NTNU – Trondheim
Norwegian University of
Science and Technology

Heterogeneous FTDT for Seismic Processing

Andreas Berg Skomedal

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Our previous project looked at how to port Yee_bench's FTDT implemntaton to CUDA. This project extends this work, including taking advantage of the processing powers of both CPU cores and a GPU, including heterogeneous scheduling, optimizing for throughput of relatively small jobs suitable for seismic processing. Fitting the implementation to EMGS' problem and improving the underlying algorithm (differential length operator) may also be included.

Assignment given: 15. January 2013
Supervisor: Anne Cathrine Elster, IDI

Abstract

In the early days of computing, scientific calculations were done by specialized hardware. More recently, increasingly powerful CPUs took over and have been dominant for a long time. Now though, scientific computation is not only for the general CPU environment anymore. GPUs are specialized processors with their own memory hierarchy requiring more effort to program, but for suitable algorithms they may significantly outperform serially optimized CPUs. In recent years, these GPUs have become a lot more easily programmable, where they in the past had to be programmed through the abstraction of a graphics pipeline.

EMGS in Trondheim is an oil-finding service working with analysis of seismic readings of the ocean floor, to provide information about possible oil reservoirs. Data-centers comprised of CPU nodes does all the work today, however GPU installations could be more cost effective and faster.

In this thesis we look at the implementation of the main part of one of their data analysis algorithms. For this we use the FDTD method implemented in Yee_bench[3] by Ulf Andersson. We look at how to adapt it for GPU using CUDA, parallelize the CPU implementations and how to run this efficiently together heterogeneously.

It is shown that this method has great potential for use on GPUs, speedups just short of 19x over single thread CPU are achieved in this work. The FDTD method we use does however have some erratic memory operations which limits our performance compared to great GPU implementations these days which can reach speedups of over 100x. However, many of them still compare to single CPU performance. The order in which we address memory is therefore even more important, we show that optimizing memory writes when half the memory reads will not coalesce still improves our performance considerably. We show that care is needed when scheduling jobs on both CPU and GPU on the same node to avoid the total performance going down. Using all available resources on the host may not be beneficial. Utilizing several parallel CUDA streams proves effective to hide a lot of overhead and delay caused by busy CPU and main memory.

This work is not a final solution for EMGS' needs for this tool, other considerations and options than those discussed are also of interest. These topics are included in the future work section.

Sammendrag

I datamaskinens barndom ble vitenskapelige kalkulasjoner gjort av spesialisert hardware. Mer nylig tok stadig mer kraftige CPUer over og har dominert i lang tid. Men nå er ikke vitenskapelige beregninger bare for det generelle CPU miljøet lenger. GPUer er spesialiserte prosessorer med egne minnehierarkier som krever mer arbeid å programmere. For passende algoritmer kan de derimot betydelig utkonkurrere de serielt optimaliserte CPUene. I nyere år har disse GPUene også blitt en god del enklere å programmere, for bare en ca. 10 år siden krevdes det at du programerte de som en grafikk prosessor med abstrakte.

EMGS i Trondheim er et firma som driver med oljesøkingstjenester, de er interesert i analyse av seismiske avlesninger fra havbunnen for å gi informasjon om mulige oljereservoarer. Datasentre med CPU-baserte servere gjør all jobben i dag, men et inntog av GPU processorer vil kunne være mer kostnadseffektivt og ha bedre ytelse.

I denne masteroppgaven vil se på implementeringen av hoveddelen av en av deres data-analyse algoritmer. For dette bruker vi FDTD-metoden som er implementert i Yee_bench[3] av Ulf Andersson. Vi ser på hvordan vi tilpasser denne til GPU ved å bruke CUDA, parallelliserer CPU implementasjonen og hvordan vi bør kjøre dette effektivt sammen heterogent.

Det vises at denne metoden har bra potensiale for bruk på GPUer, vi opplever ytelser rett under 20 ganger det av en enkel CPU kjerne i vårt resultat. FDTD metoden vi bruker har dog ikke særlig normaliserte minneaksesser som begrenser ytelsen vi oppnår i forhold til gode GPU implementasjoner i disse dager, disse når ofte ytelser over 100 ganger CPU. Men fortsatt vil mange ikke yte stort bedre enn på CPU. Rekkefølgen vi adresserer minnet er derfor enda mer viktig, vi viser at optimalisering av skriving til minnet når halvparten av lesetrafikken ikke er sekvensielle og derfor ikke sammenfalder gir oss gode ytelsesforbedringer. Vi viser at vi må være forsiktige når vi kjører oppgaver på både CPU og GPU sammtidig på en node i "scheduleren" for å unngå at den totale ytelsen ikke går ned. Om vi bruker alle tilgjengelige ressurser er det ikke nødvendigvis bra for ytelsen. Å bruke flere parallelle CUDA "streams" vises å være effektivt for å gjemme "overhead" og forsinkelser som kommer av at CPU og hovedminnet er i bruk.

Arbeidet her er ikke en fullstendig løsning på EMGS' verktøy, andre betraktninger og muligheter enn de presentert her er også av interesse. Disse emnene er inkludert i videre arbeid seksjonen.

Acknowledgments

This is a master thesis for TDT4900 - Computer and Information Science, at the Department of Computer and Information Science, the Norwegian University of Science and Technology.

I would like to thank my advisor Dr. Anne C. Elster for guiding me along, Dr. Cyril Banino-Rokkones from Electromagnetic Geoservices (EMGS) in Trondheim for providing the project and providing assistance. A great thanks to NVIDIA for providing hardware and their knowledge to the HPC-Lab through their CUDA Research Center and CUDA Teaching Center programs. I would also like to thank all my friends from Datateknikk 2013. And lastly a thanks to the guys at the HPC-Lab.

Andreas Berg Skomedal, Trondheim, Norway, May 20, 2013.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The oil and natural gas production is one of todays most important industries, especially in Norway. More than 87 million barrels are produced every day worldwide, and over two million a day in Norway alone[1]. With oil priced at up to one hundred USD a barrel this is a substantial economical market, supplying very important resources, supplying thousands of jobs, and creating the foundation of millions more.

Finding this oil is down to a number of techniques, one of which is controlled-source electromagnetic surveying (CSEM) done by EMGS, a powerful horizontal electric dipole is towed above the sea floor transmitting an electromagnetic signal into the subsurface. Electromagnetic signals propagate differently in different layers, so that it is possible to distinguish the the sea-bed's composition. Grids of seabed receivers measure the energy that has propagated through the sea and the subsurface. The received data at this point requires data processing and analysis to gain a three dimensional image of the seabed and determine drilling decisions[7].

This data processing is very computationally heavy, computing using supercomputers is therefore often the only option with large datasets or large quantities of jobs to execute. Until just a few years ago, data-centers would use almost exclusively general purpose CPUs to do this computation. However, the CPU is optimized for fast serial code, logic and running desktop programs. Recent improvements in GPU hardware, scientific GPU research and the availability of scientifically aimed compilers have made GPU more and more attractive as it's area of expertise is crunching large amounts of data in parallel. Similar work includes work done at the HPC lab earlier such as The Lattice Boltzmann Simulation on

Multi-GPU Systems[26] and CPU and GPU Co-processing for Sound[10].

A central part of the data analysis done at EMGS is running the FDTD method we will look into, running faster and more efficiently is always desirable. The prospect of running on GPUs and in heterogeneous environments is very interesting for EMGS, creating a GPU solution and a scheduler to utilize this together with existing CPU implementations will be explored. The goal is to adapt Yee_bench, an open FDTD method implementation to EMGS' needs and bring it into todays heterogeneous reality. We will adapt the CPU implementations as well to utilize multi core parallelization.

1.1 Outline

In Chapter 2 we will present background information, the rise of parallelization on GPU, the CUDA API and Yee_bench.

In Chapter 3 we introduce the CUDA Implementation of the Yee_bench code, how it was modified and the optimizations used.

Chapter 4 introduces the scheduler, our CPU modifications, the single device benchmark codes and general implementation.

Chapter 5 shows the results of the implementation and benchmarks comparing different versions to the already existing CPU version.

Lastly in Chapter 6 we conclude the findings of the thesis and discuss future work.

The Appendices include a user manual for Yeebench_CUDA in Chapter A describing how to run the scheduler and it's benchmarks as well as the plot script. In Chapter B we list some relevant source code to the project.

1.2 Setup

Throughout this thesis, timings and performances will be run on the following system unless specified otherwise. This includes compiling, development and trialruns.

Table 1.1: 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

Chapter 2

Background

In this chapter, we will introduce parallel computing of different types, from multiple computers with one CPU to the massively parallel GPUs. The foundation of the thesis Yee_bench is presented along with it's scientific basis, the FDTD method. This Chapter is built on the background section from my project in the fall of 2012[24].

2.1 Parallel Computing on GPU

Since the birth of the microprocessor at the end 1960s, it's performance has steadily increased, until a few years ago the serial performance was doubled every 18 or so months. This is related to Moore's Law stating that the number of transistors doubled approximately every 24 months. However since the number of transistors for a given area is now so large the amount of energy generated at this level is exceeding what heat sinks can realistically handle. The performance still increases to this day but at a slower pace, and in many cases due to other factors such as architecture size and optimizations. CPUs are optimized to increase the performance of serial code, focusing on single core speed, branch prediction, memory pre-fetching etc.

Todays greatest performance increases come from parallelism, commercial off-the-shelf hardware consist of CPUs consisting of several processing-cores. Each of these cores have equal processing power and increase the performance by allowing several programs to run in parallel at the same time or use parallel programs which split it's work amongst the cores. Traditional servers and HPC data centers

have had multi CPU nodes for years, but in todays environment even commodity home computers benefit from parallelism. In supercomputers there are thousands of cores, spread amongst thousands of nodes. Building carefully designed software to take use of this can be difficult, as stated by Amdahl's Law[2], the size of the serial region of a program limits it's parallel peak performance so minimizing this region and creating clever solutions is imperative to utilize the resources.

2.1.1 GPU

The GPU is a specialized processing unit aimed initially at accelerating graphics computations. At first the graphics controllers were only 2D image rendering to be used as the basis of generating the image for computer displays. However in the 1990s this evolved into GPUs with more and more functionality moved from the CPU, creating true GPUs with shader programs to allow 3D rendering with programmable functionality.

Around the turn of the millennium scientists started using GPUs to accelerate scientific applications, this offered speedups of over 100x for certain applications however programmers were limited to using graphics APIs like OpenGL to program the GPU. When using these APIs the programmer would have to relate to functions and structures such as shaders and frame buffers making it difficult and raising the bar for writing such programs. Newer fully programmable GPU hardware and API with compilers like CUDA and OpenCL opens this tremendous resource up to the rest of the world and improving previous implementations. This is where the notation of GPGPU or General-Purpose Graphics Processing Unit starts to be a broader and more accurate term than simply GPU.

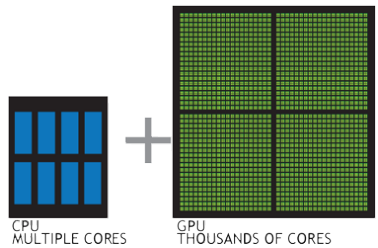

Figure 2.1: GPU vs CPU Cores [21] with permission from NVIDIA

The GPU consists of thousands of cores designed for parallel performance, but with little serial optimization making applications with a lot of control or serial

regions ineffective. Using the CPU to control the flow of an application and accelerating the the compute-intensive regions "kernels", on the GPU is a powerful combination. [21]

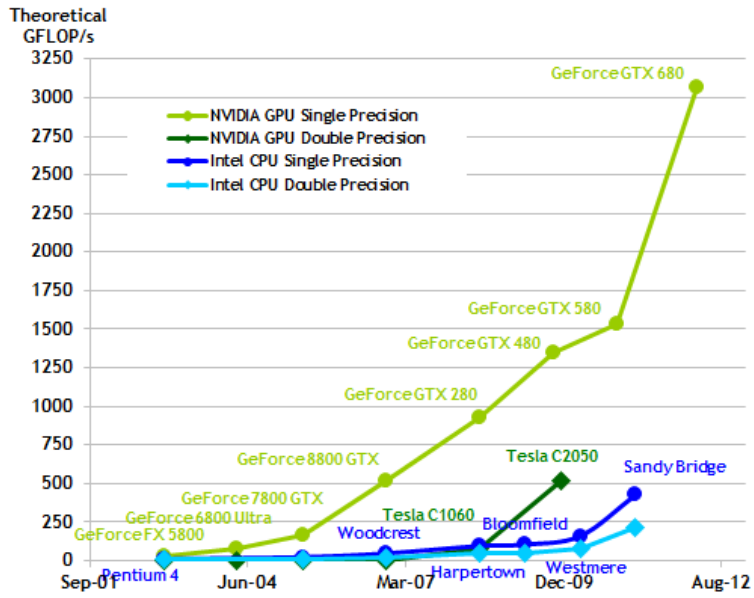

Figure 2.2: GPU vs CPU Performance [18] with permission from NVIDIA

Due to optimization for compute-intensive highly parallel computation on GPUs and reduced logic for control the GPU can achieve far higher throughput on operations, visible in the comparison above in Figure 2.2 between Intel CPUs and NVIDIA GPUs.

A GPU is as shown earlier usually at least one order of magnitude better than CPU, and in many cases two orders of magnitude, scaling more nodes instead to get the same amount of CPU power is inefficient in many ways. Using several nodes require more communication in many applications, additional nodes is more expensive than a GPU, one can pace up to 4 GPUs in one host for increased performance. In the given example of FDTD in OpenCL [25], one node of 4 GPUs is more than three times as fast as 4 nodes with 2 hyper threaded quad core CPUs from Intel.

From the results in Figure 2.3 the relative difference in power between CPU and GPU can be viewed in an actual application. In their implementation however the actual benefit from more than 4 GPUs was small, the speed increases slowly

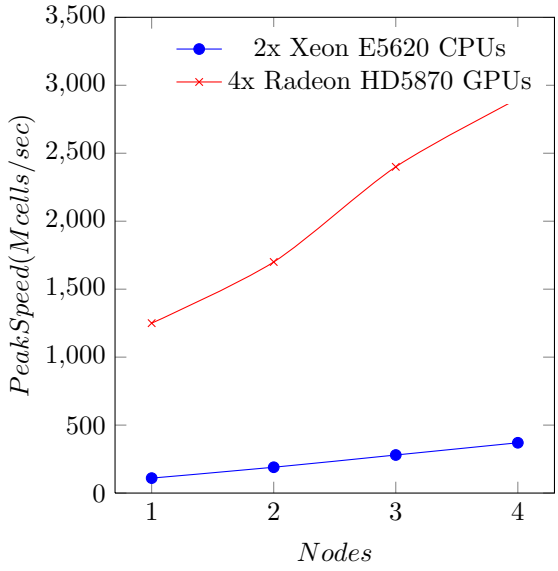

Figure 2.3: Multi CPU Node vs Multi GPU Node in OpenCL FDTD, note that the speed for GPU is on larger problem sets when increasing number of nodes. With data from article[25]

with more nodes, but on much larger problem sets, allowed by additional memory, as each GPU in their example is has 1GB of memory.

2.2 Compute Unified Device Architecture

As creating scientific programs using graphics based programming languages like OpenGL was complex and difficult, the GPU business created APIs for programming these processors in a simplified and precise way. NVIDIA developed Compute Unified Device Architecture, or CUDA in short. Programmers can also choose to use alternatives such as OpenCL, Microsoft's DirectCompute or the OpenACC directives for existing languages.

NVIDIA's "nvcc" is a C/C++ (from now on mentioned as just C) compiler that creates an extension to C that removes dependencies on knowledge about the inner workings of the graphics pipeline. The program consists of a traditional C program, initial setup like selection of GPU devices and allocation of memory on the GPU devices. To run anything on the GPU, special functions called

kernels are scheduled from the main program running on the CPU of the host. These kernels are asynchronous calls which are scheduled, the device needs to be "synchronized" in order to retrieve or write new data to the device memory. For newer models, independent "streams" of kernels can be used in parallel meaning different kernels may run at the same time on different SMs.

2.2.1 Architecture

The architecture of CUDA devices is important for how you use them. In Figure 2.4 we can see the overview for the Fermi architecture[17] with compute capability 2.0 which is present in the NVIDIA 480GTX card used in this thesis.

A GPU executes kernels in grids of thread blocks, thread blocks are executed in the streaming multiprocessors (SM). The SM consists of many CUDA cores, for Fermi there are 32 CUDA cores and 4 special function units (cosine, square root calculations etc) per SM, the SM executes 32 threads in groups of 32 called a warp. One SM has many load/store units, 16 for fermi, which is less than the number of cores but should keep it busy.

The 408GTX has 15 SMs, which in a way is like CPU SIMD core on steroids. Kernels are launched in parallel on all SMs, utilizing as many cores as possible based on the amount of memory used.

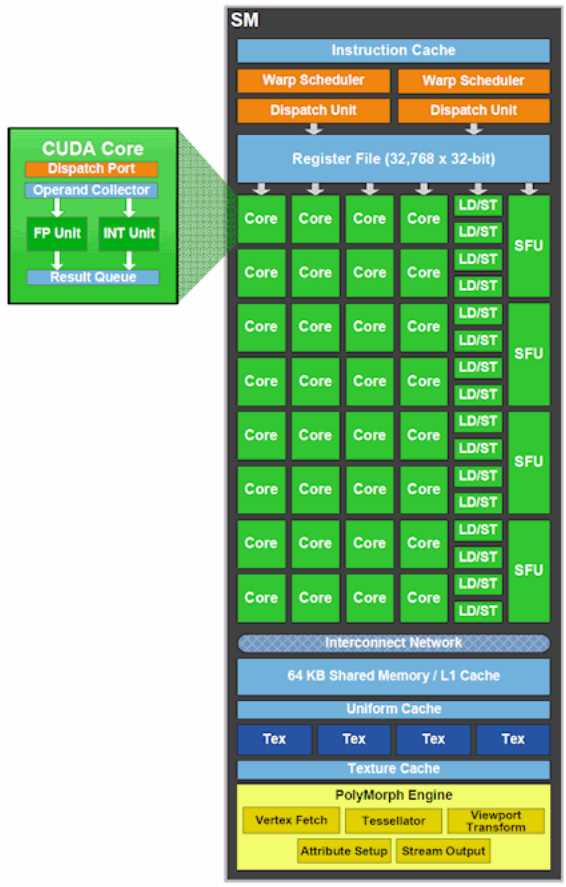

Figure 2.4: CUDA Fermi Architecture, [17] with permission from NVIDIA

Memory

CUDA GPUs have both on-chip and on-board memory, see Table 2.1. Managing and using the correct memory is the main concern when programming for CUDA.

Compute capability 2.0 units have several significant improvements in memory handling. Register spilling uses L1 cache instead of global memory (imagine suddenly having registers perform at 1/1000 of it's peak performance). An added L2 LRU cache for global memory accesses and implicit broadcast of constant data from global memory.

Table 2.1: Main CUDA Memories, GF100

Type	Speed	Perspective
Registers	8,000 GB/s	Per thread
Shared Memory	1,600 GB/s	Per thread block
Global Memory	177 GB/s	All threads and host

For GF100 (Fermi) architecture the size of the L1 cache is 16KB with 48KB shared memory, or 48KB L1 cache with 16KB shared memory with a L2 cache of 768KB. The memory from the viewpoint of a thread is depicted in Figure 2.5.

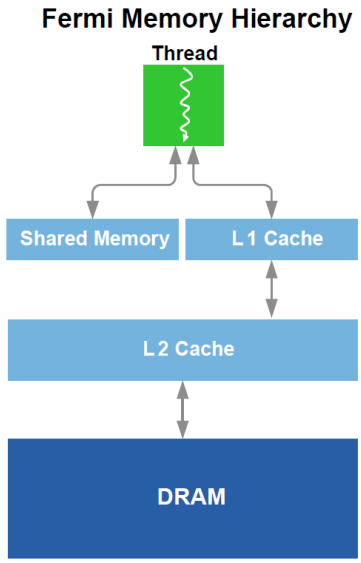

Figure 2.5: CUDA Fermi Memory Architecture for a Thread, [17] with permission from NVIDIA

2.2.2 Kepler Architecture

Also available to us is the NVIDIA Tesla K20 card[20] with compute compatibility 3.5. While GeForce cards are designed for consumer graphics the Tesla family is created for scientific parallel computations, exactly what we are working

with. The Tesla products offer us a range of features compared to the ordinary GeForce family, some of the most interesting are as follows. Far greater performance for double precision computing, up to four times the performance. Faster PCIe communication with double DMA engines and InfiniBand. Larger on-board memory to allow larger problem sizes or more parallel work to take place. These improvements as well as specialized drivers to reduce kernel overhead makes the Tesla family better suited to our needs.

Table 2.2: GeForce 480 GTX GF100 vs Tesla K20 GK110

	480 GTX	Tesla K20
CUDA Cores	480	2496
Streaming multiprocessors	15	8
Peak Double Floating Point Perf.	0.168 Tflops	1.17 Tflops
Peak Floating Point Perf.	1.35 Tflops	3.52 Tflops
Core Clock	700MHz	706MHz
Memory	1.5GB	5GB
Maximum Memory Bandwidth	177GB/s	208 GB/s

2.2.3 Utilizing GPGPU and CUDA

There are some important aspects when using GPGPU instead of CPUs. According to Farber[9] we have the three rules of GPGPU,

- Get the data on the GPGPU and keep it there
- Give the GPGPU enough work to do
- Focus on data reuse on the GPGPU to avoid memory bandwidth limitations

which from previous descriptions of how things are specialized seems natural and are in fact very important. Data transfer between host and GPU is in essence a waste of time if avoidable, and much slower than any ordinary memory lookup or similar. There is always a small overhead to launching work, the GPU will have the ability to run millions of floating point operations in the time the CPU even launches a kernel. Having the GPU under saturated means it's wasting resources when executing, as the ration between time spent launching kernels and running them can get skewed. Lastly using memory local to the CUDA core is imperative to utilize the processing power available, if each floating point operation was read

and written directly to global memory (even with perfect memory coalescing) the performance is lowered by a factor of 40-50.

Other than this we have considerations such as, memory handling, strengths and weaknesses of GPU cores compared to CPU cores, a requirement of being significantly parallelizable and the importance of launch configuration.

A CPU will have main memory which is big and slow, and then it's internally managed cache hierarchies. A GPU has it's own independent memory, a large global memory, but this is divided into different types, you have the ordinary registers but also fast shared memory within a single thread block. Using the correct memory here is vital, using the fastest and closest memory is always a best practice, not only is it faster but memory access conflicts are reduced as well.

As previously mentioned, a CPU is optimized for serial code with features such as branch predictions and memory pre-fetching. When programming the GPU these differences lead to significant performance differentiations, a number-heavy mathematical algorithm will have severe performance dips if too much control is imposed, many cycles will be wasted on diverging branches. The CUDA framework gives you large control over where you put your data and how you access it, meaning extra care is needed to utilize this freedom. For example if any data is shared inside a thread block it should be accessed as shared memory.

You also have the simple and overarching requirement that a problem is significantly parallelizable, even if just a section of code has little to no parallel regions the overall performance will drop by a lot. This is well documented by Amdahl's Law[2].

For the CUDA framework we have some limitations and specific behavior compared to ordinary C code,

- A kernel has to return void.
- Library functions are not available.
- Function calls from kernels are auto in-lined, and not actually called.
- Recursion is only supported on compute capability 2.0 and up.
- There are no static variables, values can't be kept between kernels.
- Debugging is complicated, no c out functions are available within a kernel.
- Use intrinsic functions if available as they are optimized for the NVIDIA GPU.

A relatively new problem is launch configurations, in the past days of CPU programming highly optimized libraries such as BLAS have hardware optimized code, self tuning code and the likes to really get the best out of your hardware and specific setup. For GPUs you have the added complexity of always specifying how the GPU will execute the kernels. Dimensions of threads, their order of execution, drawing properties from the amount of registers used etc., the number of CUDA threads launched needs to be a multiplum of the warp size for example. This is to ensure all compute cores in the GPU are used when it schedules threads.

For compute bound algorithms, the number of registers must be balanced with the shared memory to maximize the throughput of calculations[14]. Additionally, these resources should be maximized for a single block per multiprocessor. Techniques to optimize compute-bound algorithms include storing reusable, intermediate values in shared memory and assigning multiple data points to a thread to overcome the unused resources of idle threads.

For memory-bound algorithms, the challenge is to fit the working set of data into the fast GPU memory resources or leverage the low latency memory caches. This can be achieved by reorganizing the data into self-contained data structures and using a multi-pass approach to process a subset of these self-contained data structures during each pass. Additionally, the type of memory used should be carefully considered.

Optimized implementations are several times as fast as unoptimized implementations of CUDA algorithms. GPU acceleration of code has great potential, which we have seen to be utilized in other projects already. But carefully designing these algorithms to handle memory is imperative to achieve optimal performance.

2.3 Yee_bench

The Yee_bench[3] code is a benchmark implementing the finite-difference time-domain method (FDTD). This mathematical method was introduced by Yee in 1966. The FDTD method involved is explained further in Section 2.4. The Yee_bench code is the basis of the work performed during this thesis, and is what the result will be compared to. This section gives an overview of Yee_bench, a summary and look into the official paper published by Ulf Andersson.

Yee_bench contains complete code in Fortran90 and C, the documentation is focused on Fortran90. Code also exists in Fortran77, Matlab and C++. All

versions are available on request. Fortran90 has 6 editions with different electro magnetic field storage models.

PDC has two related parallel benchmarking codes. psycyee: A parallel implementation of the FDTD kernel using point-to-point MPI, available on request. GemsTD/frida: A hybrid time-domain solver, not available.

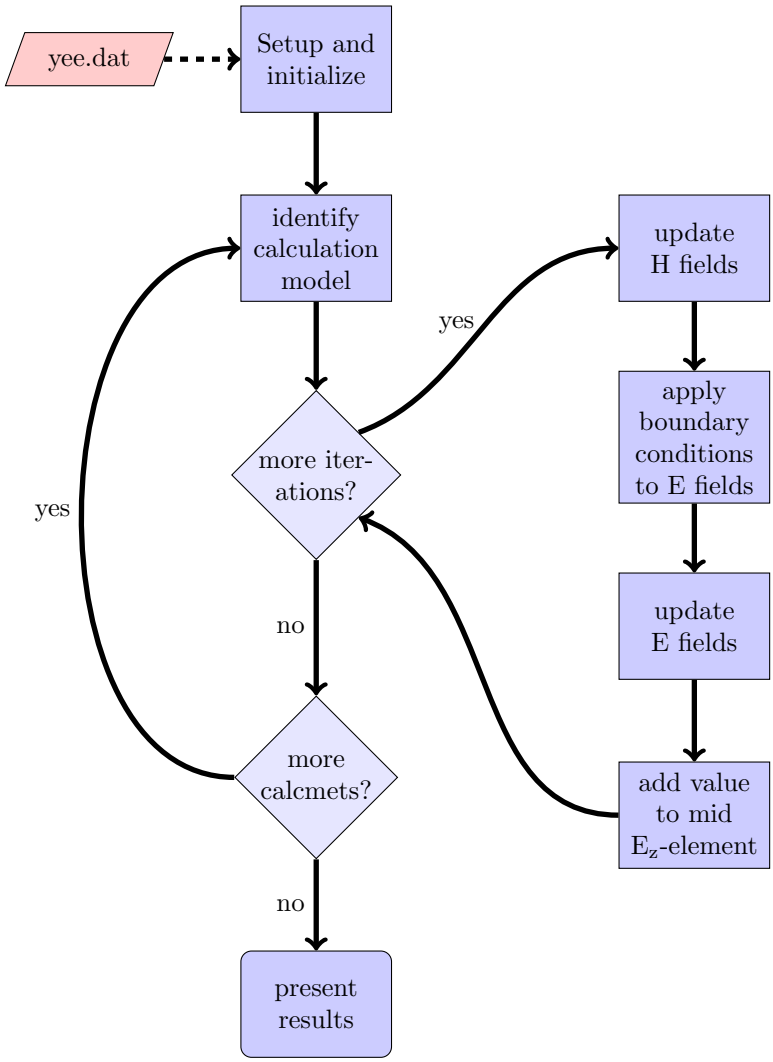

Figure 2.6: Yee_bench Flow Chart

The flow of the code is depicted in Figure 2.6. The domain lx,ly,lz is divided into N_x,N_y,N_z cells of equally sized dx,dy,dz cells with cubic computational domain, $N_x = N_y = N_z \equiv N$

Dirichlet boundary condition apply, a perfect electrical conductor (PEC) is assumed, so that the electrical field component at the outer boundary is set to zero. All initial values are set to zero as well.

For each N, N_t time steps are taken and an average time per time step is computed. Initialization overhead is excluded, along with time step 0, N_t is automatically scaled to make a problem of size N take about 20 seconds to complete.

The benchmark can repeat each time stepping, it then chooses the fastest, the initial value of N_t for first N is specified by user along with N_{min}, N_{max} and runs/repetitions per N.

Three implementations of the leap-frog update is available,[1] specified by the "calcmet" input in the settings file, see Table 2.3. Option 5 is used for results provided and has been shown to provide the best results as it optimizes register use.

Table 2.3: Calcmet Options

Calcmet	Method
3	array syntax
4	three separate loops
5	fused do loops

The memory requirement for FDTD is significant, for 64bit it is $24N^3+24(N+1)^3$ bytes reaching 4Gb at N=446. The amount of memory is often restrictive for how large a problem to create, since it is cubic and therefore increases rapidly. For 32bit the memory print will be halved.

Table 2.4: Maximum Problem Size vs Available Memory for Yee_bench.

128kbyte	2 Mbyte	4 Mbyte	1 Gbyte	2 Gbyte	4 Gbyte
N = 13	N = 34	N = 43	N = 281	N = 354	N = 446

There are 12 additions, 12 subtractions and 12 multiplications per cell per time step. Multiplication to addition/subtraction ratio is suboptimal however the

[1] leap frog method, updating velocity a halftime step after position update, "leaping over the velocity"[6].

problem is memory bound with 20 loads and six stores per cell per time step. Depending slightly on the implementations mentioned above and the compiler. Memory bandwidth can be measured with benchmark "DAXPY stream2".

The problem size in the implementation is mostly limited by memory, both memory traffic and available memory. For non parallelized implementations the compute time is not a factor.

Yee_bench is reported to be equipped with OpenMP however no results are given as to it's usefulness, multiprocess per node is given with severe performance reductions because of a memory bound problem. The same is assumed to be true for threaded runs.

2.3.1 Conclusions Made by Andersson

It is shown that the "stream2" DAXPY results can be used to predict limits for the performance of Yee_bench (with exception on one IBM computer and a SGI computer).

2.3.2 Personal Conclusions

Multi core parallelization of FDTD allegedly impossible, multi-CPU suggested to be viable still. The two way associative cache of the CPU used in 2002 shows severe memory dips at certain problem sizes, necessary to take care to avoid these. The effect is smaller for todays 8 way associative caches. Parallelization is to be made using several nodes, or memory independent compute devices. Their own conclusion seems to be more towards bandwidth prediction and bottlenecks of the implementation and less about the problem itself. The problem is a typical GPGU problem, with iteratively high amounts of floating point operations on large dataset.

As shown later in the results of Section 5 we see that many of these predictions are untrue with the modern hardware today. It should be noted that the article which originially introducees yee_bench is from 2002 and a lot has changed in the last 10 years, computing performance does rise faster than memory bandwidth but increasing the computing power of the cpu does improve performance. The code delivered did not contain any OpenMP directives.

2.4 The FDTD Method

Here follows a brief overview of the FDTD method used in the Yee_bench framework for solving Maxwell equations. FDTD refers to a specific finite-difference method, namely the leap-frog method on staggered Cartesian grids. The FDTD method is derived from Ampère's and Faraday's laws, applying central difference. A cubed domain (l_x, l_y, l_z) is divided into (N_x, N_y, N_z) equally sized cells with sizes $(\Delta x, \Delta y, \Delta z) = (l_x/N_x, l_y/N_y, l_z/N_z)$ [3]

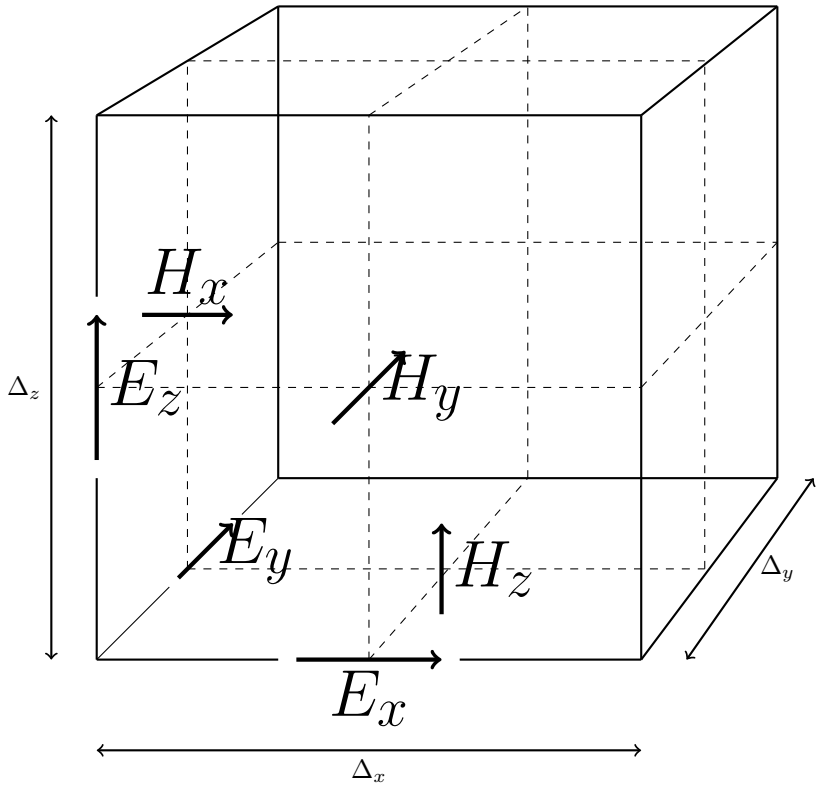

Figure 2.7: Positions of the Electric and Magnetic Field Components in a Yee Cell.

The grid consists of three magnetic fields and three electrical fields which are staggered, meaning all electromagnetic fields elements are defined at different

locations, see Figure 2.7.

$$\begin{aligned}
\epsilon\frac{\partial E_x}{\partial t} &= \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma E_x,\\
\epsilon\frac{\partial E_y}{\partial t} &= \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma E_y,\\
\epsilon\frac{\partial E_z}{\partial t} &= \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z,\\
\\
\mu\frac{\partial H_x}{\partial t} &= \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y},\\
\mu\frac{\partial H_y}{\partial t} &= \frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z},\\
\mu\frac{\partial H_z}{\partial t} &= \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}.
\end{aligned} \tag{2.1}$$

The Maxwell equations for the FDTD method can be seen in Equation 2.1. Which is the basis of the equations mentioned below.

$$H_x|^{n+\frac{1}{2}}_{i,j+\frac{1}{2},k+\frac{1}{2}} = H_x|^{n-\frac{1}{2}}_{i,j+\frac{1}{2},k+\frac{1}{2}} + \frac{\Delta t}{\mu\Delta z}\left[E_y|^{n}_{i,j+\frac{1}{2},k+1} - E_y|^{n}_{i,j+\frac{1}{2},k}\right] - \frac{\Delta t}{\mu\Delta y}\left[E_z|^{n}_{i,j+1,k+\frac{1}{2}} - E_z|^{n}_{i,j,k+\frac{1}{2}}\right] \tag{2.2}$$

$$H_y|^{n+\frac{1}{2}}_{i+\frac{1}{2},j,k+\frac{1}{2}} = H_y|^{n-\frac{1}{2}}_{i+\frac{1}{2},j,k+\frac{1}{2}} + \frac{\Delta t}{\mu\Delta x}\left[E_z|^{n}_{i+1,j,k+\frac{1}{2}} - E_z|^{n}_{i+\frac{1}{2},j,k+1}\right] - \frac{\Delta t}{\mu\Delta z}\left[E_x|^{n}_{i+\frac{1}{2},j,k+1} - E_x|^{n}_{i+\frac{1}{2},j,k}\right] \tag{2.3}$$

$$H_z|^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} = H_z|^{n-\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} + \frac{\Delta t}{\mu\Delta y}\left[E_x|^{n}_{i+\frac{1}{2},j+1,k} - E_x|^{n}_{i+\frac{1}{2},j,k}\right] - \frac{\Delta t}{\mu\Delta x}\left[E_y|^{n}_{i+1,j+\frac{1}{2},k} - E_y|^{n}_{i,j+\frac{1}{2},k}\right] \tag{2.4}$$

$$E_x|^{n+1}_{i+\frac{1}{2},j,k} = E_x|^{n}_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\epsilon\Delta z}\left[H_y|^{n+\frac{1}{2}}_{i+\frac{1}{2},j,k+\frac{1}{2}} - H_y|^{n+\frac{1}{2}}_{i+\frac{1}{2},j,k-\frac{1}{2}}\right] + \frac{\Delta t}{\epsilon\Delta y}\left[H_z|^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} - H_z|^{n+\frac{1}{2}}_{i+\frac{1}{2},j-\frac{1}{2},k}\right] \tag{2.5}$$

$$E_y|^{n+1}_{i,j+\frac{1}{2},k} = E_y|^{n}_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\epsilon\Delta x}\left[H_z|^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} - H_z|^{n+\frac{1}{2}}_{i-\frac{1}{2},j+\frac{1}{2},k}\right] + \frac{\Delta t}{\epsilon\Delta z}\left[H_x|^{n+\frac{1}{2}}_{i,j+\frac{1}{2},k+\frac{1}{2}} - H_x|^{n+\frac{1}{2}}_{i,j+\frac{1}{2},k+\frac{1}{2}}\right] \tag{2.6}$$

$$E_z|_{i,j,k+\frac{1}{2}}^{n+1} = E_z|_{i,j,k+\frac{1}{2}}^{n} - \frac{\Delta t}{\epsilon \Delta y}\left[H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} - H_x|_{i,j-\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}}\right] + \frac{\Delta t}{\epsilon \Delta x}\left[H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}} - H_y|_{i-\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}}\right] \tag{2.7}$$

At each time step the fields are updated from stencils based on equations 2.2 to 2.7. The three first are the stencils for the magnetic fields and the last three the electrical fields. The electrical fields are one half time-step ahead of the magnetic fields. These equations are based on the assumption of a homogeneous media where σ is zero. The method is explicit, meaning all values only depend on values in earlier time steps. The FDTD method is second order accurate in time and space.

Chapter 3

Yee_bench CUDA

For now we have looked at GPGPU, parallelizing and the FDTD method. In Chapter 4, we will introduce a heterogeneous scheduler for Yee_bench, which is the main goal of this thesis. Before we get to that we need something to schedule. Ulf Andersson has supplied us with a CPU edition, which leaves us to create a GPU version. In this chapter we will describe the main component of the heterogeneous scheduler, the CUDA implementation of Yee_bench.

The CUDA version took basis in the released C version of Yee_bench. This chapter describes this GPGPU implementation and the changes made to accommodate the requirements of EMGS. This Chapter describes work that was started on in my project in the fall of 2012[24], some of the text will contain exerts from that report. However a bug which affected the results of the previous report, in combination with new requirements made the specific results invalid and are therefore new.

3.1 Implementation

The existing implementation of Yee_bench consists of three 3D arrays of magnetic fields and three 3D arrays of electronic fields. These arrays consist of one array for each three-dimensional direction of the fields. The execution is performed iteratively, with each iteration including the following operations. First the electronic fields are updated, then the magnetic fields have their boundary conditions applied. Then the magnetic fields are updated followed by an effect applied to

the center element of the Z directional electrical field. After all time steps are calculated the time consumed and corresponding MFlops are displayed.

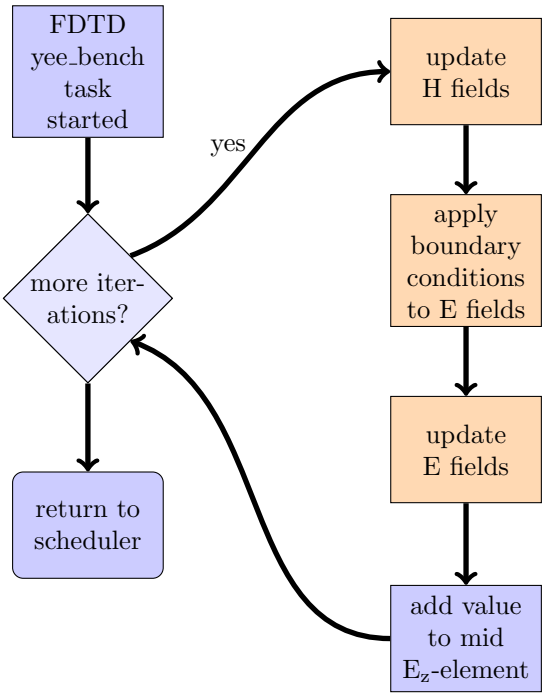

Figure 3.1: Yee_bench method Flow Chart

The overall flow of Yee_bench was unchanged in the CUDA version which is depicted in Figure 3.1. Update functions that are GPU kernels in the CUDA version are colored orange to extinguish them. In the update kernels, each field array updates it's elements with data from all three of the opposite field array type.

Other than the size of the arrays, the E field update kernel works the same way as the H field update kernel. For each element in the N^3 domain, there are 15 reads and 3 writes to global memory in each of these kernels.

3.2 Overall Design

At the start of the execution, following the declarations of variables, CUDA is initialized and device memory allocated. One CUDA array is created for each of the six field arrays, where the memory is initialized to zero on the device. The execution is launched with the configurations listed in Table 3.1, default values apply if not mentioned.

The update functions or "kernels", work mostly as before, updating the field arrays. The kernels are launched with one thread for each element in the YZ-plane, then iterating over the elements in the X dimension.

3.2.1 Launch Configuration

The block and grid configurations are squares which might suggest the processing to start at regular intervals throughout the domain. Which would happen if you have thread X,Y in iteration Z compute element X,Y,Z. Addressing inside the kernel ensures that consecutive CUDA threads are launched to compute neighboring cells to improve cache hit and data coalescing at both reads and writes.

Launching the CUDA kernels in block configurations of 16×16 produces the best effect due to warp effiency and maximizing threads. Grids are launched as described in Table 3.1, this will ensure that more threads than necessary are created, as the total number of elements is never exactly this number. This means that threads addressing elements outside the domain will be terminated after launch as they don't belong to an actual element, creating "dead" threads.

Table 3.1: Launch Configurations

Property		
Block configuration	$X \times Y = 16 \times 16$	square
Grid configuration	$X = Y = \left\lceil \sqrt[2]{\frac{n^2}{Block_y \times Block_z}} \right\rceil$	square
time steps	200	

Other CUDA settings made include setting the cache size for the L1 Cache with

```
cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)
```

This ensures that the L1 cache is larger, increasing cache hit at the expense of the shared memory size, as shared memory is not used this is a trade off without disadvantages. See Section 2.2.1.

3.2.2

To optimize the original implementation for CUDA, fewer parameters are passed to the kernels to reduce the register count, data addresses are computed locally as the problem is memory bound, for example offset variables that are originally precomputed.

All constants are passed at the end of the parameter list to utilize constant memory broadcast to the threads, with some in front of the array pointers and some after like in the original implementation this automation did not function.

The innermost loop, and the only actual loop in the CUDA implementation is changed to the X dimension. As this is the slowest moving dimension it increases memory access across threads and increases cache hits for shared data between threads.

Many other strategies and techniques were looked into during implementation that did not provide a beneficial speedup. Such as improved memory addressing calculations, but this utilized more registers and had a negative effect. Shared memory options were considered but as the stencil radius is small, actual elements needed are not very systematic and the availability of the cache function on Fermi this reuse did not justify increased complexity.

The algorithm is strongly memory bound, the memory utilization compared to theoretical maximum is fair at best as the data is accessed in a very irregular fashion, see Section 3.3 for results from code profiler. This is because calculations in the FDTD method read data from many different arrays in different directions, as listed in Section 2.4.

3.2.3 Problem Size Variations

The original specification of Yee_bench was that it was meant for problems of the size $N_x = N_y = N_z$, but the code allowed non uniform sizes.

For the CUDA version, the cleanup kernel was split up into three different kernels, each with it's own 2D-plane. For example the XY-plane as in Listing 3.1.

For the update kernels, each thread which corresponds to an element in the YZ-plane finds it's id with the same method as in the cleanup kernel in Listing 3.1.

Listing 3.1: Cleanup Kernel for XY-Plane

```
__global__ void cleanupE_xy_cuda( my_float *cHx, my_float *cHy, my_float *cEz, int
      nx, int ny, int nz, my_float Dbdx, my_float Dbdy )
{
    // Global thread id in CUDA, corresponding to the launch number
    int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
                (threadIdx.x + blockDim.x * threadIdx.y);

    int x = i/ny + 1;
    int y = i%ny + 1;

    if (x < nx && y < ny)
    EzC(x,y,0) += (HyC(x, y ,0)-HyC(x-1,y,0))*Dbdx +
                  (HxC(x,y-1,0)-HxC( x ,y,0))*Dbdy;
}
```

This addressing order is significant for the overall performance, in early tests it sped up the implementation by several hundred percent. In the profiler section we see that this is apparent for one of the cleanup kernels.

3.3 Profiler Analysis

Analysis is drawn from data gathered by running NVIDIA Visual Profiler 5.0.0[19]. The overall execution can be seen in Figure 3.2, it depicts a cutout of slightly more than one iteration. The updateE and updateH kernels take up most of the execution time, with the boundary condition kernels cleanupE only taking a few percent of the time spent executing.

The dead time between each iteration is partly due to profiling overhead, but also due to using a kernel with a single thread to calculate a single value for the middle element of E_z. However adding a conditional to an existing kernel is not worth it due to a few reasons. It would increase the number of registers needed and data copied to the kernel for all kernels to have the data needed to perform the computation, and the extra divergence will also punish performance.

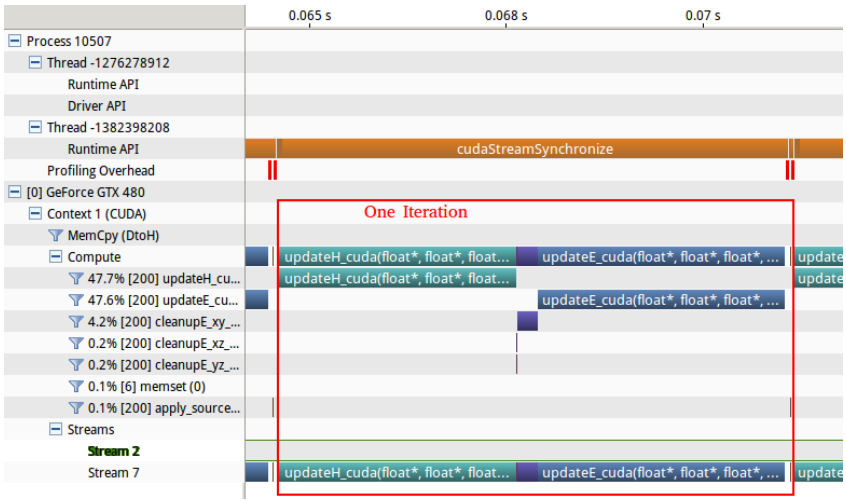

Figure 3.2: Overview of a Yee_bench Iteration in the CUDA Profiler

From Figure 3.3 we can see that cache miss is at 51%, this is due to most data being fresh for each thread with about half of it reused by other threads. As a result from this each thread will have varying execution time leading to instructions getting out of sync. This is reflected in high instruction replay, however as the main bottleneck of the execution is fetching and storing memory it is unlikely to be of importance. The cache misses leads to the main bottleneck of the program, memory.

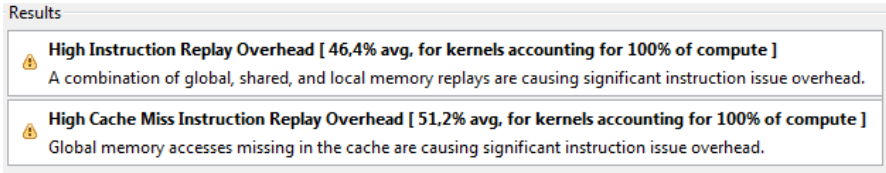

Figure 3.3: Profiler Instructions and Cache Miss of the CUDA Implementation

Figure 3.4 displays the memory throughput and efficiency for reading and writing global memory. Keep in mind the maximum throughput of the GTX480 is 177GB/s. For the memory data depicted, the maximum values apply to the two kernels that take up 95%+ of the execution time, the average numbers are a result of less effective accesses in the boundary condition kernel. It can be seen that the memory writes have a high efficiency, this is due to the fact that no

irregularities exist in the the writing addresses. Each array that is written is iterated in predictably and writes mostly coalesce. This is only untrue for one kernel, the cleanup kernel for the XY-Plane. This creates a bad writing structure and is the reason why this kernel is 4% vs 0.3% for the other two cleanup kernels. Global memory load efficiency however is limiting with an average of 56% for the two time consuming kernels. This is due to the irregular memory access in the kernels, where each thread accesses data plus one, and minus one in a direction for each field array. From before around 50% achieved cache hit, the remaining memory accesses often do not coalesce, reducing the memory performance.

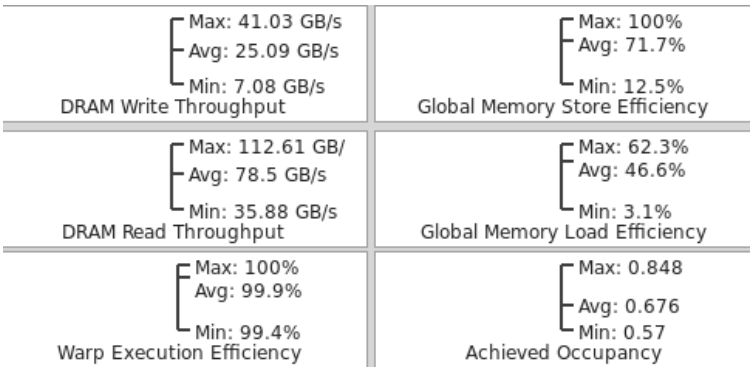

Figure 3.4: Profiler Memory and Occupancy of the CUDA Implementation

High warp execution efficiency was achieved by using proper block sizes and sufficient amount of blocks. Occupancy was commented by the profiler to be at 57%, limited by the number of threads used by the kernels. Yet occupancy only shows a part of the picture and 25% is said to be enough to give sufficient work to the GPU.

3.4 Intermediate Results and Implications

The hardware and configuration used is shown in Table 1.1, it is the hardware provided by the NTNU HPC Lab. The results from a single GPU run can be seen in Figure 3.5.

From the graph it is seen that the improvement from CPU implementation is considerable yet not as significant as many other GPGPU projects which can get close to a speedup of 100x, this solution is just shy of 20x for a single core. A

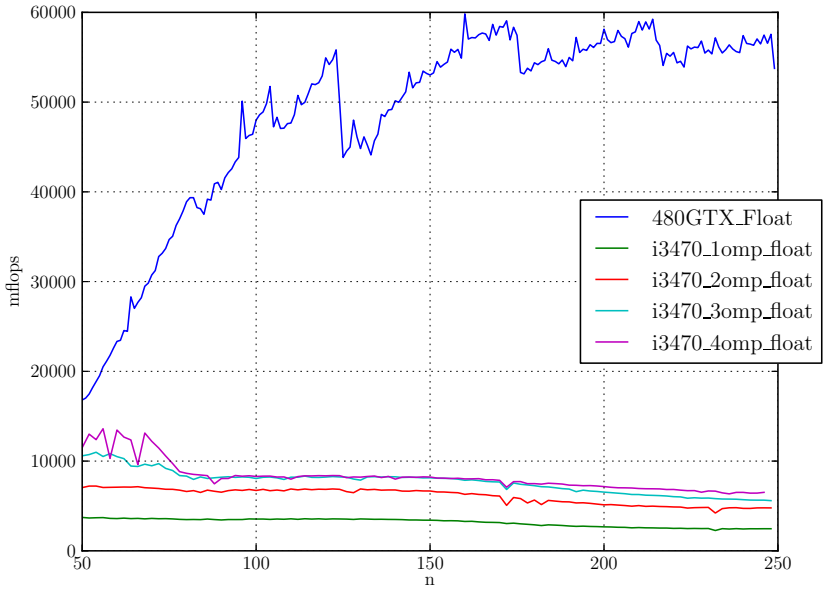

Figure 3.5: GPU Speed, Floating Point Precision for Main Loop

modern day GPU will in all likelihood perform even better than these numbers tell when comparing to a CPU. The main focus of this thesis is a heterogeneous implementation utilizing both CPU and GPU, maximizing the number of CPU cores might not be beneficial. First off the GPU implementation utilizes some CPU to keep the GPU busy, but it is also required to transfer data to and from the already strained RAM memory.

For results from the NVIDIA K20 GPU using the Kepler architecture see Section 5.5.

Chapter 4

The Heterogeneous Yee_bench Scheduler

Now that we have both a CPU and a GPU implementation of the Yee_bench code we can explore how to best use these together. In this chapter, we will describe the heterogeneous scheduler we have created to utilize both CPU- and GPU-calculation to execute FDTD based jobs.

The overall program has five different versions it can run.

1. Assignment Scheduler - The main scheduler that will run FDTD tasks in a heterogeneous environment.
2. Greedy Scheduler - A simple implementation that has each thread select a new task to perform after they complete their previous one.
3. Homogeneous Device Plotter - A benchmark created to plot performance across different problem sizes for a single device, CPU or GPU (used during development and to create graphs like Figure 3.5 in Section 3.4).
4. Heterogeneous Device Plotter - A benchmark created to plot performance for a heterogeneous scheduler, it emulates the assignment scheduler to create performance charts for different problem sizes.
5. Resource Plotter - A benchmark for comparison of different number of CPU threads and GPU threads, based off of the heterogeneous device plotter.

The one of interest is the Assignment Scheduler described in Section 4.2. For all versions the overall structure is the same, in which the program starts off

by identifying which type is to be run from the five types specified above. This is determined by the parameter to the program. The default or "greedy" for #2, "assignment" for #1, "plot" for #3, "hybridplot" for #4 and "resource" for #5.

4.1 Implementation

The overall program is no longer shaped as a simple benchmark with varying calculation methods which specified which algorithm to utilize. Yee_bench originally had several calculation methods to choose from and ran through the same problem size one or many times in a row to benchmark the method on a single core CPU.

The scheduler now supports both GPU and CPU, and with different types of parallelism.

4.1.1 Overall Design

Files in the implementation

- main.cu - Main file of the project, contains the main sequences of the program with setup and initializations.
- main.h - Header file for the project, contains includes, macros, definitions and declarations.
- jobs.cu - Contains code for running a FDTD task. CPU and GPU.
- update.cpp - Contains the CPU kernels.
- yee.cu - Contains the CUDA kernels.
- yee.dat - The configuration file containing numbers specific to the launch.
- plot.py - The plot script used to create graphs.

Common for all implementations is the general flow depicted in Figure 4.1. With minor differences or simply repeating the launch step in case of the hybrid device plotter.

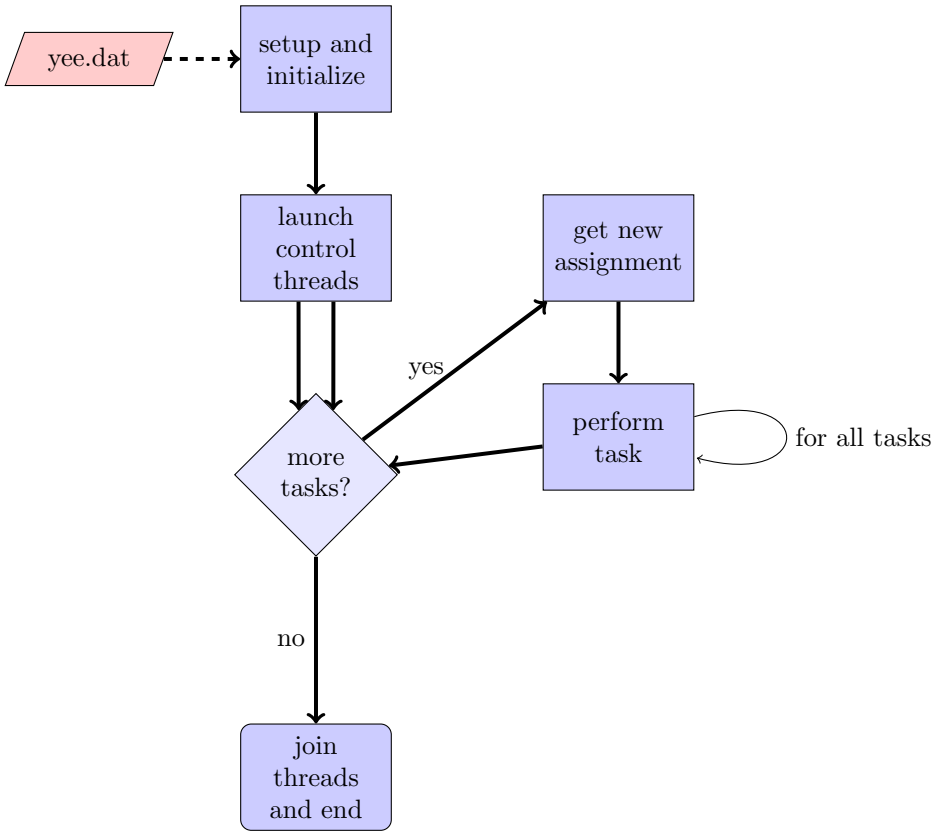

Figure 4.1: Scheduler Flow Chart

4.1.2 Plot Script

For creating graphs used with this implementation a small Python script utilizing matplotlib[11] was created. This script accepts data from "csv" files created by Yee_bench CUDA. These "csv" files are column based, where column one is the problem size or X-axis on the graph, column two is the Y-axis describing the performance in megaflop per second, and the optional third column can be used to create error bars in the Y direction. For and in-depth description look at Section A.2.

4.1.3 Computation Implementations

There are two sets of computation implementations, one for CPU and one for GPU, the GPU version is implemented in CUDA and is described in length in Chapter 3.

The program can be launched with any number of CPU or GPU threads. This is specified in the configuration file. The CPU version is based on the original C implementation of Yee_bench. The Kernels themselves are based off of the calculation method "5", which use fused for loops for the computation algorithm. The other CPU kernels are still in the code base but are not actively in use, this can be modified by changing the "calcmet_used" field in the task struct.

CPU parallelization of a single task is implemented by adding an OpenMP "for" statement before the outer "for loop" in the CPU kernel, and a "parallel" statement before the launch of this kernel. The number of OpenMP threads to be used can be specified in the configuration file that contains all the launch specific numbers. This is not an automatic number for a number of reasons, firstly allowing the user to change it to test out different configurations, but most importantly to avoid throttling the total performance by capping out all the CPU resources. CPU parallelization of a task is slightly better than parallelizing several tasks at once on the same system.

4.2 Assignment Scheduler

The assignment scheduler is based on a simple load balancing scheme. It is implemented using Pthreads, a POSIX standard for threading and mutexes to control the flow of the program. A mutex or mutual exclusion is a structure to make sure only one thread in a parallel implementation can perform a section of the code at one time. These mutexes can also be used as barriers or signals in the code to make one thread wait for an event. The flow of this scheduler can be seen in Figure 4.2. Orange boxes signify a task done by the control thread, blue boxes signify one of potentially many threads[1].

[1]The work performed in the "run task" node is as shown in Figure 3.1 in Section 3 and is the same for both GPU and CPU.

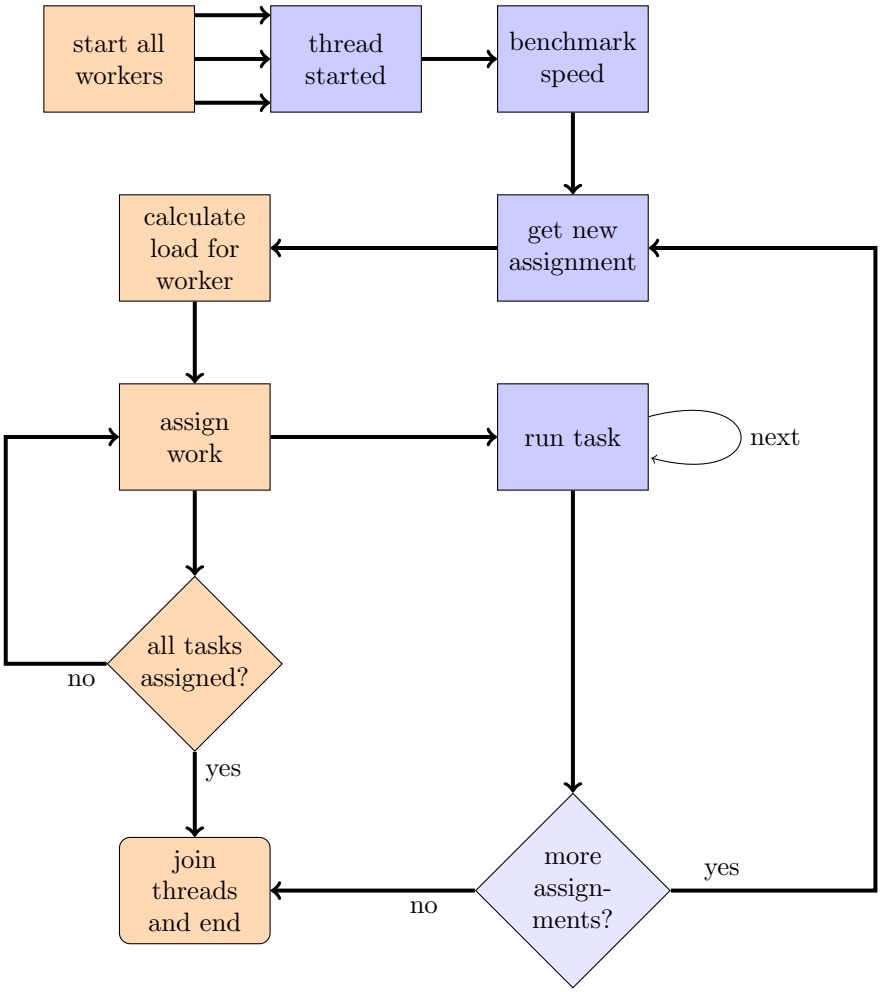

Figure 4.2: Assignment Scheduler Flow Chart

The program starts off by reading problem size parameters and CPU/GPU combination from the configuration file. A number of tasks will then be generated based on these attributes.

When this has been done, the Pthread worker threads are created and launched. While the worker threads initialize and benchmark their speed, the host thread waits. When a worker is ready it sends a signal to the host, telling it to calculate

the load it should need to be busy for X seconds. After the host has assigned work to a thread, it gives them a ready signal.

At this point the worker has received a list of task IDs based on the performance reported by to the host. Each worker will wait until these tasks are done before requesting new assignments.

If a worker receives a workload of zero tasks it means that the task list is empty and it will end, the first worker to perform this action will notify the host, which will go into the thread joining state and wait for all others workers to end as well.

The design has been implemented with the possibility of easily porting it to distributed memory and message passing frameworks such as OpenMPI in mind. With one host thread assigning work and keeping track of the others, while the workers don't directly contact the host to get new work, they communicate indirectly through shared memory.

4.2.1 Load Balancer

The load balancer was originally inspired by the concept used in "Evaluation of Likelihood Functions on CPU and GPU Devices"[13]. In the problem described in the likelihood article, parallelization is done by splitting the domain of a big problem into partitions and balancing the devices to share the work of an iteration. In this setting the distribution is started out even among devices and an iterative process is started to converge on a good distribution.

To create a balancer that would work in our environment, with potentially varying workloads per task, relative power of devices is not that important if you are instead balancing towards a static entity such as time.

In our case a single workload or task is performed on it's own and therefore the load-balancing does not need to be iterative like the original. To determine each device's compute power, a task of the same size as the first scheduled ones is run at the start to determine the initial speed S_k^0 for device k.

All tasks are ideally computed at the same time as running devices k_0 and k_1 concurrently will affect their results. However in most cases they will not be accurate for this first benchmark, as one device will complete before another and skew the numbers. We will see that this is not really a problem.

Each time a worker requests new tasks, the host takes the current speed S_k and calculates the target flop count F_k for the next batch of tasks.

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

The host then starts building a list of tasks for the worker, each with a load F_{task_i} until at least F_k load has been assigned.

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

The current load balancer could also be defined from the slowest device. This was the basis for the original balancer, but by the time it was completed a simple balancing towards a static time seems more stable.

A relative power balancer would be implemented as follows. After all devices have reported back their speed, the slowest device is identified. For K devices

$$S_{slow} = \min[S_0^0, S_1^0, ..., S_K^0]$$

Now each new partition can be computed for the next number of tasks to be performed

$$N_k = \lceil N_{slow} \times \frac{S_k^t}{S_{slow}} \rceil$$

where N_{slow} is the load to be performed by the slowest device before getting a new assignment.

4.2.2 Detailed Description of the Implementation

The main file is quite large, close to 1300 lines of code. This is mainly due to the fair amount of global variables combined with several fairly similar functions. All the five mode specific functions are in base fairly similar, they all follow the same style and structure. The two different worker functions for CPU and GPU are also quite similar. All scheduling or benchmarking is similar between the two worker functions, with two exceptions.

For the heterogeneous benchmark, the CPU thread has an “if” at the end, it checks whether the specified amount of CPU tasks have been performed, it will then exit if it has. The GPU counterpart will check a variable to see if all CPU threads have retired yet, if so, it will exit. The other exception is for the greedy scheduler, the CPU function will test if the number of remaining tasks is very low compared to it’s own performance, this is not so important as this scheduler is only meant for testing.

Data relating to a task is stored in a struct, see Listing 4.1. The first grouping of variables relate to values read from the configuration file. The second is the data arrays of the electric and magnetic fields respectively. The third grouping relates to task size and performance, while the fourth and final grouping is information about how it was run, and weather it has run yet (or did at all in the case of some of the benchmarks).

Listing 4.1: Data Struct Containing a Task

```
/* Contains data for a task */
typedef struct {
  int nts;
  int nx, ny, nz;
  int dx, dy, dz;

  my_float *ex, *ey, *ez;
  my_float *hx, *hy, *hz;

  int bytes;
  double flop;
  double mflops;
  double t[TIMERS];

  int id;
  int workerid;
  int done;
  int calcmet_used;
} task_data;
```

When launching Yeebench_CUDA the parameter will be parsed and the type of mode to run in in will be stored in the int "state" which reflects the enum in "main.h". This will be used in the worker threads to control how they acquire work.

```
/* Enum for run type */
enum { GREEDY=0, ASSIGNMENT=1, PLOT=2, HPLOT=3, RESOURCE=4};
```

4.2.3 Flow

Flow charts and overall description already exist in Section. 4.2.

First off the number of tasks specified in the configuration file is allocated as an array of the data structs specified above, the values read from configuration are also written to the tasks in this array. Arrays for worker performance and worker assignments are then allocated as well.

The number of CPU and GPU worker threads specified is then launched after all mutexes are initialized, three mutexes in total exist.

- "pmutex" - The primary mutex that governs the workers, it is there to make sure only one worker is communicating with the scheduler at a time. For a worker to request more work it needs to make claim to this mutex.
- "hostmutex" - This mutex works as a sleep interrupt for the scheduler. The host will start off polling this mutex. When it is unlocked, it will read the value in "requesting_worker", calculate the work for this worker to perform next, and go back to wait for "hostmutex" again.
- "tmutex" - When the host is done allocating work to a thread, it will unlock this mutex which the worker thread initially started waiting on after unlocking the "hostmutex".

Listing 4.2: Scheduler on Host

```
1  while (true)
2  {
3      /* Wait for worker threads */
4      pthread_mutex_lock (&hostmutex);
5      /* if requesting_worker is -2 it means all work has been assigned, so end */
6      if (requesting_worker == -2)
7      {
8          /* unlock to make sure any workers don't hang */
9          pthread_mutex_unlock(&tmutex);
10         break;
11     }
12     assign_workload(requesting_worker);
13     /* Signal the worker that the scheduler has done it's scheduling */
14     pthread_mutex_unlock(&tmutex);
15 }
```

In Listing 4.2 we see the host side of the scheduler with the work assignement in Listing 4.3, in Listing 4.4 we have a cutout of the worker-thread function that requests work to be done.

Listing 4.3: Work Assigner

```
1  void assign_workload(int worker_id)
2  {
3      if (next_task >= number_of_tasks)
4      {
5          printf("#HOST# All assignments assigned already\n");
6          /* Tell worker it has zero assignments to perform */
7          TaskNum(worker_id) = 0;
8          return;
9      }
10     /* Target flop for a worker to perform
```

```
11      * will try to make it as close to but at least
12      * SECONDS_TARGET seconds
13      */
14     double flop_target = workerperf[worker_id] / (((double)tasks[next_task].nts
            -1.0)/(SECONDS_TARGET*1000000.0));
15     printf("target for %d is %f\n", worker_id, flop_target/1000000);
16     double flopassigned = 0;
17     int tasks_assigned = 0;
18
19     while (flopassigned < flop_target && tasks_assigned < MAX_TASKS_PER_THREAD &&
            next_task < number_of_tasks)
20     {
21         /* Assign task */
22         TaskId(worker_id, tasks_assigned) = next_task++;
23         /* Keep track of how much work there is in the tasks we have assigned so
               far */
24         flopassigned += get_flop(tasks[TaskId(worker_id, tasks_assigned++)]);
25     }
26     /* How many tasks in the tasklist for a worker */
27     TaskNum(worker_id) = tasks_assigned;
28     printf("#HOST# Assigning %d jobs to worker %d\n", TaskNum(worker_id),
            worker_id );
29 }
```

At the start of a worker thread, it runs a small job of similar problem size to the first job in the task list. It will not run the full length of it, only 40 iterations of the normal 200+ iterations. They will then start requesting work from the scheduler by using the mutexes described above. This whole process is quite similar to message passing, in message passing we would replace the mutexes with messages containing the relevant data instead. This data would be worker id for the request, and the response a number of tasks with a list of tasks. These tasks would include their individual task-data instead of reading these values from a global array.

Listing 4.4: Task Acquisition on Worker

```
1
2  /* If the worker has completed all the currently scheduled tasks */
3  if (tasks_performed_for_assignment == tasks_assigned)
4  {
5      pthread_mutex_lock (&pmutex);
6
7      // Send signal to host
8      workerperf[threadid] = lastperf;
9      requesting_worker = threadid;
10     pthread_mutex_unlock (&hostmutex);
11
12     // Barrier to wait for host to assign
13     pthread_mutex_lock (&tmutex);
14
```

```
15      tasks_assigned = TaskNum(threadid);
16      tasks_performed_for_assignment = 0;
17      pthread_mutex_unlock (&pmutex);
18  }
19
20  /* Next task to perform */
21  my_task = TaskId(threadid, tasks_performed_for_assignment);
```

If a worker gets an assignment of zero tasks to perform it means the scheduler ran out of tasks to run. This worker then sets the "requesting_worker" variable to -2, -1 denotes no threads have yet requested work. The worker will also unlock the host and exit the thread, the host will read the "requesting_worker" variable, end the assignment phase and start joining threads. When other workers end their assignments, they will also read the "requesting_worker" variable, see it's value and exit. At the end, the host thread will present performance results based on the tasks run, broken down by CPU and GPU as well as total performance.

4.3 Other versions

As mentioned in the intro a few other version exist that perform their own small roles, mostly for presenting results or as part of the development cycle.

4.3.1 Greedy Scheduler

The greedy scheduler is the initial version and a very simple implementation. In fact no real scheduling is done at all. All threads are started in parallel at the start, when the host thread releases the mutex each thread will in order select the next available task until the task list is empty. After all threads have joined up the performance for the run will be listed, including timings for each part of the task; setup, compute, data copy and total for both CPU and GPU.

It was implemented with a simple mechanism to halt the slowest devices (CPU threads) slightly before all tasks are performed as to not let some slow CPU tasks run alone and ruin the average.

This version will produce fair output for performance but has some obvious drawbacks, as individual tasks will end at close to any given moment. This will negatively affect the reported performance, only for lengthy runs will this negligible. In other words to produce results for a single problem size the scheduler would have to run for as much as ten to thirty minutes.

Also it does not produce output on it's own, like the main scheduler it requires you to run it manually for all problem sizes. This is where the device plotters come in.

4.3.2 Homogeneous Device Plotter

The homogeneous device plotter is made to benchmark single devices, allow parallelizing of a single task and create plots of the performance over a wide problem size domain. It requires you to only launch the program with one device active, or it will give incorrect data. The homogeneous plotter will also produce undefined performance output for devices parallelizing several tasks at once. The heterogeneous device plotter could in theory do the same work but it will not for GPU plotting as it is based on matching CPU workloads.

The plotter starts off by creating tasks equal to the number of tasks specified in the code, splitting the problem size equally among these tasks and running them in sequence on the available resources specified. At the end of the run it will output data to a CSV file and plot it using the Python plot-script.

4.3.3 Heterogeneous Device Plotter

The heterogeneous device plotter based around the assumption that the CPU workers are slowest and closely match each others speed. Like the previous plotter it runs tasks equal to the number of tasks specified in the code, splitting the problem size equally among these tasks. However it will create several tasks per problem size, it will schedule a specific number of tasks per CPU worker and constantly resupply the GPU workers until the CPU workers are done. The uncertainty is larger than desirable when the number of CPU tasks per problem size is only one, but already at two the error margin is low.

Depending on the number of task level parallel threads per device the relative performance will vary. Even with close predictions of matching the number of tasks run per device, some variations will occur on when each device will end it's work. To compensate for this, error estimation was added to this plotter to close the gap.

Listing 4.5: Error Correction of Hybrid Device Plotter

```
1 double extraflop = 0;
  // Sort data of the last tasks performed with the oldest first
3 qsort(ending_data, nthreads, sizeof(enddata), cmp_enddata);
  double modifier;
```

```
for (int j = 0; j < nthreads - 1; j++)
{
    if (ending_data[j].calcmet == 7)
            modifier = 0.4;
        else
            modifier = 1.0;
    // Difference in time from when it ended vs the entire iteration
    double diff = ( ending_data[nthreads-1].end_time - ending_data[j].end_time ) /
          ending_data[j].tot_time;
    extraflop += diff * ending_data[j].flop * modifier;
}

mflops = (extraflop/2+flop) * (timesteps-1.0) / (run_time*1000000.0);
est_error = extraflop/2 * (timesteps-1.0) / (run_time*1000000.0);
```

As shown in Listing 4.5 the timestamps for the last task performed by each device thread is collected. Error correction is added by modifying the amount of work to have been performed to emulate the last task of the device thread continuing to work until the end of the last. This is then halved and added back as an error bar. As in the case of CUDA threads, the performance of other threads ending will improve the performance for the rest, therefore the error correction has been reduced to 40% for these, meaning only 20% in the end after cutting it in half.

This estimation will be visualized with the lower end of the error bar being where the performance was actually measured, and the top of the error bar signifying where the performance would have been in a best case scenario. This is illustrated in Figure 4.3. The top performance is notably unlikely as the last tasks running might perform better than they otherwise should as they are competing less for the same resources. In some cases the first task to actually get started might have slightly better performance, but this does not affect the error estimation done at the end. In most cases the error would not impact the performance too much if the benchmark is launched with reasonable parameters and resources.

4.3.4 Resource Plotter

A final mode of the benchmarks is the resource plotter. This version will utilize the problem size listed in the configuration file, create a number of tasks to schedule and try out different combinations of CPU threads and GPU threads to find the optimal combination. This is also meant as a predecessor to self tuning for the scheduler.

The overall flow of this plotter is the same as for the heterogeneous plotter of the previous section, with difference that now instead of varying the problem size it

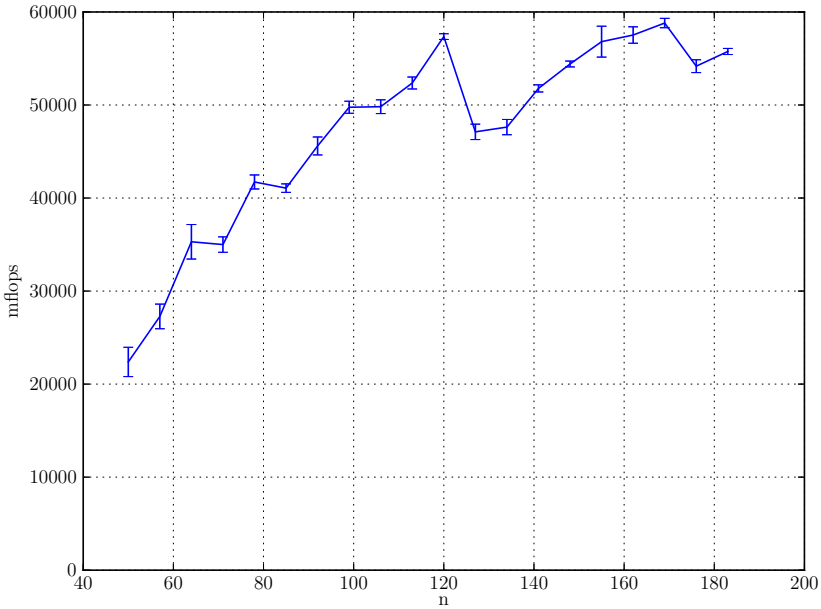

Figure 4.3: Error Bar Example

varies thread counts for CPU and GPU. To look at all reasonable configurations of threads does take it's time however, and might prove difficult in a production environment if set up incorrectly. This is especially true if the need to increase the number of CPU tasks before it rests is high.

A specific weakness to this model is that the results produced may be sufficiently different for different problem sizes for this to cause you to make the wrong decision. This problem arises from the fact that different hardware hit different performance peaks and dips at different times. Combination X might also cause the different compute units to affect each other at other points in the execution than combination Y. Usually the difference is within a fair distance, and between the optimal configurations the performance is close.

An example of this plotter is Figure 5.2 in Section 5.1.

Chapter 5

Results

This chapter describes the results of the implementation, the CUDA implementation as well as the overall scheduler. Yee_bench does not use an accurate representation of actual CPU or GPU performance. It is a synthetic performance based on the actual work of the FDTD method. The performance in MFlop/second is the performance of the implementation relative to the method, internal operations, addressing or partial calculations are omitted. This is based off of the original memory operations of the CPU implementation, which in theory should be the same for the GPU implementation.

5.1 CPU Performance, Resources Used Compared to Results

The relative performance of the CUDA implementation compared to different levels of CPU parallelization can be seen in Figure 3.5 in Section 3.4. From this we can see that CPU parallelization adds a decent amount of performance but small compared to the CUDA implementation when approaching the core limit.

In Figure 5.1 we see the performance of different number of OpenMP threading on our quad core processor for different even cubic problem sizes.

If we look at performances for $N = 150$, we get Table 5.1, this is a relevant problem size just short of the performance peak of our GPGPU implementation at $N = 153$. This area also hits right before the performance starts dipping

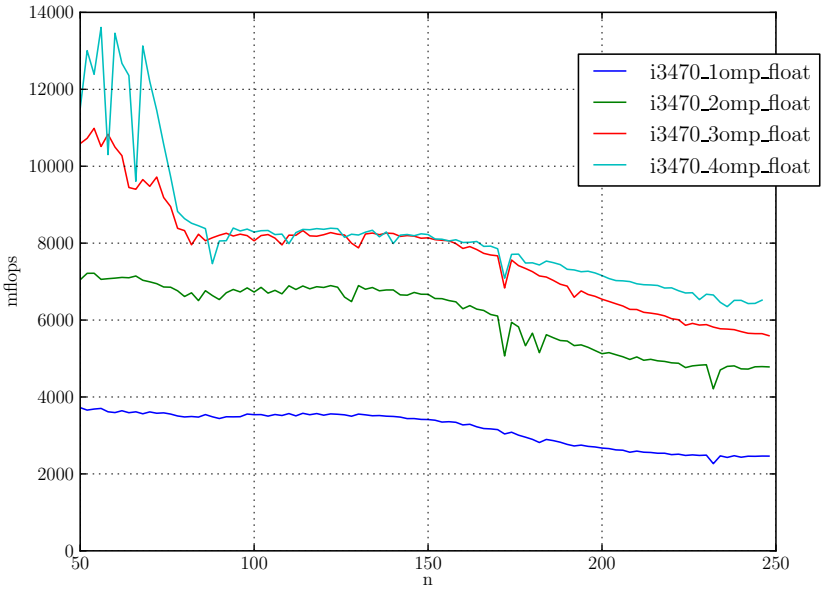

Figure 5.1: CPU Speed for Main Loop

on the CPU implementation as well, this is likely due to inner data structures getting pushed out of L2 cache. According to valgrind[16] the miss rate for L2 and L3 together was only very slightly higher at 250 compared 150, this further suggests it's a L2 cache issue. We see that the performance gain from one to two cores is almost doubled, from three to four is a 22% increase, and three to four almost negligible. This is due to the already discussed memory restrictions of this method. It appears that the memory buss reaches it's limit when using 3 threads. This gives reason to suspect that using all available CPU cores might not be beneficial.

Table 5.1: Performance for N = 150 on CPU

	1 CPU	2 CPU	3 CPU	4 CPU
MFlops	3417	6669	8138	8223
Speedup from 1 CPU	1.0	1.95	2.38	2.41

When we lower the CPU frequency from 3.2Ghz to 1.6Ghz we observe some interesting results. This is listed in Table 5.2. At this frequency using multithreading

is almost linear, using three cores is almost three times the speed of a single core. Even when using all four cores we get a speedup of 3.59 compared to one core. This confirms our earlier claims that the memory is the limiting factor.

Table 5.2: Performance for N = 150 on CPU, 50% of Maximum Frequency

	1 CPU	2 CPU	3 CPU	4 CPU
MFlops	2136	4220	6246	7674
Speedup from 1 CPU	1.0	1.98	2.92	3.59

Figure 5.2 gives us a quick look at the performance of different combinations of resources allocated for the benchmark at a set problem size. This graph was created with the benchmark described in Section 4.3.4. In this case we have $N_x \times N_y \times N_z \times = 150 \times 150 \times 150$ and we see that two OMP threads and three CUDA streams works best. This is however is not universally true for the entire problem size.

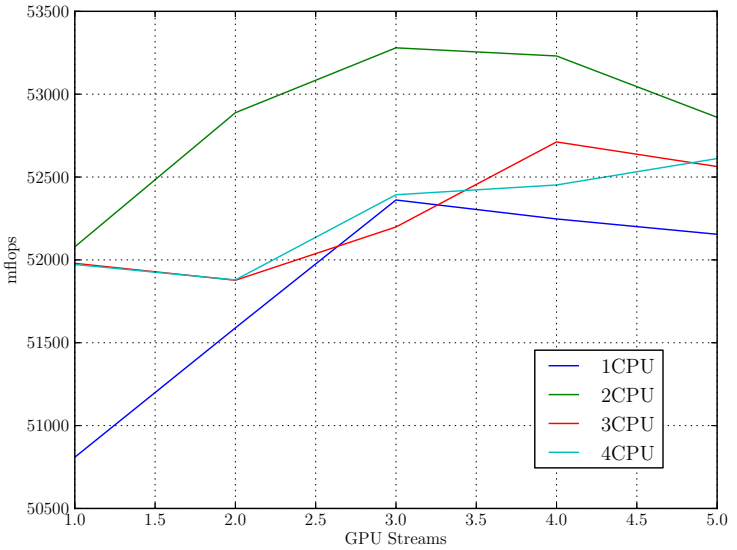

Figure 5.2: Performance with Different Combinations of OpenMP Threads and CUDA GPU Streams

For the most part 2 OMP threads turns out as the best option for CPU configuration, and in the events it isn't the difference is fairly small. It is also worth

contemplating that if using all CPU cores only rewards you with perhaps a single percent performance increase at some problem sizes, this will use a fair bit more energy. It will use more energy on the CPU, and also for cooling. If the system is used for anything else in the background, be that other intentional tasks or implicit operating system tasks. Utilizing all available resources in this setting will most likely have a much larger impact on the performance than if only some of the resources are used.

5.2 Memory Implications

The memory requirement for Yee_bench is cubic, for 32bit it is $12N^3+12(N+1)^3$ bytes. There are three H arrays of floats with the size N^3 and three E array of floats with the size $(N+1)^3$, where one float is 4 bytes. This reaches the limit for our NVIDIA 480 GTX at 1535MB at N=405 if all dimensions are of the same size.

$$12n^3 + 12(n+1)^3 byte = 1535MB \times 1024^2 byte/MB$$

$$n = 405.786$$

We will instead look at some numbers where the Z dimension is locked at $N_z = 150$, as this is the most common case for EMGS. Equipment to measure the fields in the seabed rarely go beyond this distance.

$$12n^2 \times 150 + 12(n+1)^2 \times 151 byte = 1535 \times 1024^2 byte/MB$$

$$n = 606.040$$

Interesting dimensions in the X and Y directions for EMGS are mostly within the $N = 100...250$. We see that this is well within the available memory on our device and most devices in the same category of the future. But running simply one task at a time is not the full scope of our problem. We want to run multiple tasks at once, and schedule many more while others are still in memory. A "large" task would then be of the size $250 \times 250 \times 150$ with a memory print of

$$12byte \times 250^2 \times 150 + 12byte \times (250+1)^2 \times 151 = 216.16MB$$

The total number of concurrent tasks in memory of our GPU would then be

$$\frac{1535MB}{216.16MB} = 7.1$$

From our experience the best performance is achieved when using two GPU streams which from our calculations is well within our potential. When one tasks is done on the GPU the data is copied back to main memory before it is released and another task started. For main memory we have 16Gb in total leaving us available to use

$$\frac{16GB \times 1024MB/GB}{216.16MB} = 75.8$$

75 tasks if no other memory is occupied. Obviously it is but this a considerably large number giving us freedom to keep many tasks in memory. Our benchmark or scheduler does not take into account storing the results to disk or otherwise.

5.3 Performance in Main Loop and in the Big Picture

Throughout this thesis performances are listed as "main loop" performance when not looking at a scheduler. This is due to the original Yee_bench implementation listing performances as such. This is the performance of the method on a device within the main loop of the program, and doesn't list the overhead there is for a task. A task has an initial setup of allocating memory, deallocating memory at the end and for GPU tasks copying data back to main memory. These times are necessary to include to get the full impact of running one task.

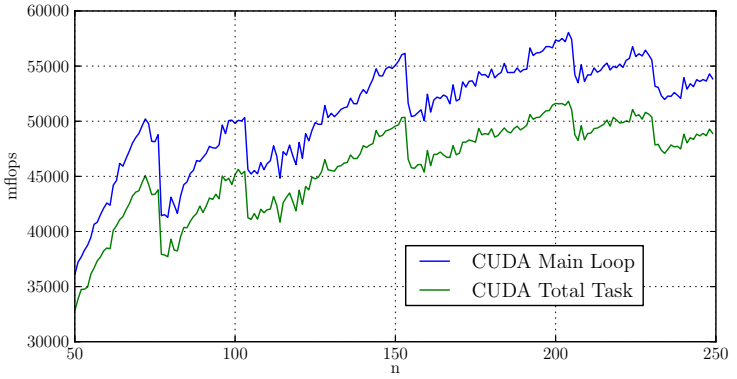

Figure 5.3: Main Loop vs. Total Time for One GPU Stream

With relatively few iterations compared to many other physics or mathematical problems, and at smaller problem sizes this overhead will be considerable for the total time. An example of this is seen in Figure 5.3 where $N_x \times N_y \times N_z = 250 \times 250 \times 150$.

5.3.1 Runtime Breakdown

The total runtime of a task is divided into four parts. Setup deals with calculating parameters used in the FDTD method based on constants and problem size, as well as initializing memory to be used. Compute contains the most intensive part with the calculations taking place. For the CUDA implementation the act of copying data back to main memory takes some time. After all these steps are completed there is overhead for all actions within the life of the task execution as well as deallocating the memory used for this task, and example run can be seen in Table 5.3.

Table 5.3: Time Spent in Each Stage of a Task. With N = 150, 1 CPU thread with 2 OMP threads, 2 GPU threads with their own streams

	CUDA	**CPU**
Setup	0.035 sec.	0.032 sec.
Compute	0.853 sec.	3.537 sec.
Copy	0.044 sec.	0.000 sec.
Other	0.014 sec.	0.022 sec.
Total	0.946 sec.	3.611 sec.

For CPU we can observe that the total time spent outside compute is less than 1.5% of the total time and there is no copying. In the case of the CUDA implementation we have some copying taking place, but this is unavoidable. The time spent in setup and other is for CUDA about 5%, considerably more than CPU as the total runtime is far lower.

A scheme of having the scheduler preallocate the memories to bring down this effect was considered. For the CPU tasks we observe that this effect would be of little use to us, but the CUDA version does slow down considerably because of this dilemma. However, when we utilize two CUDA streams, the overhead time consumed dealing with memory is pipelined parallel between the streams. When one stream is performing memory copy or allocations the other stream will benefit from the available compute resources.

5.4 Scheduler Performance

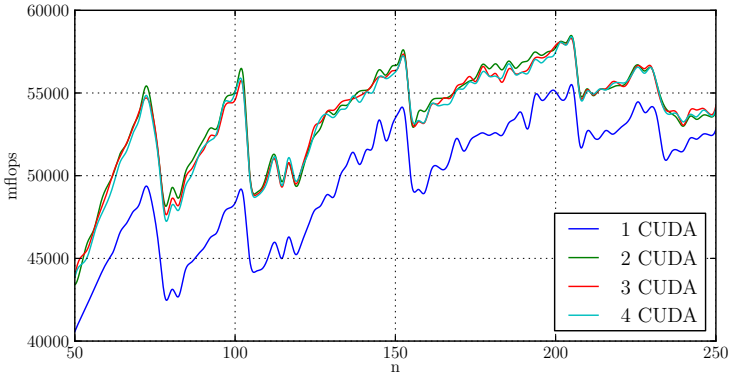

Figure 5.4: Performance with Different Number of CUDA GPU Streams for 2 OMP Threads

In Figure 5.4 we present full results when using different number of GPU streams, again we have $N_x \times N_y \times N_z \times = 150 \times 150 \times 150$. From this we see that the performances are in fact quite similar for most number of combinations. For large N we can observe that 2 GPU streams perform the best. At small N utilizing more GPU streams to benefit from parallelizing the overhead of copying data to host as well as other memory operations and continually launching many small kernels is beneficial.

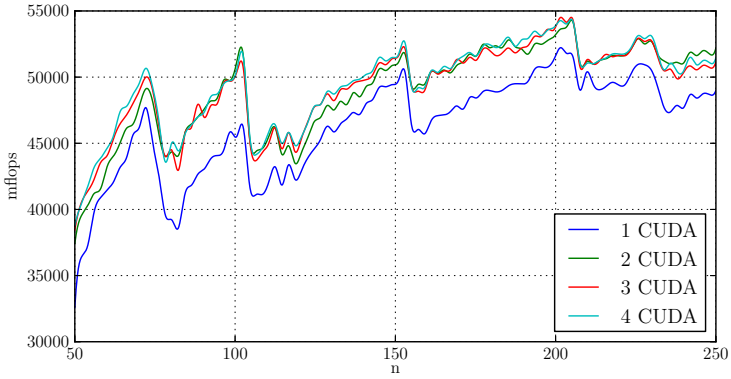

Figure 5.5: Performance Comparison for Different Number of CUDA Streams

From what we've previously seen, as shown in Figure 5.5 the performance of two concurrent CUDA streams compared to only one improves the results. When one stream has downtime doing memory operations or other CUDA overhead, the other stream will keep the GPU busy.

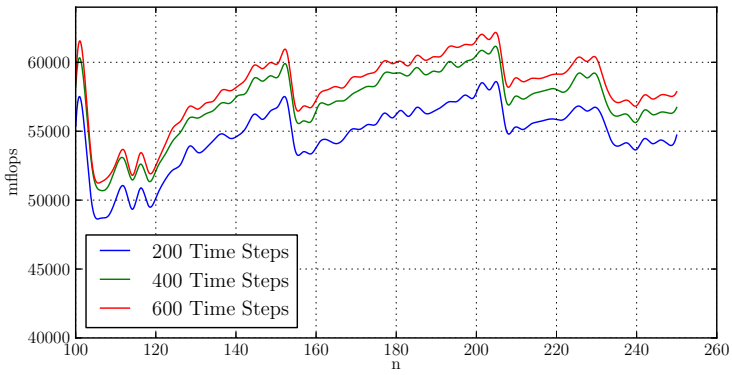

Figure 5.6: Different Number of Iterations for 2 OMP Threads and 2 CUDA Streams

So far we've mostly looked at the standard case of 200 time steps when running as this is what Yee_bench originally used. The number of time steps or iterations can go well beyond 200 depending on the highest resistivity value in the FDTD model. Runtime for with only 200 iterations is as we've seen so far fairly short. At $N = 150^3$ for example it's still less than a second even when running two parallel GPU streams. In Figure 5.6 we see look at longer runs with more iterations. At 400 iterations we get a fair performance boost, a slight improvement is alos observed at 600 but we are approaching the point where overhead seems to be minimalized. If we were still using only one GPU stream we could probably still benefit from more iterations.

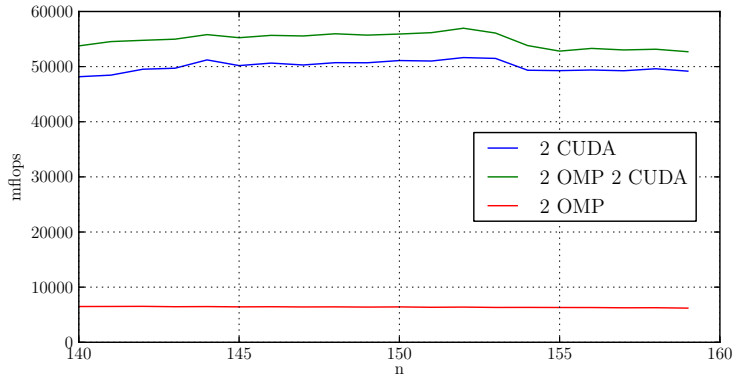

Figure 5.7: Performance Comparison for CPU Alone and Together With 2 CUDA

From Figure 5.7 we see that using CPU resources in addition to the GPU improves total performance. This is what we desire in a heterogeneous scheduler. The total performance is understandably not the same as adding them together as one will impact the other. The decline in CUDA performance with CPU load is not related to lower GPU performance, however the kernel launches and system memory operations will affect the GPU to have longer overall runtime[5], also transfering data to and from memory does affect CPU performance as well.

This difference is shown in Table 5.4, when we use all four cores we see a huge difference in individual performance for the devices.

We see the relative performance for each device when running alone and in combination, from before 2 CPU threads and 2 CUDA streams yields the best performance most of the times. We keep the Z-axis at 150 for all tests. It is apparent that the devices are competing too much for memory and CPU resources when we use too many cores. We also see that when we use two CPU threads, the performance for CPU is almost the same while GPU performance is slightly less forgiving.

Table 5.4: Relative Device Performance

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

5.5 Kepler Performance

We did some testing on the NVIDIA Tesla K20 GPU. Although the K20 is a lot better on paper, there is a downside. The architecture is not necessarily an improvement for all types of algorithms. There are fewer streaming multiprocessors, the memory is more optimized for coalesced reads and writes. The memory speed is is around 17.5% faster while the compute power is is almost tripled compared to the GTX 480 Fermi GPU.

With our algorithm, which is strongly memory bound, and with large amount of uncoalesced memory transactions this is detrimental to the performance as a whole. The performance on preliminary testing on the K20 card was around 5-10% less that our presented results.

It is possible that our implementation would work satisfactory on the Kepler architecture still, but more time would have to be put down into the work needed to get good results.

Chapter 6

Conclusion and Future Work

Scientific computations have been a cornerstone of many engineering businesses for quite some time, from the days of punch cards to the supercomputers of today. An emerging field in the past decade is the use of GPGPU processing power, originating as graphical pipelines to create accelerated 2D and 3D projections. These massively parallel processors are perfect for solving parallel and iterative algorithms and scientific problems. With today's ecosystem, a GPU can be several orders of magnitude faster than a CPU. With the possibility of adding several GPU's to a node, the overhead cost of a node compared to the performance unleashed by the processor drops while the speed increases as well.

The goal of this work was to implement a GPU solver for Yee_bench in CUDA, parallelize the CPU implementation and create a scheduler to utilize these. Yee_bench implements the basis of certain calculations done by EMGS in Trondheim to increase drilling success in search for hydrocarbons.

The CUDA solution on our three year old GPU did achieve good speedups compared our contemporary CPU. The erratic nature of memory addressing in the FDTD method however limits this compared to the great examples these days of with speedups past 100x. Yee_bench CUDA has a speedup of 7.5x compared to quad-core parallelization on the CPU and just shy of 19x for the single core implementation.

Speedup is mainly achieved by optimizing memory stores by addressing CUDA threads in execution order. This also improves loads but by getting elements in

different direction across different arrays is difficult to do optimally. These loads will have issues getting many coalesced values. With varying dimensions making sure enough threads are launched, and in a good order is important. Having the Z dimension fairly well situated around 150 guarantees a fair size of parallel CUDA threads. Especially below problem sizes of 100 the performance is very much all over the place as the problems are so small.

Using several CUDA streams to compute several FDTD tasks at once is important to saturate the GPU to take use of downtime during overhead. Considering the high amount of irregular data access and lack of explicit cache, the implementation might suffer greatly if executed on a device with compute capability below 2.0 as they have no L2 cache.

The scheduler does however reveal certain things when it comes to CPU and GPU being utilized concurrently. As noted at the very start the problem is memory bound, utilizing the fourth core in the CPU barely affects results at all, and reducing CPU performance shows the parallel nature of the method very well. The GPU performance dips when utilizing more CPU cores, and CPU performance is reduced to around 2/3 of it's optimal performance when the GPU is working. This shows us that the peak performance is not necessarily achieved when pumping all our resources at the problem.

CPU does not scale fully when on top of the GPU performance, but does add to the overall throughput. The GPU needs some CPU resources as well as main memory to schedule kernels and move data around. High CPU load delays some of this resulting in longer runtime even if the GPU performs at the same top speed. The system is more prone to performance loss if all resources are used to achieve the same speed or even lower in many cases. Modern Intel processors benefit from Turbo Boost Technology[12], meaning the serial performance is higher when fewer cores are active. This feature further gives credit to using fewer CPU cores.

6.1 Future Work

This work is meant as an exploration of GPU and heterogeneous solutions to the FDTD method. While we have looked at many relevant aspects there are still some issues to address or at least give some thought to. Some outside the scope of the work, but some work that was not possible to complete within the time.

6.1.1 Distributed Memory Version

This scheduler is a shared memory single node scheduler, however implemented with distributed memory in mind. Porting it to for example an OpenMPI solution should be fairly trivial, at least compared to the work required to implement the rest of the solution to put it into production.

6.1.2 Self Tuning

To be an operational scheduler it needs an important feature if the hardware across nodes are different. Either it identify what hardware it has available and make decisions for how to configure the thread configuration, or it will run self tuning live. It should at startup identify the number of CUDA devices, and launch the right number of streams relative to that.

From our results it might seem like a good idea to run a node with zero CPU threads and two CUDA streams per CUDA device. This is both simple and also most likely a very good configuration. Say we have as much as 4 CUDA devices on a node, the performance battle for main memory and CPU time for these is large. Optimally we might squeeze out 6-7000 MFlops from the CPU, but each GPU will put out 50 000-60 000 MFlops or more depending on the hardware. We quickly see that the performance gain, if it at all exists is not that useful on a heterogeneous node with this much GPU power. However in the case of a data center with many CPU nodes already in place, these old nodes may very well not have GPU power at all. We already discussed the uncertainty of live benchmarking, a decision making algorithm might very well be the best solution.

6.1.3 Stencil Size

Our version remains with the standard Yee_bench stencil, EMGS uses a slightly different one that wasn't implemented in this work. For larger stencils, or stencils with larger distances the performance will most likely change. If these new data elements are neighboring values the performance is likely to improve over the standard five point stencil used in this implementation. Further care when looking at the domain for the calculation would be needed as well as new formulas for the number of floating point operations per iteration.

6.1.4 Comparison of Different GPU Implementations

In this project the solution was created using CUDA to accelerate the code on GPU. EMGS has already received a version of the original Yee_bench in FORTRAN, modified for acceleration with OpenACC[22] PGI directives from NVIDIA. PGI is only one of several compilers which compile OpenACC "#pragma" directives to CUDA or OpenCL code, others include Cray, CAPS and Laguna.

These directive standards works in a similar fashion to the already established OpenMP[23] which simplifies multi threading of code segments. Using these directives instead of a full CUDA or OpenCL implementation simplifies the implementation and understandability of the code. The performance of implementations using these methods, as well as OpenCL implementations might be interesting for future development in the future.

6.1.5 Extension to Full EMGS Implementation

As of right now the project envelops only a non-complete implementation of the FDTD method explained earlier, the production implementation has a few more details to it, which could potentially also impact the design. Such issues would have to addressed and solved, and the algorithms wrapped in the existing program.

6.1.6 Alternative Maxwell Solvers and Redefined Mathematical Problem

The FDTD method is a solution of some Maxwell equations, several solvers exist. Yee_bench uses vector field iterations to solve them. Other solutions have been used to solve similar equations, for example FFT[6]. It could be interesting to see if our problem would be possible to redefine to fit such solutions as we already have highly optimized algorithms for them.

The possibility of redefining the problem to only use one type of field vectors has also been mentioned. Only one type is interesting as a final product, and the other is essentially a partial computation, used one at a time. Reducing the algorithm to only one type of field vectors would alleviate memory and potentially unnecessary memory writes.

6.1.7 Other GPU Architectures

As shown already, we did not have time to get a good version for the Kepler architecture. New considerations must be done, potentially launch configurations or kernel addressing need be changed to get good performance. This is a valid problem for any future architectures as well. Work on self tuning GPU algorithms across different GPUs and architectures is a field on it's own with on-going projects.

Bibliography

[1] International Energy Agency. Key world energy statistics, 2011. 1

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilitie, 1967. 6, 13

[3] Ulf Andersson. Yee bench – a pdc benchmark code, 2002. i, iii, 14, 18

[4] Ulf Andersson and Philip Mucci. Analysis and optimization of yee bench using hardware performance counters. *Parallel Computing: Current and Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, 33:179–186, 2006.

[5] M. Bobrov, R. Melton, S. Radziszowski, and M. Lukowiak. Effects of gpu and cpu loads on performance of cuda applications. 2011. 51

[6] Anne Cathrine Elster. *Parallelization issues and particle-in-cell codes.* PhD thesis, Cornell University, 1994. 16, 56

[7] EMGS. Official website. http://www.emgs.com, 2012. 1

[8] Anne C. Elster Erik Smistad and Frank Lindseth. Gpu accelerated segmentation and centerline extraction of tubular structures from medical images. NTNU, 2013. 68

[9] Rob Farber. *CUDA Application Design and Development.* Elsevier inc., 2011. 12

[10] Aleksander Gjermundsen. Cpu and gpu co-processing for sound, 2010. 2

[11] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. 31

[12] Intel. Intel turbo boost technology - on-demand processor performance. http://www.intel.com/technology/turboboost/, 2013. 54

[13] Sverre Jarp, Julien Leduc Alfio Lazzaro, and Yngve Sneen Lindal Andrzej Nowak. Evaluation of likelihood functions on cpu and gpu devices. *Journal of Physics: Conference Series*, 368, 2012. 34

[14] Daren Lee, Ivo D. Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W. Toga. Cuda optimization strategies for compute- and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, 106(3):175–187, 2012. 14

[15] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. Technical report, NVIDIA, 2009.

[16] Robert Walsh Nicholas Nethercote and Jeremy Fitzhardinge. Building workload characterization tools with valgrind. *Invited tutorial, IEEE International Symposium on Workload Characterization (IISWC 2006), San Jose, California, USA*, October 2006. 44

[17] NVIDIA. Nvidias next generation cuda compute architecture fermi, 2010. 9, 10, 11

[18] NVIDIA. From graphics processing to general purpose parallel computing. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2012. 7

[19] NVIDIA. Cuda profiler. https://developer.nvidia.com/nvidia-visual-profiler, 2013. 25

[20] NVIDIA. High performance computing, accelerating science with tesla gpus. http://www.nvidia.com/object/tesla-supercomputing-solutions.html, 2013. 11

[21] NVIDIA. History of gpu computing. http://www.nvidia.com/object/what-is-gpu-computing.html, October 2012. 6, 7

[22] OpenACC. Official website. http://www.openacc.org, 2012. 56

[23] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. 56

[24] Andreas Berg Skomedal. Gpu-accelerated seismology using cuda. NTNU, 2012. 5, 21

[25] N. Chavannes T. P. Stefański and N. Kuster. Parallelization of the fdtd method based on the open computing language and the message passing interface. *Microw. Opt. Technol. Lett.*, 54:785–789, 2012. 7, 8

[26] Thor Kristian Valderhaug. The lattice boltzmann simulation on multi-gpu systems, 2011. 2

Appendices

Appendix A

User Manual

This section is a description of how to use the benchmark / scheduler and specifics to it's implementation.

A.1 How to Run the Scheduler

As described in Section 4 the program has five different versions available. Running one of them is simply down to the argument passed to Yeebench_cuda at start. An entry in the makefile also exists for each of the versions

Table A.1: Runnable Makefile Targets

Number	Name	Program Parameter	Makefile Target
1	Assignment Scheduler	assignment	run
2	Greedy Scheduler		run_greedy
3	Homogeneous Device Plotter	plot	plot
4	Heterogeneous Device Plotter	hybridplot	hplot
5	Resource Plotter	resource	resource_plot

There is a compile flag in "main.h" that denotes running it in verbose mode

```
/* Run the benchmark with verbose output */
#define VERBOSE
```

This will output more information while running, similar to that of the original Yee_bench but with some extra information especially connected to running the CUDA versions. If run in verbose it will output a checksum for each calculation based on the E_z array. This should be sufficient to see if the output is correct but with single precision it is slightly inaccurate and so the numbers will not match 100%.

If more certainty to the correctness is needed there is also a compile flag in "main.h" for this.

```
/* Run thourough check on GPU output,
   requires that output from gpu and cpu are in tasks 0 and 1 */
#define gpucheck
```

The requirement is simple to achieve, simply set the number of tasks to two, set the thread configuration to be one CPU control thread and one GPU thread. When running the greedy scheduler the two outputs will reside in tasks 0 and 1.

Each version does however have some requirements connected to the values in the configuration file "yee.dat".

```
250 250 150     ! N_x, N_y, N_z: problem size (called nx, ny and nz in code)
1.0 1.0 1.0     ! Delta_x, Delta_y, Delta_z: cell size (dx, dy and dz in code)
200             ! N_t: Number of time steps (called nts in code)
1               ! Number of CPU Threads
2               ! Number of OpenMP Threads
3               ! Number of CUDA Streams
200             ! Number of Tasks to perform
```

The three benchmark versions listed in this section will produce an output file called "data.csv" which is a tab separated value file. This file can be used together with the plot script described in Section A.2.

A.1.1 Schedulers

For the two schedulers you will get what you ask for in the configuration file. Set problem size, set number of tasks run on the set number of resources given. There are no requirements for the deltas or the number of time steps, however they will affect the runtime and results given.

They will both produce similar output at the end, listing how many tasks each of the CPU and GPU performed.

```
Total   Launchtime = 44.797 sec. (0:44:797)
Total   MFLOPS     = 57540.79 MFlops/s

------------------CUDA-TASKS- 55-------------------
Setup           0.050 sec.
Compute         1.477 sec.
Copy            0.070 sec.
Other           0.019 sec.

Average MFlops 26601.940598

------------------CPU--TASKS-  5--------------------
Setup           0.058 sec.
Compute         8.276 sec.
Copy            0.000 sec.
Other           0.042 sec.

Average MFlops 5129.412769

Run completed successfully
```

But also underway, the only difference is how they acquire work.

A.1.2 Homogeneous Device Plotter

This benchmark will require only one device to be activated or the numbers will not be useful. This means one CPU thread or one GPU thread, several OMP threads however will still work. It is possible to use this benchmark with for X times one device and then simply multiply the results by X, but this will most likely be unreliable.

In order to specify the range of problem sizes calculated by this benchmark a combination of problem size and number of tasks is used. The Benchmark will plot from $N_X = 50 N_y = 50$ to the values specified in the configuration file as N_x, with number of data points in between being the number of tasks. In the current version N_z is locked at what is specified in the configuration file for all plot values, but this can be changed. If the number of tasks is greater than the number of actual integers between 50 and what is specified, the benchmark will still continue to perform this many tasks, but each data point will always be at least one greater than the previous. In other word if the configuration file says $N_x = 55$ but the number of tasks is 7, the benchmark will plot for

$N = [50, 51, 52, 53, 54, 55, 56]$.

A.1.3 Heterogeneous Device Plotter

This benchmark will try to plot data similar to the output of an optimal scheduler.

The requirements are the same as the homogeneous plotter for problem sizes and will behave in the same manner. The goal for this benchmark is to plot the coexistence of the different hardware together and therefore any number of resources will work unless the system runs out of resources. In the function named "cpu_thread(...)" of "main.cu" there is an entry at the top with

```
int plot_precision = 2;
```

at the top. This specifies the number of tasks a CPU thread will complete before resting. When this number is one the error estimation is slightly higher but can be used instead for a faster benchmark.

A.1.4 Resource Plotter

This benchmark will try to accurately display the performance for different number of CPU and GPU threads for one specific problem size denoted in the configuration file. There are no requirements for this benchmark.

The maximum number of resources it will test is specified at the top of the "resource_bench(...)" function of "main.cu".

```
int max_cpu = 4;
int max_gpu = 5;
```

The combination zero and zero will not be used.

A.2 Python Plot Script

The script requires python-matplotlib, latex and scipy. Latex can be avoided if the part about using the latex backend is switched back to it's default setting. Scipy is only needed for graph smoothing.

The python script for creating graphs works as follows.

```
$ ./plot.py --help

usage: plot.py [-h] [-f FILE [FILE ...]] [-a] [-d DIR] [-s] [-t FILETYPE] [-v]
               [-e] [-r] [-n FIRSTN] [-m]

Plot CSV files for yee_bench

optional arguments:
  -h, --help            show this help message and exit
  -f FILE [FILE ...], --file FILE [FILE ...]
                        specify csv input filename, default is data.csv
  -a, --all             plot all local csv files
  -d DIR, --dir DIR     directory for all argument
  -s, --save            save output to file, default is png
  -t FILETYPE, --filetype FILETYPE
                        file type for saved output
  -v, --verbose         be verbose
  -e, --errorbars       include error bars
  -r, --resource        test of resource allocation
  -n FIRSTN, --firstn FIRSTN
                        start the graph at n
  -m, --smoothing       graph smoothing, the number of m signifying the number
                        of consolidated data points
```

This can be used to graph data given by the benchmarks, remember to copy the output file and/or rename it to keep the results between runs. If the Makefile was used it will do this for you, for the first two benchmarks. to create a file name such as "1x2omp_4gpu_150x_150y_150z.csv". This means that it was run with 1 CPU control thread with 2 OMP threads, accompanied by 4 CUDA streams, run on the listed problem size.

Examples of using the plot script.

Create an Encapsulated PostScript file of all runs with 2 OMP threads run on one CPU thread.

```
$ ./plot.py -f 1x2omp_*.csv -st eps
```

View the graph of all local csv files.

```
$ ./plot.py -a
```

Create graph for the resource benchmark with error bars, when the file name has not been changed yet and then save it to a png file.

```
$ ./plot.py -rse
```

View smoothed graph of plots in a folder named “powersave”, average two and two data points.

```
$ ./plot.py -ad powersave/ -mm
```

Keep in mind error bars only work with version #4 and #5, as #3 does not create any error estimation.

Some things may still be desirable to change manually to get the exact graph you want, such as the minimum value for the Y-axis, positioning of the legend if the standard is not good enough and the total size of the graph. These settings are controlled by the lines in Listing A.1 respectively.

Listing A.1: Manual settings in Plot Script

```
plt.ylim(ymin=40000) # Y-min value
ax.legend(loc='center left', bbox_to_anchor=(0.5, 0.2)) # Location of the Legend
fig.set_size_inches(8.0,4.0) # Size of the graph
```

A.3 Comments on Kepler

The temporary results from Kepler are discussed in Section 5.5. To run on the K20 card a few things need be changed. In the Makefile, see Listing B.4, architecture must be changed to 35 from 20. Block size should be increased considerably as discovered by other projects[8], suggestions would be 32x32 or 32x16 from our 16x16, see Listing B.3.

Appendix B

Source Code

Some relevant source code to the project. Full source code will be provided by Dr. A. C. Elster.

B.1 Yeebench

Listing B.1: Configuration File

```
250 250 150  ! N_x, N_y, N_z: problemsize (called nx, ny and nz in code)
1.0 1.0 1.0 ! Delta_x, Delta_y, Delta_z: cell size (dx, dy and dz in code)
200 ! N_t: Number of time steps (called nts in code)
1    ! Number of CPU Threads
2     ! Number of OpenMP Threads
2     ! Number of CUDA Streams
200    ! Number of Tasks to perform
```

Listing B.2: CUDA Kernels

```
/* CUDA Kernels */
#define CUDAFILE
#include "main.h"

__global__ void updateH_cuda(my_float *cHx, my_float *cHy, my_float *cHz, my_float
      *cEx, my_float *cEy, my_float *cEz, int nx, int ny, int nz, my_float Cbdx,
     my_float Cbdy, my_float Cbdz )
{

        int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
              (threadIdx.x + blockDim.x * threadIdx.y);
```

```
9
10          int y = i/nz;
11          int z = i%nz;
12
13          if (!(y < ny && z < nz)) return;
14
15          for(int x=0; x<nx; x++)
16          {
17                  __syncthreads();
18                  HxC(x,y,z) += (EyC(x,y,z+1)-EyC(x, y ,z))*Cbdz +
19                                                  (EzC(x,y, z )-EzC(x,y+1,z))*Cbdy;
20
21                  HyC(x,y,z) += (EzC(x+1,y,z)-EzC(x,y, z ))*Cbdx +
22                                                  (ExC( x ,y,z)-ExC(x,y,z+1))*Cbdz;
23
24                  HzC(x,y,z) += (ExC(x,y+1,z)-ExC( x ,y,z))*Cbdy +
25                                                  (EyC(x, y ,z)-EyC(x+1,y,z))*Cbdx;
26          }
27  }
28
29  __global__ void updateE_cuda( my_float *cHx, my_float *cHy, my_float *cHz,
        my_float *cEx, my_float *cEy, my_float *cEz, int nx, int ny, int nz, my_float
         Dbdx, my_float Dbdy, my_float Dbdz)
30  {
31
32          int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
                (threadIdx.x + blockDim.x * threadIdx.y);
33
34          int y = i/nz + 1;
35          int z = i%nz + 1;
36
37          if (!(y < ny && z < nz)) return;
38
39          for(int x=1; x<nx; x++)
40          {
41                  __syncthreads();
42                  ExC(x,y,z) += (HzC(x,y, z )-HzC(x,y-1,z))*Dbdy +
43                                                  (HyC(x,y,z-1)-HyC(x, y ,z))*Dbdz;
44
45                  EyC(x,y,z) += (HxC( x ,y,z)-HxC(x,y,z-1))*Dbdz +
46                                                  (HzC(x-1,y,z)-HzC(x,y, z ))*Dbdx;
47
48                  EzC(x,y,z) += (HyC(x, y ,z)-HyC(x-1,y,z))*Dbdx +
49                                                  (HxC(x,y-1,z)-HxC( x ,y,z))*Dbdy;
50
51          }
52  }
53
54  __global__ void cleanupE_yz_cuda( my_float *cHy, my_float *cHz, my_float *cEx, int
        nx, int ny, int nz, my_float Dbdy, my_float Dbdz )
55  {
56
```

```
57        int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
              (threadIdx.x + blockDim.x * threadIdx.y);
58
59        int a = i/nz + 1;
60        int b = i%nz + 1;
61
62        if (a < ny && b < nz)
63        ExC(0,a,b) += (HzC(0,a, b )-HzC(0,a-1,b))*Dbdy +
64                                  (HyC(0,a,b-1)-HyC(0, a ,b))*Dbdz;
65  }
66
67  __global__ void cleanupE_xz_cuda( my_float *cHx, my_float *cHz, my_float *cEy, int
        nx, int ny, int nz, my_float Dbdx, my_float Dbdz )
68  {
69
70        int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
              (threadIdx.x + blockDim.x * threadIdx.y);
71
72        int a = i/nz + 1;
73        int b = i%nz + 1;
74
75        if (a < nx && b < nz)
76        EyC(a,0,b) += (HxC( a ,0,b)-HxC(a,0,b-1))*Dbdz +
77                                       (HzC(a-1,0,b)-HzC(a,0, b ))*Dbdx;
78  }
79
80  __global__ void cleanupE_xy_cuda( my_float *cHx, my_float *cHy, my_float *cEz, int
        nx, int ny, int nz, my_float Dbdx, my_float Dbdy )
81  {
82
83        int i = (blockIdx.x + gridDim.x * blockIdx.y) * (blockDim.x * blockDim.y) +
              (threadIdx.x + blockDim.x * threadIdx.y);
84
85        int a = i/ny + 1;
86        int b = i%ny + 1;
87
88        if (a < nx && b < ny)
89        EzC(a,b,0) += (HyC(a, b ,0)-HyC(a-1,b,0))*Dbdx +
90                                       (HxC(a,b-1,0)-HxC( a ,b,0))*Dbdy;
91  }
92
93  __global__ void apply_source( my_float *cEz, my_float p_source_factor, int ts,
        my_float dt, int is, int js, int ks, int ny, int nz )
94  {
95      EzC(is, js, ks) += - p_source_factor *( (ts*dt-3.0*Tk)/(Tk*Tk) )
96             *exp( -1.0*(ts*dt-3.0*Tk)*(ts*dt-3.0*Tk)/(Tk*Tk) );
97  }
```

Listing B.3: Header File

```
1  #ifndef yeecudah
2  #define yeecudah
```

```
3
4  #include <pthread.h>
5  #include <omp.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <string.h>
10 #include <sys/types.h>
11
12
13 #ifndef _WIN32
14 #include <sys/time.h>
15 #include <unistd.h>
16 #else
17 #include <cuda_runtime.h>
18 #include <cuda.h>
19 #endif
20
21 /* Block Configuration for CUDA */
22 #define BLOCKX 16
23 #define BLOCKY 16
24
25
26 /* Use floating point precision */
27 #define use_float
28
29 #ifdef use_float
30 typedef float my_float ;
31 #else
32 typedef double my_float ;
33 #endif
34
35 #define Ex(I,J,K) Ex1D[ioff_e*(I) + (nz+1)*(J) + (K)]
36 #define Ey(I,J,K) Ey1D[ioff_e*(I) + (nz+1)*(J) + (K)]
37 #define Ez(I,J,K) Ez1D[ioff_e*(I) + (nz+1)*(J) + (K)]
38 #define Hx(I,J,K) Hx1D[ioff_h*(I) + nz*(J) + (K)]
39 #define Hy(I,J,K) Hy1D[ioff_h*(I) + nz*(J) + (K)]
40 #define Hz(I,J,K) Hz1D[ioff_h*(I) + nz*(J) + (K)]
41
42 #define cmEx(I,J,K) cEx[ioff_e*(I) + (nz+1)*(J) + (K)]
43 #define cmEy(I,J,K) cEy[ioff_e*(I) + (nz+1)*(J) + (K)]
44 #define cmEz(I,J,K) cEz[(ny+1)*(nz+1)*(I) + (nz+1)*(J) + (K)]
45 #define taskEz(I,J,K) task->ez[(task->ny+1)*(task->nz+1)*(I) + (task->nz+1)*(J) +
       (K)]
46 #define cmHx(I,J,K) cHx[ioff_h*(I) + nz*(J) + (K)]
47 #define cmHy(I,J,K) cHy[ioff_h*(I) + nz*(J) + (K)]
48 #define cmHz(I,J,K) cHz[ioff_h*(I) + nz*(J) + (K)]
49
50 #define Ez1(I,J,K) tasks[0].ez[ioff_e*(I) + (nz+1)*(J) + (K)]
51 #define Ez2(I,J,K) tasks[1].ez[ioff_e*(I) + (nz+1)*(J) + (K)]
52 #define debugout(I,J,K) cEz[(ny+1)*(nz+1)*(I) + (nz+1)*(J) + (K)]
53
```

```
54 #define ExC(I,J,K) cEx[(ny+1)*(nz+1)*(I) + (nz+1)*(J) + (K)]
55 #define EyC(I,J,K) cEy[(ny+1)*(nz+1)*(I) + (nz+1)*(J) + (K)]
56 #define EzC(I,J,K) cEz[(ny+1)*(nz+1)*(I) + (nz+1)*(J) + (K)]
57 #define HxC(I,J,K) cHx[ny*nz*(I) + nz*(J) + (K)]
58 #define HyC(I,J,K) cHy[ny*nz*(I) + nz*(J) + (K)]
59 #define HzC(I,J,K) cHz[ny*nz*(I) + nz*(J) + (K)]
60
61 #define eps0 (my_float)8.8541878E-12 /* permittivity in vacuum */
62 #define mu0 (my_float)1.256637061E-6 /* permeability in vacuum */
63 #define c0 (my_float)2.99792458E+8 /* speed of light in vacuum */
64 #define Const (my_float)1.0e-10
65 #define Tk (my_float)2.0e-9
66
67 /* Macros for accessing assignment scheduler data */
68 /* Number of tasks assigned */
69 #define TaskNum(I) workerassignment[nthreads * MAX_TASKS_PER_THREAD + nthreads - I
       ]
70 /* Id of task J in current assignment for workerthread I */
71 #define TaskId(I,J) workerassignment[I * MAX_TASKS_PER_THREAD + J]
72
73 /* enum for timer array in task struct below */
74 #define TIMERS 5
75 enum { SETUP=0, COMPUTE=1, COPY=2, TOTAL=3, END=4 };
76
77 /* Fields in CSV file for benchmarks (does not apply to resource plot)*/
78 #define CSVFIELDS 3
79 enum { CSVN=0, CSVMFLOPS=1, CSVERR=2 };
80
81 /* Enum for run type */
82 enum { GREEDY=0, ASSIGNMENT=1, PLOT=2, HPLOT=3, RESOURCE=4};
83
84 /* Contains data for a task */
85 typedef struct {
86         int nts;
87         int nx, ny, nz;
88         int dx, dy, dz;
89
90         my_float *ex, *ey, *ez;
91         my_float *hx, *hy, *hz;
92
93         int bytes;
94         double flop;
95         double mflops;
96         double t[TIMERS];
97
98         int id;
99         int workerid;
100        int done;
101        int calcmet_used;
102 } task_data;
103
104 /* Struct for containing information when calculating
```

```
105  * error estimation for some benchmarks
106  */
107 typedef struct {
108        double end_time;
109        double tot_time;
110        int calcmet;
111        double flop;
112 } enddata;
113
114 /* Read in main file, needed in jobs.cu, Number of OpenMP threads to be used */
115 extern int nomp;
116
117 /* Run the benchmark with verbose output */
118 //#define verbose
119
120 /* Run thourough check on GPU output, requires that output from gpu and cpu are in
        tasks 0 and 1 */
121 //#define gpucheck
122
123 int init_task(task_data *task);
124 int destroy_task_arrays(task_data *task);
125
126 double msecond();
127 double get_flop(task_data task);
128
129 void launch_threads(pthread_t *pthreads);
130 void join_threads(pthread_t *pthreads);
131
132 void assignment_scheduler( int nts, int nx, int ny, int nz, my_float dx, my_float
      dy, my_float dz);
133 void greedy_scheduler(  int nts, int nx, int ny, int nz, my_float dx, my_float dy,
       my_float dz);
134 void plot_single_device( int nts, int nx, int ny, int nz, my_float dx, my_float dy
      , my_float dz);
135 void plot_hybrid_setup(  int nts, int nx, int ny, int nz, my_float dx, my_float dy
      , my_float dz);
136 void resource_bench(  int nts, int nx, int ny, int nz, my_float dx, my_float dy,
      my_float dz);
137
138 void assign_workload(int i);
139 void printresults(int nts, double tot_time);
140
141 void *gpu_thread(void *arg);
142 void *cpu_thread(void *arg);
143 void run_cpu_job(char *output, int threadid, task_data *task);
144
145
146 void StartCounter();
147 double GetCounter();
148 int cmp_enddata(const void *a,const void *b);
149
150
```

```
151 void updateH_homo(int calcmet, int nx, int ny, int nz, int ioff_e, int ioff_h,
152        my_float *Hx1D, my_float *Hy1D, my_float *Hz1D,
153        my_float *Ex1D, my_float *Ey1D, my_float *Ez1D,
154        my_float Cbdx, my_float Cbdy, my_float Cbdz);
155
156 void updateE_homo(int calcmet, int nx, int ny, int nz, int ioff_e, int ioff_h,
157        my_float *Hx1D, my_float *Hy1D, my_float *Hz1D,
158        my_float *Ex1D, my_float *Ey1D, my_float *Ez1D,
159        my_float Dbdx, my_float Dbdy, my_float Dbdz);
160
161 /* Since there is only one header file
162  * and the compilation uses different compilers
163  * we specify which files need the CUDA entries in main.h
164  */
165 #ifdef CUDAFILE
166 void CUDA_Success(cudaError_t error, char *str);
167 void CUDA_Success(cudaError_t error);
168
169 void run_gpu_job(char *output, int threadid, task_data *task, cudaStream_t stream)
      ;
170
171 __global__ void tid(my_float *H, int nx, int ny, int nz, my_float *np);
172 __global__ void updateH_cuda( my_float *cHx, my_float *cHy, my_float *cHz,
      my_float *cEx, my_float *cEy, my_float *cEz, int nx, int ny, int nz, my_float
       Cbdx, my_float Cbdy, my_float Cbdz );
173 __global__ void updateE_cuda( my_float *cHx, my_float *cHy, my_float *cHz,
      my_float *cEx, my_float *cEy, my_float *cEz, int nx, int ny, int nz, my_float
       Dbdx, my_float Dbdy, my_float Dbdz );
174
175 __global__ void cleanupE_yz_cuda( my_float *cHy, my_float *cHz, my_float *cEx, int
       nx, int ny, int nz, my_float Dbdy, my_float Dbdz );
176 __global__ void cleanupE_xz_cuda( my_float *cHx, my_float *cHz, my_float *cEy, int
       nx, int ny, int nz, my_float Dbdx, my_float Dbdz );
177 __global__ void cleanupE_xy_cuda( my_float *cHx, my_float *cHy, my_float *cEz, int
       nx, int ny, int nz, my_float Dbdx, my_float Dbdy );
178
179 __global__ void apply_source( my_float *cEz, my_float p_source_factor, int ts,
      my_float dt, int is, int js, int ks, int ny, int nz );
180
181 #endif
182 #endif // yeecudah
```

B.2 Utility

Listing B.4: Makefile

```
1 DEBUG = 0
2
```

```
3  NCFLAGS = -c -O2 -Xcompiler -fopenmp -arch=compute_20 -code=sm_20
4  CFLAGS = -c -O2 -Wall -fopenmp
5  CLIBS = -lm -lpthread -lgomp -L/usr/local/cuda/lib -lcudart
6  OBJ = yee.o update.o jobs.o main.o
7  NCC = nvcc
8  CC = g++
9
10 .PHONY: plot plot_file hplot hplot_file clean
11
12 all: $(OBJ)
13         $(CC) -o yeebench_cuda $(OBJ) $(CLIBS)
14
15 main.o: main.cu
16         $(NCC) $(NCFLAGS) main.cu
17
18 yee.o: yee.cu
19         $(NCC) $(NCFLAGS) -lineinfo yee.cu
20
21 jobs.o: jobs.cu
22         $(NCC) $(NCFLAGS) jobs.cu
23
24 update.o: update.cpp
25         $(CC) $(CFLAGS) update.cpp
26
27 $(OBJ): main.h
28
29 run: all
30         ./yeebench_cuda assignment
31
32 run_greedy: all
33         ./yeebench_cuda
34
35 # Plot for single device from n = 50 to what is specified in yee.dat
36 # NUMTASKS in main.h will determine number of datapoints gathered
37 plot: all
38         ./yeebench_cuda plot
39         .plots/plot.py
40
41 # Same as above but save results to file
42 plot_file: all
43         ./yeebench_cuda plot
44         ./rename.sh
45
46 # Plot for hybdrid setup
47 # Same as above, but instead of one task per datapoint;
48 # one cpu task will be run per datapoint, filled with gputasks
49 # untill the cpu tasks retire
50 hplot: all
51         ./yeebench_cuda hybridplot
52         .plots/plot.py -e
53
54 # Same as above but save results to file
```

```
55 hplot_file: all
56         ./yeebench_cuda hybridplot
57         ./rename.sh
58
59 resource_plot: all
60         ./yeebench_cuda resource
61         ./plot.py -re
62
63 clean:
64         rm -f *.o yeebench_cuda
```