0] = (MPI_Aint)&task.m_hy[-yz_offset_h] - (MPI_Aint)&task;
    displacements[1] = (MPI_Aint)&task.m_hz[-yz_offset_h] - (MPI_Aint)&task;

    MPI_Type_create_struct(number_of_blocks, block_lengths, displacements,
        types, down_h_datatype);
    MPI_Type_commit(down_h_datatype);
}
```

Listing 3.3: Node::ExchangeH_Borders(...)

```
void Node::ExchangeH_Borders(const MPI_Datatype up_h_datatype,
    const MPI_Datatype down_h_datatype, Task *task)
{
    if (mpi_rank == 0) {
        MPI_Send(task, 1, up_h_datatype, mpi_rank+1,
            FIELD_BORDER_TAG, MPI_COMM_WORLD);
    }
    else if (mpi_rank == mpi_size-1) {
        MPI_Recv(task, 1, down_h_datatype, mpi_rank-1,
            FIELD_BORDER_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else {
        MPI_Sendrecv(task, 1, up_h_datatype, mpi_rank+1, FIELD_BORDER_TAG,
            task, 1, down_h_datatype, mpi_rank-1, FIELD_BORDER_TAG,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Listing 3.4: Worker::InternalThreadEntry()

```
void Worker::InternalThreadEntry()
{
    pthread_mutex_lock(&m_mutex);
    while (true)
    {
        switch (m_state)
        {
            case UPDATE_H:
                UpdateH_Internal();
                break;
            case UPDATE_E:
                UpdateE_Internal();
                break;
            case SOURCE_EXCITATION:
                SourceExcitationInternal();
                break;
            case BENCHMARK:
                MeasurePerformanceInternal();
                break;
            case STOP:
                m_state = IDLE;
                pthread_cond_signal(&m_condition);
                goto stop;
            case IDLE:
                break;
        }

        m_state = IDLE;
        pthread_cond_signal(&m_condition);
        pthread_cond_wait(&m_condition, &m_mutex);
    }
    stop:
    pthread_mutex_unlock(&m_mutex);
}
```

Listing 3.5: Worker::Wait()

```
void Worker::Wait()
{
    pthread_mutex_lock(&m_mutex);
    while (m_state != IDLE)
        pthread_cond_wait(&m_condition, &m_mutex);
    pthread_mutex_unlock(&m_mutex);
}
```

borders of the neighboring sub-domains. These borders are loaded before each update of either electrical or magnetic field values in every time-step. In relation, the borders of the sub-domains contained by GpuWorker objects will also have to be copied onto the system memory after each update. This is required for the borders to be available to other Workers, and also for them to be included in the MPI sends during exchanges between nodes.

Listing 3.6 shows the GpuWorker class implementation of the *UpdateH_Internal()* function, which corresponds to step 6(a), 6(b) and 6(c) of the flow in Figure 3.2. This function has a call to the UpdateH_cuda() kernel, which will use the GPU to do the actual updating of the magnetic field values. Before calling this kernel, the borders of the electrical field values are copied into the device's memory from the host (system) memory.[8] After the magnetic field values have been updated, the upper magnetic field borders are copied to their related addresses in system memory. Note that the global variable: *g_single_worker*, prevents border exchanges from being done in cases where only one worker is used.

The GpuWorker class also implements a function called *UpdateE_Internal()*, which corresponds to step 6(f), 6(g) and 6(h) in Figure 3.2. It works similar to the UpdateH_Internal() function, and will therefore not be included. The difference is that the upper magnetic borders of the preceding sub-domain are copied into device memory before the electrical fields are updated, and the lower borders of the updated electrical field values are afterwards copied back to the system memory. Figure 3.3 demonstrates the directions the borders are copied between Workers.

[8]The member variables *m_from* and *m_to* are used to find the addresses of the borders in system memory. They hold the *from* and *to* coordinates in the X-dimension of the Worker's sub-domain related to the Node's sub-domain.

Listing 3.6: GpuWorker::UpdateH_Internal()

```
void GpuWorker::UpdateH_Internal()
{
    cudaSetDevice(m_device_number);

    const int ny = m_task->m_num_y;
    const int nz = m_task->m_num_z;

    if (g_single_worker == false)
    {
        /* Load upper E-fields borders into device memory */
        const my_float *host_ey = m_task->m_ey;
        const my_float *host_ez = m_task->m_ez;
        const int yz_offset_e = (ny+1) * (nz+1);
        const int e_border_size = yz_offset_e * sizeof(my_float);
        cudaMemcpy(&device_ey[m_local_x_size*yz_offset_e],
            &host_ey[m_to*yz_offset_e], e_border_size, cudaMemcpyHostToDevice);
        cudaMemcpy(&device_ez[m_local_x_size*yz_offset_e],
            &host_ez[m_to*yz_offset_e], e_border_size, cudaMemcpyHostToDevice);
    }

    const int yz_offset_h = ny * nz;

    const int grid_yz = ceil(sqrt(yz_offset_h / (float)m_block_size));
    const dim3 dim_grid_yz(grid_yz, grid_yz);

    GpuWorkerKernels::UpdateH_cuda<<<dim_grid_yz, m_block_size>>>(
        m_local_x_size, ny, nz, m_task->m_Cbdx, m_task->m_Cbdy, m_task->m_Cbdz,
        device_ex, device_ey, device_ez, device_hx, device_hy, device_hz);

    if (g_single_worker == false)
    {
        /* Un-load upper H-fields borders onto host memory */
        my_float *host_hy = m_task->m_hy;
        my_float *host_hz = m_task->m_hz;
        const int h_border_size = yz_offset_h * sizeof(my_float);
        cudaMemcpy(&host_hy[(m_to-1)*yz_offset_h],
            &device_hy[(m_local_x_size-1)*yz_offset_h], h_border_size,
            cudaMemcpyDeviceToHost);
        cudaMemcpy(&host_hz[(m_to-1)*yz_offset_h],
            &device_hz[(m_local_x_size-1)*yz_offset_h], h_border_size,
            cudaMemcpyDeviceToHost);
    }
}
```

Chapter 4

Results

This chapter presents various results from executing the implementation using different program and hardware configurations. It will include results covering the overhead from doing border exchanges, optimal load distribution between multiple execution units, and a comparison to Skomedal's original implementation [1]. It will also include results comparing executions on the NVIDIA Fermi and Kepler architectures, and results from running on an improvised multi-node setup.

4.1 Test setup

All results in this chapter have (unless otherwise stated) been obtained using the following configurations: Three different domain sizes have been used: N=100, N=200 and N=400. Where the total number of Yee-cells are N^3, and the required memory is 23 MB for N=100, 184 MB for N=200, and 1470 MB for N=400. The number of time-steps is 200 for all sizes.

In order to get accurate results, averages of several runs have been used: 40 runs for N=100, 20 runs for N=200 and 3 runs for N=400. Different number of runs have been used for each of the domain sizes since the execution time variance less on larger sizes.

The performance is measured in FLOPS, which is calculated from the number of floating-point operations required for a FDTD simulation and the total execution time (including data transfers) of that simulation. The number of required floating-point operations is calculated using Equation 4.1.

$$\begin{aligned} \text{FLOPS} = ((N_y - 1) \times (N_z - 1) \times 6 \\ + (N_x - 1) \times (N_z - 1) \times 6 \\ + (N_x - 1) \times (N_y - 1) \times 6 \\ + (N_x - 1) \times (N_y - 1) \times (N_z - 1) * 18 \\ + N_x \times N_y \times N_z \times 18) \\ \times TS \end{aligned} \tag{4.1}$$

Where N_x, N_y, N_z is the domain size and TS is the number of time-steps

4.1.1 Execution platforms

Table 4.1 presents the hardware and software configurations of the execution platforms used to gather the results. The three test systems uses three different generations of Intel i7 micro-architectures: Nehalem (System 3), Ivy Bridge (System 1) and Haswell (System 2). All of these processors have four physical cores and a similar clock frequency, their performance do however differ due to different cache sizes and implemented features and optimizations.

The systems also contains different GPU configurations: System 1 is equipped with two NVIDIA Tesla K20c of the Kepler GK110 architecture. System 2 has a NVIDIA Tesla K40c, also of the Kepler GK110 architecture, and a GTX 760 of the slightly older Kepler GK104 architecture. System 3 is primarily included in order to test and discuss the performance differences between the Fermi and Kepler architectures. It is equipped with a GTX 470 GPU of the Fermi GF100 architecture and a GTX 280 of the obsolete GT200 architecture.

Appendix A contains an overview of the GPU architectures and a comprehension of the differences in theoretical performance. However, these performance numbers represents the GPUs' theoretical peaks when factors like memory bandwidth is not an issue, and the results presented in this chapter is therefore not expected to be comparable to these values.

The use of Error-Correcting Code (ECC) memory has been disabled during all tests, since it can come with a slight cost to memory performance.

Table 4.1: Test systems

	System 1 (K20c)	**System 2 (K40c)**	**System 3 (GTX470)**
Hardware:			
Motherboard	MSI Z77A-G45	MSI Z87-G45 GAMING	EVGA X58 SLI
CPU	Intel Core i7-3770K, 3.5GHz	Intel Core i7-4771, 3.5GHz	Intel Core i7-950, 3.07GHz
Memory	4x DDR3 8GB, 1333MHz	4x DDR3 8GB, 1333MHz	6x DDR3 2GB, 1600MHz
Storage	Intel SSD 330, 120GB	Seagate SSD 600, 240GB	WD HDD Green, 500GB
GPU0	Tesla K20c, 5GB	Tesla K40c, 12GB	GTX 470, 1280MB
	PCIe 2.0 x8	PCIe 2.0 x8	PCIe 2.0 x16
GPU1	Tesla K20c, 5GB	GTX 760, 4GB	GTX 280, 1GB
	PCIe 2.0 x8	PCIe 2.0 x8	PCIe 2.0 x8
Software:			
Operating System	Ubuntu 12.04 LTS, 64-bit	Ubuntu 12.04 LTS, 64-bit	Ubuntu 12.04 LTS, 64-bit
NVIDIA Driver Version	331.62	331.20	331.62
GCC Version	4.6.4	4.6.4	4.6.4
NVCC Version	6.0	5.5	6.0

4.2 CPU performance

The implementation allows for several possible ways to configure the use of the CPU: A single multi-core system can take advantage of its multi-threaded capabilities by running either multiple OpenMP threads, CpuWorkers, Nodes, or any combination of the mentioned. Running multiple Nodes basically means running several MPI processes, and running multiple CpuWorkers equals to running several POSIX threads.

Figure 4.1 and 4.2 shows the performance scaling across multiple cores on System 1 and 2, respectively. The results are similar to those provided in Table 2.5, and shows

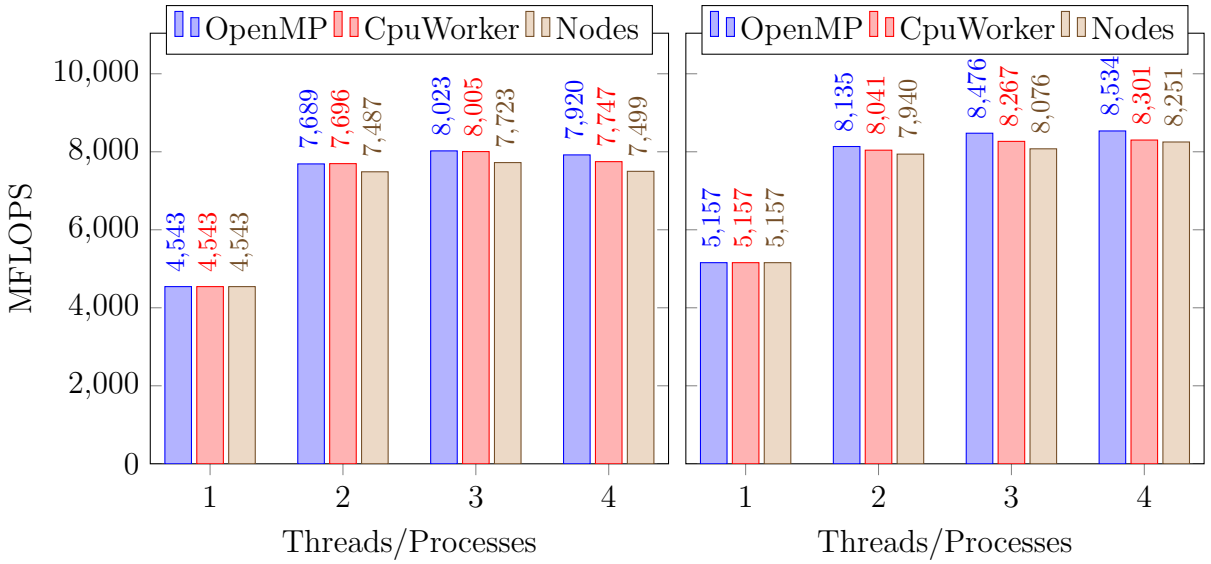

Figure 4.1: Performance of different CPU configurations on System 1 (Ivy Bridge), using a domain of size N=150

Figure 4.2: Performance of different CPU configurations on System 2 (Haswell), using a domain of size N=150

that using more than two cores gives a negligible increase in performance due to memory bandwidth being a limitation. Because of these observations, only two threads will be used when the CPU is used together with GPU(s).

The figures also shows a comparison between using multiple OpenMP threads, CpuWorkers and Nodes. Even though the differences are minor, the results indicates that using multiple OpenMP threads is the most efficient way to utilize a multi-core CPU. This is as expected, since the implementation is designed to parallelize the FDTD algorithm on CPUs by launching multiple OpenMP threads. Using several CpuWorkers or Nodes on a single system is merely a possibility, and will introduce an unneeded overhead.

4.3 Border exchange overhead

In order for FDTD simulations to be done collectively on multiple execution units, borders have to be exchanged during execution. This introduces a noteworthy overhead and decrease in performance. Figure 4.3 shows a comparison of executions with and without border exchanges on a single K20c GPU on System 1. Executing on a single GPU does not require borders to be exchanged during execution, and the implementation will withhold from doing so if it registers that there is only a single execution unit in use. Border exchanges have therefore been enforced during these measurements, in order to show the overhead.

The results in Figure 4.3 shows a reduction in performance of 18.7%, 7.7% and 3.6%, for domain sizes of N=100, N=200 and N=400, respectively. This is an expected and acceptable difference, because a total of four border exchanges is being done during every time step. The results also indicates that the significance of the overhead becomes smaller as the domain becomes larger.

Due to the exchanges: the interconnect, chipset, CPU and memory can have an impact on the performance. Profiling shows that each of the border exchanges are of 160 KB for domains of size N=200, and their average transfer throughput is 3.17 GB/s on System 1

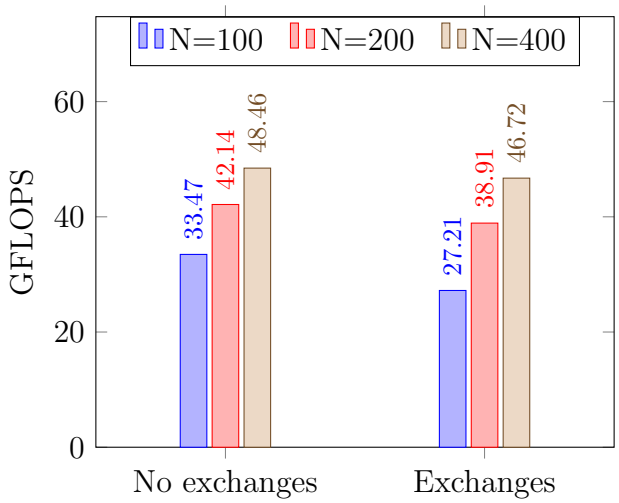

Figure 4.3: Overhead of border exchanges on a NVIDIA K20c GPU

with PCI-Express 2.0 x8. In comparison, a throughput of 5.95 GB/s was registered on System 3 which uses a PCI-Express 2.0 x16 interconnect.

4.4 Load distribution

The strength of the new implementation is its ability to have multiple execution units collaborate on a single FDTD simulation. All the execution units are however required to synchronize between every update of the electromagnetic values, and execution time is therefore limited to the unit that completes its updates last. The implementation is designed to work on heterogeneous systems where workers can have different performance. Work should therefore be distributed among workers according to their performance, in order to achieve the best overall execution time.

The implementation uses a micro-benchmark to individually measure the performance of the workers. The results from running this micro-benchmark is used to determined the load distribution between the workers, which is represented in Figure 4.4-4.7 as an orange vertical line. However, the results shows that this load distribution is not always the most efficient.

Figure 4.4-4.7 shows how the performance varies with different load distributions when divided between multiple heterogeneous execution units. The measurements done with a domain of size N=100 stands out from the others, this is likely due to the problem being too small to take advantage of multiple execution units. For large domain sizes (N=400), the results in Figure 4.4 and 4.5 shows a peak in combined performance when the CPU is scheduled 10% of the work load.

Figure 4.6 shows that the benefit of using the CPU in addition to two GPUs is negligible, and more likely to reduce the overall performance.

Figure 4.7 shows an example of executions on two different GPUs. The results suggests that a load distribution of 70% and 30% between a K40c and a GTX 760 gives the optimal performance for domains of size N=400. This is reasonable in regards to their specified performances.

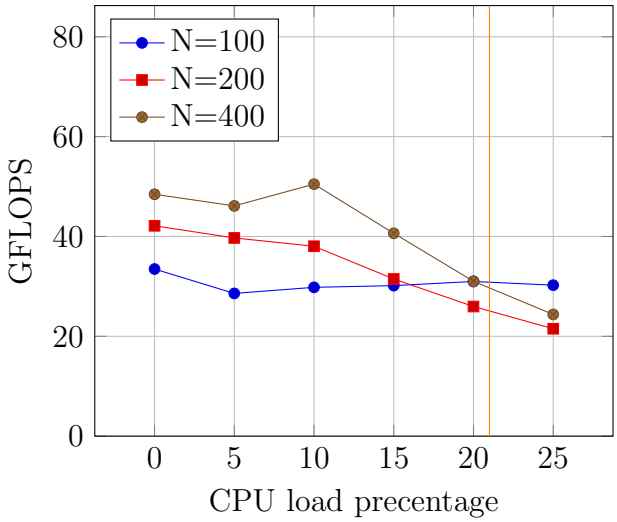

Figure 4.4: Load balance between Tesla K20c and 2xCPU

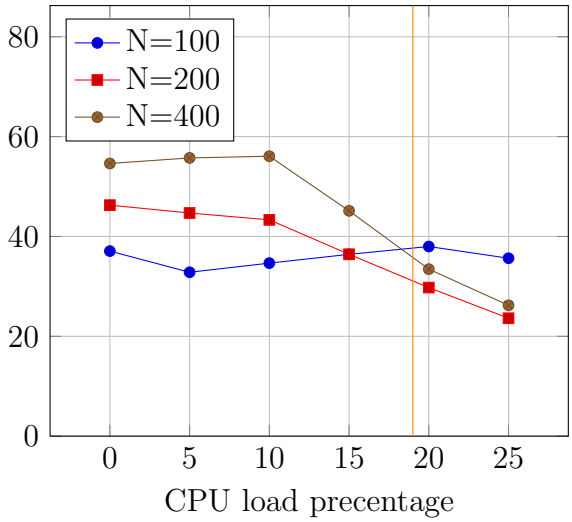

Figure 4.5: Load balance between Tesla K40c and 2xCPU

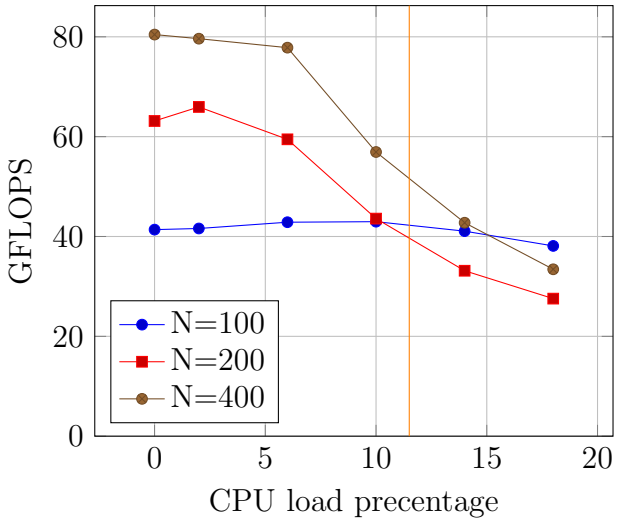

Figure 4.6: Load balance between 2xK20c and 2xCPU

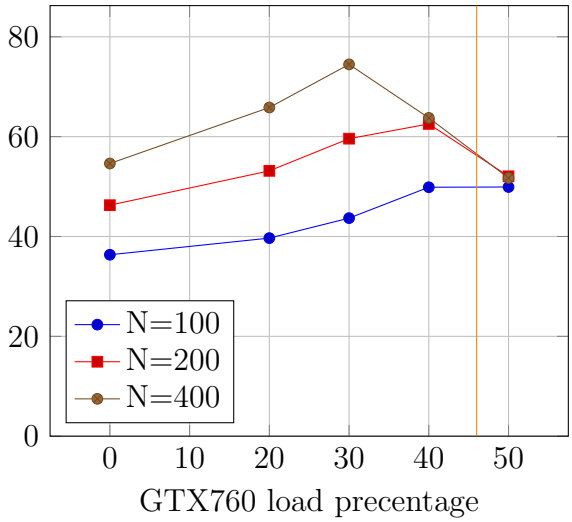

Figure 4.7: Load balance between Tesla K40c and GTX 760

4.5 Performance

A wide comparison of performance measurements when running different configurations of execution units is shown in Figure 4.8. The *4xCPU* results refers to using four OpenMP threads on the Haswell processor on System 2. The results from executing on two heterogeneous execution units where obtained by using the load balances that proved to be most efficient according to the results in Figure 4.4-4.7.

The results in Figure 4.8 shows a slight improvement when using the CPU in addition to the GPU on large domains (N=400). However, performance is either unchanged or reduced when using domains of size N=100 or N=200.

Using multiple GPUs shows a performance improvement for all domain sizes. Employing two K20c compared to one K20c GPU improves the performance by 23.6%, 50.5% and 66.0% for domains of size N=100, N=200 and N=400, respectably.

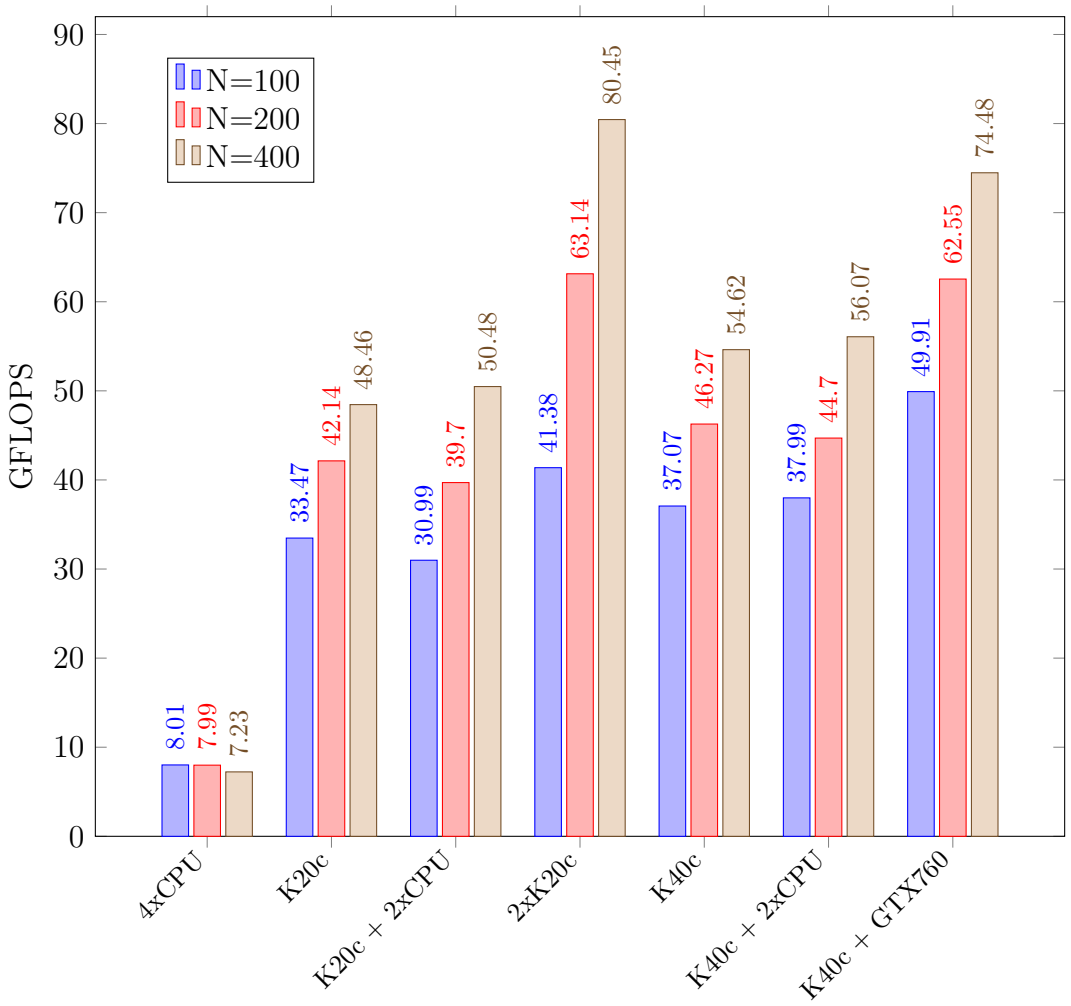

Figure 4.8: Performance of different configurations using optimal load balance

4.6 Comparison to original implementation

Figure 4.9 shows a comparison between the original [1] and the new implementation running on the same system. In order to make the implementations comparable, the code of the original implementation has been altered to also transfer the FDTD domain *to* device memory, prior to execution. Initially, this implementation only allocated and initialized the field values on device memory to zero, thereby avoiding the overhead of transferring. The code has also been modified to launch CUDA kernels with 1024 threads, equal to what is used in the new implementation. The original implementation was tested using two CUDA streams and two OpenMP threads, which has been proven to be the optimal configuration [1].

The results in Figure 4.9 compares the overall performance of simulating 20 FDTD domains on System 1. The performance is calculated by using the Unix *time* command to measure the whole execution time of the program, and then divide the combined FLOPS of all the domains by the measured time. The results indicates that the original implementation performs better when dealing with many small domains compared to the new implementation when only using one of the GPUs. However, simulations of N=400 sized domains appears to perform similar. Larger sizes has not been included because the original implementation failed to execute them. Using the multi-GPU capability of the

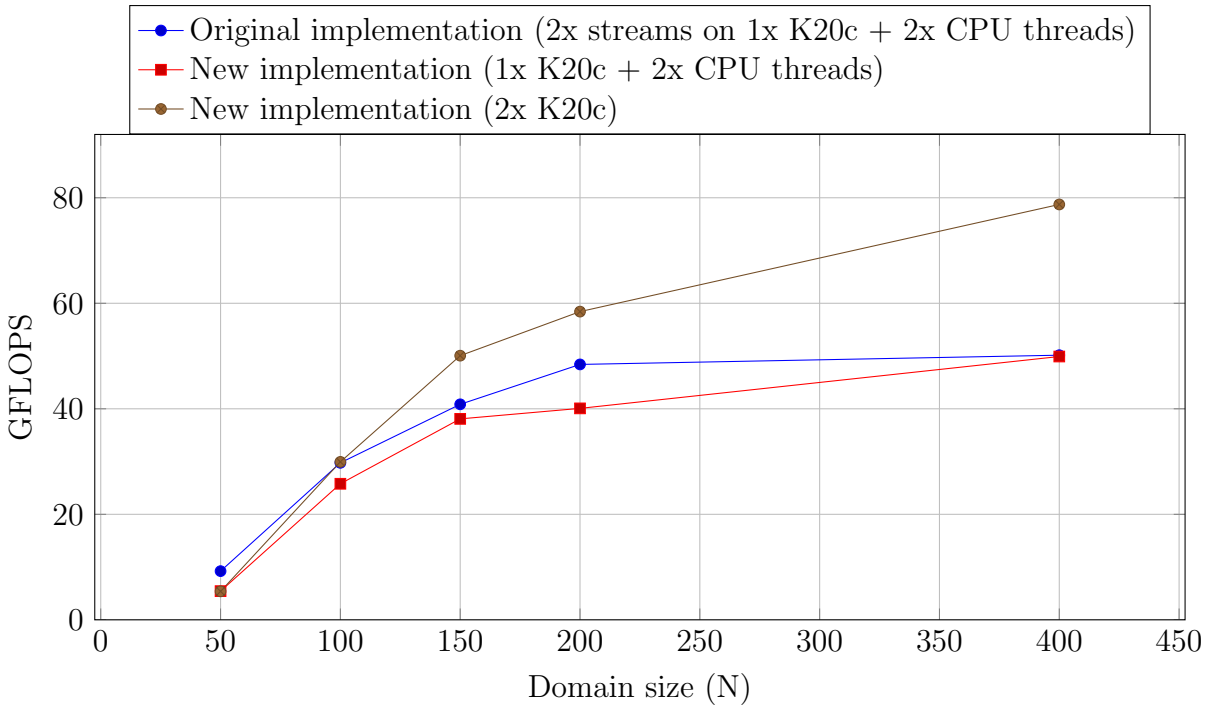

Figure 4.9: Comparison of overall performance of 20 FDTD simulations on System 1

new implementation shows a notable increase in performance compared to the original, although the original implementation still performs slightly better on very small domains (N=50).

4.7 Fermi versus Kepler

Skomedal provides a performance comparison between the Kepler and the Fermi architecture as a part of his thesis [1]. His results shows a 5-10% decrease in performance on the NVIDIA Tesla K20c Kepler GPU compared to the NVIDIA GTX 480 Fermi GPU, even though the memory speed on K20c is 17.5% faster and the compute power is almost triple. He suggests that this is due to differences in the architectures, without mentioning anything specific.

The Fermi and Kepler architectures are explained in Section 2.2.3 and 2.2.3, respectably. One of the differences is how the L1 cache is utilized. The use of L1 cache can be disabled during compiling by using the compile flag: *-Xptxas -dlcm=cg*, and Figure 4.10 shows how this affects the results. Note that because of the limited amount of usable memory on the GTX 470, a domain size of N=330 is used instead of N=400.

Table 4.2 contains the average of the memory and cache usage of the two main kernels: *UpdateH_cuda* and *UpdateE_cuda*. The data is gathered using the NVIDIA Visual Profiler, and shows that the Fermi architecture is making good use of the L1 cache. It also suggests that the K20c's larger and faster L2 cache is an essential part of the reason to why it performs better than the GTX 470 without L1 cache.

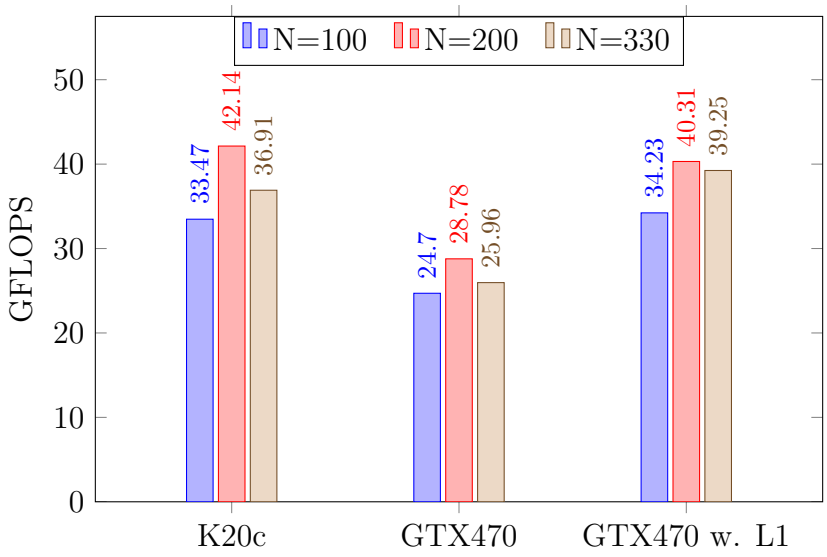

Figure 4.10: Performance impact of L1 cache

Table 4.2: Combined memory and cache usage for N=200

	K20c	**GTX470**	**GTX470 w. L1**
L1 Cache			
Hit Rate	0%	0%	74.8%
Reads	195.215 GB/s	133.282 GB/s	302.935 GB/s
Writes	36.035 GB/s	24.595 GB/s	35.915 GB/s
Total	231.251 GB/s	157.875 GB/s	338.852 GB/s
L2 Cache			
Hit Rate	65.1%	55.9%	10.15%
Reads	195.215 GB/s	133.282 GB/s	77.082 GB/s
Writes	36.035 GB/s	24.595 GB/s	35.915 GB/s
Total	231.251 GB/s	157.875 GB/s	112.995 GB/s
Device Memory			
Reads	68.115 GB/s	59.591 GB/s	69.89 GB/s
Writes	33.542 GB/s	22.785 GB/s	33.088 GB/s
Total	101.655 GB/s	82.375 GB/s	102.980 GB/s

4.8 Multi-node execution

Figure 4.11 shows the measured performance of different configurations when executed on both System 1 and 2. The systems were connected using a regular 100 Mbit/s Ethernet, and the work was scheduled evenly between the two systems. Although the FDTD simulations produced the correct outputs, the performance is shown to be severely limited by the interconnect between the systems.

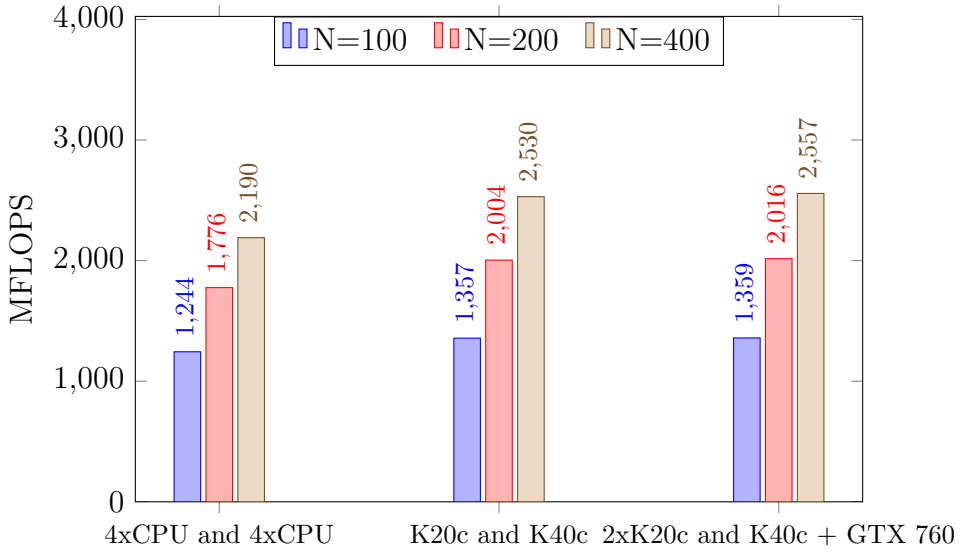

Figure 4.11: Performance of collaborate execution on both System 1 and 2

Chapter 5

Discussion

This chapter contains discussions regarding the results presented in Chapter 4. This includes possible explanations to the measured results, in addition to potential ways to improve the implementation related to these results. The chapter also presents some of the implementation's known limitations.

5.1 CPU performance

Figure 4.1 and 4.2 indicates that there is little to gain from using more than two cores, and similar results have been reported when executing the original implementation [1]. It has also been proven that memory bandwidth is limiting the benefit of using more than two cores. This is done by reducing the clock frequency of the cores to half, followed by running additional tests, which have shown a notable speed-up from using both three and four cores.

The memory bandwidth bottleneck is the main reason to why only two threads are used in CPU+GPU(s) configurations. Other reasons includes: Setting aside more resources to border exchanges and background processes, and being comparable to results from the original implementation [1]. It has also been suggested that only using two threads is more energy efficient [1].

5.2 Border exchange overhead

The advantage of the new implementation its ability to have multiple compute units cooperate on a single FDTD simulation. Compute units with dedicated memory (GPUs) will have to send and receive borders during every time-step, which will obviously introduce an overhead. The results presented in Section 4.3 shows the overhead when enforcing border exchanges during a single-GPU execution.

The results in Figure 4.3 shows that the impact of the overhead becomes smaller when using larger domains. This is most likely due to the number of computations becoming much larger as the size of the domain grows, compared to the size of the borders. The size of the borders grows with N^2 while the number of computations grows with N^3, when using grid decomposition and domains of size N^3.

5.2.1 Ways to reduce the overhead

Several potential techniques that can reduce the overhead from border exchanges are discussed in this section. The techniques are described in relation to intra-communication and CUDA GPUs, but most of them also applies to inter-communication and multi-node execution.

Faster interconnect

The average transfer throughput of a 160 KB border was measured to be 3.17 GB/s with PCI-Express (PCIe) 2.0 x8, and 5.95 GB/s using PCIe 2.0 x16. Even though these measurements were done on different systems using different GPUs, they indicate that the throughput is limited by the speed of the interconnect. The speed of the PCIe 2.0 bus is specified to be 500 MB/s (5 GT/s) per lane, in each direction. The maximum peak throughput of PCIe x8 is therefore 3.91 GB/s, and 7.81 GB/s for PCIe x16. It should be noted that these values are the theoretical peak bandwidth, and even though the measured speeds are notable slower, they are still within what can be expected. The overhead of border exchanges can potentially be reduced by using a faster interconnect, such as PCIe 3.0 x16 with a theoretical peak of 15.39 GB/s per direction. A faster interconnect would also improve the initial transfer of the sub-domain onto device memory and the transfer of the completed sub-domain back to system memory.

Concurrent copy and execution

The border exchange overhead could be significantly reduced by concurrently updating the electromagnetic field values and copying the borders. All CUDA GPUs with compute capability 1.1 or higher supports this through the use of multiple streams. Every update of either the electrical or the magnetic field values requires one border to be received before updating and another to be sent after updating. The updating of field values which depends on borders to be received can be postponed until the end of the kernel. Correspondingly, the updating of field values that needs to be sent after updating can be done first. This way, sending, receiving and computations can be done concurrently.

The order of the memory reads caused by the discussed approach might have a negative impact on cache utilization. This hit can be avoided by explicit cacheing using shared memory.

Page-locked memory

A known technique to improve transfer speeds between system and device memory is to use page-locked (pinned) memory, which enables the GPU to request transfers to and from the system memory without involving the CPU. However, the results in Table 5.1 from the project: *The Evolution and Current State of CUDA GPGPU* [8], suggests that there is nothing to gain from using pinned memory with small transfers.

Box decomposition

As discussed in Section 3.4, the implementation uses strip decomposition to create sub-domains. Another and arguably more complex approach is to use box decomposition, where a domain is divided along all the axis as opposed to only one. This will result in more borders and exchanges, but the size of the borders will become smaller as the number

Table 5.1: Pinned memory performance

Host Memory	Unpinned	Pinned	Unpinned	Pinned
Type	Host to Device	Host to Device	Host to Device	Host to Device
Duration	176.402 us	176.753 us	3.528 ms	2.79 ms
Size	1 MB	1 MB	16 MB	16 MB
Throughput	5.54 GB/s	5.53 GB/s	4.43 GB/s	5.6 GB/s
Host Memory	Unpinned	Pinned	Unpinned	Pinned
Type	Device to Host	Device to Host	Device to Host	Device to Host
Duration	167.057 us	167.088 us	11.838 ms	2.639 ms
Size	1 MB	1 MB	16 MB	16 MB
Throughput	5.85 GB/s	5.84 GB/s	1.32 GB/s	5.92 GB/s

of sub-domains increases. In comparison, strip decomposition has the limitation that the border sizes remains constant regardless of the number sub-domains. This might not be significant in regards to the domain sizes used in this thesis, but it might be a necessity when using very large domains executing on many nodes with multiple execution units.

5.3 Load distribution

The results in Section 4.4 shows that the load distribution derived from the micro-benchmark is sub-optimal for domain sizes other than N=100. This makes sense, since the micro-benchmark itself works by executing a temporal domain of size N=100 on each of the workers. Adapting a larger size for domains used in the micro-benchmark might give better results. Alternatively, the best load balance for different sizes can be calibrated prior to the actual execution, by letting each of the workers execute a multitude of domains of different sizes. The results can then be stored in a file for later use, thereby making it a one time operation.

According to the results for domains of size N=400 in Figure 4.4 and 4.5, the optimal balance between CPU and GPU is 10% and 90%, respectively. This balance is similar to the balance suggested in the article: *GPU-Accelerated Parallel FDTD on Distributed Heterogeneous Platform* [7], where a problem of size $512 \times 256 \times 256$ is executed on a NVIDIA Tesla K20m GPU and two threads on a Intel XEON E5-2670 CPU.

The results in Figure 4.6 shows that there is little or nothing to gain from using the CPU together with two K20c GPUs. In addition to the GPUs being very fast compared to the CPU, the CPU and system memory are already under load due to the required border exchanges. In conclusion, it might be better to omit the use of the CPU when multiple fast GPUs are already in use on the same system.

Except for the obvious benefit of spreading the computational load among multiple execution units, there are other benefits to load distribution. In the first place, using multiple execution units will allow for larger domain sizes if the units have dedicated memory. Secondly, using multiple GPUs can decrease the overhead of transferring the data to and from device memory, since the domain is divided and transferred to or from both GPUs in parallel. This also somewhat applies to CPU+GPU configurations, since the amount of data that is transferred to and from the GPU is reduced. On the other hand, using multiple execution units also adds the overhead of border exchanges.

5.4 Performance

The results presented in Figure 4.5 shows how the performance varies on different configurations of CPUs, GPUs and domain sizes. The results indicates that the overall performance of executions involving GPU(s) increases as the domain gets larger. This can be explained by the fact that in order for CUDA GPUs to perform optimally, they require a lot of threads, which in turn allows for warp scheduling (see Section 2.2.1). Having enough warps to schedule between effectively hides the latency of memory reads, thereby improving the overall performance.

Another observation is how the benefit of using multiple execution units increases as the domain becomes larger, this applies to both GPU+CPU and multi-GPU configurations. There are multiple possible explanations to this: Firstly, this can also be related to larger domains offering better warp scheduling. Secondly, the overhead of transferring to and from GPU(s) is reduced, because some of the domain remains in system memory in a GPU+CPU configuration, or the domain is divided and transferred simultaneously in a multi-GPU configuration. Thirdly, as discussed in Section 5.2, the overhead of the border exchanges grows slower than the amount of computations, in relation to the domain size. This can also explain why GPU+CPU configurations performs worse on small domains, compared to only executing on the GPU.

5.5 Comparison to original implementation

The new implementation differs from the original implementation [1], in that it focuses on decomposing large domains and executing them on multiple execution units and nodes. On the other hand, the original implementation focuses on scheduling many small individual domains on one multi-threaded CPU and one GPU. Even though the new implementation allows for larger domain sizes and faster execution of large domains, it also introduces an overhead due to the border exchanges.

The results in Figure 4.9 indicates that the original scheduling implementation is slightly better at dealing with many small domains compared to the new decomposition implementation. One reason to this is the added overhead from border exchanges, another is the use of multiple streams in the original implementation. Multiple streams are used to execute several domains simultaneously, thereby overlapping transfers and executions on the GPU, and improve performance. Although this feature is possible in the new decomposition implementation, it is not implemented. The main reason to this is that the new implementation is focused on executing large domains, and taking advantage of multiple streams in the same way as the scheduling implementation would require two (or more) domains to reside in device memory simultaneously. Another way to have concurrent transfers and execution in the new implementation is already discussed in Section 5.2.1.

The new implementation does surpass the original when dealing with large domains, primary due to its ability to use several execution units. Domains of size N=400 performs almost identical when only using the CPU and one GPU. Using both the installed GPUs improves the performance by 22.57% on N=200 sized domains, and 55.56% on N=400. Using more than two GPUs is also possible, as well as larger domains.

Why the decomposition approach was chosen instead of extending the scheduling implementation to also work on multiple GPUs and nodes is already explained in Section

2.6.3 and 3.1.

5.6 Fermi versus Kepler

Compared to the GTX 470's performance in Figure 4.10, the Tesla K20c shows a 6.34% decrease in performance when executing domains of size N=330 using the new implementation. The K20c performs 4.54% better on domains of size N=200m, and slightly worse on N=100. These results are similar to the 5-10% performance decrease reported using the original implementation [1], considering those results compared the K20c to the GTX 480, which is slightly more powerful than the GTX 470.

As discussed in Section 2.2.3, the Fermi architecture implicitly uses the L1 cache for all global memory accesses, similar to how the L2 cache works. This is different in the Kepler architecture, where the L1 cache is dedicated to local memory accesses only. Since there is no register spilling in the new implementation, the L1 cache remains unused on the Kepler architecture. Disabling the L1 cache on the GTX 470 leads to a significant reduction in performance, indicating that the performance is memory bound. Looking at Table 4.2 suggests that more L2 cache hits and higher L2 bandwidth are the reasons to why the Kepler architecture performs better than Fermi without L1 cache. This is reasonable, since the Kepler architecture has twice the amount of L2 cache compared to the Fermi architecture.

The results shows that due to how the L1 cache is utilized, the GTX 470 can perform similar to the much more powerful Tesla K20c. This suggests that the performance of the K20c is severely limited by memory bandwidth, and can be expected to improve significantly with the use of shared memory and explicit caching. Shared memory is neither used in the new or the original implementation, but should be considered as a future improvement. The article: *CUDA Based FDTD Implementation* [6], presents a technique to eliminate uncoalesced memory accesses by using shared memory.

5.7 Multi-node execution

One of the goals of this thesis has been to develop an implementation that can utilize multiple nodes. This has been done, and also tested to work on two desktop computers with conventional hardware and a regular 100 Mbit/s Ethernet connection. However, Figure 4.11 shows that the performance results from these tests are inferior compared to executing on a single system. It's reasonable to assume that the interconnect between the systems is limiting the performance, because the results show a very minor improvement in performance when using more compute power.

High Performance Computers usually connects the nodes using networks faster than conventional 100 Mbit/s Ethernet, such as 10 Gigabit Ethernet (10GigE) or InfiniBand. 10GigE can provide a bandwidth of 10 Gbit/s, while InfiniBand can provide bandwidths up to 300 Gbit/s, depending on the data rate and number of lanes used. In comparison, the PCI-Express 2.0 x8 used to connect the GPUs in the test systems has a bandwidth of 40 Gbit/s. Some of the techniques discussed in Section 5.2.1 can also be used to reduce the overhead of border exchanges between nodes in a multi-node system, specifically the overlapping of transfers and computations, and the use of box decomposition for large domains. These techniques, combined with a faster network connection, can make the

performance improvements from using multiple nodes much more compelling than the results shown in Figure 4.11.

5.8 Limitations

There are various limitations to the current implementation: Most apparent is the fact that it is only a benchmarking code, and not yet usable for real world simulations. One feature that would have to be added for it to be more relevant for real world problems is the ability to use heterogeneous materials. Currently, the implementation uses a homogeneous material, specifically the material constants of empty space. Using a heterogeneous materials is possible by assigning material constants to each of the Yee cells. A memory efficient way to do this is through the use of a lookup table, and only storing an index with each of the Yee cells. This table of material constants can be stored in the GPUs' constant memory, which is also suggested in the articles: *CUDA Based FDTD Implementation* [6], and *GPU-Accelerated Parallel FDTD on Distributed Heterogeneous Platform* [7]. Another thing lacking in the implementation, related to real world usefulness, is some sort of visual feedback of the simulation.

There is currently a known limitation to the size of the executable domains, outside of the available memory. Most of the code that deals with dividing up the domains or transmitting sub-domains stores the offsets and number of bytes in 32-bit signed integers. Bytes are used because it enables the implementation to seamlessly work with electromagnetic field values of both single and double precision. However, very large problems might face issues due to integer overflow(s): A 32-bit signed integer can store a number of bytes equaling 2 GB. Even though the implementation will work with domain sizes far larger than 2 GB, the possibility of integer overflow should be considered. Hence, the implementation might need some adjustments if it shall be used for very large domains.

Chapter 6

Conclusion

The goal of this thesis has been to extend Skomedal's original FDTD implementation [1] into also utilizing multiple nodes and GPUs. The original implementation was able to utilize both the CPU and a CUDA GPU on a single system, by scheduling small individual FDTD domains to each of them. This scheduling implementation was first extended into also using multiple GPUs and nodes. However, that approach was abounded in favor of a decomposition approach, since it would allow for better scaling across multiple nodes and execution units.

The new implementation can be configured to utilize several CUDA GPUs within a single system, and is only limited by the number of GPUs supported by the system. It divides the FDTD domains into several sub-domains according to the number of nodes and execution units. This allows for much larger domains than the original scheduling approach, because it required the entire domain to fit in the memory of a single GPU. The new implementation also improves performance by using multiple GPUs: Executing a set of 20 domains of size 400^3 has been proven to improve performance by 55.56% compared to the original implementation running on the same system.

Another major feature of the work done in this thesis is the ability to use multiple nodes, and thereby allow for a combined computing power beyond what is possible on a single system. However, due to network limitations, the results provided in this thesis has not been able to prove this improvement.

Because of the many differences, the new implementation uses very little of the original code, which was written in the C programming language. The new implementation has instead been written in the C++ language, which has allowed for an understandable, modular and easily extendable code. Most noteworthy is the possibility to add new types of workers (based on APIs such as OpenCL or OpenACC), without significant alterations to the excising code.

The contribution from this project is a working implementation of the FDTD algorithm, that can be executed on a cluster of heterogeneous systems with a multi-core CPU, and one or several CUDA-capable GPUs. Due to reasons already explained in Section 5.8, this implementation is not sufficient for real world FDTD simulations. It is instead meant to be used as a basis for future work, or as an example on how to do FDTD on a cluster of heterogeneous multi-GPU systems.

6.1 Recommendations for future work

Numerous ideas for potential improvements and features have surfaced during the work on this thesis. Not all of these have been included as part of the work, either in the respect of the limited time or because they have been outside the scope of the focus area. Following is a selection of some of these ideas recommended for future work.

6.1.1 Extension to a full FDTD implementation

As discussed in Section 5.8, the implementation is currently limited to only being a benchmarking code for simulating electromagnetic fields using the FDTD method. Multiple features will have to be added to make it useful for real world simulations, such as the abilities to use heterogeneous materials and to get visual feedback. Users who frequently uses the FDTD method for simulating electromagnetic fields should be consulted, in order to get a better understanding of the needs and requirements of real world use cases.

6.1.2 Overlapping exchanges and computations

The overhead from doing border exchanges between multiple execution units and nodes can be critical when using slow interconnects. Techniques to reduce this overhead is discussed in Section 5.2.1. A particular interesting technique is to overlap transfers and computation, since it can be applied to both the intra-communication between workers and the inter-communication between nodes. This will require the use of multiple CUDA streams and non-blocking message passing routines.

6.1.3 Optimize for Kepler

The original implementation [1] which this work is based upon is developed and optimized with the NVIDIA Fermi architecture in mind. The newer Kepler architecture is capable of much better performance, but also differs from Fermi in many ways. The code will therefore have to be adjusted according to these differences. Optimizing for the Kepler architecture can be done by adjusting the current GPU worker, although it might be more convenient to add a new Kepler tailored worker. Following is a couple of suggestions to optimizations that might improve performance on the Kepler architecture.

Shared memory

The Kepler architecture does not implicitly use the L1 cache in the same manner as Fermi, it is instead required to be explicitly used through shared memory. The results in Section 4.7 demonstrates the significance of memory bandwidth and cache usage, which also suggests that the use of shared memory will lead to a commendable improvement in performance.

Dynamic parallelism

Dynamic parallelism is a new feature introduced in the Kepler architecture which lets a kernel launch new kernels. This is a technique that might be beneficial in this implementation, specially in the updating of electrical field values, since it requires multiple kernels to be launched consecutively.

6.1.4 Develop new workers

The code is designed to be easily extendable by adding new workers. A possible future work can be the development and incorporation of new workers outside the currently implemented CPU and CUDA GPU workers. This can for example be workers based on OpenCL, OpenACC or DirectCompute, which can allow for the code to execute on a wider range of hardware and software configurations.

6.1.5 Combining of scheduling and decomposition

The results in Section 4.6 suggests that it might be better to schedule individual FDTD domains when dealing with many small domains. If this is an important and reoccurring use case, a potential extension would be to combine the scheduling and decomposition approach into a single program.

Bibliography

[1] Andreas Berg Skomedal. Heterogeneous FTDT for Seismic Processing. Master's thesis, Norwegian University of Science and Technology, June 2013. URL http://www.diva-portal.org/smash/get/diva2:655628/FULLTEXT01.pdf.

[2] Ulf Andersson. Yee_bench - A PDC benchmark code. November 2002. URL https://www.pdc.kth.se/research/publications/pdc-technical-report-series/trita-repository/PDC_TRITA_2002_1.pdf.

[3] Kane Yee. Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media. *IEEE Transactions on Antennas and Propagation*, 14(3):302–307, May 1966. URL http://ecee.colorado.edu/~mcleod/teaching/nmip/references/Numerical%20Solution%20of%20Initial%20Value%20Problems%20of%20Maxwells%20Equations%20Yee%201966.pdf.

[4] Dan Yangl, Jie Xiong, Cheng Liao, and Lang Jen. A Parallel FDTD Algorithm Based on Domain Decomposition Method Using the MPI Library. In *Parallel and Distributed Computing, Applications and Technologies*, pages 730–733. IEEE, August 2003. URL http://www.math.zju.edu.cn/xlhu/teaching/ComputEM/FDTD/ParallelFDTD-MPI.pdf.

[5] X.-M. Guo1, Q.-X. Guo, W. Zhao, and W.-H. Yu. Parallel FDTD Simulation Using NUMA Acceleration Technique. *Progress In Electromagnetics Research Letters*, 28: 1–8, November 2011. URL http://www.jpier.org/PIERL/pierl28/01.11101706.pdf.

[6] Veysel Demir and Atef Z. Elsherbeni. Compute Unified Device Architecture (CUDA) Based Finite-Difference Time-Domain (FDTD) Implementation. *The Applied Computational Electromagnetics Society*, 25(4):303–314, April 2010. URL http://www.engineering.olemiss.edu/~atef/pdfs/journal_papers/2010/CUDA_Based_FDTD_Implementation.pdf.

[7] Ronglin Jiang, Shugang Jiang, Yu Zhang, Ying Xu, Lei Xu, and Dandan Zhang. GPU-Accelerated Parallel FDTD on Distributed Heterogeneous Platform. *International Journal of Antennas and Propagation*, February 2014. URL http://www.hindawi.com/journals/ijap/2014/321081/.

[8] Eirik Myklebost. The Evolution and Current State of CUDA GPGPU. January 2014.

[9] Russel Fish. The Future of Computers, November 2011. URL http://www.edn.com/design/systems-design/4368705/The-future-of-computers--Part-1-Multicore-and-the-Memory-Wall.

[10] *Design Guide: CUDA C Programming Guide*. NVIDIA, July 2013. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[11] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, 2010.

[12] Leif Christian Larsen. Utilizing GPUs on Cluster Computers. December 2006. URL http://www.idi.ntnu.no/~elster/master-studs/leifchl/report/report.pdf.

[13] *Reference Guide: CUDA Compiler Driver NVCC*. NVIDIA, July 2013. URL http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.

[14] *Application Guide: Parallel Thread Execution ISA*. NVIDIA, July 2013. URL http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf.

[15] *Design Guide: CUDA C Best Practices Guide*. NVIDIA, July 2013. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[16] Fermi Whitepaper. Technical report, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[17] NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, November 2006. URL http://www.cse.ohio-state.edu/~agrawal/788-su08/Papers/week2/GPU.pdf.

[18] NVIDIA GeForce GTX 200 GPU Architectural Overview. Technical report, May 2008. URL http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.

[19] Kepler GK110 Whitepaper. Technical report, 2012. URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[20] NVIDIA GeForce GTX 680 Whitepaper. Technical report, 2012. URL http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[21] Peter Moczo, Johan O. A. Robertsson, and Leo Eisner. The Finite-Difference Time-Domain Method for Modeling of Seismic Wave Propagation. *Advances in Geophysics*, 48:421–516, 2007. URL ftp://158.195.84.50/pub/Papers/Moczo_etal_AG_2007.pdf.

[22] Andreas Berg Skomedal. GPU-Accelerated Seismology Using CUDA. 2012.

[23] Ryan Smith. NVIDIA's GeForce GTX 580: Fermi Refined, November 2010. URL http://www.anandtech.com/show/4008/nvidias-geforce-gtx-580/2.

Appendix A

Comparison of CUDA Supported Architectures

Included in this Appendix is a table containing an extensive comparison between the CUDA supported architectures and the generations within each major architecture. All the specifications, except from Double-Precision (DP) units and DP GFLOPS, are from the fastest GTX GPUs within each generation. While the number of DP units and DP GFLOPS are from the fastest Tesla GPUs within each generation.

The following GPUs have been used as references:

- **G80:** 8800 GTX
- **G92b:** 9800 GTX+
- **GT200:** 280 GTX and Tesla C1060
- **GF100:** 480 GTX and Tesla M2070
- **GF110:** 580 GTX and Tesla M2090
- **GK104:** 680 GTX and Tesla K10
- **GK110:** 780 Ti GTX and Tesla K40

Table A.1: Comparison of CUDA supported architectures [16–20, 23]

				Fermi		**Kepler**	
GPU	G80	G92b	GT200	GF100	GF110	GK104	GK110
Release date	11.2006	07.2008	06.2008	03.2010	11.2010	03.2012	02.2013
Transistors	681 million	754 million	1.4 billion	3.2 billion		3.54 billion	7.1 billion
Performance							
SP GFLOPS	518	705	1062.72	1345	1581.1	3090.4	5046
DP GFLOPS	n/a		77.76	515.2	666.1	95	1500
Compute units							
Core clock	575 Mhz	738 MHz	648 MHz	700 MHz	772 MHz	1006 MHz	928 MHz
Shader clock	1350 MHz	1836 MHz	1476 MHz	1401 MHz	1544 MHz	n/a	
SMs	16		30	15	16	8	15
CUDA cores (per SM)	128 (8)		240 (8)	480 (32)	512 (32)	1536 (192)	2880 (192)
SFUs (per SM)	8 (2)		15 (2)	60 (4)	64 (4)	256 (32)	480 (32)
DP units (per SM)	n/a		30 (1)	480 (32)	512 (32)	64 (8)	960 (64)
Load/store units (per SM)	8		10	240 (16)	256 (16)	256 (32)	480 (32)
Warp schedulers per SM	1			2		4	
Main memory							
DRAM type	GDDR3			GDDR5			
Memory clock	1800 MHz	2200 MHz	2484 MHz	3696 MHz	4008 MHz	6008 MHz	7000 MHz
Memory bus width	384-bit	256-bit	512-bit	384-bit		256-bit	384-bit
Memory bandwidth	86.4 GB/s	70.4 GB/s	159 GB/s	177.4 GB/s	192.4 GB/s	192.3 GB/s	336 GB/s
Memory size	768 MB	512 MB	1024 MB	1536 MB	3072 MB	4096 MB	6144 MB
Load/store address width	32-bit			64-bit			
ECC memory support	No			Yes			
On-chip memory and cache							
L1 cache per SM	n/a			16 KB or 48 KB		16 KB, 32 KB or 48 KB	
L2 cache	(texture only) 256 KB			768 KB		1536 KB	
Constant memory	64 KB						
Constant memory per SM	8 KB						
Shared memory per SM	16 KB			16 KB or 48 KB		16 KB, 32 KB or 48 KB	
Registers per SM	8192		16384	32768		65536	
Registers per thread	128			63			255
Texture memory per SM	8 KB			12 KB			48 KB unified
CUDA							
Compute capability	1	1.1	1.3	2		3	3.5
Max threads per block	512			1024			
Max threads per SM	768		1024	1536		2048	
Max blocks per SM	8					16	
Max Warps per SM	24		32	48		64	
Warp size	32						
Concurrent kernels	n/a			16			32
Supported technologies							
Hyper-Q	No						Yes
Dynamic parallelism	No						Yes
Unified memory access	No			Yes			

Appendix B

User Manual

This appendix will explain how to execute the code associated with this thesis, and the required hardware and software configurations.

Compilation

A *makefile* has been created and included to simplify the compilation and execution of the code. The makefile provides four make targets:

- **all:** Builds the program without executing it.
- **clean:** Removes the following files (if they exist): The executable, the profile output file, and all object files.
- **run:** Executes the program. Will also build or re-build the executable before execution, if necessary.
- **mpirun:** Multi-node execution of the program using MPI. Launches the program as a MPI process on each of the hosts listed in the *host_file* file. Will also build or re-build the executable before execution, if necessary.
- **profile:** Uses the GNU GCC profiling tool (gprof) to produce profiling data while executing the program. The data is outputted to *profile.txt*. Will also build or re-build the executable before execution, if necessary.

Compiling the program requires an installation of the GNU Compiler Collection (GCC), Open MPI, and the NVIDIA CUDA Compiler (NVCC). Testing and development has been done on a platform with Ubuntu version 12.04 64-bit, using GCC version 4.6.3, OpenMPI version 1.4.3, and the CUDA toolkit version 6.0. Older or newer versions might also work, as well as other implementations of MPI, such as MPICH. Table B.1 contains a complete list of all the libraries linked in a working executable.

The implementation can be toggled to use either single or double-precision. The default behavior is to use single-precision, and is controlled by the compile flag: *use_float*, which is defined in the *globals.h* source file. Double-precision can be toggled by removing the definition of *use_float*, the entire program also has to be re-compiled.

Table B.1: Linked libraries

Libraries	Definitions
linux-vdso.so.1	Virtual dynamic shared object
libpthread.so.0	POSIX threads library
libgomp.so.1	GNU Open MP library
libcudart.so.6.0	CUDA runtime library
libmpi_cxx.so.0	Part of Open MPI
libmpi.so.0	Part of Open MPI
libstdc++.so.6	The GNU standard C++ library
libm.so.6	C math library
libgcc_s.so.1	GCC low-level runtime library
libc.so.6	Standard C library
ld-linux-x86-64.so.2	Common X extensions library
librt.so.1	POSIX.1b realtime extensions library
libdl.so.2	Dynamic linking library
libopen-rte.so.0	Part of Open MPI
libopen-pal.so.0	Part of Open MPI
libutil.so.1	Utility functions from BSD systems

Execution

The program uses a command-line only interface, and is executed through the use of the makefile. It does not accept any command line arguments, all user inputs are instead entered in the *config* file. The config file is loaded by the program, and specifies both the system and FDTD task configurations. System configurations consists of the amount of information output and the number of execution units (CPU and GPUs). Editing the FDTD task configurations can change the number of FDTD tasks, and the size and time-steps of the FDTD domains.

The *host_file* file is loaded during multi-node runs, and must contain the IP-addresses or names of all the nodes that will run the program. Each node also needs a copy of the executable, which must be located in a directory with the same path on every node. Further explanation on how to run MPI jobs can be found at:
http://www.open-mpi.org/faq/?category=running.

The program can be executed on a system without a CUDA capable GPU by setting *maxgpus* in the *config* file to zero. The CUDA toolkit is however still required.

Comments on load distribution

The program uses a micro-benchmark to measure the performance of each execution unit (worker), and uses this value to divide and distribute the FDTD domains. The thesis includes results from executions with load distributions other than those suggested by the micro-benchmark. Doing so requires changes in the *Node.cpp* and *WorkerPool.cpp* source files: First, the micro-benchmark will have to be disabled by commenting out (or removing) `<m_workers.MeasurePerformance();>`, at line 28 in *Node.cpp*. Secondly, the load distributions will have to be manually set. This is done by inserting:

<m_collective_performance = 100;>, at line 36. This line must then be followed by multiple insertions of the line: <m_workers[INDEX]->m_performance = PERCENTAGE;>, depending on the number of workers to be used, and where *INDEX* and *PERCENTAGE* are specified by the user: *INDEX* is the workers index and *PERCENTAGE* is the worker's load percentage. The sum of the load percentages must be 100. Listing B.1 contains an example of a custom load distribution among a CPU and two GPUs.

Listing B.1: Custom load distrubtion example

```
33      ...
34      const int subdomain_x_size = to - from;
35      int remaining_x;
36
37      m_collective_performance = 100;
38      m_workers[0]->m_performance = 10; // CPU
39      m_workers[1]->m_performance = 45; // GPU0
40      m_workers[2]->m_performance = 45; // GPU1
41
42      if (m_collective_performance == 0)
43      {
44      ...
```

Appendix C

Program Documentation

This appendix contains a documentation of all the classes, functions, constants and variables in the program. The documentation is generated using Doxygen version 1.8.6.

Contents

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

4 Class Documentation

4.1 CpuWorker Class Reference

A Worker sub-class specialized for doing FDTD on a multi-core CPU.

`#include <CpuWorker.h>`

Inherits Worker.

Public Member Functions

- CpuWorker (const int number_of_threads=1)

  Constructor.

Private Member Functions

- void UpdateH_Internal ()

  Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- void UpdateE_Internal ()

  Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- void SourceExcitationInternal ()

  Updates the source point (m_i_source) in the FDTD domain of m_task.

Private Attributes

- const int m_number_of_threads

  The number of OpenMP threads used during executions.

Additional Inherited Members

4.1.1 Detailed Description

A Worker sub-class specialized for doing FDTD on a multi-core CPU.

A sub-class of the abstract Worker class. Contains functions for updating the electromagnetic field values using a multi-core CPU.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 CpuWorker::CpuWorker (const int *number_of_threads* = `1`)

Constructor.

Will create a CpuWorker instance using the specified number of OpenMP threads.

Parameters

in	*number_of_threads*	An optional parameter which specifies the number of OpenMP threads to use.

4.1.3 Member Function Documentation

4.1.3.1 void CpuWorker::SourceExcitationInternal ( ) [private], [virtual]

Updates the source point (m_i_source) in the FDTD domain of m_task.

This function will update the source point of the domain. If this function is called it is already assumed that the worker is responsible for the sub-domain containing the source point.

Implements Worker.

4.1.3.2 void CpuWorker::UpdateE_Internal ( ) [private], [virtual]

Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This function will update the electric field values of the sub-domain associated with the CpuWorker. The updating is accelerated on multi-core systems by using OpenMP.

Implements Worker.

4.1.3.3 void CpuWorker::UpdateH_Internal ( ) [private], [virtual]

Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This function will update the magnetic field values of the sub-domain associated with the CpuWorker. The updating is accelerated on multi-core systems by using OpenMP.

Implements Worker.

The documentation for this class was generated from the following files:

- src/Workers/CpuWorker.h
- src/Workers/CpuWorker.cpp

4.2 GpuWorker Class Reference

A Worker sub-class specialized for doing FDTD on a CUDA-capable GPU.

`#include <GpuWorker.h>`

Inherits Worker.

Public Member Functions

- GpuWorker ()

  Constructor.

Static Public Member Functions

- static int GetNumberOfDevices ()

  Returns the number of avaliable CUDA-capable GPUs.

Private Member Functions

- void PrepareForNewTaskInternal ()

  Preparations for a new Task, executed by the internal thread.
- void UpdateH_Internal ()

  Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- void UpdateE_Internal ()

  Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- void SourceExcitationInternal ()

  Updates the source point (m_i_source) in the FDTD domain of m_task.
- void FinalizeTaskInternal ()

  Finalization of the Task done by the internal thread.

Private Attributes

- const int m_device_number

  The device number of the GPU associated with this GpuWorker.
- int m_block_size

  The CUDA block size used during kernel calls.
- my_float ∗ **device_hx**
- my_float ∗ **device_hy**
- my_float ∗ **device_hz**
- my_float ∗ **device_ex**
- my_float ∗ **device_ey**
- my_float ∗ **device_ez**

Static Private Attributes

- static int s_number_of_devices

  The number of avaliable CUDA-capable GPUs.
- static int s_next_device_number

  The number of the next unused device.

Additional Inherited Members

4.2.1 Detailed Description

A Worker sub-class specialized for doing FDTD on a CUDA-capable GPU.

A sub-class of the abstract Worker class which contains functions for updating the electromagnetic field values using a CUDA-capable GPU.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 GpuWorker::GpuWorker ( )

Constructor.

Will set m_device_number and m_block_size when called, and increment s_next_device_number.

4.2.3 Member Function Documentation

4.2.3.1 void GpuWorker::FinalizeTaskInternal ( ) [private], [virtual]

Finalization of the Task done by the internal thread.

This function will copy all the electromagnetic field values of the sub-domain to system memory, and then remove (free) them from device memory.

Reimplemented from Worker.

4.2.3.2 static int GpuWorker::GetNumberOfDevices ( ) [static]

Returns the number of avaliable CUDA-capable GPUs.

Will update s_number_of_devices the first time it's called.

4.2.3.3 void GpuWorker::PrepareForNewTaskInternal ( ) [private], [virtual]

Preparations for a new Task, executed by the internal thread.

This function will allocate the necessary memory on the GPU, before copying the electromagnetic field values into the GPU's device memory.

Reimplemented from Worker.

4.2.3.4 void GpuWorker::SourceExcitationInternal ( ) [private], [virtual]

Updates the source point (m_i_source) in the FDTD domain of m_task.

This function will call one CUDA kernel using a single CUDA core to update the source point of the domain. If this function is called it is already assumed that the worker is responsible for the sub-domain containing the source point.

Implements Worker.

4.2.3.5 void GpuWorker::UpdateE_Internal ( ) [private],[virtual]

Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This function will copy the lower magnetic field borders into device memory, before updating the electric field values by calling four CUDA kernels. The lower electric field borders will afterwards be copied to the system memory. Border exchanges will not be done if only a single worker is used, or if the worker is responsible for the bottom sub-domain (in relation to how the domain is divided among nodes and workers).

Implements Worker.

4.2.3.6 void GpuWorker::UpdateH_Internal ( ) [private],[virtual]

Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This function will copy the upper electrical field borders into device memory, before updating the magnetic field values by calling a CUDA kernel. The upper magnetic field borders will afterwards be copied to the system memory. Border exchanges will not be done if only a single worker is used.

Implements Worker.

The documentation for this class was generated from the following file:

- src/Workers/GpuWorker.h

4.3 Node Class Reference

An all static class that provides all the MPI functionality of the program.

`#include <Node.h>`

Public Member Functions

- Node (const int num_cpus, const int num_omps, int max_gpus)

  Constructor which also creates workers according to the parameters.

Static Public Member Functions

- static void ExecuteTask (Task *task)

  Executes a Task using all avaliable Nodes and Workers.

- static void RunAsSlave ()

  Puts the Node in a slave state where it will wait for work from the host node.

- static bool BroadcastEndSignal (bool end=true)

  Signals all the slave nodes to exit their slave state.

Static Public Attributes

- static int mpi_rank

  The MPI rank of the node. Is used to differentiate the nodes from each other. Rank 0 represents the host node.
- static int mpi_size

  The total number of MPI processes in the communicator. Also referred to as the total number of nodes.

Private Types

- enum MyMpiTag { **FIELD_DATA_TAG**, **FIELD_BORDER_TAG** }

  Tags used to differentiate the MPI messages. Is not really needed, but included for readability.

Static Private Member Functions

- static void CreateDatatypeFromTask (const int slave_rank, const Task &task, MPI_Datatype ∗datatype)

  Creates and commits a MPI_Datatype for the electromagnetic field values of a Task object.
- static void CreateH_BordersDatatypes (const int local_x_size, const Task &task, MPI_Datatype ∗up_h_datatype, MPI_Datatype ∗down_h_datatype)

  Creates datatypes for the top and bottom boundary data of the magnetic field sub-domains.
- static void CreateE_BordersDatatypes (const int local_x_size, const Task &task, MPI_Datatype ∗up_e_datatype, MPI_Datatype ∗down_e_datatype)

  Creates datatypes for the top and bottom boundary data of the electric field sub-domains.
- static void ExchangeH_Borders (const MPI_Datatype up_h_datatype, const MPI_Datatype down_h_datatype, Task ∗task)

  Exchanges the borders of the magnetic field component values with adjacent nodes in the X direction.
- static void ExchangeE_Borders (const MPI_Datatype up_e_datatype, const MPI_Datatype down_e_datatype, Task ∗task)

  Exchanges the borders of the electric field component values with adjacent nodes in the X direction.

Static Private Attributes

- static WorkerPool m_workers

  Container for all the node's workers.

4.3.1 Detailed Description

An all static class that provides all the MPI functionality of the program.

This purpose of this class is to serve as an utility class, which in this case means that it will handle the top level control of the program through the use of the rest of the implemented classes. This

class will also contain all the program's functions that relies on MPI. The class is implemented as an all static class in order to stress that there should only be one of it (per node). Using an all static class is a subjective choice, as it could also have been implemented as a namespace or as a singleton class.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Node::Node (const int *num_cpus,* const int *num_omps,* int *max_gpus*)

Constructor which also creates workers according to the parameters.

This function will create Workers according to the input parameters, store the workers in m_workers, and measure their performances by calling WorkerPool::MeasurePerformance(). The function will also initialize MPI by calling MPI_Initialize, and should therefore only be called once!

Parameters

in	*num_cpus*	The number of CPU workers to create.
in	*num_omps*	The number of OpenMP threads each CPU worker will have.
in	*max_gpus*	The maximum number of GPU workers to create. Will be changed to the number of installed GPUs if a number higher than the number of installed GPUs is used.

4.3.3 Member Function Documentation

4.3.3.1 bool Node::BroadcastEndSignal (bool *end* = `true`) [static]

Signals all the slave nodes to exit their slave state.

This function uses MPI to broadcast a boolean to all the slave nodes, telling them if they should exit their slave state or continue to wait for new tasks. Calling this function is the only way (except for errors) to stop nodes executing the RunAsSlave() function. If a true value is broadcasted; all the nodes (including the host node) will call MPI_Finalize() and terminate all their workers by calling WorkerPool::KillAllWorkers().

Parameters

in	*end*	The bool value that should be broadcasted (only matters if called by the host node)

4.3.3.2 void Node::CreateDatatypeFromTask (const int *slave_rank,* const Task & *task,* MPI_Datatype * *datatype*) [static], [private]

Creates and commits a MPI_Datatype for the electromagnetic field values of a Task object.

This function should only be called by the host node. It will create a MPI derived datatype representing the electromagnetic field values that should be handled by the slave node specified by `slave_rank`. It will use mpi_size and the provided `slave_rank` argument to calculate a suitable sub-domain size for the slave node. The purpose of creating the derived datatype is to be able to combine six electromagnetic field arrays into one MPI_Send(), and to be able to later receive the completed sub-domain from the associated slave node. The function also commits the datatype, but it is left to the caller to later free it, when it is no longer needed (MPI_Type_-

free()).

Parameters

in	*slave_rank*	The rank of the slave node that the datatype will be made for.
in	*task*	The task object containing the electromagnetic field arrays that the datatype will be a associated with.
out	*datatype*	A pointer to the datatype that will contain the created and commited datatype.

4.3.3.3 void Node::CreateE_BordersDatatypes (const int *local_x_size,* const Task & *task,* MPI_Datatype * *up_e_datatype,* MPI_Datatype * *down_e_datatype*) [static], [private]

Creates datatypes for the top and bottom boundary data of the electric field sub-domains.

This function creates two MPI derived datatypes: one for the top and one for the bottom border of the sub-domain's electric field values. These datatypes are used to simplify the border exchanges between adjacent nodes, by bundling together borders of both the Y and Z directions. The datatypes are also committed, and should later be freed by the user (when no longer needed) by calling MPI_Type_free().

Parameters

in	*local_x_size*	The size in the x direction of the node's local subdomain.
in	*task*	The local copy of the task object.
out	*up_e_datatype*	Pointer to the datatype that will contain the created and committed datatype, for the uppermost (in the X direction) borders of the electric field values in the Y and Z directions.
out	*down_e_datatype*	Pointer to the datatype that will contain the created and committed datatype, for the lowest (in the X direction) borders of the electric field values in the Y and Z directions.

4.3.3.4 void Node::CreateH_BordersDatatypes (const int *local_x_size,* const Task & *task,* MPI_Datatype * *up_h_datatype,* MPI_Datatype * *down_h_datatype*) [static], [private]

Creates datatypes for the top and bottom boundary data of the magnetic field sub-domains.

This function creates two MPI derived datatypes: one for the top and one for the bottom border of the sub-domain's magnetic field values. These datatypes are used to simplify the border exchanges between adjacent nodes, by bundling together borders of both the Y and Z directions. The datatypes are also committed, and should later be freed by the user (when no longer needed) by calling MPI_Type_free().

Parameters

in	*local_x_size*	The size in the X direction of the node's local sub-domain.
in	*task*	The local copy of the task object.
out	*up_h_datatype*	A pointer to the datatype that will contain the created and committed datatype for the uppermost (in the X direction) borders of the magnetic field values in the Y and Z directions.

out	*down_h_datatype*	Pointer to the datatype that will contain the created and committed datatype for the lowest (in the X direction) borders of the magnetic field values in the Y and Z directions.

4.3.3.5 void Node::ExchangeE_Borders (const MPI_Datatype *up_e_datatype,* const MPI_Datatype *down_e_datatype,* Task * *task*) [static],[private]

Exchanges the borders of the electric field component values with adjacent nodes in the X direction.

CreateE_BordersDatatypes() should be called prior to this function, since it depends on the datatypes it creates.

Parameters

in	*up_e_datatype*	Datatype for the uppermost (in the X direction) borders of the electric field values in the Y and Z directions.
in	*down_e_datatype*	Datatype for the lowest (in the X direction) borders of the electric field values in the Y and Z directions.
in,out	*task*	The local task object, used as a base address by the datatypes.

4.3.3.6 void Node::ExchangeH_Borders (const MPI_Datatype *up_h_datatype,* const MPI_Datatype *down_h_datatype,* Task * *task*) [static],[private]

Exchanges the borders of the magnetic field component values with adjacent nodes in the X direction.

CreateH_BordersDatatypes() should be called prior to this function, since it depends on the datatypes it creates.

Parameters

in	*up_h_datatype*	Datatype for the uppermost (in the X direction) borders of the magnetic field values in the Y and Z directions.
in	*down_h_datatype*	Datatype for the lowest (in the X direction) borders of the magnetic field values in the Y and Z directions.
in,out	*task*	The local task object, used as a base address by the datatypes.

4.3.3.7 void Node::ExecuteTask (Task * *task*) [static]

Executes a Task using all avaliable Nodes and Workers.

This function will execute a Task using all the available nodes and workers, and should only be called by the host Node after it has been initialized with at least one Worker. Only the electromagnetic field values of the task object will be modified.

Parameters

in,out	*task*	A pointer to the task that will be executed.

4.3.3.8 void Node::RunAsSlave ( ) [static]

Puts the Node in a slave state where it will wait for work from the host node.

This function should be called by every node except from the host node. Calling this function puts the slave node in a wait state, which implies running a loop with an execution path similar to that of the ExecuteTask() function called by the host node. The slave node will remain in this loop, where it will either execute tasks or wait for new tasks, until the host node calls BroadcastEndSignal().

The documentation for this class was generated from the following files:

- src/Node.h
- src/Node.cpp

4.4 Task Class Reference

Container class for all the data associated with a FDTD domain.

`#include <Task.h>`

Public Member Functions

- Task (int nts, int nx, int ny, int nz, int dx, int dy, int dz)

  Constructor.
- void InitializeFieldComponents ()

  Dynamically allocates memory for the field components.
- void UpdateFieldPointers (const int local_x_size, const char *buffer)

  Updates the Task's pointers to point to the data stored in the local `buffer`.
- void CleanUp ()

  Frees the memory allocated by InitializeFieldComponents(). Must be called before ~Task().
- void PrintResults ()

  Prints the Task's performance results.

Public Attributes

- bool m_done

  A flag that indicates if the Task has been completed.
- const int m_id

  An unique ID automatically generated during object construction.
- const int m_num_time_steps

  The number of time steps.
- const int m_num_x

  The number of Yee-Cells in the X dimension.
- const int m_num_y

  The number of Yee-Cells in the Y dimension.
- const int m_num_z

  The number of Yee-Cells in the Z dimension.

- long long m_bytes

  The total number of bytes allocated for the field components. Will only be set after a call to InitializeFieldComponents().

- int m_is

  The X-coordinate of the source excitation, also known as the point source.

- int m_js

  The Y-coordinate of the source excitation, also known as the point source.

- int m_ks

  The Z-coordinate of the source excitation, also known as the point source.

- my_float m_p_source_factor

  The point source factor.

- my_float m_dt

  The time of one time-step: Δt.

- my_float m_Cbdx

  Constant: $\frac{\Delta t}{\mu_0 \Delta x}$.

- my_float m_Cbdy

  Constant: $\frac{\Delta t}{\mu_0 \Delta y}$.

- my_float m_Cbdz

  Constant: $\frac{\Delta t}{\mu_0 \Delta z}$.

- my_float m_Dbdx

  Constant: $\frac{\Delta t}{\varepsilon_0 \Delta x}$.

- my_float m_Dbdy

  Constant: $\frac{\Delta t}{\varepsilon_0 \Delta y}$.

- my_float m_Dbdz

  Constant: $\frac{\Delta t}{\varepsilon_0 \Delta z}$.

- Timer m_setupTimer

  Contains the time (in seconds) used by the program to get the Task ready for execution.

- Timer m_computeTimer

  Contains the time (in seconds) used for the required computations of the Task.

- Timer m_totalTimer

  Contains the total time (in seconds) used to complete the Task.

- double m_flops_work

  The number of floating-point operations required to complete the Task.

- double m_mflops_performance

  The number of million floating-point operations per second the Task was executed with.

- my_float $*$ m_ex

  Electrical field components in X direction.

- my_float $*$ m_ey

  Electrical field components in Y direction.

- my_float $*$ m_ez

  Electrical field components in Z direction.

- my_float * m_hx

  Magnetic field components in X direction.
- my_float * m_hy

  Magnetic field components in Y direction.
- my_float * m_hz

  Magnetic field components in Z direction.

Static Private Attributes

- static int s_next_id = 0

  A static counter used to generate the unique m_id.
- static const my_float s_const = 1.0e-10

  A constant used to set the m_p_source_factor.
- static const my_float s_mu0 = 1.256637061E-6

  Relative permeability of vacuum: μ_0. The degree of magnetization that a material obtains in response to an applied magnetic field.
- static const my_float s_eps0 = 8.8541878E-12

  Relative Permittivity of vacuum: ε_0. The ability of a substance to store electrical energy in an electric field.
- static const my_float s_c0 = 2.99792458E+8

  The speed of light in vacuum.

4.4.1 Detailed Description

Container class for all the data associated with a FDTD domain.

Each instance of the Task class represents an individual FDTD domain, and contains all the data associated with that domain. Executing a Task simulates the work required for a real FDTD simulation. Details about the FDTD domain, such as its size and Yee cell details, are specified by the user during object construction. The domains are currently limited to only using a homogeneous media, meaning that there is only a single material across the entire domain, this material is set to empty space (vacuum). All the electromagnetic fields are initialized to zero. A point source is located at the center of the domain.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Task::Task (int *nts,* int *nx,* int *ny,* int *nz,* int *dx,* int *dy,* int *dz*)

Constructor.

Creates an instance of the Task class using the specifications provided as arguments. Will generate an unique ID: m_id, and calculate the total required FLOPs of the Task: m_flops_work. Will also set the point source location: m_is, m_js, m_ks, to the center of the domain. Calling this function will not allocate memory for the electromagnetic field values, it should therefore be followed by a call to: InitializeFieldComponents(), before doing calculations on the Task.

Parameters

in	*nts*	The number of time-steps.
in	*nx,ny,nz*	Specifies the number of Yee-Cells in each dimension. Also known as the domain size or grid size.
in	*dx,dy,dz*	The size of each Yee-Cell in each dimension $(\Delta x, \Delta y, \Delta z)$.

4.4.3 Member Function Documentation

4.4.3.1 void Task::InitializeFieldComponents ( )

Dynamically allocates memory for the field components.

Will dynamically allocate the required memory for the electromagnetic field components, depending on the specified dimensions: #dx, #dy, #dz, and the precision of the values (32/64 bit). All the values will also be initialized to zero. Must be freed by calling CleanUp() before $\sim$Task().

4.4.3.2 void Task::PrintResults ( )

Prints the Task's performance results.

Should only be called after a Task has been executed by a Worker. If the Task uses the test configuration of: 200 time steps, 100$*$100$*$100 Yee-Cells, and 1.0$*$1.0$*$1.0 Yee-Cell size. It will also print the sum of the m_ez array and compare it to the known correct result of said test configuration.

4.4.3.3 void Task::UpdateFieldPointers (const int *local_x_size,* const char $*$ *buffer*)

Updates the Task's pointers to point to the data stored in the local `buffer`.

This function is used when the Task has been received from another process. Because the pointers are transferred "as is", they will still use the addresses in the sender's memory space, which will result in undefined behavior if used by the receiver. Calling this function will update the pointers to point to data in the local memory space.

Parameters

in	*local_x_size*	The number of Yee-Cells in the X dimension of the local subdomain of the Task, will be used instead of the *m_nx*, and together with *m_ny* and *m_nz*, to calculate the offsets between the field component arrays in the `buffer`.
in	*buffer*	A pointer to the start of the `buffer`, where all the local field components are stored contiguously in memory.

The documentation for this class was generated from the following files:

- src/Task.h
- src/Task.cpp

4.5 Timer Class Reference

A helper class to simplify timing and preserve code readability.

`#include <Timer.h>`

Public Member Functions

- void Start ()

  Starts the timer.
- void Stop ()

  Stops the timer, must be called after Start().
- double GetTime () const

  Retrieves the measured time in seconds between a call to Start() and Stop().

Static Private Member Functions

- static double TimeNow ()

  Returns the current time in seconds. Used by Start() and Stop().

Private Attributes

- double **m_seconds**

4.5.1 Detailed Description

A helper class to simplify timing and preserve code readability.

Measures and stores the time (in seconds) between a call to Start() and Stop(). The time is stored in seconds using double precision, it should therefore be sufficiently accurate.

The documentation for this class was generated from the following files:

- src/Timer.h
- src/Timer.cpp

4.6 Worker Class Reference

An abstract class for workers.

`#include <Worker.h>`

Inherited by CpuWorker, and GpuWorker.

Public Member Functions

- Worker ()

  Constructor.
- virtual ∼Worker ()

  Destructor.

- void PrintInformation ()

  Prints information about the worker.
- void PrepareForNewTask (const bool has_bottom_subdomain, const int from, const int to, Task ∗new_task)

  Prepares the worker for a new Task by updating the necessary member variables.
- void UpdateH ()

  Signals the internal thread of the Worker that it should call UpdateH_Internal().
- void UpdateE ()

  Signals the internal thread of the Worker that it should call UpdateE_Internal().
- void SourceExcitation (const int time_step, const int i_source)

  Signals the internal thread of the Worker that it should call SourceExcitationInternal().
- void FinalizeTask ()

  Signals the internal thread of the Worker that it should call FinalizeTaskInternal().
- void MeasurePerformance ()

  Signals the internal thread of the Worker that it should call MeasurePerformanceInternal().
- void Wait ()

  Waits until the internal thread of the Worker reaches an IDLE state.
- void Stop ()

  Signals the internal thread of the Worker that it should stop.

Public Attributes

- const int m_id

  An unique ID automatically generated during object construction.
- int m_performance

  The measured performance of the Worker in million floating-point operations per second.

Protected Member Functions

- virtual void PrepareForNewTaskInternal ()

  Preparations for a new Task, executed by the internal thread.
- virtual void UpdateH_Internal ()=0

  Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- virtual void UpdateE_Internal ()=0

  Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.
- virtual void SourceExcitationInternal ()=0

  Updates the source point (m_i_source) in the FDTD domain of m_task.
- virtual void FinalizeTaskInternal ()

  Finalization of the Task done by the internal thread.

Protected Attributes

- char m_name [256]

  Container for the name of the Worker.
- bool m_has_bottom_subdomain

  A flag that indicates if the Worker has been assigned the bottom sub-domain of the FDTD domain.
- int m_current_time_step

  The current time-step of the simulation. Is updated by SourceExcitation() and used by SourceExcitationInternal().
- int m_i_source

  The X-coordinate of the source point. Is updated by SourceExcitation() and used by SourceExcitationInternal().
- int m_local_x_size

  The size of the range between m_from and m_to.
- int m_from

  The X-coordinate in the node's sub-domain in which the worker's currently assigned sub-domain starts.
- int m_to

  The X-coordinate in the node's sub-domain in which the worker's currently assigned sub-domain ends.
- Task ∗ m_task

  Pointer to the worker's currently assigned Task.

Private Types

- enum State {
  IDLE, UPDATE_H, UPDATE_E, SOURCE_EXCITATION,
  BENCHMARK, STOP }

  The states in which the internal thread can be in.

Private Member Functions

- void InternalThreadEntry ()

  The main loop of the internal thread.
- void MeasurePerformanceInternal ()

  Measures the performance of the Worker and updates m_performance accordingly.

Static Private Member Functions

- static void ∗ InternalThreadEntryFunc (void ∗this_worker)

  The entry point of the internal thread.

Private Attributes

- State m_state

  The state of the worker's internal thread.
- pthread_t m_thread

  The internal thread. Is initialized and started during a call to Worker().
- pthread_mutex_t m_mutex

  The mutex governing access to the member variables between the host and the internal thread.
- pthread_cond_t m_condition

  A conditional variable used for signaling the internal thread.

Static Private Attributes

- static int s_next_id = 0

  A static counter used to generate the unique m_id.

Friends

- class WorkerPool

  Needed because WorkerPool::SourceExcitation() needs access to m_from and m_to.

4.6.1 Detailed Description

An abstract class for workers.

This class serves as an abstract class that can be used to create workers, which are specialized or optimized for a particular execution unit. It will also incorporate all the threading functionality needed to execute multiple workers simultaneously as several threads.

4.6.2 Member Enumeration Documentation

4.6.2.1 enum Worker::State [private]

The states in which the internal thread can be in.

Enumerator

IDLE Set by the internal thread when it has completed its current work and awaits new work.

UPDATE_H Set by UpdateH().

UPDATE_E Set by UpdateE().

SOURCE_EXCITATION Set by SourceExcitation().

BENCHMARK Set by MeasurePerformance().

STOP Set by Stop().

4.6.3 Constructor & Destructor Documentation

4.6.3.1 Worker::Worker ( )

Constructor.

This function will implicitly be called by the constructor of the workers which inherits this class. It will create a new POSIX thread and have it enter a wait state, where it will wait for signals from the master thread through calls to the Worker class' public functions.

4.6.3.2 Worker::~Worker ( ) [virtual]

Destructor.

This destructor will free all the POSIX thread resources allocated by Worker(). The internal thread must be exited by calling: Stop(), prior to destroying the object.

4.6.4 Member Function Documentation

4.6.4.1 void Worker::FinalizeTask ( )

Signals the internal thread of the Worker that it should call FinalizeTaskInternal().

Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

4.6.4.2 void Worker::FinalizeTaskInternal ( ) [protected], [virtual]

Finalization of the Task done by the internal thread.

This is a virtual function that does nothing and can optionally be implemented by classes inheriting this class. If implemented, it should clean up any resources allocated by the PrepareForNewTaskInternal() function. The function has to be implemented by workers that uses execution units with dedicated memory (such as GPUs), since they need to copy the final data to system memory at the end of the execution.

Reimplemented in GpuWorker.

4.6.4.3 void Worker::InternalThreadEntry ( ) [private]

The main loop of the internal thread.

This function contains the main loop which the internal thread will execute. The internal thread will not use polling, but rather wait (using a conditional variable) for m_state to change. Depending on m_state, the internal thread can execute the internal functions: PrepareForNewTaskInternal(), MeasurePerformanceInternal(), UpdateH_Internal(), UpdateE_Internal(), SourceExcitationInternal() and FinalizeTaskInternal().

4.6.4.4 void * Worker::InternalThreadEntryFunc (void * *this_worker*) [static], [private]

The entry point of the internal thread.

This function is needed because pthread_create() cannot take a member function as a parameter. This function circumvents this limitation by being a static function that takes a pointer to a worker object as a parameter, then calls the worker object's member function: InternalThread-

Entry(), which contains the internal thread's main loop.

Parameters

in,out	*this_worker*	A pointer to the worker object that owns the thread executing this function.

4.6.4.5 void Worker::MeasurePerformance ( )

Signals the internal thread of the Worker that it should call MeasurePerformanceInternal().

Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

4.6.4.6 void Worker::MeasurePerformanceInternal ( ) [private]

Measures the performance of the Worker and updates m_performance accordingly.

This function will measure the performance of the worker by letting it execute a dummy task. A dummy task is both created and destroyed during a call to this function.

4.6.4.7 void Worker::PrepareForNewTask (const bool *has_bottom_subdomain,* const int *from,* const int *to,* Task * *new_task*)

Prepares the worker for a new Task by updating the necessary member variables.

This function will prepare the worker for a new task, and must therefore be called before simulations can be done on a FDTD sub-domain. Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

Parameters

in	*has_bottom_subdomain*	A flag that indicates if the sub-domain assigned to the worker is the bottom sub-domain, in relation to how the FDTD domain is divided among nodes and workers along the X-axis.
in	*from,to*	Specifies the range of the node's sub-domain in which the worker's sub-domain exists.
in	*new_task*	A pointer to the task which the worker should prepare for. Meaning that future calls to UpdateH(), UpdateE(), SourceExcitation() and FinalizeTask() will make changes to this task.

4.6.4.8 void Worker::PrepareForNewTaskInternal ( ) [protected],[virtual]

Preparations for a new Task, executed by the internal thread.

This is a virtual function that does nothing and can optionally be implemented by classes inheriting this class. It has to be implemented by workers that uses execution units with dedicated memory (such as GPUs), since they need to load data prior to execution.

Reimplemented in GpuWorker.

4.6.4.9 void Worker::PrintInformation ( )

Prints information about the worker.

This function will print the ID, the name and the measured performance of the worker. However, it will not output anything if the global variable: g_output, is set to ERROR.

4.6.4.10 void Worker::SourceExcitation (const int *time_step,* const int *i_source*)

Signals the internal thread of the Worker that it should call SourceExcitationInternal().

This function will signal the worker to update the source point in a domain. A check should be done prior to calling this function to make sure that sub-domain assigned to this worker actually contains the source point. Member variables will be updated with the arguments to this function, so that the thread calling: SourceExcitationInternal(), will have access to them. Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

Parameters

in	*time_step*	The current time-step of the FDTD simulation. Will update m_current_time_step.
in	*i_source*	The X-coordinate of the source point. Will update m_i_source.

4.6.4.11 virtual void Worker::SourceExcitationInternal ( ) [protected],[pure virtual]

Updates the source point (m_i_source) in the FDTD domain of m_task.

This is a pure virtual function, meaning that it must be implemented by any class inheriting this class.

Implemented in GpuWorker, and CpuWorker.

4.6.4.12 void Worker::Stop ( )

Signals the internal thread of the Worker that it should stop.

This function will wait until the internal thread completes whatever it is currently doing. It must be called before destroying the worker, so that the internal thread can exit its loop and be joined. A call to this function will only return when the internal thread has been joined, a call to Wait() is therefore not needed and will in fact create a deadlock.

4.6.4.13 void Worker::UpdateE ( )

Signals the internal thread of the Worker that it should call UpdateE_Internal().

This function will signal the worker to update the electric field values of a sub-domain. Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

4.6.4.14 virtual void Worker::UpdateE_Internal ( ) [protected],[pure virtual]

Updates the electric field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This is a pure virtual function, meaning that it must be implemented by any class inheriting this class.

Implemented in GpuWorker, and CpuWorker.

4.6.4.15 void Worker::UpdateH ( )

Signals the internal thread of the Worker that it should call UpdateH_Internal().

This function will signal the worker to update the magnetic field values of a sub-domain. Since a call to this function only signals the internal thread before returning, a call to Wait() should follow to make sure that the internal thread has completed its execution.

4.6.4.16 virtual void Worker::UpdateH_Internal ( ) [protected], [pure virtual]

Updates the magnetic field values in the specified range: m_from to m_to, in the FDTD domain of m_task.

This is a pure virtual function, meaning that it must be implemented by any class inheriting this class.

Implemented in GpuWorker, and CpuWorker.

The documentation for this class was generated from the following files:

- src/Workers/Worker.h
- src/Workers/Worker.cpp

4.7 WorkerPool Class Reference

A helper class used to interact with a set of Worker objects.

`#include <WorkerPool.h>`

Public Member Functions

- void AddWorker (Worker ∗new_worker)

  Adds a new Worker to the m_workers vector, nothing else.
- void MeasurePerformance ()

  Calls the Worker::MeasurePerformance() function for all the workers in m_workers.
- void PrepareForNewTask (const int from, const int to, Task ∗new_task)

  Calls the Worker::PrepareForNewTask() function for all the workers in m_workers.
- void UpdateH ()

  Calls the Worker::UpdateH() function for all the workers in m_workers.
- void UpdateE ()

  Calls the Worker::UpdateE() function for all the workers in m_workers.
- void SourceExcitation (const int time_step, const int i_source)

  Calls the Worker::SourceExcitation() function for the Worker who has the point source in its sub-domain.
- void FinalizeTask ()

  Calls the Worker::FinalizeTask() function for all the workers.
- void KillAllWorkers ()

  Calls the Worker::Stop() function for all the workers. A cleanup function that calls Worker::Stop() on all the workers before deleting them.

Private Member Functions

- void PrintWorkerLoad (const unsigned int worker_id, const int subproblem_size, const int problem_size)

  Prints the precentage of the worker's sub-domain size related to the total domain size.

Private Attributes

- std::vector< Worker * > m_workers

  A vector of pointers to Worker objects.
- std::vector< int > m_load_sizes

  A vector of all the load sizes, corresponding to m_workers.
- int m_collective_performance

  The collective performance of all the workers in m_workers.
- int m_subdomain_from

  X-coordinate of where the WorkerPool's sub-domain starts in the FDTD domain.
- int m_subdomain_to

  X-coordinate of where the WorkerPool's sub-domain ends in the FDTD domain.

4.7.1 Detailed Description

A helper class used to interact with a set of Worker objects.

This class is only meant to simplify interactions with multiple workers at the same time by providing a set of functions corresponding to the Worker class' public functions. There should generally only exist one instance of this class per Node. It has however not been made all static like the Node class, since future development might require several instances per node. This class will internally handle all the needed synchronizations between workers by calling Worker::Wait() when necessary.

4.7.2 Member Function Documentation

4.7.2.1 void WorkerPool::FinalizeTask ( )

Calls the Worker::FinalizeTask() function for all the workers.

This function is the counterpart to the PrepareForNewTask() function, and should be called after the task is considered done. All the workers will be synchronized by calling Worker::Wait(), before returning from this function.

4.7.2.2 void WorkerPool::KillAllWorkers ( )

Calls the Worker::Stop() function for all the workers. A cleanup function that calls Worker::Stop() on all the workers before deleting them.

Calling this function will stop all the workers, delete them, and do the necessary cleanup. It should be called at the end of the program execution in order to terminate all the threads and free resources.

4.7.2.3 void WorkerPool::MeasurePerformance ( )

Calls the Worker::MeasurePerformance() function for all the workers in m_workers.

This function will measure the performance of all the workers in the pool by calling the Worker::-MeasurePerformance() function for all the workers in m_workers. It will also set the m_collective-_performance, so that it contains the collective performance of all the workers in the WorkerPool. All the workers will be synchronized by calls to Worker::Wait(), before returning from this function.

4.7.2.4 void WorkerPool::PrepareForNewTask (const int *from,* const int *to,* Task * *new_task*)

Calls the Worker::PrepareForNewTask() function for all the workers in m_workers.

This function will prepare all the workers in m_workers for the new Task by calling Worker::-PrepareForNewTask() on each of them. This function must be called before UpdateH(), Update-E(), SourceExcitation() or FinalizeTask(). FinalizeTask() is the counterpart to this function, and should be called after the task is considered done. Calling this function will divide the task's domain into a number of sub-domains equal to the number of workers in the m_workers vector. The size of the sub-domains is determined by the measured performance of each of the workers after a call to MeasurePerformance(). If MeasurePerformance() has not been called prior to calling this function, the domain will be divided evenly among the workers. All the workers will be synchronized by calling Worker::Wait(), before returning from this function.

Parameters

in	*from,to*	Specifies the range of the original domain which the Worker-Pool should work on. If the original domain has been divided among multiple nodes, this range should specify the range of the sub-domain assigned to the node who owns this instance of the WorkerPool class.
in	*new_task*	A pointer to the new Task which the workers will prepare for. Every worker will store a copy of this pointer. Calling this function alone will not make any changes to the task instance.

4.7.2.5 void WorkerPool::PrintWorkerLoad (const unsigned int *worker_id,* const int *subproblem_size,* const int *problem_size*) [private]

Prints the precentage of the worker's sub-domain size related to the total domain size.

This function will not print anything if g_output is set to ERROR.

Parameters

in	*worker_id*	The ID of the worker.
in	*subproblem_-size*	The size of the sub-domain, specifically the size in the X dimension.
in	*problem_size*	The size of the complete domain, specifically the size in the X dimension.

4.7.2.6 void WorkerPool::SourceExcitation (const int *time_step,* const int *i_source*)

Calls the Worker::SourceExcitation() function for the Worker who has the point source in its sub-domain.

This function will do a check to see if the `i_source` exists in the range previously set by calling the PrepareForNewTask() function. If the `i_source` exists in the range, the worker who is responsible for the sub-domain containing the point source will be found and signaled. The worker will then update the source point by calling Worker::SourceExcitation(). This function should be called once for every Task::m_num_time_steps. All the workers will be synchronized by calling Worker::Wait(), before returning from this function.

Parameters

`in`	*time_step*	The current time-step. Included because the algorithm calculating the new point source needs it.
`in`	*i_source*	The coordinate of the source point in the X-dimension. Needed in order to find the worker responsible for the sub-domain containing the source point. Because the domain is only divided along the X-dimension, the Y and Z coordinates do not need to be passed to this function. The Y and Z coordinates are however still needed, but can later be fetched from the Task object by the worker doing the actual source update.

4.7.2.7 void WorkerPool::UpdateE ( )

Calls the Worker::UpdateE() function for all the workers in m_workers.

This function will do one update of the electric fields using the Task and range previously set by a call to the PrepareForNewTask() function. Should be called once for every Task::m_num_time_steps. All the workers will be synchronized by calling Worker::Wait(), before returning from this function.

4.7.2.8 void WorkerPool::UpdateH ( )

Calls the Worker::UpdateH() function for all the workers in m_workers.

This function will do one update of the magnetic fields using the Task and range previously set by a call to the PrepareForNewTask() function. Should be called once for every Task::m_num_time_steps. All the workers will be synchronized by calling Worker::Wait(), before returning from this function.

The documentation for this class was generated from the following files:

- src/WorkerPool.h
- src/WorkerPool.cpp

5 File Documentation

5.1 src/globals.h File Reference

Header file containing all the global variables.

Macros

- #define use_float

  Decides between float and double precision. A compiler flag that decides if the my_float type should use float (32-bit) or double (64-bit) precision. Double precision will be selected if this line is commented away or removed. This exists only to provide a convenient way to switch between float and double precision.

Typedefs

- typedef float **my_float**

Enumerations

- enum Output { ERROR, INFO, VERBOSE }

  Enum type describing the amount of information output.

Variables

- bool g_single_worker

  Global variable that indicates if there is only one worker available for execution. Used to omit border exchanges if there is only one worker in use. Is initialized in Node.cpp and set during Worker creation.
- Output g_output

  Global variable that defines the amount of information output to the screen during execution. Is initialized in main.cpp and set when loading the system configuration.

5.1.1 Detailed Description

Header file containing all the global variables.

5.1.2 Enumeration Type Documentation

5.1.2.1 enum Output

Enum type describing the amount of information output.

Enumerator

ERROR Only output error messages and the end of execution results.

INFO Output minimal, but useful information.

VERBOSE Output everything.

5.2 src/main.h File Reference

Helper functions for main.

```
#include "globals.h"
#include <vector>
```

Functions

- void LoadSystemConfig (int *ncpus, int *nomps, int *max_ngpus)

  Loads the Worker configurations from the `config` *file in the root folder.*
- void LoadTaskConfig (int *ntasks, int *nts, int *nx, int *ny, int *nz, my_float *dx, my_float *dy, my_float *dz)

  Loads the Task configurations from the `config` *file in the root folder.*
- void PrintTaskConfiguration (const int ntasks, const int nts, const int nx, const int ny, const int nz, const my_float dx, const my_float dy, const my_float dz)

  Prints the Task configuration.
- void PrintEndResults (const std::vector< Task * > &tasks)

  Prints the end results of the program execution.

5.2.1 Detailed Description

Helper functions for main.

5.2.2 Function Documentation

5.2.2.1 void LoadSystemConfig (int * *ncpus,* int * *nomps,* int * *max_ngpus*)

Loads the Worker configurations from the `config` file in the root folder.

This function will load the user specified configurations from the config file, which includes: The number of CPU workers, OpenMP threads per CPU worker, and the maximum allowed number of GPU workers. These values will be loaded from the file and stored in the corresponding pointers passed as arguments. This function should be used in combination with the Node class, since Node::Node() requires these values as arguments.

Parameters

`out`	*ncpus*	The number of CPU workers.
`out`	*nomps*	The number of OpenMP threads per CPU worker.
`out`	*max_ngpus*	The maximum allowed number of GPU workers.

5.2.2.2 void LoadTaskConfig (int * *ntasks,* int * *nts,* int * *nx,* int * *ny,* int * *nz,* my_float * *dx,* my_float * *dy,* my_float * *dz*)

Loads the Task configurations from the `config` file in the root folder.

This function will load the Task configurations from the config file and store them at the addresses pointed to by the parameters, which are intentionally equivalent to the input parameters of Task-::Task(). This function should only be called by the host node, since it is responsible for creating the Tasks and distribute the workload among the slave nodes. The loaded task configurations are printed by a call to: PrintTaskConfiguration().

Parameters

out	*ntasks*	The number of tasks.
out	*nts*	The number of time-steps.
out	*nx,ny,nz*	The size of each domain.
out	*dx,dy,dz*	The size of each Yee-cell in a Task.

5.2.2.3 void PrintEndResults (const std::vector< Task * > & *tasks*)

Prints the end results of the program execution.

Will print a summary of the program execution.

Parameters

in	*tasks*	Pointer to a vector containing all the completed Tasks.

5.2.2.4 void PrintTaskConfiguration (const int *ntasks,* const int *nts,* const int *nx,* const int *ny,* const int *nz,* const my_float *dx,* const my_float *dy,* const my_float *dz*)

Prints the Task configuration.

Will print the Task configurations supplied as parameters. Will also calculate and print the number of floating-point operations of the Task.

Parameters

in	*ntasks*	The number of tasks.
in	*nts*	The number of time-steps.
in	*nx,ny,nz*	The size of each domain.
in	*dx,dy,dz*	The size of each Yee-cell in a Task.

Index