NTNU

Norwegian University of
Science and Technology

High-Performance X-ray Scattering Simulations

Mathias Havdal

Master of Science in Computer Science
Submission date: August 2018
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Dag Werner Breiby, Institutt for fysikk (NV, NTNU)

Norwegian University of Science and Technology
Department of Computer Science

Problem description

In this project, we will analyze and profile an application that does compute-intensive calculations for X-ray scattering from nanostructures. The project will focus on one or more of the following topics:

- High-performance computing towards real-time simulations
- Visualization of the phase and amplitude of the X-ray field in nanoscale environments
- Machine learning applied to phase retrieval from intensity measurements

Machine learning techniques may also be considered once we have access to sufficient number of images to have both a decent training set as well as data sets to interpret.

Abstract

This thesis is part of a larger project in the field of computational microscopy. Its main contribution is to facilitate the implementation of high performance tools to assist in the understanding and simulation of microscopic coherent x-ray imaging of objects.

The main focus of this thesis is improving the performance of PaXPro, which is a C++ library focused on solving the paraxial wave equation in 2D and 3D. Some optimizations are made to an existing solver implementing the Split-Step Fourier Transform Method on the CPU. A GPU-based implementation of this solver is also created, using CUDA.

Benchmarking tools for PaXPro are created, and docker is used to measure performance on a wide range of systems, with a total of 4 different CPUs and 5 GPUs. Optimizations to the CPU-based solver achieve a significant speedup of more than 6x on an 8-core CPU with SMT. Consumer-grade GPUs achieve a speedup of around 2-4x when compared with the optimized CPU solver. Professional-grade GPUs achieve over 10x speedup.

Several ideas for future work that could further increase performance or expand functionality/usability are also included.

Sammendrag

Denne oppgaven er den del av et større prosjekt innen beregningsmikroskopi. Hovedbidraget til oppgaven er å fasilitere for implementasjonen av verktøy med høy ytelse som kan assistere i å skape bedre forståelse og simuleringer av mikroskopisk koherent røntgenavbildning av objekter.

Hovedfokuset i oppgaven er å forbedre ytelsen til PaXPro, som er et C++ bibliotek med fokus på å løse den paraksiale bølgeligningen. Noen optimaliseringer blir gjørt på en eksisterende løser som implementerer en splitt-trinns fourier transform metode på CPU. En GPU-basert implementasjon av denne løseren blir også laget, med CUDA.

Benchmark vertøy for PaXPro blir laget, og docker brukes for å måle ytelse på et bredt spekter av systemer, med totalt 4 forskjellige CPUer og 5 GPUer. Optimaliseringer på den CPU-baserte løseren oppnår en ytelses-forbedring på over 6x på en 8-kjerners CPU med SMT. Forbruker-rettede GPUer oppnår en ytelses-forbedring på 2-4x, sammenliknet med den optimaliserte CPU løseren. Proff-rettede GPUer oppnår over 10x ytelses-forbedring.

Flere ideer for fremtidig arbeid som kan gi ytterligere ytelsesøkninger eller utvidet funksjonalitet/brukervennlighet blir også foreslått.

Acknowledgements

I would like to thank my supervisor Dr. Anne C. Elster for her valuable guidance and feedback while working on this thesis. I would also like to thank my co-supervisor from the Department of Physics at NTNU, Prof. Dag W. Breiby, for his aid in understanding the physics theory needed to make this thesis possible. Also from the Department of Physics, I would like to thank David Kleiven for his work on developing the PaXPro library and helping me understand the inner workings and structure of the codebase for further development.

I am also grateful to have been a part of the HPC-Lab at NTNU, and would like to thank NTNU and IBM for supporting the lab with the wide range of high-end GPU equipment that has been essential to the development and benchmarking aspects of this thesis. Last but not least, I would like to thank my fellow HPC-Lab members for good discussions and aid in both theoretical and practical topics. They have helped make the past year an enjoyable and memorable experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is a small part of a larger project in the field of computational microscopy. In this larger project, the objective is to perform microscopic coherent x-ray imaging of objects. To facilitate understanding in this area and aid in the creation of simulations, software tools are needed for:

1. Measuring the accuracy of approximations and simplifications of physical phenomena for various scenarios.
2. Performing near real-time simulations to compare with experimental results.
3. Rapidly iterating through parameter adjustments to match experimental results.

PaXPro is a C++ library created by David Kleiven [23] to help provide these tools.

1.1 Project goals and contributions

PaXPro in its current state has the accuracy needed to fulfill the first task in the list. To make the second and third tasks possible, high performance is needed in addition to accuracy. Because of this, the main goal for this thesis is to improve the performance of the PaXPro library as much as possible in pursuit of making it useful for implementing the tools for these tasks.

Since PaXPro is currently only able to use the CPU for calculations, the main approach for achieving the goal of higher performance will be to implement a GPU-based version of one of the existing solvers. Using GPUs for general-purpose computing has exploded in popularity in the last 10 years and the potential performance improvements for highly parallel workloads are significant [34][5][18].

1.2 Report outline

Chapter 2 contains a brief historical view of the evolution of architectures for parallel computing. The emergence of general-purpose computing on graphics processing units (GPUs) is discussed, as well as some of the features and functionality of NVIDIAs state of the art CUDA toolkit for heterogeneous computing with GPUs.

Chapter 3 focuses on the PaXPro library for solving the paraxial wave equation. A short summary of the physics phenomenon simulated by PaXPro are covered, and an overview of the structure of the codebase is provided.

Chapter 4 focuses on the work done to improve the performance of PaXPro. Some profiling is done on the existing implementation, and optimizations are made to the existing code. A CUDA implementation of one of the solver methods is discussed and carried out.

Chapter 5 focuses on benchmarking and results attained with the optimizations and CUDA-based solver implemented in chapter 3. Challenges relating to benchmarking on different systems with different software configurations are discussed and docker is used as a solution for obtaining comparable results.

Finally, Chapter 6 summarises the work carried out in this thesis, the results obtained and the insights gained from benchmarking results. Several candidates for future work are also outlined.

Chapter 2

GPU Computing

This chapter covers a brief history how architectures for parallel computing have evolved, leading up to the advent of general-purpose GPU computing. Some of the key features and functionality in the CUDA toolkit for heterogeneous compute with GPUs is also covered.

2.1 Parallel computing

In parallel computing, there are many different aspects to consider. The most fundamental element is the architecture of the hardware itself. According to Flynn [15], hardware can be divided into four categories:

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

Consumer grade processors up until the mid 2000s were mostly SISD architectures, with a single stream of instructions operating on a single stream of data. For continued year-over-year performance increases, it was apparent that this type of architecture would no longer be sufficient. Increasing the amount of retired instructions per clock cycle (IPC) was becoming increasingly difficult, requiring complex designs with deep pipelines, branch prediction and memory hierarchies with multiple cache levels. The increasing complexity of these designs gave diminishing returns for energy efficiency and performance. Some of the techniques developed to improve IPC (branch prediction, speculative execution) have also been proven to have implications for security, with the most recent examples being the Meltdown [26] and Spectre [24] exploits.

While improvements in IPC were stagnating, there were also major challenges in improving the clock frequency. Increasing the clock frequency often requires increasing the voltage to reduce the setup and hold time of the logic circuits [36]. Since the dynamic power of a circuit scales with the frequency multiplied by the

square of the supply voltage, frequency can only scale so far before the power consumption becomes prohibitively high both in terms of efficiency and thermals. The ever-increasing transistor density goes some way to counteract the need for a higher voltage, but increasing the transistor density also means increasing the power density, which again leads to thermal issues [41]. With the performance of SISD architectures stagnating, the solution was to add multiple cores to a single processor chip. Each core is able to execute an independent stream of instructions on an independent stream of data (MIMD).

While SISD and MIMD architectures are dominant in consumer grade hardware, SIMD has played an important role in computing in the past and has had a resurgence in the last 10 years in the form of graphics processing units (GPUs). SIMD architectures are based primarily around vector operations, where the exact same instruction needs to be carried out for several elements of data. An example of this can be seen in Equation 2.1. On a SIMD architecture, several elements of the vector c would be calculated by a single "add" instruction.

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.1}$$

While MIMD architectures proved to be a (temporary) solution to the scaling issues of SISD architectures, GPUs now seem to be the solution to the scaling limitations of MIMD architectures. By shifting the focus from low-latency operation for a few parallel tasks to highly parallel throughput, GPUs achieve higher performance and energy efficiency (for suitable workloads). Figure 2.1 shows the practical implications of this difference in design philosophy.

Figure 2.1: Comparison of how transistors are used in a CPU compared with a GPU. From *CUDA C Programming Guide* [7] with permission.

It should be noted that modern GPUs cannot truly be described as having a SIMD architecture. In reality they are more of a hybrid approach between SIMD and MIMD. Individual instructions operate in a SIMD-like fashion, but multiple separate streams of SIMD-like instructions can operate on separate streams of data. In the same vein, modern CPUs also support SIMD instructions, albeit with a lower width (meaning fewer data elements processed per instruction) than what is typically seen on GPUs.

2.2 General-purpose GPU computing

As the name suggests, GPUs were originally created specifically to accelerate real-time graphics rendering. In real-time graphics, images are composed from thousands, if not millions of geometric primitives. Each primitive is composed of several vertices, and these vertices must be individually transformed to give a correct sense of orientation, perspective, position and scale. For a 3D scene, these transformations typically boil down to Equation 2.2 where v is the original vertex in 3D homogeneous coordinates, v' is the transformed vertex and A is the transformation matrix. Because homogeneous coordinates are used, and the fourth component of v is always 1, A can represent any combination of translation, rotation and scaling transformations, as well as a perspective transformation.

$$v' = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} \tag{2.2}$$

In addition to transforming each vertex, some per-vertex calculations are needed to determine lighting and other visual effects. The operation in Equation 2.2 is carried out by what is commonly referred to as a vertex shader. Lighting calculations are mostly carried out in a fragment shader[1]. Vector-vector and matrix-vector operations are a core part of this process as well. Vertex and fragment shading have been a core part of the graphics rendering pipeline in some shape or form since the early 90s (ex. OpenGL 1.0). In older GPUs, these stages (and essentially every other stage of the pipeline) had dedicated, fixed-function hardware that only allowed adjusting the input parameters.

Fixed function graphics processing hardware is obviously not useful for general-purpose computation. However, in pursuit of enabling more complex 3D graphics, the vertex and fragment shader stages were eventually made programmable. Early programmable GPUs used separate hardware with different instruction sets for the vertex and fragment shaders, but this was an important step towards general-purpose computing. Current GPUs have a so-called "unified shading architecture", where all programmable stages of the graphics pipeline run on the same hardware using the same instruction set. [34]

An individual thread of execution on a modern GPU roughly matches that of a simple CPU in terms of capabilities. Both integer and floating point arithmetic are supported, arbitrary memory reads/writes are possible as well as conditional/unconditional branches. The main difference between them lies in the performance characteristics. On a CPU, individual threads execute quickly and with low latency. Branching has a low penalty, and inter-thread communication is unrestricted. Concurrently executing threads are counted in the 10s on high-end systems and servers. This means that CPUs are suited to workloads that are highly interactive and have significant serial/weakly scaling sections of code. On a GPU, individual threads execute slowly and with high latency. Branching can be very expensive, and inter-thread communication is very restricted. However, a modern high-end GPU

[1]Fragment shader is the term used in OpenGL. DirectX uses the term pixel shader.

can have 1000s of threads executing concurrently (and 10000s of threads in-flight). This means that the overall throughput of a GPU is much higher than a CPU under ideal circumstances.

The takeaway is that while a GPU is capable of executing programs in much the same way as a CPU, it achieves high performance in a very different manner. As a result, only a certain class of problems is suited to a GPU. In real-time graphics rendering, the exact same operations are performed independently for millions of elements of data (vertices, pixels, etc.). This is what is commonly referred to as an "embarrassingly parallel" problem. For a more general problem to be suited to a GPU, it has to meet this classification to some extent.

2.3 NVIDIA CUDA

CUDA is a state of the art toolkit for heterogeneous compute created by NVIDIA for use with their GPUs. It was launched in 2007 and today it is one of the two most widely used platforms[2] for leveraging GPUs in general-purpose computing.

2.3.1 Programming model

CUDA allows the user to define kernels that execute on the device[3]. The same kernel is executed by hundreds or even thousands of threads over a computational domain. Typically each thread operates independently on a single unit of data in one or more large arrays. The dimensionality and size of the computational domain correspond to the dimensionality and size of the array(s) being operated on.

In CUDA, the computational domain of a kernel is represented as shown in Figure 2.2. Threads are grouped into blocks, and blocks are grouped into a grid. Both the blocks and the grid can have up to three dimensions. Each thread has a unique index within a block, while each block has a unique index in the grid. The thread and block indices combined with the block dimensions are typically used to determine which data element(s) a single thread should operate on.

The justification for representing the computational domain as shown in Figure 2.2 comes down to the underlying hardware architecture of modern NVIDIA GPUs. The basic building block of the architecture is the streaming multiprocessor (SM), shown in Figure 2.3. The NVIDIA Tesla P100 GPU has a total of 60 SMs[4] as shown in Figure 2.4. When executing a kernel, each thread block in the grid is assigned to an SM. Because of the hardware limitations of an SM, a block may contain no more than 1024 threads and thread synchronization is only possible between threads in the same block. For each block, threads are executed in a SIMD-like fashion in groups of 32 referred to as warps. In the case of the Pascal SM shown in Figure 2.3, two warps can execute concurrently.

[2]With the other being OpenCL, also supported by the CUDA toolkit.

[3]In the CUDA documentation, the GPU is commonly referred to as the device, while the CPU is referred to as the host.

[4]4 of the SMs are disabled to increase manufacturing yields, resulting in a total of 56 usable SMs [30].

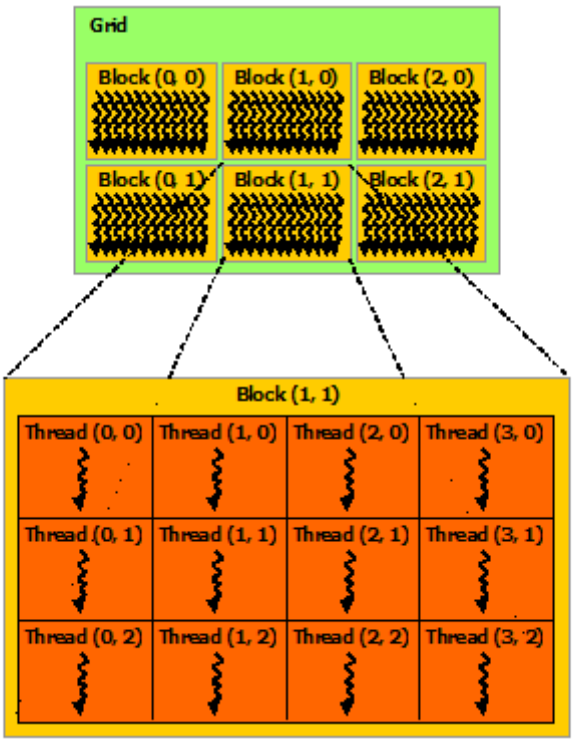

Figure 2.2: The computational domain of a kernel, represented by a grid of thread blocks. From *CUDA C Programming Guide* [7] with permission.

NVIDIA refers to warp execution as single instruction, multiple thread (SIMT), because unlike SIMD the threads can diverge at branches and execute different instructions. Up until NVIDIAs latest GPU architecture, Volta, warps shared a single program counter [7]. This meant that divergent branching within a warp led to serialized execution of each path taken. Threads not on a given path are masked out. This means that if two different paths with roughly equal amounts of computation are taken, the effective throughput can be cut in half. In other words, branching can be very costly if it is not used carefully.

Figure 2.3: Architectural overview of a single SM in the Pascal GP100 GPU. From *NVIDIA Tesla P100* [30] with permission.

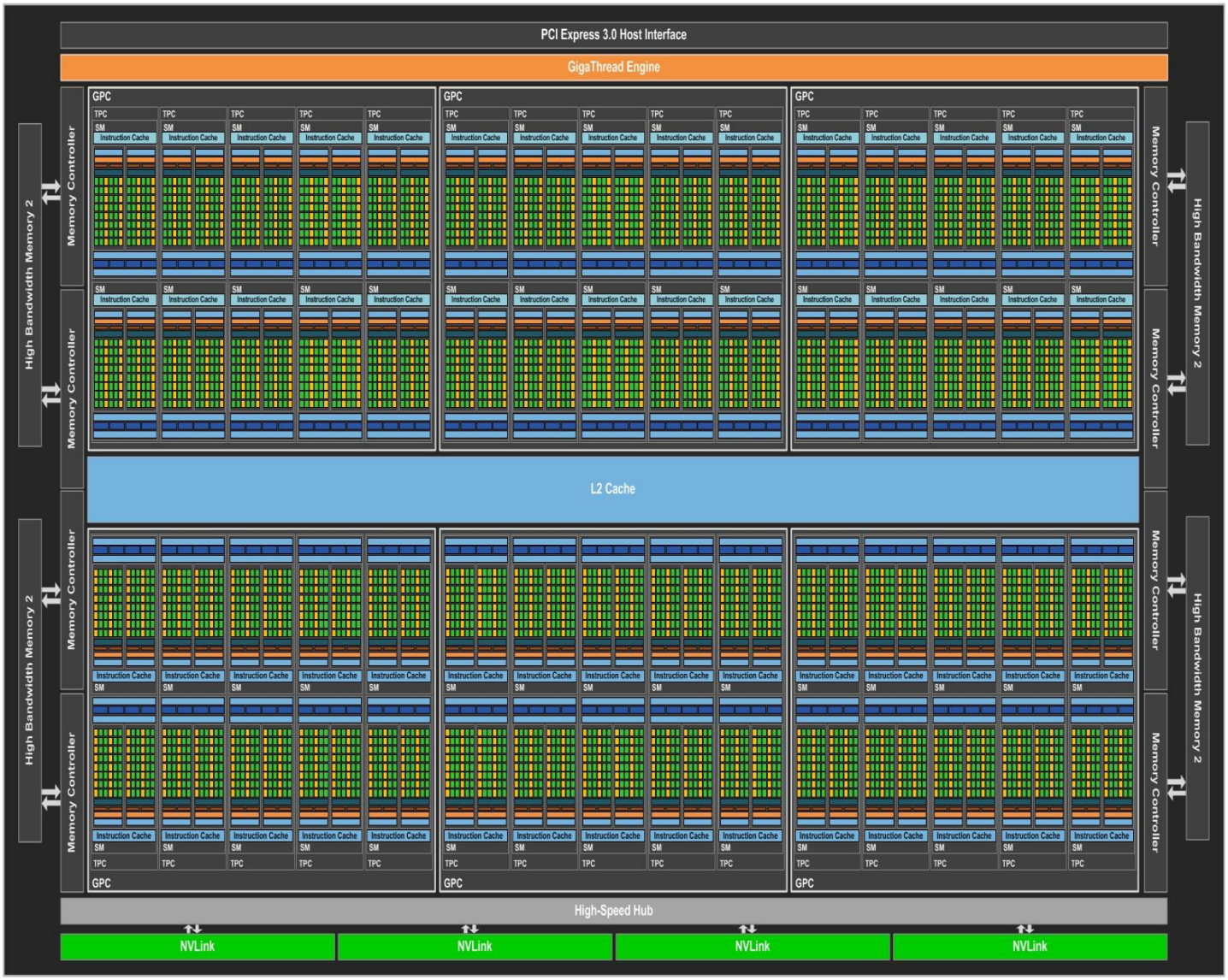

Figure 2.4: Architectural overview of the Pascal GP100 GPU found in the NVIDIA Tesla P100. From *NVIDIA Tesla P100* [30] with permission.

2.3.2 Memory types

Figure 2.5 shows the different types of memory that can be accessed by threads during kernel execution. The largest and slowest memory type is global memory. Accesses to global memory are coalesced by the SM into 32-, 64- or 128-byte accesses. This greatly reduces the number of memory accesses needed when threads in the same warp make contiguous memory accesses. Unfortunately, it can also waste a lot of memory bandwidth if for instance only a few bytes out of the smallest possible 32-byte memory access is used.

Shared memory is essentially a per-block programmer-controlled L1 cache. It is much faster than global memory, but also much smaller. Shared memory is ideal to use for shared data within a thread block. Like with global memory, specific memory access patterns are required to achieve optimal performance. Shared memory is divided into equally sized banks that can be accessed in parallel. If two or more accesses are made to different addresses the same bank, the accesses are serialized and the effective bandwidth is cut in half or worse.

Variables allocated on the stack in a kernel are usually mapped to per-thread registers in the SM. In cases where this is not possible (arrays or just too many variables to fit in registers), per-thread local memory is used. Local memory resides in device memory and is really just a per-thread variant of global memory, with the same performance characteristics and coalescing behaviour.

There are two more memory types not shown in Figure 2.5. The first is constant memory. Constant memory, like global memory, resides in device memory. The difference between the two is that constant memory has its own cache. Constant memory is primarily used for kernel launch arguments.

The second type of memory not mentioned in Figure 2.5 is texture memory. Texture memory is also backed by device memory, and like constant memory it has its own cache (on some devices). The unique thing about texture memory is that caching has 2D spatial locality. It also allows the programmer to make use of the texture interpolation hardware for interpolating values in 1D/2D/3D arrays.

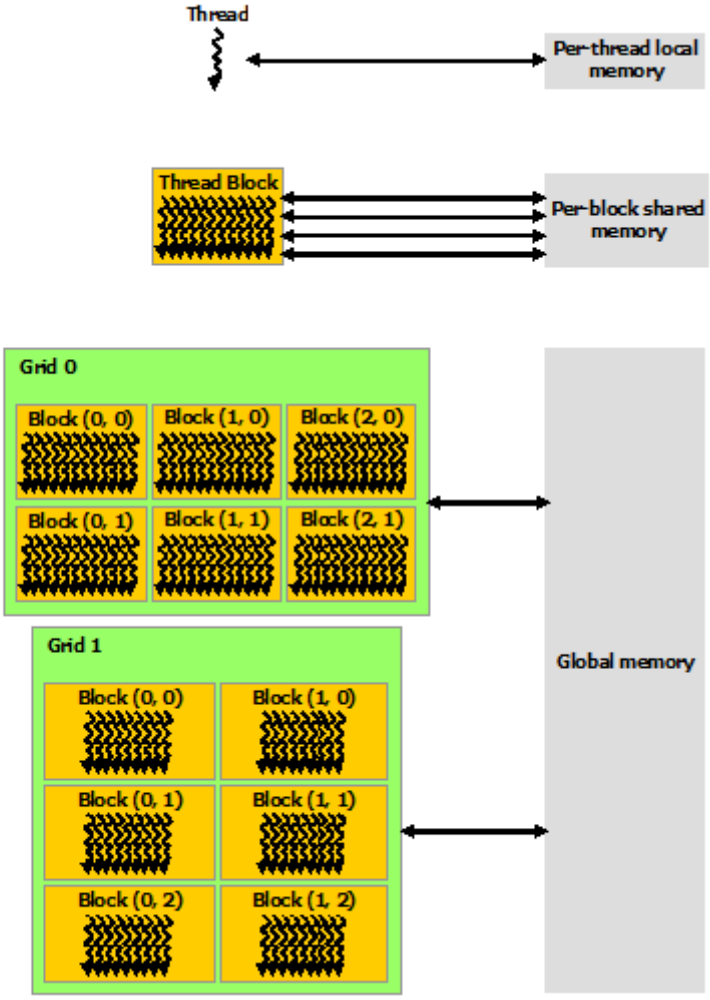

Figure 2.5: Memory hierarchy for kernel execution. From *CUDA C Programming Guide* [7] with permission.

2.3.3 Streams and concurrency

In addition to the massive parallelism within kernels themselves, CUDA also allows multiple different kernels to be executed concurrently. This is accomplished with the use of streams. Streams are essentially work queues, where kernels and other operations are executed in a given order. Streams can be synchronized with one another and the host using events. Depending on the capabilities of the device, operations from several streams can be carried out concurrently. Stream concurrency is not just limited to kernels, it also applies to `memcpy` operations.

Chapter 3

About PaXPro

PaXPro (Paraxial X-ray Propagator) was created by David Kleiven as a part of his master's thesis. It is a library written in C++ for solving the paraxial wave equation. The original purpose of the library was to aid in simulating X-ray waveguides, but it was later expanded to allow for more general scattering simulations. In this chapter, we give a short overview of the physics phenomenon simulated by PaXPro as well as an overview of the structure of the existing codebase.

3.1 X-ray physics

When propagating a plane wave of X-rays through objects with different refractive indices, the rays are deflected according to the incidence angle and the difference in refractive index between two mediums [37]. The wave travels at an increased speed when not in a vacuum, resulting in a phase shift relative to the parts of the plane wave outside the object. Lastly, some of the wave energy is absorbed by the objects that it passes through. These are the three main phenomenon simulated by PaXPro when solving the paraxial wave equation.

3.2 Codebase structure

Running simulations with PaXPro usually directly involves three classes. The first of the three is the `ParaxialSimulation` class (or a derivative). This class acts as a wrapper for the simulation, and is configured for specific parameters/problems by the user. In this thesis, x-ray scattering simulations are the main focus. This means that the `GenericScattering` class, inheriting from `ParaxialSimulation` as shown in Figure 3.1, is used.

The `ParaxialSimulation` and derivative classes do not handle the calculations for the simulation itself. This is where the `Solver` class shown in Figure 3.2 comes into the picture. This class and its derivatives store the relevant simulation data and handle the calculations, employing a range of different solver methods for both 2D and 3D problems.

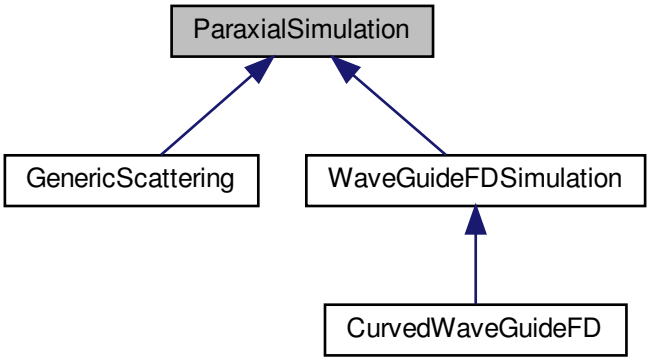

Figure 3.1: `ParaxialSimulation` class diagram.

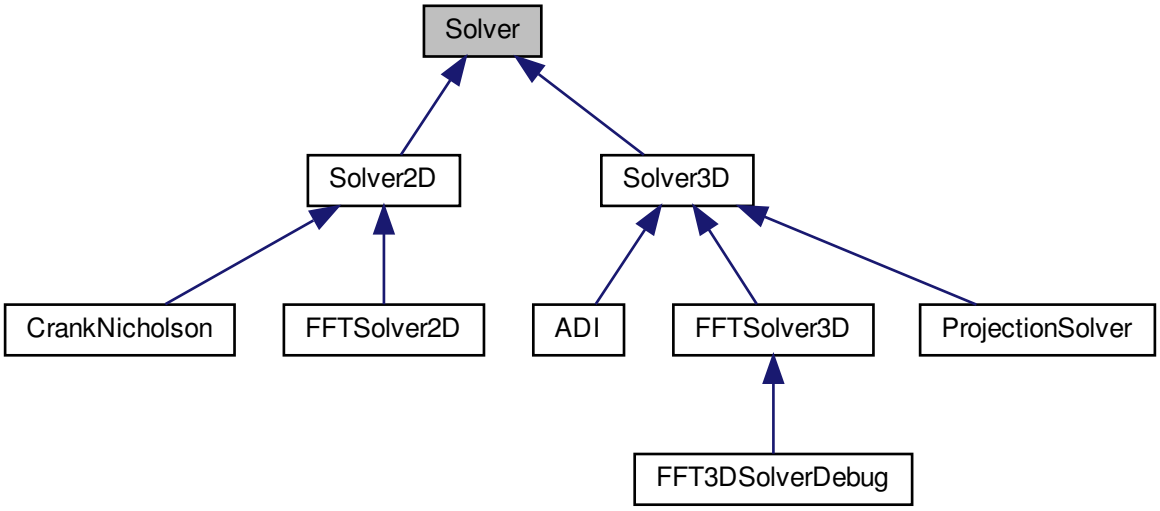

Figure 3.2: Solver class diagram.

Although the `ParaxialSimulation` classes do not handle any of the calculations directly, there is a two-way communication with the `Solver` class. `ParaxialSimulation` sets problem size and initial conditions in the `Solver` object. When running the simulation, the `ParaxialSimulation` object is queried for the discretization, position and refractive index moving through the simulation domain. This interaction is shown in Figure 3.3.

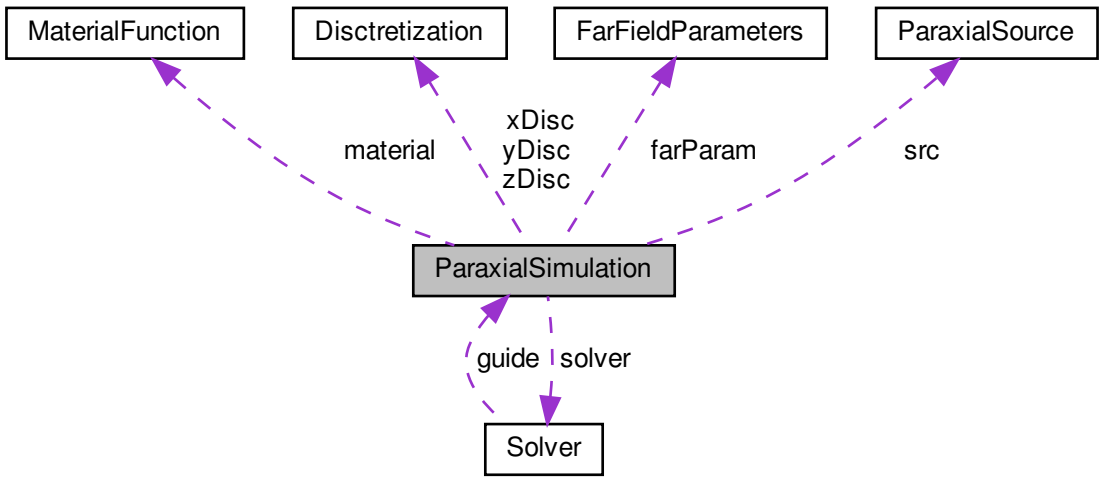

Figure 3.3: `ParaxialSimulation` collaboration diagram.

The `MaterialFunction` class is always involved for x-ray scattering simulations. This class has the sole purpose of returning the refractive index for all coordinates within the simulation domain. `MaterialFunction` is an abstract class, so the user must create their own class inheriting from it, to represent the objects within the simulation domain. The benefit of this solution is that many different representations of objects in the simulation domain are possible, ranging from a simple inside/outside test consisting of a few if-statements, to a full voxel-grid.

The `FFTSolver3D` class (shown in Figure 3.4) is the main focus of this thesis. It implements the Split-Step Fourier Transform Method for solving the paraxial wave equation [25]. As the name suggests, fourier transformations are a core part of this method. The current implementation runs on the CPU, using the FFTW library for FFTs. FFTs are known to be fast on GPUs[5][18], so the goal is to port this class to run the entire simulation on a GPU.

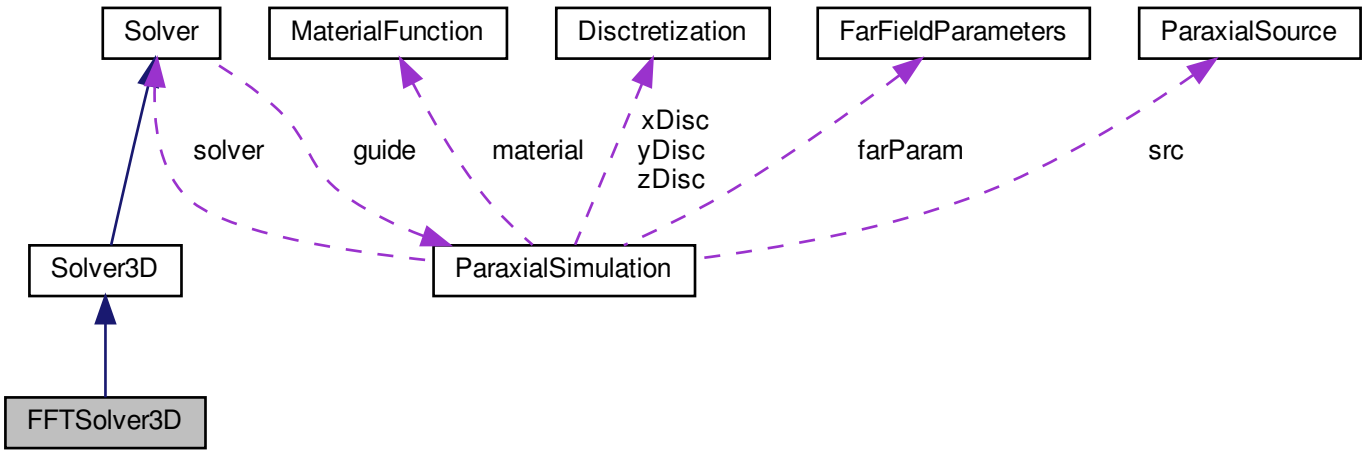

Figure 3.4: `FFTSolver3D` collaboration diagram.

Chapter 4

Improving the Performance of PaXPro

This chapter focuses on the work that was done to improve the performance of PaXPro. First we cover performance profiling and optimization of the existing CPU-based SSFTM implementation, before moving on to the challenges associated with adding a GPU-based version of the solver and achieving maximum performance.

4.1 Performance profiling

When attempting to improve the performance of a program, it is important to focus the parts of the code that have the most impact on the overall performance. Making a tenfold performance improvement in a part of the code that only represents a few percent of the total run time still only improves the overall performance by a few percent. This is especially important to keep in mind when considering the use of accelerators such as GPUs. As discussed in section 2.2, not all problems are suited for GPUs. The parts of the code that are suited a GPU must represent a significant part of the total execution time, or the performance improvement will be too small to justify the development effort.

To determine how suitable the `FFTSolver3D` class is for implementation on a GPU, the performance of a simulation using this solver has been profiled with Valgrind's Callgrind tool. Callgrind records function calls and instructions executed by a program, and uses this information to generate a call-graph. This gives the user a good indication of where most of the execution time is spent [4]. The data output by Callgrind can be visualized by the GUI tool KCachegrind. Figure 4.1 shows the top of the call-graph for `main()`.

Since PaXPro is a shared library and uses OpenMP for multithreading, two environment variables have been set for the profiling run. The first is `LD_BIND_NOW=1`. This tells the dynamic linker, `ld`, to resolve all symbols from shared libraries when the program starts, instead of waiting until the first time a function call to a given shared library is made. This prevents the dynamic linker from showing up in the call graph at some point under `main()`. The second enviroment variable is `OMP_NUM_THREADS=1`. This is set to prevent OpenMP from creating additional threads. The problem with the worker threads created by OpenMP is that they

follow a different path on the call-graph into parallel regions compared to the main thread. The result is that instructions executed in worker threads are not attributed to the functions that led to the OpenMP parallel region. This makes the profiling results confusing to interpret. As an example, if 60% of the total executed instructions are executed in OpenMP worker threads, only 40% of the total instructions will be attributed to `main()`, which is supposed to be the entry point to the program. This is especially problematic for parallel regions within functions, because the parallel regions show up as similarly named but separate functions[1]. This means that functions containing parallel regions can appear to be underrepresented in terms of their total cost.

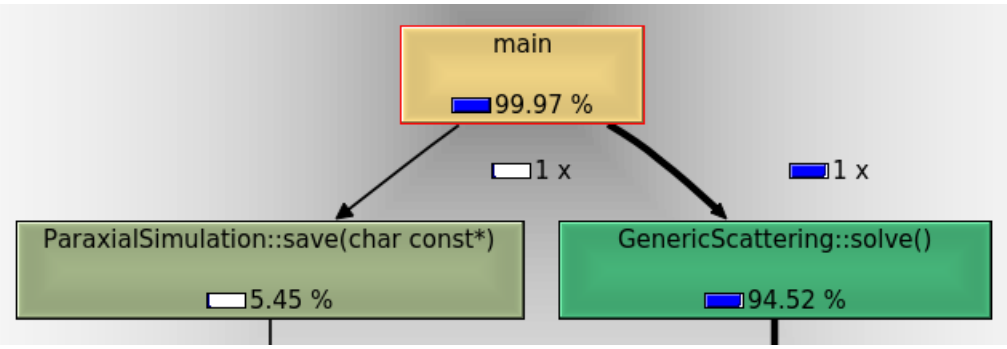

Figure 4.1: Execution instructions in functions called from `main()`.

Figure 4.1 shows that nearly 95% of the instructions are executed within `GenericScattering::solve()`. This leads to the unsurprising conclusion that the solver is important to the performance of the simulation. The remaining 5% are attributed to post-processing and saving the simulation output in `ParaxialSimulation::save()`.

Looking further down the call-graph in Figure 4.2 reveals that FFTW takes up less than 25% of the total execution time. This means that simply executing the FFTs on a GPU will not give much of a performance improvement. Due to the latency involved in transferring data to and from the GPU, the ideal solution is that all the work done in `FFTSolver3D::solveStep()` is carried out on the GPU, with no intermediate processing on the CPU between propagation steps. Because more than 90% of all the instructions are executed within this function, there should be potential for a significant speedup with a GPU.

[1]This is likely compiler-specific to an extent.

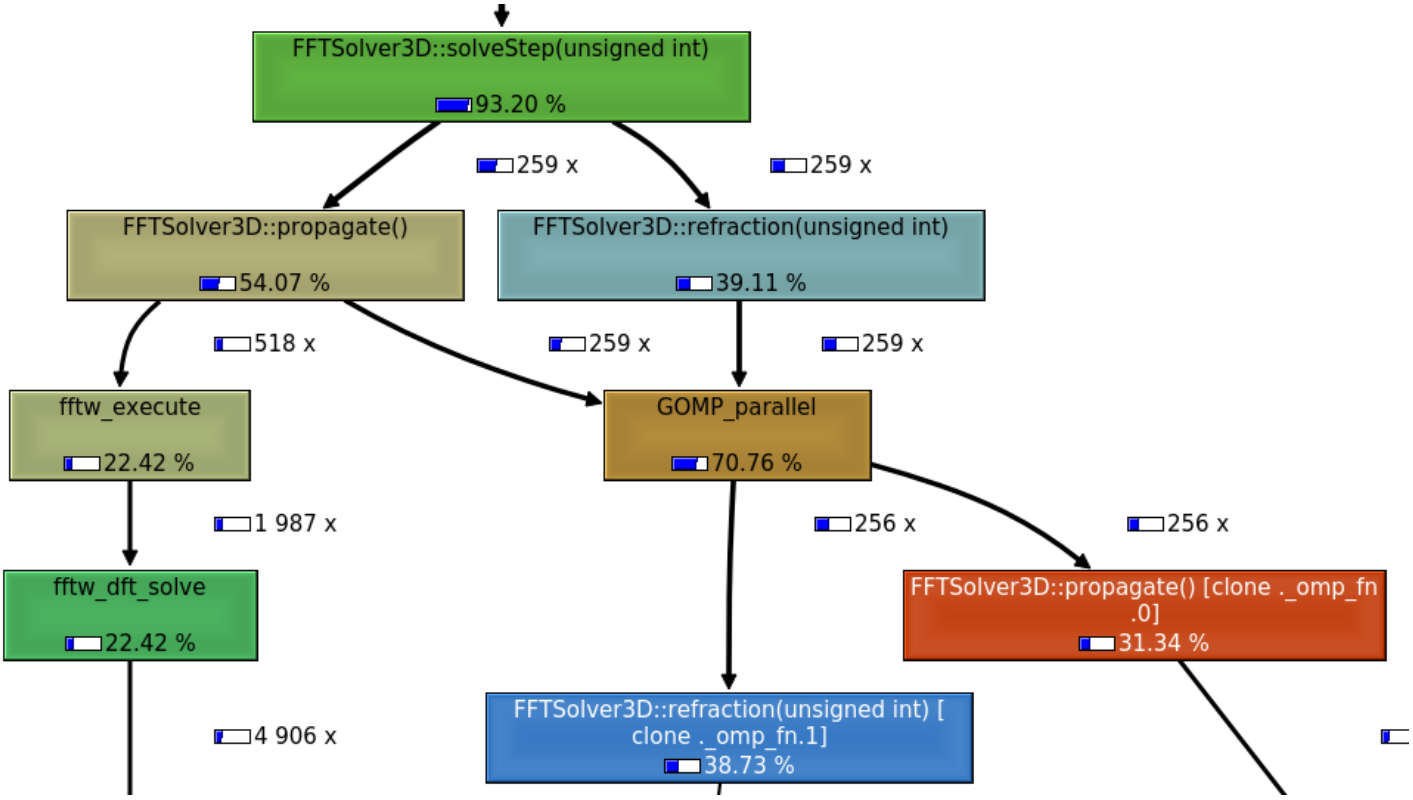

Figure 4.2: Executed instructions in functions called from `FFTSolver3D::solveStep()`.

4.2 Optimizations

In the interest of getting a fair comparison between GPU and CPU performance, an effort was made to optimize the solver in the `FFTSolver3D` class. A couple of low hanging fruit related to FFTW were discovered during the development process. First and foremost, it was discovered that only a single thread was being used for FFTW plans. Considering that most modern desktop processors have 4 or more cores (often with SMT), this leaves a lot of performance untapped. The second issue was that the cheapest planning approach was being used, potentially leading to plans with sub-optimal performance.

4.2.1 FFTW threads

Unfortunately, enabling multiple threads in FFTW is not entirely straightforward. FFTW threads have to be initialized with a call to `fftw_init_threads()` and cleaned up with `fftw_cleanup_threads()` before terminating the program. Because FFTW is used within C++ classes, this is problematic. Multiple objects of classes using FFTW can exist, which means that initializing and cleaning up FFTW threads within any class could lead to issues (cleaning up before finished, initializing multiple times). One option is to force the user to call `fftw_init_threads()` and `fftw_cleanup_threads()` before/after using any classes depending on FFTW. This is cumbersome, so an alternative solution was chosen where all classes make a call

to two global functions before/after using FFTW. These two functions take care of initializing and cleaning up the FFTW threads, and keep track of how many objects are currently using FFTW (so that threads are not cleaned up prematurely). The number of threads used by FFTW is set to be equal to the maximum number of OpenMP threads (which defaults to the number of logical processors on the machine). See section 5.4 for benchmark results showing the speedup with multiple FFTW threads.

4.2.2 FFTW planning rigor

The plan chosen by FFTW has a significant impact on the performance of the CPU-based solver. FFTW has a few different flags that can be passed when generating a plan [13]. The planning rigor flags influence how FFTW picks a plan. This allows the user to make a tradeoff between planning time and performance. The default behaviour in PaXPro is to use the `FFTW_ESTIMATE` flag. With this flag, FFTW creates a plan based on simple heuristics. This results in a short planning time, but the performance of the plan may not be optimal. Another benefit of this planning mode is that the data in the arrays used by the plan is not altered during planning.

In an effort to extract the maximum performance from the solver, optional support for the `FFTW_MEASURE` planning flag has also been added. This support is enabled by defining `PAXPRO_FFTW_MEASURE=1` when running CMake. When using this planning mode, FFTW will run actual performance tests to determine a plan that has close to optimal performance for the problem size and the hardware being used[2]. There some drawbacks to this approach. The first is that running tests takes time, meaning that the overall run time might end up being longer despite the plan itself being faster. The number of times a plan is run is proportional with the problem size, so while this is a potential issue for small problem sizes, the investment is generally worthwhile for larger problem sizes.

Another drawback is that the tests destroy the data in the arrays. Because the plan is created before the first propagation step is computed (meaning that the initial conditions are already placed in the array), a temporary copy has to be created so the initial conditions can be restored. The root cause of this issue is that PaXPro implements several different solvers and is not designed specifically to cater to the needs of FFTW.

The final and perhaps most important drawback is that the tests run during planning are sensitive to background activity from other programs. Background activity disturbs the timing measurements taken during the tests, and can lead FFTW to create a plan that performs poorly. In some cases FFTW created plans that performed worse than the `FFTW_ESTIMATE` mode. This is the main reason the `FFTW_ESTIMATE` mode is the default. `FFTW_MEASURE` is only suited to environments where the planner can run undisturbed. See section 5.4 for benchmark results comparing the performance of the two planner flags.

[2]The flags `FFTW_PATIENT` and `FFTW_EXHAUSTIVE` work the same way as `FFTW_MEASURE`, but will run more tests in an attempt to get closer to optimal performance at the cost of increased planning time.

4.3 CUDA implementation

Other than the obvious goal of improving performance, there were two underlying goals when developing the `CUDAFFTSolver3D` class, which is the CUDA-based version of the `FFTSolver3D` class. The first was that the codebase should only be extended in ways that still make it possible to compile and run PaXPro without CUDA support. Secondly, no breaking changes should be made to the existing functionality and solvers.

As a computer-science student with no real background in physics other than the fundamentals, another important focus was to ensure the correctness of the CUDA implementation. The complexity and depth of the physics theory needed to implement an accurate X-ray scattering simulation makes it prohibitively difficult to ensure correctness by any other method than comparing the results with a known good implementation. For this reason, verification tests comparing results with the `FFTSolver3D` class (implemented by physics student Kleiven [23]) were a high priority.

4.3.1 Challenges

Host/device keywords

CUDA code requires that functions declaration specify whether the function should be compiled for the host, device or both. This is done using the `__host__` and `__device__` keywords (`__host__` is the default when unspecified). This is problematic for classes that need to be accessed from plain C++ code and CUDA device code, when the goal is to avoid making CUDA a required dependency. The solution to this is to wrap the use of `__host__` and `__device__` in preprocessor macros. When `nvcc` is detected, the macros are set to the respective keywords. When `nvcc` is not used, the macros are defined as empty to avoid unknown keyword errors.

Functions in `ParaxialSimulation/GenericScattering`

`FFTSolver3D` uses several functions in `ParaxialSimulation/GenericScattering` to get parameters and calculate certain values for each propagation step. These functions can not realistically be made accessible from device code, because passing the `ParaxialSimulation/GenericScattering` object to the device is unfeasible (member variables of types without CUDA support, issues with virtual functions as outlined below). To get around this issue, the `CUDASimulation` helper class was created with all the necessary functions.

Virtual functions in classes

Although CUDA supports many of the modern C++ language constructs, there are some limitations that apply when passing objects to the device and launching kernels (among other things). With regards to implementing the `FFTSolver3D` class in CUDA, there are some problematic limitations related to classes with virtual functions. If an object of a class with virtual functions is created in host code and copied to a device (or vice-versa), calling any of the virtual functions (in device code)

will result in undefined behaviour [7]. This happens because the virtual function pointer table in the object points to host functions, which are not accessible/valid on the device. Understandably, this issue also applies for arguments to kernels. The implication of is that accessing the user-implemented `MaterialFunction` child class becomes problematic.

The solution to this was to turn `CUDAFFTSolver3D` and `CUDASimulation` into template classes, where the template parameter is the user-implemented `MaterialFunction` child class. All kernels invoked by `CUDAFFTSolver3D` also had to be templated.

4.3.2 Implementation

The Split-Step Fourier Transform Method (SSFTM) used by `FFTSolver3D` can be divided into two parts. For the first part, X-rays are propagated in fourier space. This involves one FFT, a kernel to handle the propagation and an IFFT back to real space. The cuFFT library bundled with the CUDA toolkit is used to perform the FFTs.

The second step consists of computing the refraction integral for elements where the refractive index changes, meaning that a material boundary has been crossed or that the x-rays are moving through a non-homogeneous material. This is handled in a single kernel. There is also an optional third step to tackle the periodic boundary conditions that are a characteristic of SSFTM. In this step, a transmission function is applied to make the waves decay at the horizontal/vertical edges of the domain. This step is also handled by a single kernel.

After computing a step, `FFTSolver3D` computes a downsampled version of the result in a 3D array that will eventually contain a downsampled version of the entire simulation. Depending on the downsampling factor, the size of this array can be quite significant. On top of that, the values in this array are not used by the simulation itself. They are only stored for post-processing and output. Due to the limited amount of memory available on a GPU, it is not desirable to store this array on the device. This means that for each propagation step, the solution must be copied back to the host. This must be done asynchronously to avoid a performance penalty.

Asynchronous `memcpy` operations in CUDA can only be performed under certain conditions [27]. For transfers from device to host memory, the host memory must be pinned. This is done using `cudaHostRegister()` for memory that has already been allocated or `cudaHostAlloc()` for new allocations. When it comes to the CUDA streams, there are multiple options. The most fundamental requirement is that the `memcpy` runs in a different stream from the kernels and cuFFT plans. When using the legacy default stream[3], all events cause implicit synchronization (events meaning kernel launches, memcpy, etc.). This means that either the kernels and cuFFT plans must run in their own explicitly created stream, or the `memcpy` stream must be created with the `cudaStreamNonBlocking` flag. Alternatively, the per-thread default stream[4] can be used. In this case, two explicitly created threads

[3]The "legacy default stream" is the actual default, despite the name suggesting that it is not.

[4]NVIDIA using the term "default" rather loosely, as the per-thread default stream must be

are used, with one thread handling the execution of kernels and cuFFT plans and the other handling asynchronous `memcpy` operations.

To enable the GPU to continue solving the next propagation step while the previous one is copied to the host, three buffers are used. Two of the buffers are used for computing the next step, while the third holds the solution to the previous step. At the end of a step, an asynchronous `memcpy` to host is queued and the buffers are rotated by swapping the pointers. The kernels and cuFFT plans for the next step are then queued before waiting for the `memcpy` to complete.

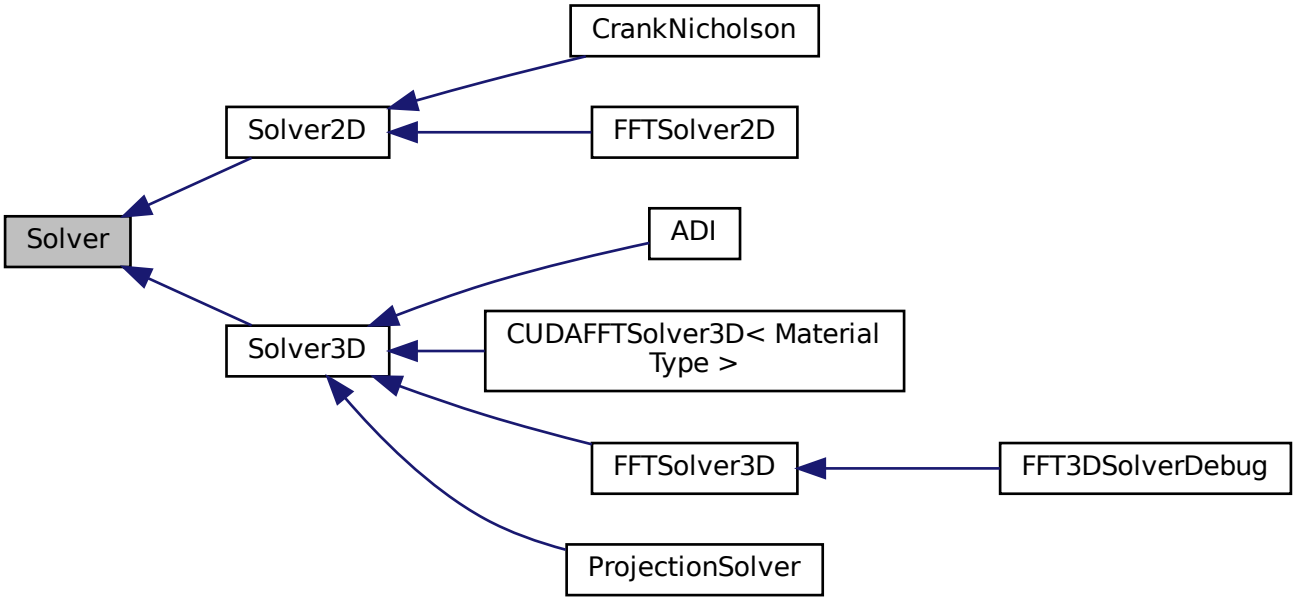

Figure 4.3: `Solver` class diagram with `CUDAFFTSolver3D`.

enabled per-file at compile time.

Chapter 5

Benchmarking and Results

5.1 Benchmarking method

This section describes the approach used to gauge the performance of the GPU solver versus the existing CPU solver. A wide range of hardware was tested, which brings on some challenges.

5.1.1 Problem

The problem used for benchmarking consists of propagating a plane wave of x-rays through a solid sphere. In physics terms, this is a simple enough problem that it can be solved analytically. However, it shows the capabilities of the solver and can be scaled with ease. In this case, there are two parameters that can be used to scale the problem size in terms of computational demands. The first is the radius of the sphere itself, given in nm. Increasing the radius of the sphere means that the computational domain also has to increase in size to encapsulate the entire sphere. The second parameter is the discretization step, also given in nm. By setting a smaller step, we increase the resolution and accuracy of the solution along with the computational demands. Since increasing the sphere radius and reducing the discretization step has the same effect in terms of the computational demands, only the sphere size has been varied when generating the benchmarking results in this chapter. The discretization step has been fixed at 3 nm.

The size of the computational domain D is given by Equation 5.1, where r is the radius of the sphere. The domain is bigger along the horizontal and vertical (x and y) axes because the Split-Step Fourier Transform Method, has periodic boundary conditions [23][25]. This means that x-rays that exit the domain will reappear on the opposite side.

$$D = \begin{cases} x \in [-1.5r, 1.5r] \\ y \in [-1.5r, 1.5r] \\ z \in [-1.05r, 1.05r] \end{cases} \tag{5.1}$$

Determining the number of elements in the computational domain is done by dividing the length of the domain along each axis by the discretization step s as shown in Equation 5.2 for the x-axis. Since propagation is done along the z-axis,

this means that the number of elements processed in each propagation step n_{prop} is given by Equation 5.3. The number of elements in the whole domain is given by Equation 5.4.

$$x_{steps} = \frac{x_{max} - x_{min}}{s} = \frac{1.5r - (-1.5r)}{s} = \frac{3r}{s} \tag{5.2}$$

$$n_{prop} = x_{steps} \cdot y_{steps} = \frac{9r^2}{s^2} \tag{5.3}$$

$$n = x_{steps} \cdot y_{steps} \cdot z_{steps} = \frac{18.9r^3}{s^3} \tag{5.4}$$

Each propagation step involves an FFT and an IFFT. This means that the computational complexity for a single propagation step is $\mathcal{O}(n_{prop} \log(n_{prop}))$. That puts the total computational complexity at $\mathcal{O}(n \log(n_{prop}))$. Since we use a constant discretization step for the benchmarks in this chapter, the computational complexity can be expressed as $\mathcal{O}(r^3 \log(r))$.

Memory complexity is divided into two parts. The solver stores the whole solution (all propagation steps), which has a memory complexity of $\mathcal{O}(r^3)$ assuming a constant discretization step. However, the full 3D solution is not always interesting. For instance, the visualizations in figures 5.1 and 5.2 are computed only from the output of the very last propagation step. For that reason, the solver allows the user to set a downsampling factor f applied to each axis in the complete 3D solution[1]. This gives a memory complexity of $\mathcal{O}(\frac{r^3}{f^3})$. In the benchmark, f is set to such a high value that the memory usage for the 3D solution is insignificant. It is also important to note that the complete 3D solution is only stored on the host, so it has no impact on the memory complexity on the GPU itself for the GPU-based solver. If we ignore the memory complexity for the full 3D solution, the memory complexity is given by $\mathcal{O}(r^2)$ (derived from n_{prop}), again assuming a constant discretization step.

In Figures 5.1 and 5.2 we can see a visualization of the data output by the benchmark simulation for a 500 nm and 3000 nm sphere respectively. The "Comparison with form factor" plot shows a comparison with an approximate method that is only accurate for smaller sphere sizes.

[1]The downsampling factor can be set individually for each axis, but this is not done for the benchmark

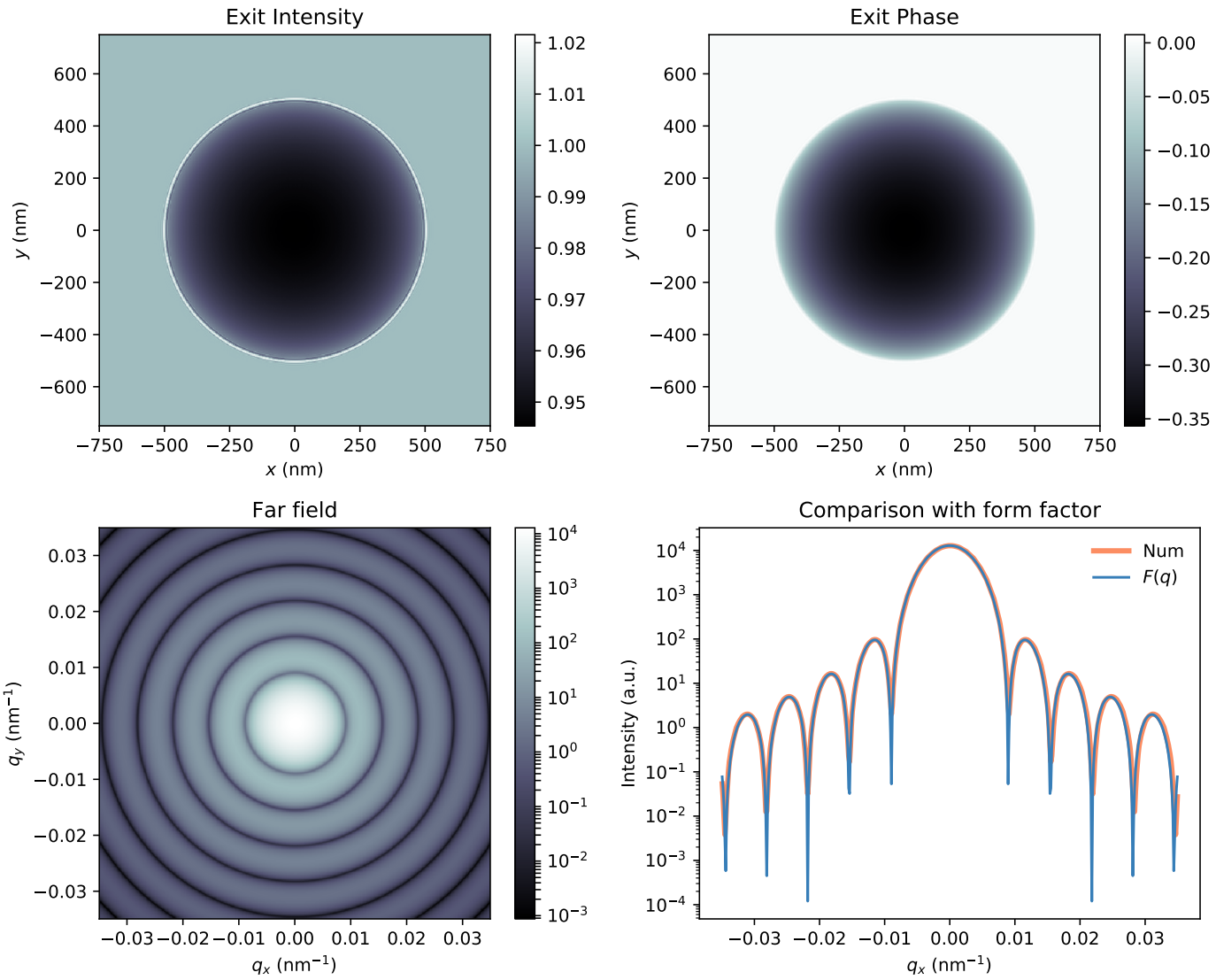

Figure 5.1: Visualization of simulation results from a 500 nm sphere.

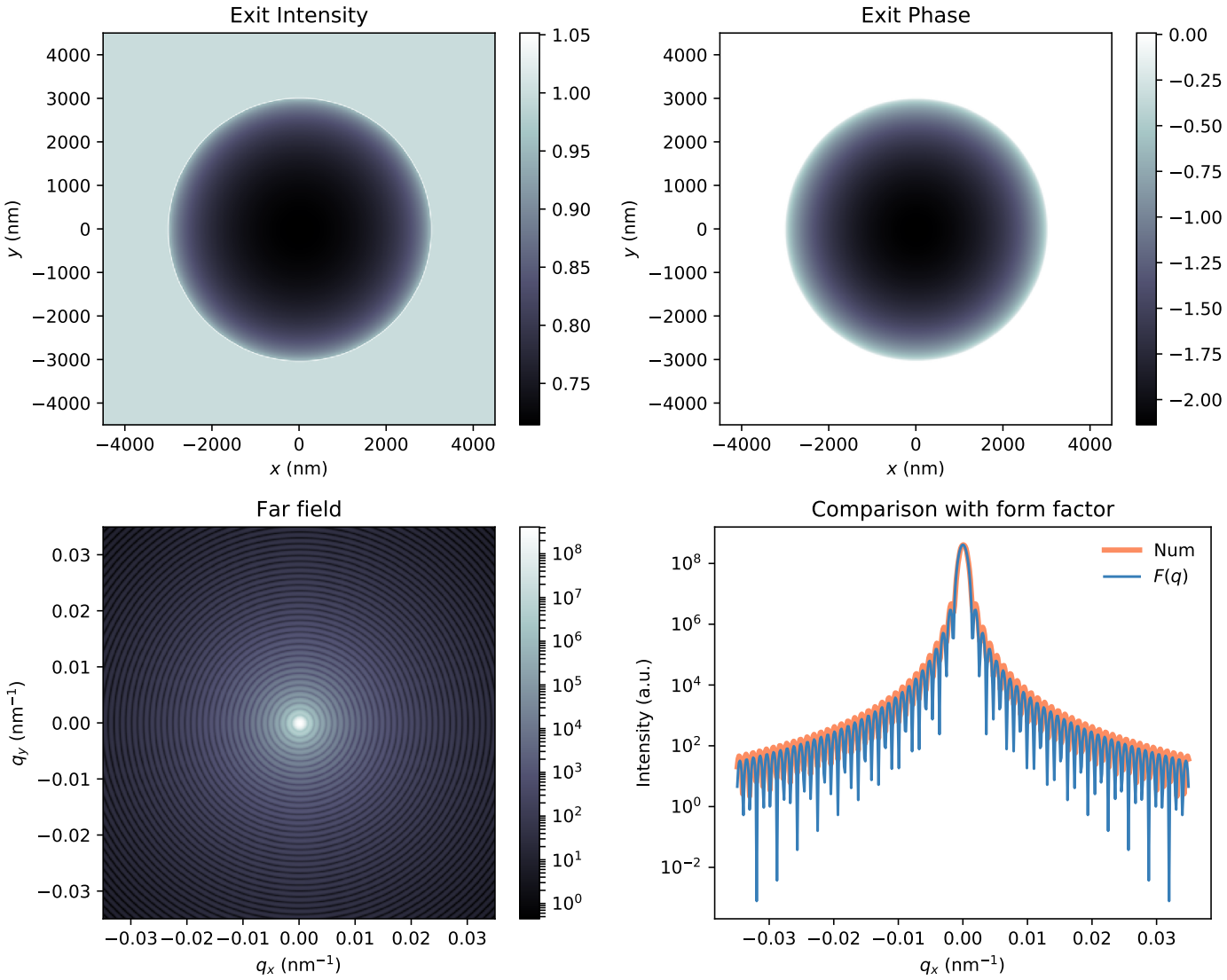

Figure 5.2: Visualization of simulation results from a 3000 nm sphere.

5.1.2 Problem sizes and collecting results

Table 5.1: Benchmarking problem sizes.

Group	Sphere radius	Discretization
SMALL	50 nm	3 nm
SMALL	100 nm	3 nm
SMALL	250 nm	3 nm
SMALL	500 nm	3 nm
SMALL	750 nm	3 nm
SMALL	1000 nm	3 nm
MEDIUM	1500 nm	3 nm
MEDIUM	2000 nm	3 nm
MEDIUM	2500 nm	3 nm
LARGE	3000 nm	3 nm
LARGE	5000 nm	3 nm
LARGE	7500 nm	3 nm
HUGE	10 000 nm	3 nm
HUGE	12 000 nm	3 nm

Benchmarking problem sizes are divided into four groups as shown in Table 5.1. The small problem sizes are intended to be small enough to run fast (sub-minute) on CPUs. Medium problem sizes are intended to be challenging for CPUs, with run times of a few minutes. Large problem sizes should take up to a few hours on CPUs. Finally, the "huge" problem sizes are only intended for compute-oriented GPUs and test performance with high memory usage (observed to use >10GB GPU memory).

Table 5.2: Number of passes per benchmark group.

Group	Passes	Solver
SMALL	5	CPU/GPU
MEDIUM	3	CPU/GPU
LARGE	2	CPU/GPU
HUGE	1	GPU only

Benchmarking results are calculated as the average run time over a number of passes. Only the solver itself is timed, meaning that initialization, post-processing and exporting simulation data is ignored. There are two main reasons for this. The first is that only the solver itself has been implemented in CUDA. The second is that most of the run time is spent in the solver anyway, with the exception of some of the smallest problem sizes on GPUs. Initialization and post-processing could also be sped up with CUDA, but this gives minimal benefit outside of small problem sizes. Due to the wide range of run times, a different number of passes is used for each benchmark size group as shown in Table 5.2.

Running the benchmark program is handled through a script (found at `benchmark/run_benchmark.sh` in the PaXPro git repo). The script allows the user to configure a number of options:

- Problem size groups to run.
- Use CPU or GPU solver.
- Which GPU to use (if multiple GPUs are present).
- Number of threads used by OpenMP/FFTW.
- On NUMA systems, force running on a specific NUMA node.

For each problem size, the script directs the output of the benchmark program to a file in a directory structure that follows a hierarchy of:

`results/<group>/<sphere_radius>_<discretization>/<pass_number>`

For example:

`results/LARGE/7500_3/2`

To save the user the trouble of digging through the full program output to find out how long the solver took, another script has been created to parse the results (found at `benchmark/read_results.sh` in the PaXPro git repo). This script supports two different output modes. The first is made to be human readable and shows the solver time for each pass as well as the average (with time converted to hours, minutes and seconds as needed) and standard deviation. The second output follows a CSV (comma separated values) style, and is suited for further processing or use in LaTeX. It is also possible for the user to filter the output based on the size group. A third script (`benchmark/calc_speedup.py`) has also been made to calculate speedup based on two or more CSV outputs.

5.1.3 Running on multiple systems

There is a wide range of hardware available to run benchmarks at the NTNU HPC-Lab . The challenge is that many of these systems are shared between multiple users. This means that the operating system and software/libraries must meet the needs of all the users. For that reason, there is a policy that all shared systems use the latest Ubuntu LTS (Long-Term Support) release (16.04 at the start of this thesis).

PaXPro dependencies

Ubuntu 16.04 LTS was released in April, 2016 [35]. However, PaXPro was developed using a mix of Fedora 25 (released in November, 2016 [10]) and Debian 9 (released in June, 2017 [9]). As a consequence, PaXPro depends on certain libraries with versions that are newer than what is available in Ubuntu 16.04 LTS. This means that several other libraries must be compiled by the user for PaXPro to work. This also leads to a tedious workflow for compiling PaXPro itself, because CMake must be manually pointed to the user-compiled libraries.

In addition to the existing dependency issues, the CUDA version bundled with Ubuntu 16.04 is quite outdated (version `7.5`, newest at time of writing is `9.2`).

NVIDIA officially supports newer CUDA versions on Ubuntu 16.04 through several different install options, but as with any third-party packages, this is not seamless. For instance, an incompatible kernel update caused CUDA to stop working on several machines in the lab.

To get the latest version of CUDA and avoid dependency issues with PaXPro, Arch Linux (https://www.archlinux.org/) was chosen for the development machine. Arch Linux is a rolling release distribution, meaning that it generally has the latest versions of all software.

Docker

While Arch Linux was a viable solution for my personal development machine, it is not a good solution for any of the shared machines in the lab. The constant version changes of a rolling release are fine when the system only has one user, because that user is in control of when updates are installed. On a shared machine, this is no longer the case, leading to potential unwelcome surprises when a library or some other software changes.

Docker is a container technology that strikes a middle ground between native/bare-metal and virtualization. Unlike a virtualized environment, docker containers run directly on the host OS. Each container is isolated through functionality in the host kernel, meaning that processes from the host and other containers are hidden. Containers also have independent filesystems, meaning that containers can have different configurations, libraries and runtimes. This is obviously very useful in cases like this one, where an application/library (PaXPro) has dependencies that are not easily available on the host OS (Ubuntu 16.04). It also has the advantage of providing the exact same environment, regardless of the host configuration. This means that performance differences will be down to hardware as opposed to software optimizations from different library versions and compilers.

As a part of the isolation from the host, docker containers by default don't have access to any of the host hardware/peripherals. This includes GPUs, which means that CUDA does not work within a normal docker container. Fortunately, NVIDIA has made their own docker plugin/wrapper that takes care of setting up the necessary sysfs entries within the container so that GPUs can be accessed for use with CUDA.

5.1.4 Docker setup

Docker containers are based off images. An image is essentially a snapshot of a root filesystem. This snapshot usually contains the equivalent of a minimal OS installation, minus some core components like the kernel. In addition, it includes dependencies and configuration for a specific application. Images are built from instructions contained in a dockerfile. The dockerfile essentially specifies a base image, files to copy from the host and commands to run. Commands consist of typical things you would do to configure a native OS install for a specific application. That includes using the package manager (`apt`/`apt-get` in Ubuntu), fetching configurations and building dependencies.

Initially the plan was to base the PaXPro docker image off an image of the newly released Ubuntu 18.04 LTS. Ubuntu 18.04 has all the dependencies PaXPro needs, to setting up this image would be a simple matter of installing dependencies through the package manager and copying over the source files for VISA and PaXPro. Unfortunately, NVIDIA only provides CUDA-prepared Ubuntu 18.04 docker images for `x86_64` machines [31]. Since the Minsky (see section 5.2) is a `ppc64le` machine, Ubuntu 18.04 was not an option [32].

Fortunately, dealing with the Ubuntu 16.04 dependency issues is easier in a docker image. The single use-case nature of the image means that you can disregard some of the best practices for installing and maintaining libraries on the system. CUDA is pre-installed in the Ubuntu 16.04 images from NVIDIA, and Table 5.3 shows the dependencies that could be installed straight from the package manager.

Table 5.3: PaXPro satisfied dependencies on Ubuntu 16.04 LTS.

Dependency	Package name	Version
GCC	`gcc-5`	`5.4.0`
ARPACK	`libarpack2-dev`	`3.3.0`
BLAS	`libblas-dev`	`3.6.0`
FFTW	`libfftw3-dev`	`3.3.4`
GSL	`libgsl-dev`	`2.1`
SFML	`libsfml-dev`	`2.3.2`
HDF5	`libhdf5-dev`	`1.8.16`

Although the FFTW version found in the official Ubuntu 16.04 is sufficient for PaXPro, the header file `fftw3.h` has a bad interaction with the CUDA compiler, `nvcc` [14]. This issue has been fixed in FFTW version `3.3.8`, but building FFTW is somewhat complex due to the number of compilation options with a potential performance impact. To make matters worse, compilation has to work for both `x86_64` and `ppc64le` systems. It was deemed easier to simply patch the `fftw3.h` header provided by the package manager.

The remaining dependencies are shown in Table 5.4. All these libraries are built from source and installed in system directories by running `make install` as root. This is normally seen as bad practice, because there is no easy way to uninstall/remove files placed in system directories when using this method. To add to this, all other files in the system directories are managed by the package manager (`apt`/`apt-get` for Ubuntu). This could lead to conflict if the user installs multiple versions of the same software, potentially causing serious breakage of the system. However, because this is a single-use docker image, maintainability is not a concern like on a normal system.

For CMake, at least version `3.8` is required (`3.5.1` is provided in Ubuntu 16.04). This dependency stems from PaXPro using the first-class language support for CUDA that was added in CMake version `3.8` [6]. For parity with the Arch Linux development system, CMake `3.11.4` was used (latest version at the time of creating the dockerfile).

SuperLU is a dependency of Armadillo. PaXPro uses Armadillo functionality added in version `7.500`. This version of Armadillo in turn depends on Superlu

Table 5.4: PaXPro unsatisfied dependencies on Ubuntu 16.04 LTS.

Dependency	Git repository	Branch/tag
CMake	`https://gitlab.kitware.com/cmake/cmake.git/`	`v3.11.4`
SuperLU	`https://github.com/xiaoyeli/superlu.git/`	`v5.2.1`
Armadillo	`https://gitlab.com/conradsnicta/armadillo-code.git/`	`8.500.x`
Google Test	`https://github.com/google/googletest.git/`	`release-1.8.0`
JsonCpp	`https://github.com/open-source-parsers/jsoncpp.git/`	`master`

version `5.x`. Again, for parity with the Arch Linux development system, the latest versions available at the time were used.

JsonCpp is the only dependency that doesn't use a versioned branch/tag. The reason for this is that JsonCpp has a bad interaction with the CUDA compiler (`nvcc`) [22]. A fix exists in the `master` branch, but it is currently not found in any released version.

NVIDIA provides several Ubuntu 16.04 images with different CUDA versions. Originally the idea was to use the image with the latest CUDA version (`9.2`). Unfortunately, although the CUDA runtime is decoupled from the host system, the NVIDIA driver version is not. All CUDA releases have a minimum driver version, so this means that the CUDA version used in a docker container is restricted by the drivers on the host. Ubuntu 16.04 officially provides NVIDIA driver version `384.xx`, which only supports CUDA versions up to `9.0`. This meant that an Ubuntu 16.04 image with CUDA 9.0 had to be used as a base for the PaXPro docker image. More specifically, `nvidia/cuda:9.0-devel-ubuntu16.04` was used on `x86_64` systems and `nvidia/cuda-ppc64le:9.0-devel-ubuntu16.04` was used on `ppc64le` systems. For the full dockerfile used to generate a PaXPro image, see the git repo at https://github.com/acelster/PaxPro-Docker.

5.2 Systems used

Four different systems were used to obtain the benchmarking results in this chapter. Both the CPU(s) and GPU(s) in each system were tested. The specs of each system can be found in tables 5.5, 5.6, 5.7 and 5.8.

Table 5.5 shows the specs for my machine at the NTNU HPC-Lab . This machine was used for most of the development work. There are a couple of things worth pointing out about this machine. First of all, it has two GPUs: a GTX 980 and a Tesla K40. The GTX 980 is a typical high-end gaming GPU, while the Tesla K40 is a slightly older compute-oriented GPU. The most obvious difference between the two is that the K40 has three times as much memory as the GTX 980. Another important difference is that the GTX 980 has much lower FP64 rate than the K40. This is typical for a gaming-oriented GPU, because real-time 3D graphics typically don't need more than precision than FP32. The second thing to note about this

[2]Boost must be enabled manually on the Tesla K40.

Table 5.5: Development system [1][17][40]

System	
ISA	`x86_64`
CPU	AMD Ryzen 7 1800X
Clock speed	3.6GHz base, 4.0GHz boost
Cores	8 with 2x SMT
Memory	16GB DDR4 @ 2933MHz
GPU 1	
Interconnect	PCIe 3.0 x8
GPU	NVIDIA GTX 980
Architecture	Maxwell
Clock speed	1126MHz base, 1216MHz boost
FP64 rate	1/32 FP32
CUDA cores	2048
Memory	4GB GDDR5
Memory bandwidth	224GB/s
GPU 2	
Interconnect	PCIe 3.0 x8
GPU	NVIDIA Tesla K40
Architecture	Kepler
Clock speed	745MHz base, 810/875MHz boost[2]
FP64 rate	1/3 FP32
CUDA cores	2880
Memory	12GB GDDR5
Memory bandwidth	288GB/s
Software	
OS	Arch Linux
Kernel	`4.17.6`
NVIDIA driver	`396.25`
Docker version	`18.05.0`

system, is that both GPUs are only using 8 lanes of the PCIe 3.0 link[3]. This is due to a limitation of the X370 chipset on the motherboard in this system [8]. When one PCIe 3.0 slot is in use, the full 16 lanes are available. With two slots in use, the lanes are shared between the two slots, giving 8 lanes per slot.

The machine in Table 5.5 is my home computer. Around 2013, this was a high-end gaming-PC/workstation. It has been included to give an indication of how PaXPro performs on older hardware.

A Titan V was newly acquired at the NTNU HPC-Lab . The specs for this GPU and the system it is installed in can be seen in Table 5.7. With 5120 CUDA cores

[3]Profiling has shown that the reduced bandwidth is not a performance bottleneck for either GPU.

[4]Overclocked to 4.4GHz, normal clocks are 3.5GHz base, 3.9GHz boost.

[5]This is a factory overclocked ASUS GTX 780, stock clocks for a reference design GTX 780 are 863MHz base, 900Mhz boost.

Table 5.6: Home system [20][16][3]

System	
ISA	`x86_64`
CPU	Intel i7 3770k
Clock speed	4.4GHz[4]
Cores	4 with 2x SMT
Memory	16GB DDR3 @ 1600MHz
GPU	
Interconnect	PCIe 3.0 x16
GPU	NVIDIA GTX 780
Architecture	Kepler
Clock speed	889MHz base, 941MHz boost[5]
FP64 rate	1/24 FP32
CUDA cores	2304
Memory	3GB GDDR5
Memory bandwidth	288.4GB/s
Software	
OS	Arch Linux
Kernel	`4.17.6`
NVIDIA driver	`396.25`
Docker version	`18.05.0`

and a memory bandwidth of 653GB/s, the Titan V has by far the most raw compute power out of the systems tested.

The last system is an IBM "Minsky", with specs found in Table 5.8. This is the only non-desktop/workstation system tested, and there are several noteworthy things about this system. First and foremost, it is the only system that does not use the `x86_64` ISA. Secondly, it is the only NUMA system, with its two POWER8 CPUs. Last but not least, it has four Tesla P100 GPUs connected with NVLink.

[6]System can also be configured with 4x/8x or no SMT.
[7]256GB per CPU, non-uniform memory access.

Table 5.7: Titan V system [21][39]

System	
ISA	`x86_64`
CPU	Intel i7 7700k
Clock speed	4.2GHz base, 4.5GHz boost
Cores	4 with 2x SMT
Memory	32GB DDR4 @ 2133MHz
GPU	
Interconnect	PCIe 3.0 x16
GPU	NVIDIA TITAN V
Architecture	Volta
Clock speed	1200MHz base, 1455MHz boost
FP64 rate	1/2 FP32
CUDA cores	5120
Memory	12GB HBM2
Memory bandwidth	653GB/s
Software	
OS	Ubuntu 16.04 LTS
Kernel	`4.4.134`
NVIDIA driver	`384.130`
Docker version	`18.03.1`

Table 5.8: IBM Minsky [42][38]

System	
ISA	`ppc64le`
CPUs	2x 8335-GTB POWER8
Clock speed	2.860GHz base, 3.492GHz boost
Cores	10 per CPU with 2x SMT[6]
Memory	512GB[7] DDR4 @ 1600MHz
GPUs	
Interconnect	NVLink
GPU	4x NVIDIA Tesla P100 SXM2
Architecture	Pascal
Clock speed	1328MHz base, 1480MHz boost
FP64 rate	1/2 FP32
CUDA cores	3584 (per GPU)
Memory	16GB HBM2 (per GPU)
Memory bandwidth	720GB/s (per GPU)
Software	
OS	Ubuntu 16.04 LTS
Kernel	`4.4.73`
NVIDIA driver	`384.66`
Docker version	`17.09.0`

5.3 Docker vs. native performance

As established in subsection 5.1.3, docker is needed to get a fair comparison of the performance running on a wide range of shared systems. To get an idea of how much overhead/performance penalty is introduced when running CUDA programs in a docker container, some tests were run on the development machine (see Table 5.5). It is important to point out that there are significant software differences other than of just docker vs. native, as shown in Table 5.9.

Table 5.9: Software versions in PaXPro docker container and on development machine.

Software	Docker version	Native version
CUDA	`9.0.176`	`9.2.148`
GCC	`5.4.0`	`7.3.1`
ARPACK	`3.3.0`	`3.6.0`
BLAS	`3.6.0`	`3.8.0`
FFTW	`3.3.4`	`3.3.8`
GSL	`2.1`	`2.5`
SFML	`2.3.2`	`2.5.0`
HDF5	`1.8.16`	`1.10.2`

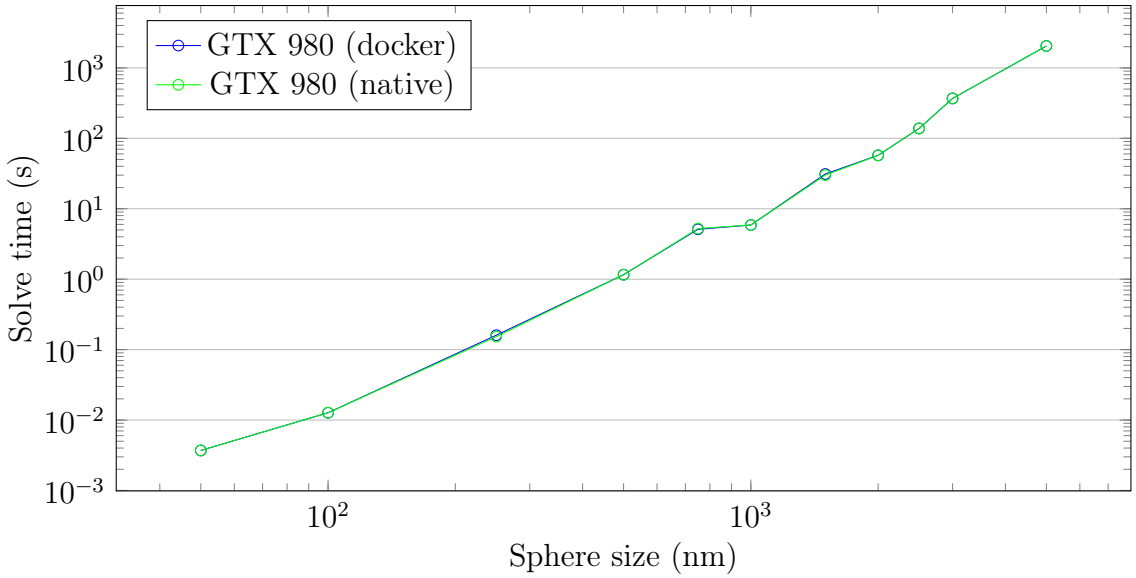

Figure 5.3: Docker vs. native performance for GTX 980.

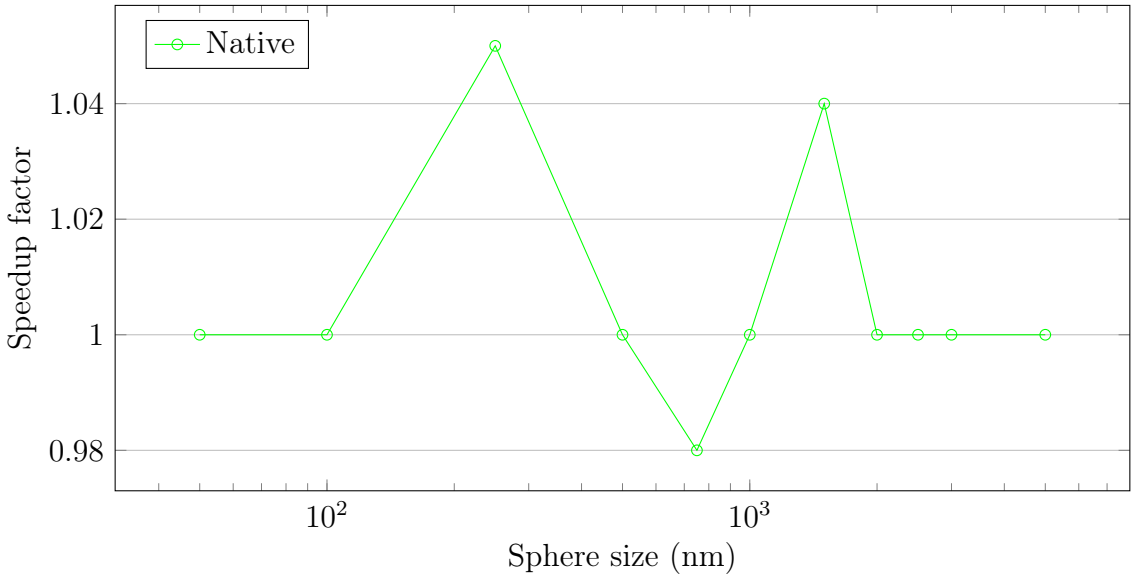

Figure 5.4: Docker vs. native speedup for GTX 980.

Figures 5.3 and 5.4 show that there was at most a 5% performance difference between docker and native for the GTX 980. Considering all the software differences with a potential performance impact, this is astonishingly close.

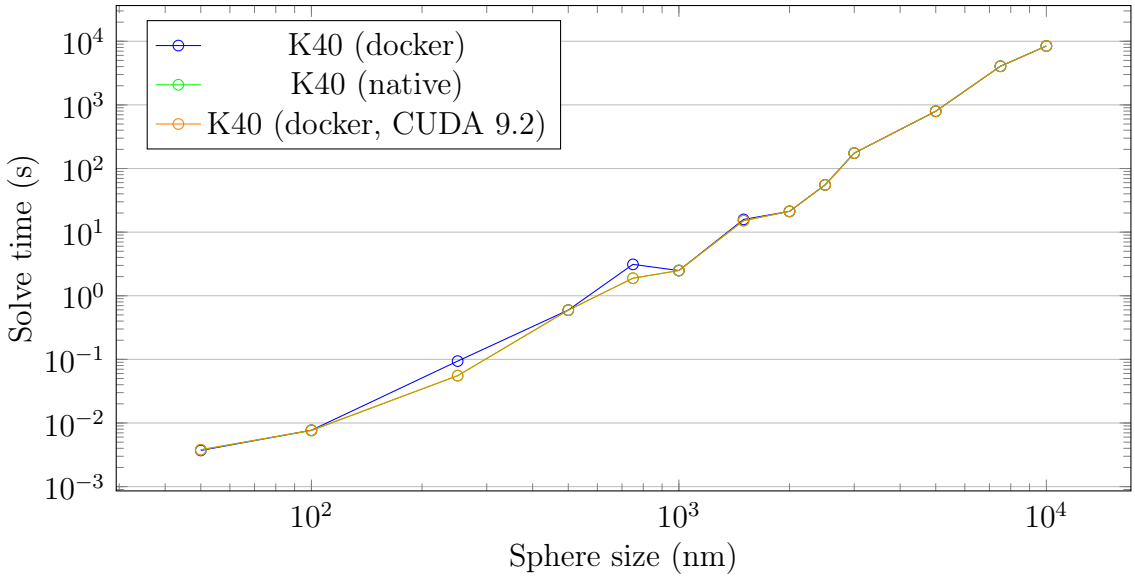

Figure 5.5: Docker vs. native performance for Tesla K40.

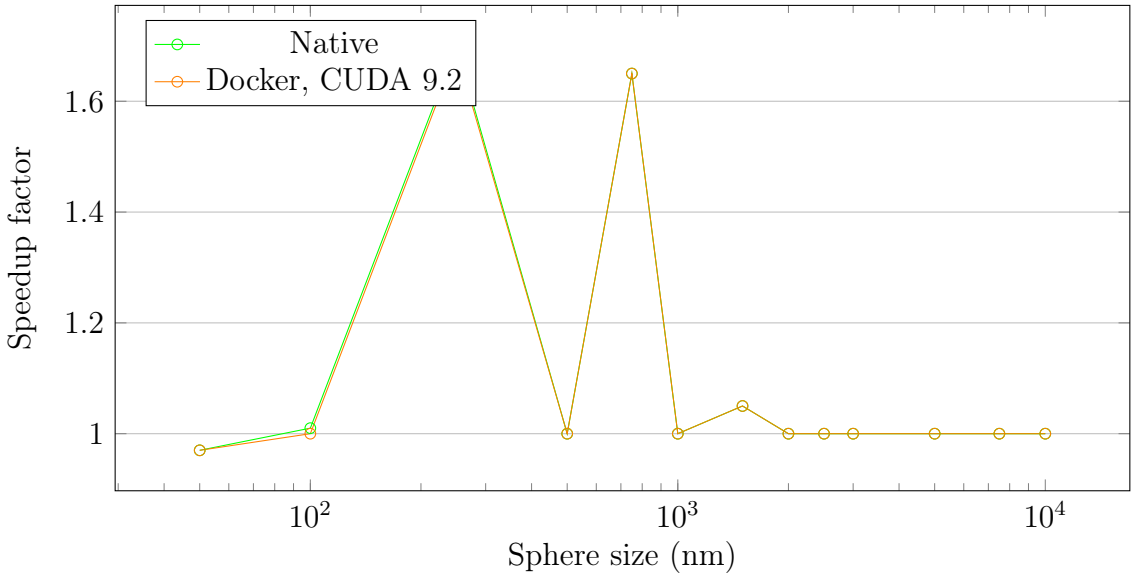

Figure 5.6: Docker vs. native speedup for Tesla K40.

In figures 5.5 and 5.6, we can see that the story is quite different for the Tesla K40. For sphere radius 250 nm and 750 nm, the speedup is over 50%. The other problem sizes are in line with the GTX 980 results. Fortunately, the up-to-date NVIDIA driver on the development machine meant that it was also possible to test a docker image with CUDA version `9.2`. This revealed that the performance difference was almost entirely down to the CUDA version. Looking at CUDA changelogs for versions `9.1` and `9.2`, it appears that performance improvements were made to the cuFFT Bluestein kernels that are used for the problem sizes in question [28][29].

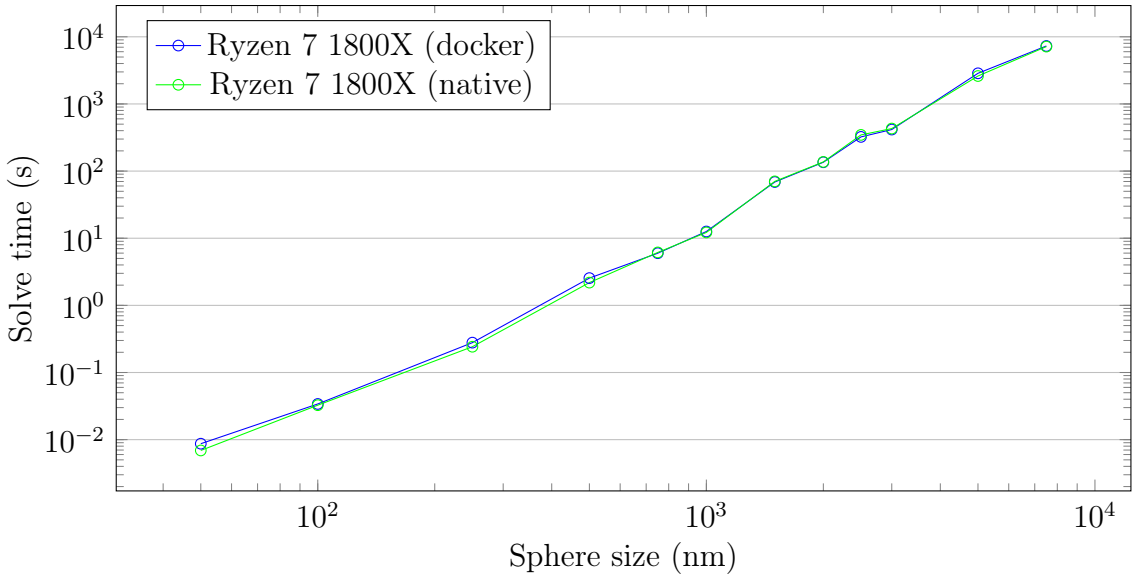

Figure 5.7: Docker vs. native performance for Ryzen 7 1800X.

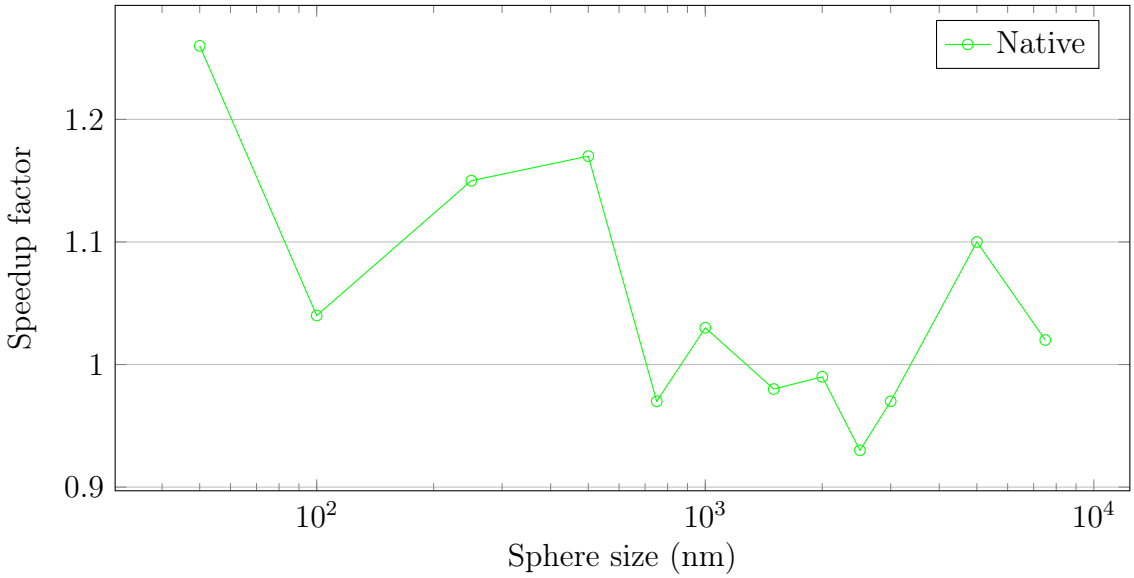

Figure 5.8: Docker vs. native speedup for Ryzen 7 1800X.

Figures 5.7 and 5.8 show that the Ryzen 7 1800X in the native environment mostly ranges from 10-15% faster to 5% slower. Part of the performance difference is likely down to the GCC version. The bulk of the run time of the CPU solver is spent in either FFTW or OpenMP parallel regions interacting with data in Armadillo matrices. Armadillo is primarily a template-based library, so there should be a potential for compiler optimizations to make a difference here. The FFTW version is also likely to have an impact on the performance, as the `FFTW_MEASURE` planner flag was used to obtain these results. This flag does not explore all possible planning options, so minor changes to the planning strategy between FFTW versions could have a noticeable impact on performance.

In conclusion, there are some minor performance differences between docker and native, but they appear to be mostly attributed to version differences in each environment. Docker is known to have some overhead for I/O heavy workloads, but this does not appear to be an issue for more compute-oriented workloads running on the CPU or a GPU [11].

5.4 Performance on CPU

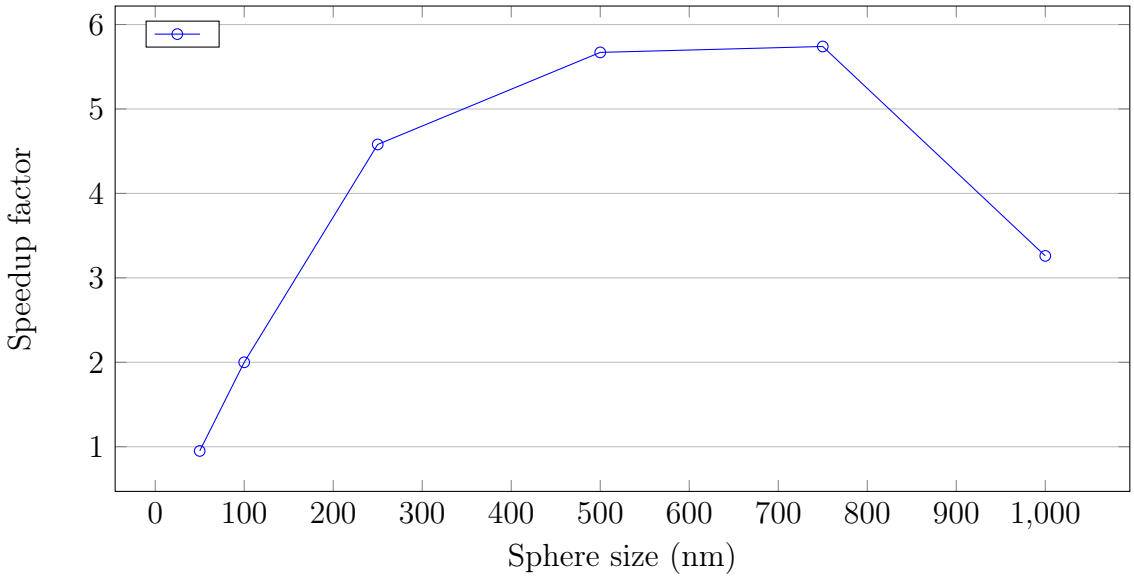

Figure 5.9: Speedup when enabling multiple threads in FFTW for Ryzen 7 1800X. Number of FFTW threads equal to default number of OpenMP threads, 16 in this case. `FFTW_ESTIMATE` planner flag was used.

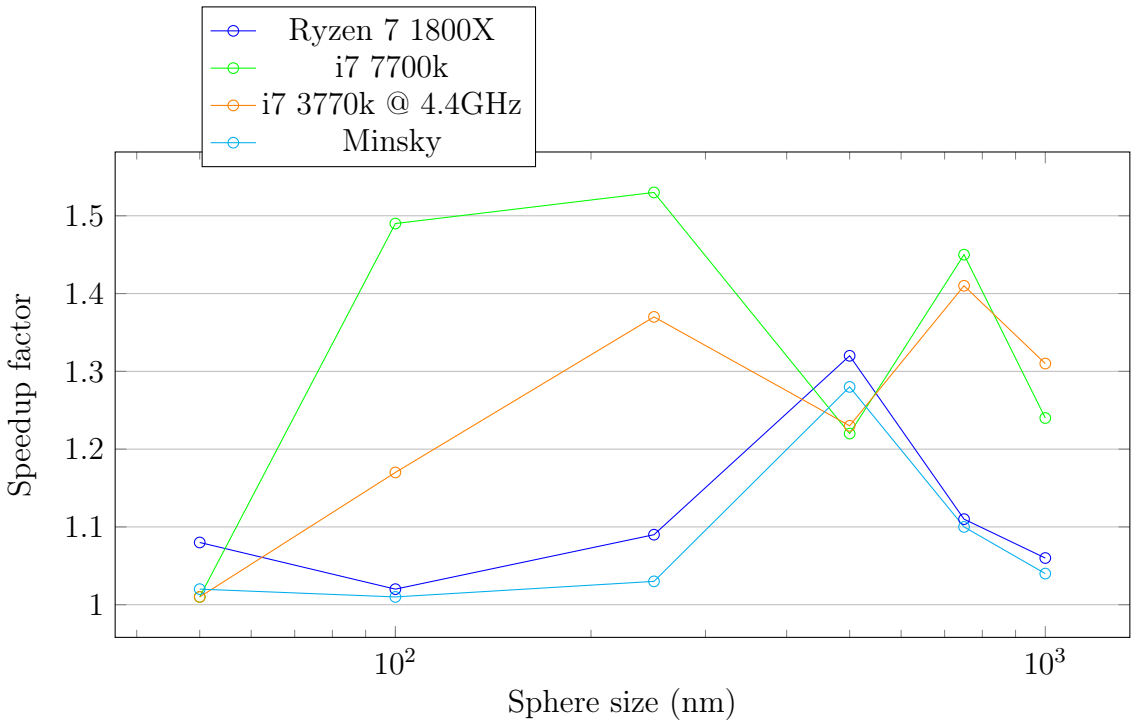

Figure 5.10: Speedup when using `FFTW_MEASURE` planner flag compared with `FFTW_ESTIMATE` (single thread).

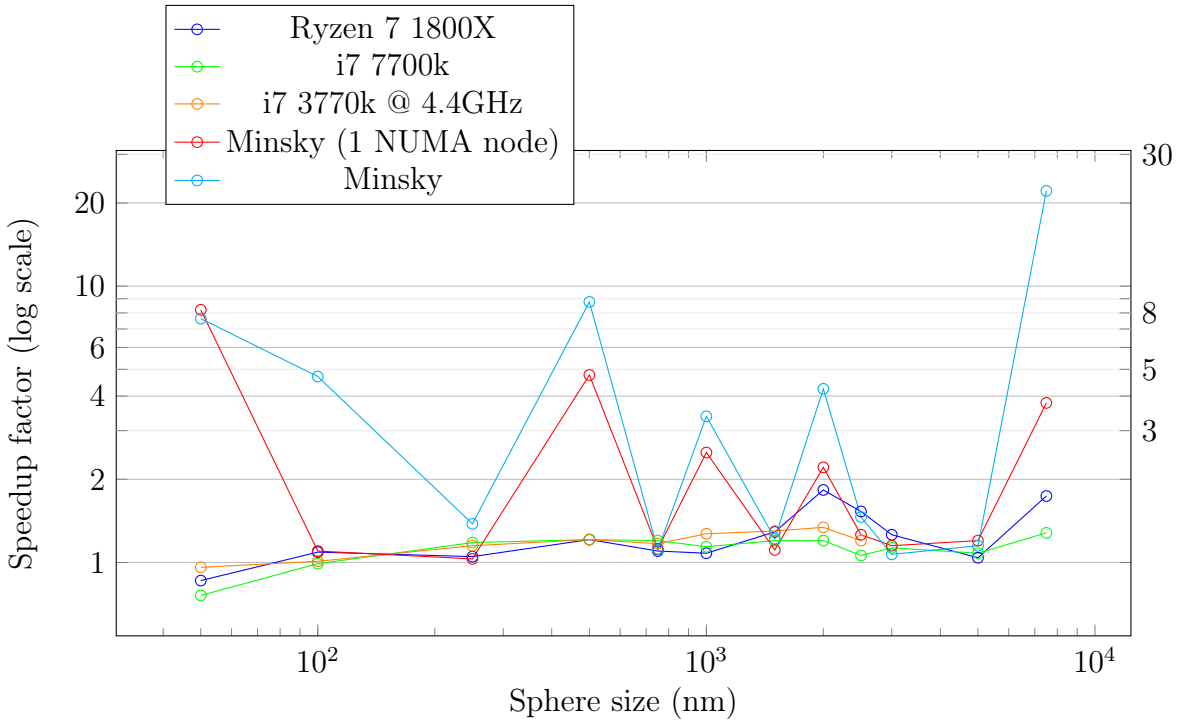

Figure 5.11: Speedup when using `FFTW_MEASURE` planner flag compared with `FFTW_ESTIMATE` (multithreaded).

Figure 5.9 shows that a large performance improvement is made when enabling multithreading in FFTW[8]. This is Amdahl's law in practice [2]. The massive speedup for all problem sizes except the smallest indicates that the serial FFT/IFFT was causing weak scaling.

Figures 5.10 and 5.11 show the speedup achieved by using `FFTW_MEASURE` compared to `FFTW_ESTIMATE`. When running the solver on a single thread, the speedup ranges from fairly minor to just over 50%. The multithreaded scenario paints a very different picture however. While the Ryzen 7 1800X and the two i7s remain well below a 100% speedup, the Minsky shows huge fluctuations in speedup. It seems that the FFTW heuristics do a very inconsistent job of predicting the performance of multithreaded workloads on this machine. To get a reasonably accurate representation of the CPU performance of all four systems, the `FFTW_MEASURE` flag has been used to obtain all the remaining results in this section.

An interesting thing to note is that in Figure 5.11, the i7s and the Ryzen 7 1800X are slower on the smallest problem size when using `FFTW_MEASURE`. This is likely a result of the extra copies made to preserve the initial conditions when the planner is working. Although the planning cost is also higher, the plan is initially created when doing a reference run (simulating the propagation of x-rays through a vacuum). Recreating an identical plan in the timed part of the solver is cheap, so this should not be much of a factor [12].

[8]In addition to OpenMP multithreading.

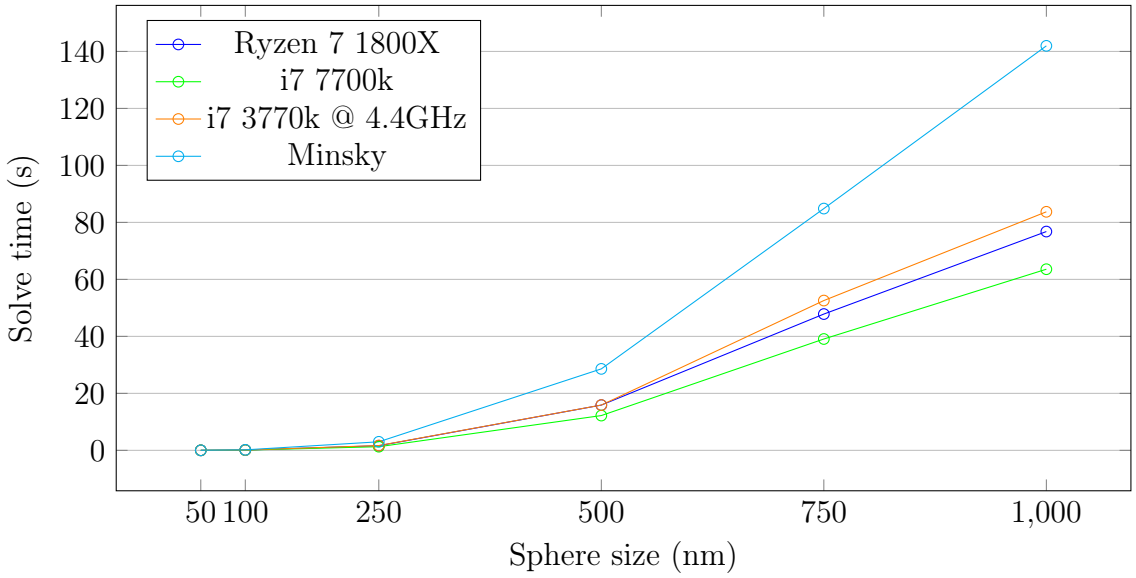

Figure 5.12: Single-thread CPU performance.

Looking at the single-thread performance shown in Figure 5.12 clearly shows the difference between the desktop/workstation oriented processors and the Minsky. A desktop workload often involves interactive programs where parallelism is limited, so good single-thread performance is essential. The Minsky clearly lags behind in this department, but it makes up for this deficit by having two CPUs and a total number of cores greater than the other three systems combined.

Comparing the overclocked i7 3770k with the similarly clocked i7 7700k, we can see that Intel has made a total IPC improvement of around 30% in the last four generations of i7 processors. The 3770k even comes close to the lower clocked Ryzen 7 1800X, although it lags behind as the problem size increases. The 7700k is likely more closely matched with the Ryzen 7 1800X in terms of IPC, and comfortably pulls ahead thanks to its superior clock speed.

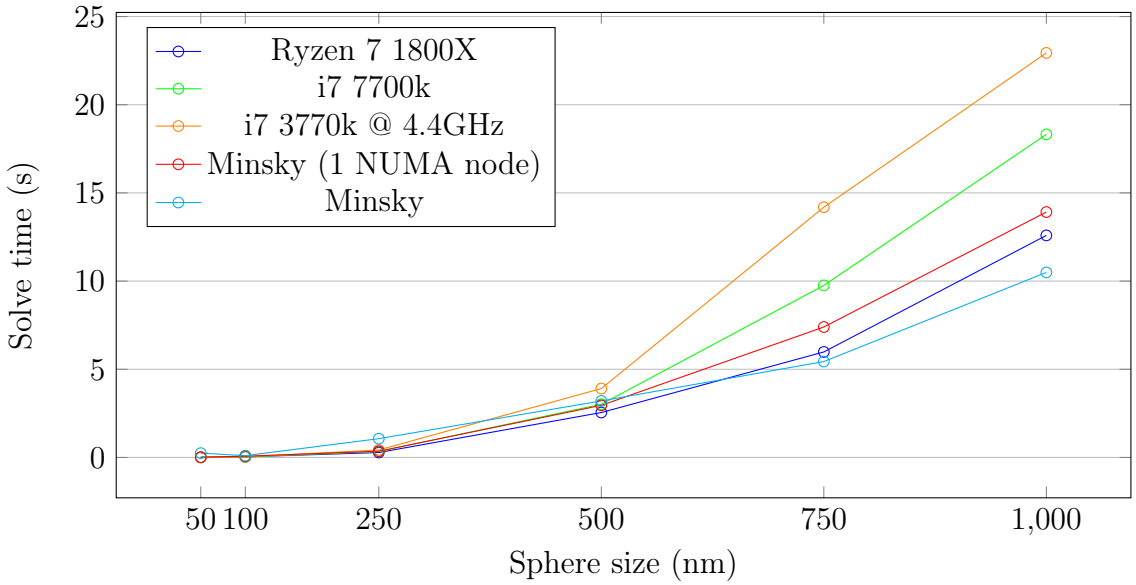

Figure 5.13: Small problem sizes on CPUs.

As Figure 5.13 shows, bringing multiple threads into the equation changes up the order quite drastically. As the problem size increases, the i7s are relegated to being the slowest, limited by only having 4 cores. As expected, the Minsky performs much better. Due to the low single-thread performance, a single Minsky CPU is slightly slower than the Ryzen 7 1800X, despite having two more cores. When using both CPUs, the Minsky takes a clear lead as the problem size increases. For the smallest problem sizes, the overhead caused by the large number of threads combined with the non-uniform memory access for the two CPUs is apparent. Up until a sphere radius of 750 nm, a single Minsky CPU actually performs better.

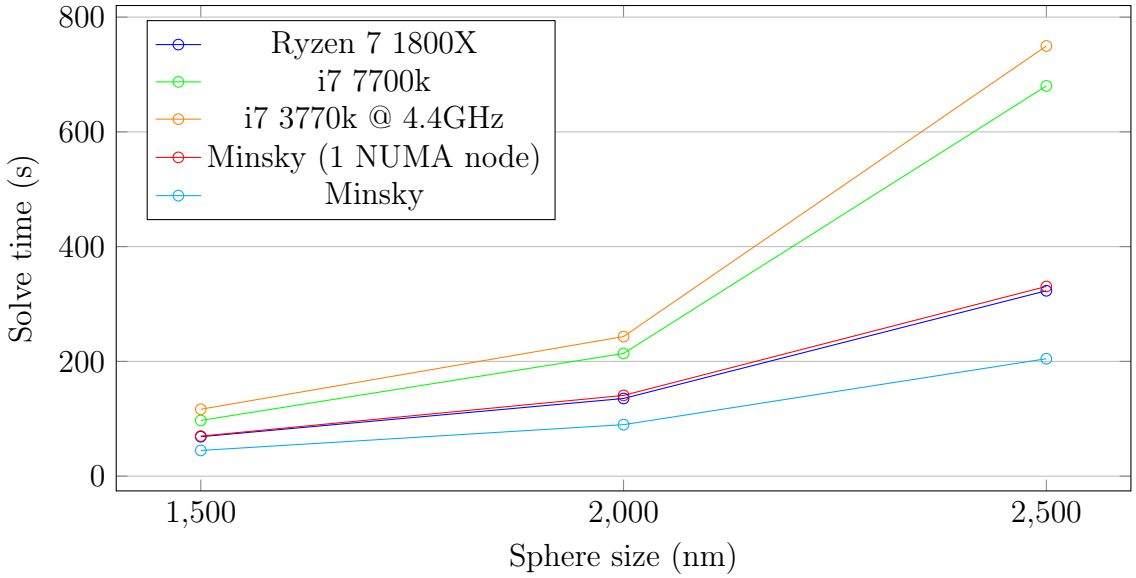

Figure 5.14: Medium problem sizes on CPUs.

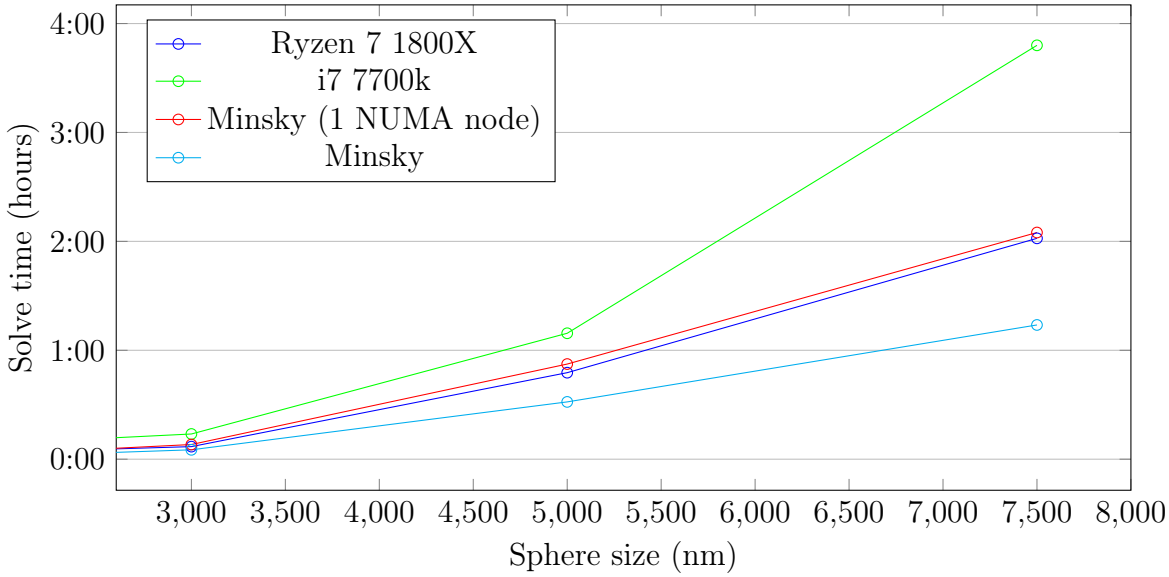

Figure 5.15: Large problem sizes on CPUs.

The trends established for the small problem sizes in Figure 5.13 continue in figures 5.14 and 5.15. As the problem size grows larger, the relative overhead from the number of threads continues to decrease. For the last of the large problem sizes, the Ryzen 7 1800X and single Minsky CPU show nearly double the performance of the i7 7700k. The Minsky is nearly twice as fast using both CPUs compared to a single CPU. This is Gustafson's law in action [19].

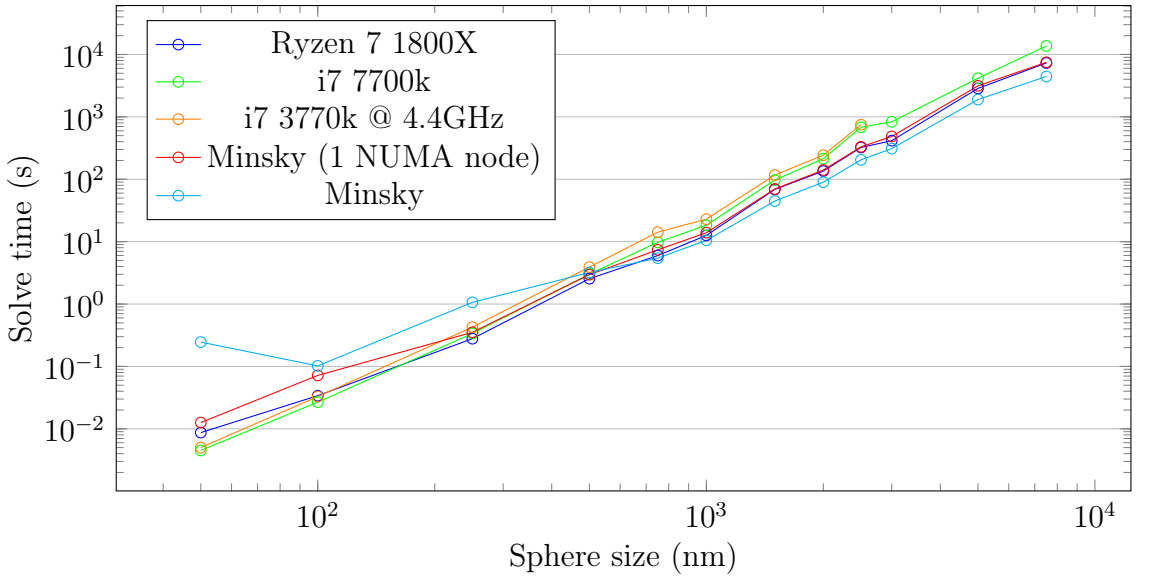

Figure 5.16: Small, medium and large problem sizes on CPUs.

5.5 Performance on GPU

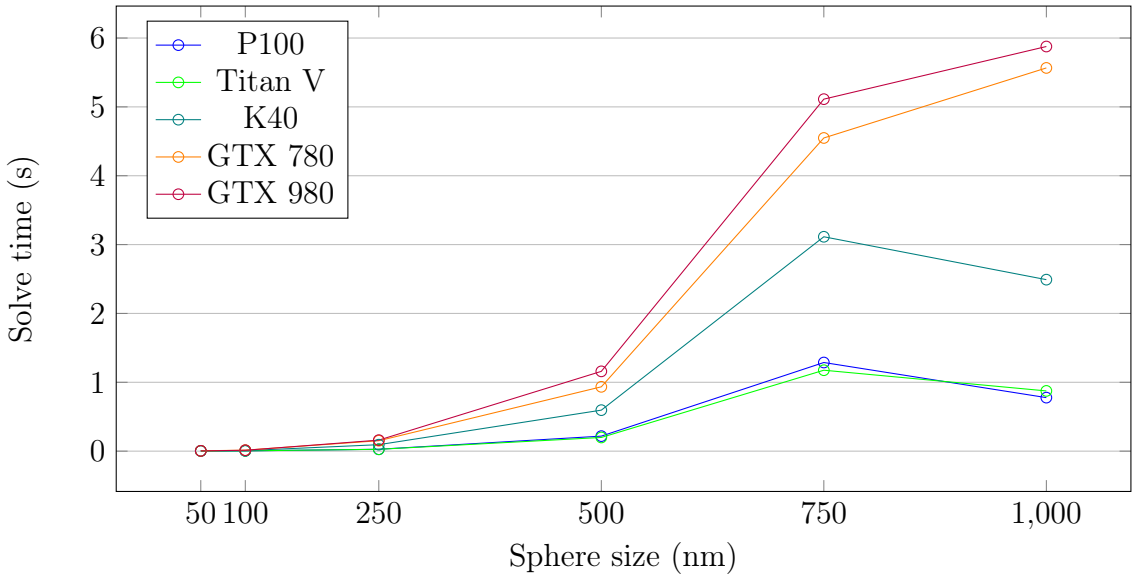

Figure 5.17: Small problem sizes on GPUs.

For the two smallest problem sizes shown in Figure 5.17, all five GPUs are fast enough to be practically indistinguishable. From sphere radius 250 nm and up, a clear pattern starts to emerge. Somewhat surprisingly, the GTX 780 is consistently faster than its successor, the GTX 980. Looking at the specs in section 5.2, we can see that the GTX 980 has around 26/29% (base/boost) more clock speed, while the GTX 780 has just over 12% more CUDA cores. For an FP32 workload, this should work out in favour of the GTX 980. However, this is an FP64 workload and the GTX 780 has 1/3 higher FP64 rate than the GTX 980, giving the overall edge to the GTX 780.

The importance of the FP64 rate is highly apparent when comparing the performance of the professional-grade Tesla K40 with the consumer-grade GTX 780, both based on the Kepler architecture. They have roughly the same amount of memory bandwidth, while the K40 has a lower clock speed (roughly 16% lower base clock), but slightly more CUDA cores (25%). Naively, this would lead you to think that the performance of the K40 should be around 5% faster than the GTX 780. The main difference between these two GPUs is the FP64 rate (and memory size), which is 8 times higher on the K40. This results in the K40 being anywhere from 50% to over 100% faster.

Figure 5.17 also shows a bit of an anomaly for the 750 nm sphere radius, with the K40, Titan V and P100 being slower than on the 1000 nm sphere radius. This likely relates directly to the findings in section 5.3, where CUDA `9.2` showed a significant speedup over CUDA `9.0` for sphere radius 250 nm and 750 nm. In other words, it is a performance problem with cuFFT. That still leaves the question of why the GTX 780/980 is not affected. It could be that cuFFT uses a different plan on these GPUs. Alternatively, it could be that the poor performance is caused by bad memory access patterns in global or shared memory (lack of coalescing or shared

memory bank conflicts). Due to the terrible FP64 rate of the GTX 780/980, these GPUs might be compute-bound even with bad memory access patterns, effectively masking the problem.

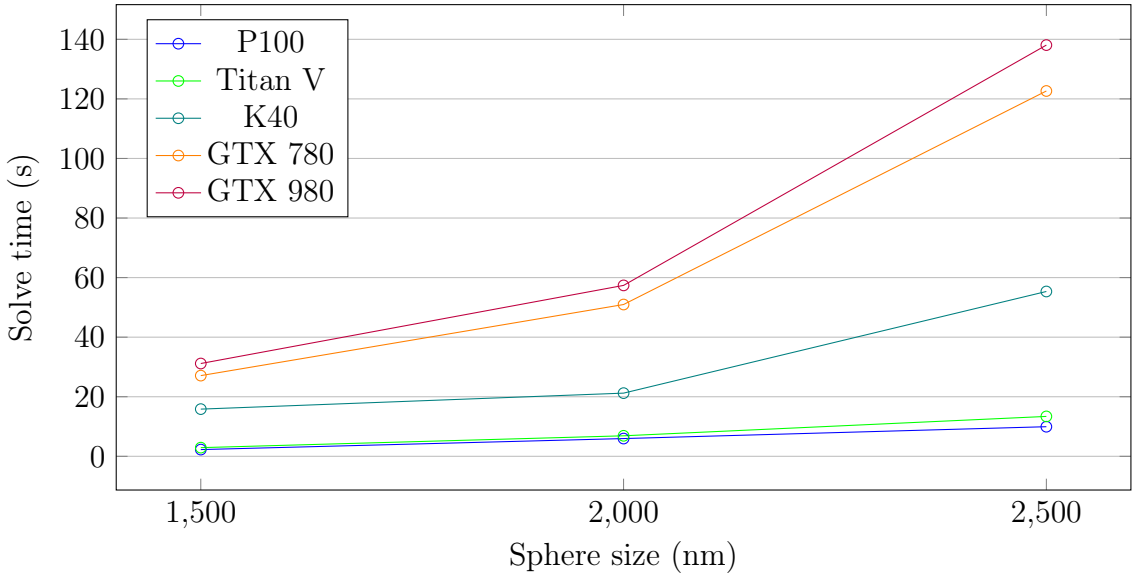

Figure 5.18: Medium problem sizes on GPUs.

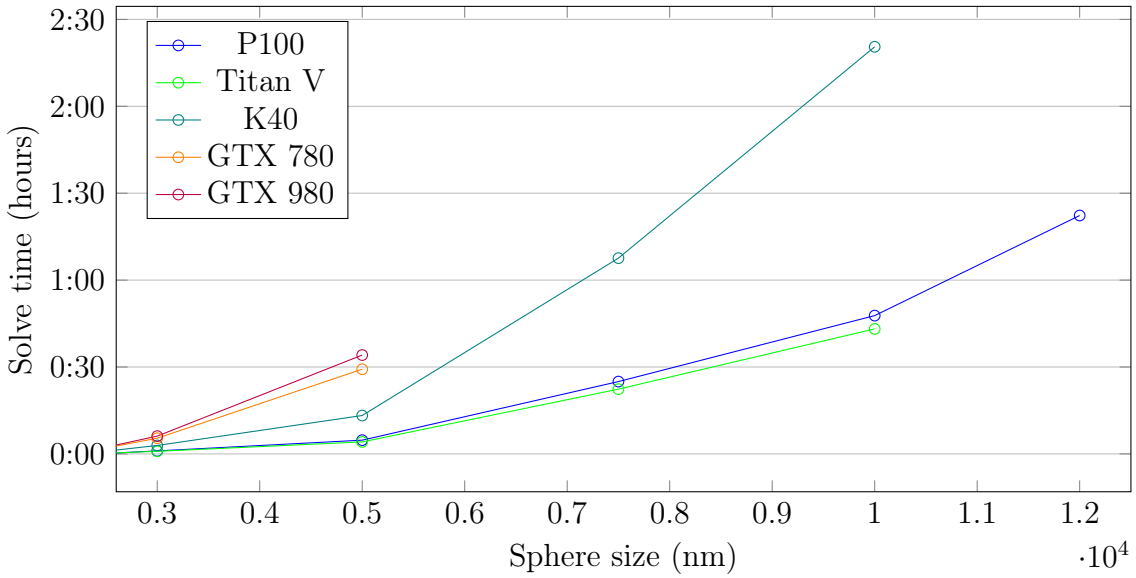

Figure 5.19: Large and huge problem sizes on GPUs.

While Figure 5.18 does not reveal anything particularly interesting, we can see that in Figure 5.19, the plots come to an end at different problem sizes. This is simply a result of running out of GPU memory and not being able to run the simulation. The GTX 780/980 make it to the 5000 nm radius with respectively 3/4GB of memory. Both the K40 and Titan V make it to 10 000 nm, with both

GPUs having 12GB of memory. Finally, the P100 makes it to 12 000 nm before exhausting its 16GB of memory.

An interesting trend that is present through all problem sizes, is that the P100 and Titan V are very closely matched. Looking at the specs (see section 5.2), it seems that the Titan V should beat the P100 by a significant margin. Although the Titan V has a slightly lower base clock, the boost clocks are closely matched. Memory bandwidth only favours the P100 by around 10%. This means that the Titan V's >40% CUDA core advantage should be enough to make a significant difference. With the P100 and Titan V trading blows in terms of being the fastest, this is obviously not the case. The intuitive conclusion might be that the workload is memory-bound, but profiling showed that this was not the source of the problem.

Figure 5.20 shows a timeline of `memcpy` and kernel execution on the P100. For every step that is solved, the solution is copied back to the host to be downsampled and stored for post-processing. This happens asynchronously, so that the next step can be solved while the previous one is being copied back to the host. For the GTX 780/980, K40 and P100, solving a step takes longer than copying the solution back to the host. As we can see in Figure 5.21, this is not the case for the Titan V.

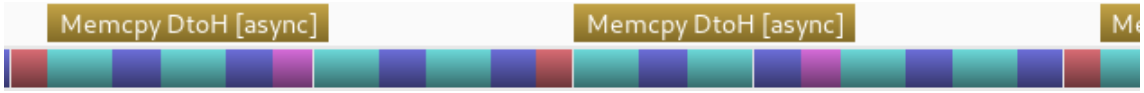

Figure 5.20: Excerpt from P100 profiling timeline showing two steps for sphere radius 2500 nm (top: `memcpy`, bottom: kernel execution).

Figure 5.21: Excerpt from Titan V profiling timeline showing two steps for sphere radius 2500 nm (top: `memcpy`, bottom: kernel execution).

Looking at the `memcpy` bandwidth for these two profiling runs, the Titan V achieves an average of 12.68GB/s. Meanwhile, the P100 achieves an average bandwidth of 33.24GB/s. This shows the massive difference in bandwidth between NVLink and PCIe 3.0, and makes it clear that the Titan V is being bottlenecked by the interconnect. The profiling data for these particular runs shows that the Titan V spent approximately 13.8 seconds copying data to the host, but only around 7.1 seconds were spent actually executing kernels. This means that a faster interconnect could nearly double the performance. For reference, the P100 spent around 9.9 seconds executing kernels for the same problem size[9], meaning that the Titan V is nearly 40% faster in terms of pure compute. This aligns much better with the specs of the two GPUs.

[9]Profiling also showed that they were running the same FFT kernels, meaning that cuFFT generated identical plans for both GPUs.

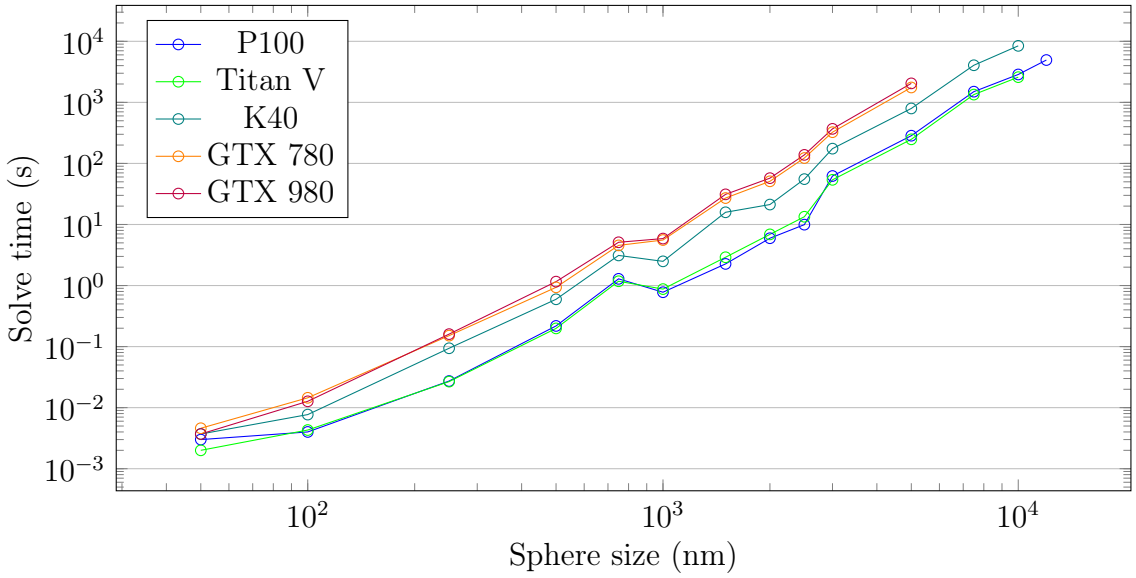

Figure 5.22: All problem sizes on GPUs.

5.6 Comparison between CPU and GPU performance

Figures 5.23, 5.24 and 5.25 show the speedup each GPU achieves compared to the i7 7700k, Ryzen 7 1800X and Minsky CPUs, using the `FFTW_MEASURE` planner flag. With the exception of the smaller problem sizes where the Minsky struggles, the overall trend is consistent. All GPUs have a dip in speedup at the 750 nm sphere radius, which is down to a performance issue with cuFFT in CUDA version `9.0`, as discussed in section 5.3. The best speedup is generally achieved for the problem sizes ranging from radius 1000 nm to 2500 nm. From 3000 nm and up, performance appears to drop off relative to the CPUs.

cuFFT typically takes up more than 80% of the execution time of the GPU-based solver (sometimes more than 90%, depending on the problem size). Because cuFFT is a closed-source library, there is very little insight into the internal workings of the planner. This in turn makes it difficult to reason about the performance characteristics in relation to the problem size.

Speaking in general terms, the key to achieving good performance on GPUs is to have as many resident warps as possible on each SM (typically referred to as occupancy). Having many resident warps (high occupancy) makes it possible for the SM to always stay busy, because there are always warps that are ready to execute. With only a few resident warps (low occupancy), there may be situations where all warps are waiting for memory accesses or other dependencies, causing the SM to sit idle. Each SM has two main resources that are divided between resident warps: registers and shared memory. If a kernel requires a large amount of registers or shared memory, this will cause poor occupancy and likely poor performance. Conversely, if the problem size is small (meaning a small number of blocks and threads), occupancy will be low regardless of the register and shared memory usage.

With these general observations, it makes sense that the speedup of the GPUs relative to the CPUs increases with the problem size up to a certain limit (reaching optimal occupancy), before declining again (contention for registers and/or shared memory hurts occupancy).

A less technical justification for the performance scaling behaviour of cuFFT could simply be that NVIDIA has spent more time optimizing for a certain range of problem sizes. As shown in section 5.3, the performance of cuFFT is a constant work in progress.

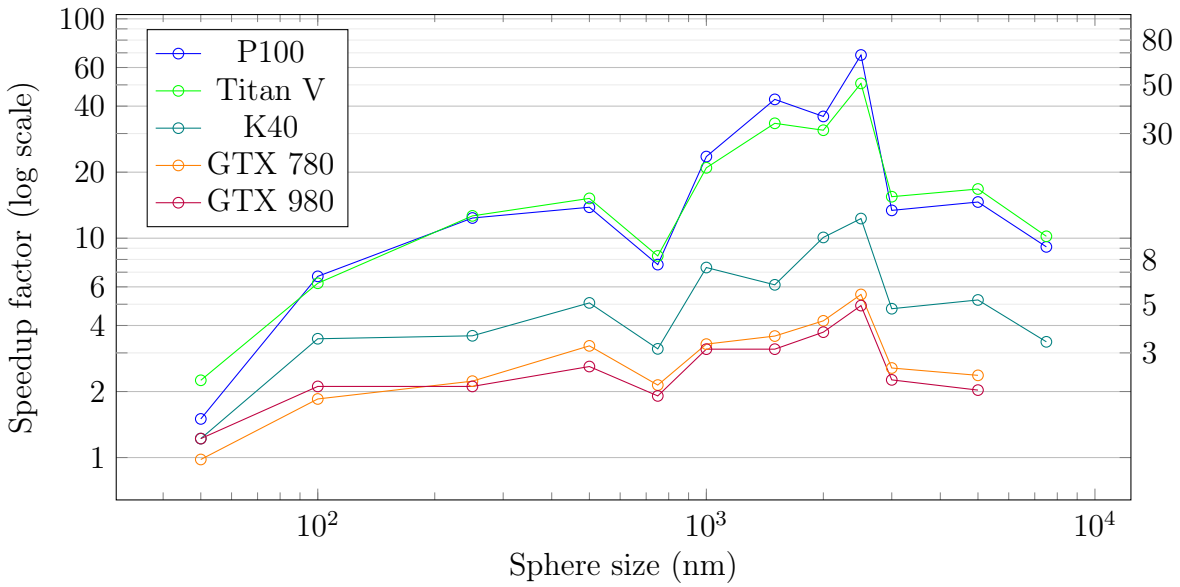

Figure 5.23: GPU speedup compared to i7 7700k.

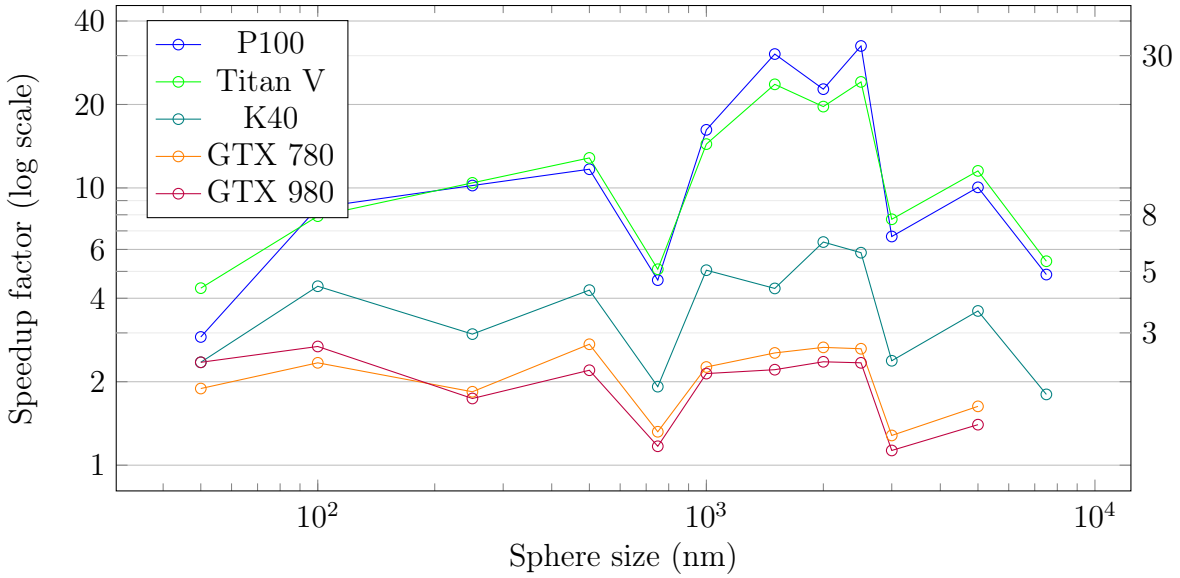

Figure 5.24: GPU speedup compared to Ryzen 7 1800X.

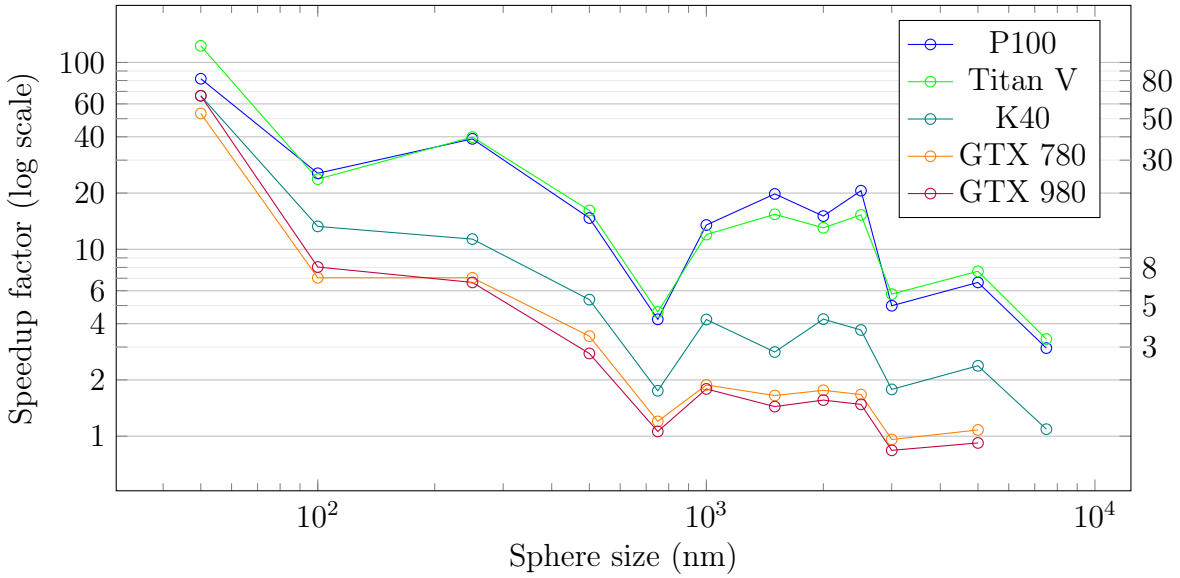

Figure 5.25: GPU speedup compared to Minsky CPUs.

Focusing on the affordable consumer-grade CPUs (i7 7700k, Ryzen 7 1800X) and high-end gaming GPUs (GTX 780/980), a respectable speedup of at least 2x is achieved for most problem sizes. For the 7700k, only the smallest problem size falls significantly below 2x, while the peak speedup is just over 5x. As mentioned in section 5.5, the FP64 performance is a real weak point for the GTX 780/980. It is likely they would have a speedup closer to the Tesla K40 if an FP32 version of both the CPU and GPU-solver was implemented. While the CPUs would likely also get some speedup with FP32, the difference should not be nearly as dramatic.

At the opposite end of the affordability-spectrum, the Tesla P100 and Titan V are faster by more than an order of magnitude for many of the problem sizes. Thanks to the price and consumer availability of these GPUs, it is not entirely fair to compare them to consumer-grade CPUs. However, even against the dual CPUs in the Minsky they still manage to be an order of magnitude faster for the moderately large problem sizes ranging from 1000 nm to 2500 nm. The Minsky CPUs fare better against the GTX 780/980 and K40, nearly matching and even beating the GPUs for the largest problem sizes.

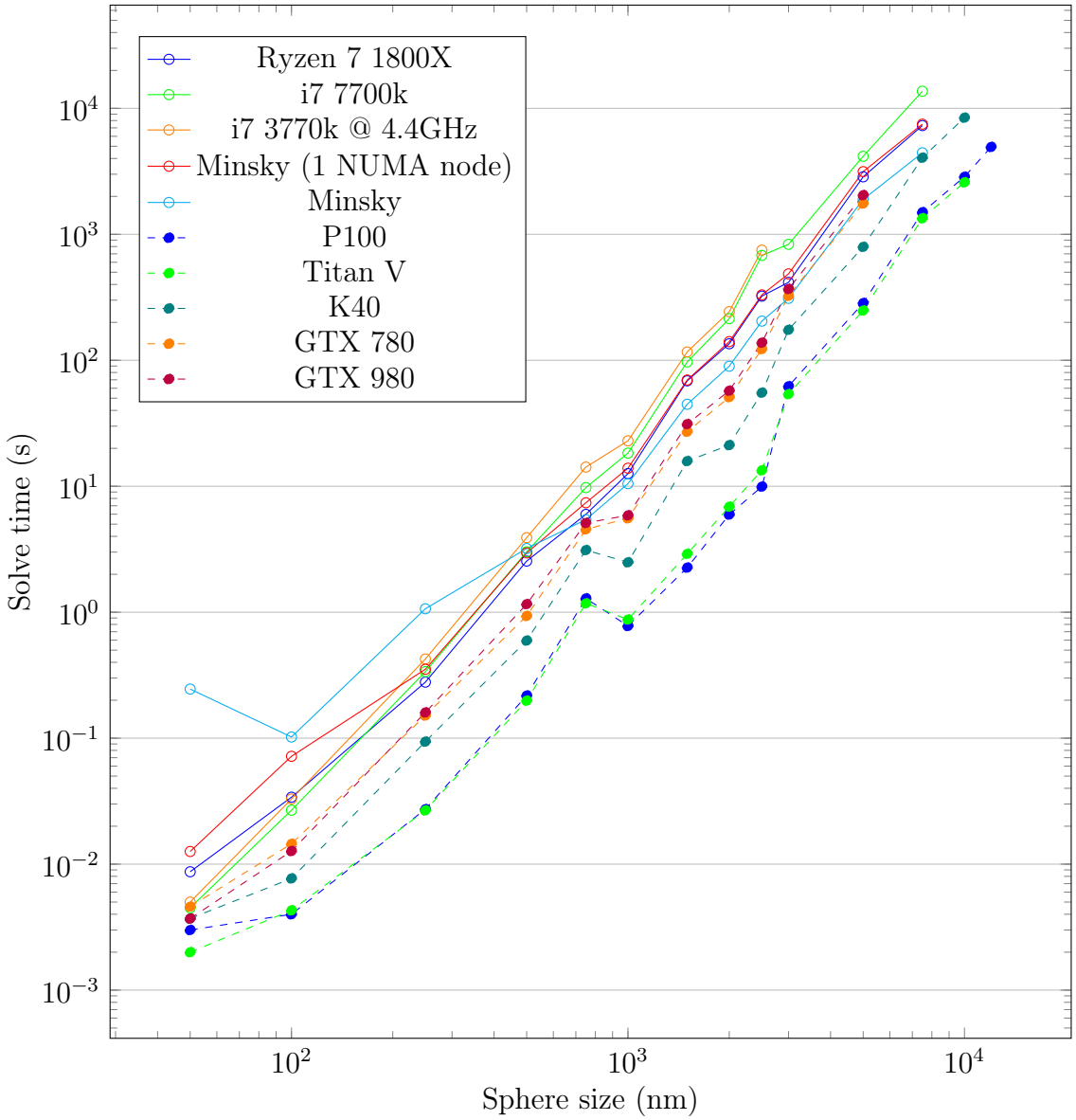

Figure 5.26: All problem sizes on all hardware.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

A GPU-based solver implementing the Split-Step Fourier Transform Method (SSFTM) was successfully implemented in PaXPro. Verification tests comparing the simulation results with the existing CPU-based implementation were used to show that the GPU-based implementation was giving correct results.

Tools for benchmarking the CPU and GPU based SSFTM solvers in PaXPro were created to be able to analyze the performance. The benchmark program itself was created to allow for a wide range of problem sizes specified at run time. Several scripts were created to automate running the benchmark for a range of problem sizes, and help the user analyze the results.

A docker image was created to allow the PaXPro benchmark to run with a near-identical configuration in terms of dependencies across a wide range of systems. The overhead added by using docker was shown to be negligible due to the compute-bound nature of the workload. This allowed a fair comparison between CPUs and GPUs to be made across a range of different systems, emphasizing the differences between the hardware and not the software.

The benchmarking results show that a good speedup was achieved for the existing SSFTM CPU solver through optimizing the usage of FFTW. Performance differences between the different CPUs tested for single thread and multithreaded benchmarks indicate that the solver has strong scaling characteristics when increasing the number of CPU cores available.

GPUs were able to achieve significant speedups, even against the optimized CPU solver. While the speedup achieved on the consumer-grade GPUs was respectable, the professional-grade GPUs were significantly more impressive. Considering that professional-grade NVIDIA GPUs are often an order of magnitude more expensive than consumer-grade NVIDIA GPUs (in some cases it is not even possible to buy standalone GPUs) somewhat diminishes these results, as they will not be attainable for the average user.

An interesting discovery was made for the Titan V GPU. Profiling revealed that the PCIe 3.0 interconnect was severely bottlenecking the performance. No other GPUs using PCIe 3.0 had issues with the bandwidth, but the Titan V has far more compute power than the other GPUs. This shows the importance of NVLink and

the coming PCIe 4.0 for future high-end GPUs [33].

6.2 Future work

As discussed in section 5.5, FP64 performance is a severely limiting factor for affordable consumer-grade GPUs. Due to the drastic difference between the FP32 and FP64 rate on these GPUs, it seems likely that a significantly better speedup could be achieved by reducing the precision. Reducing the precision does of course reduce the accuracy of the simulation results, and this is a tradeoff that has to be considered. It may not make sense to reduce the floating point precision if performance is only an issue for problem sizes large enough that the floating point precision can cause a noticeable difference in the simulation results.

In this thesis, only one of several solver methods were implemented in CUDA. Each of the solver methods have different characteristics [23] and may be interesting for different problems. For this reason, implementing the other solvers in CUDA could also be a topic for future work.

The NTNU HPC-Lab has several systems available with multiple GPUs. It could be feasible to extend the current implementation of the GPU-based SSFTM solver to work with multiple GPUs. This could potentially enable greater speedup and allow for running even larger problem sizes (assuming the simulation data can be split evenly across all GPUs).

One of the current weaknesses of the GPU-based SSFTM solver is that the user is forced to implement their own CUDA compatible `MaterialFunction` derived class to represent the refractive index. PaXPro already has the `CSGMaterial` class that supports using geometry created in OpenSCAD (`http://www.openscad.org/`), but this class in its current state is not compatible with CUDA in the sense that it can actually be used in device code. There are two main reasons for this. First and foremost, it uses some functionality from the C++ standard library, none of which is supported in CUDA device code. This functionality can likely be replaced by libraries bundled with CUDA. The second issue is that the representation of the geometry is stored in a tree structure. When copying an object of the `CSGMaterial` class to the device, this tree structure must be recreated on the device. This is not straightforward, because the nodes in the tree are objects of a class with virtual functions. As discussed in subsection 4.3.1, there are some limitations that apply when transferring objects with virtual functions from host to device (or vice versa). This means that the tree structure must be recreated with kernels that create the correct nodes on the device, as opposed to merely copying the node objects from the host and updating the pointers.

References

[1] *AMD Ryzen™ 7 1800X Processor*. URL: https://www.amd.com/en/products/cpu/amd-ryzen-7-1800x (visited on 24/07/2018).

[2] Gene M. Amdahl. 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities'. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[3] *ASUS GeForce® GTX 780 DirectCU II*. URL: https://www.asus.com/Graphics-Cards/GTX780DC2OC3GD5/specifications/ (visited on 24/07/2018).

[4] *Callgrind: a call-graph generating cache and branch prediction profiler*. URL: http://valgrind.org/docs/manual/cl-manual.html (visited on 01/08/2018).

[5] B. Cloutier, B. K. Muite and P. Rigge. 'Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms'. In: *2012 Symposium on Application Accelerators in High Performance Computing*. July 2012, pp. 145–148. DOI: 10.1109/SAAHPC.2012.24.

[6] *CMake 3.8 Release Notes*. URL: https://cmake.org/cmake/help/v3.8/release/3.8.html (visited on 23/07/2018).

[7] *CUDA C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 01/08/2018).

[8] Ian Cutress. *The AMD Zen and Ryzen 7 Review: A Deep Dive on 1800X, 1700X and 1700*. 2nd Mar. 2017. URL: https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700 (visited on 24/07/2018).

[9] *Debian "stretch" Release Information*. URL: https://www.debian.org/releases/stretch/ (visited on 22/07/2018).

[10] *Fedora 25 Schedule*. URL: https://fedoraproject.org/wiki/Releases/25/Schedule (visited on 22/07/2018).

[11] W. Felter et al. 'An updated performance comparison of virtual machines and Linux containers'. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[12] *FFTW FAQ - Section 3: Using FFTW*. URL: http://www.fftw.org/faq/section3.html (visited on 29/07/2018).

[13] *FFTW3 Docs: Planner Flags*. URL: http://www.fftw.org/fftw3_doc/Planner-Flags.html (visited on 28/07/2018).

[14] *FFTW3 GitHub Issue #18: " error: identifier "__float128" is undefined " when using CUDA*. URL: https://github.com/FFTW/fftw3/issues/18 (visited on 23/07/2018).

[15] M. J. Flynn. 'Some Computer Organizations and Their Effectiveness'. In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[16] *GeForce GTX 780 - Specifications*. URL: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications (visited on 24/07/2018).

[17] *GeForce GTX 980 - Specifications*. URL: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications (visited on 24/07/2018).

[18] N. K. Govindaraju et al. 'High performance discrete Fourier transforms on graphics processors'. In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5213922.

[19] John L. Gustafson. 'Reevaluating Amdahl's Law'. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[20] *Intel® Core™ i7-3770K Processor*. URL: https://ark.intel.com/products/65523/Intel-Core-i7-3770K-Processor-8M-Cache-up-to-3_90-GHz (visited on 24/07/2018).

[21] *Intel® Core™ i7-7700K Processor*. URL: https://ark.intel.com/products/97129/Intel-Core-i7-7700K-Processor-8M-Cache-up-to-4_50-GHz (visited on 24/07/2018).

[22] *JsonCpp GitHub Issue #486: nvcc + gcc -> error*. URL: https://github.com/open-source-parsers/jsoncpp/issues/486 (visited on 23/07/2018).

[23] David Kleiven. 'Simulation of X-ray Propagation in Guiding Structures and Light Scattering From Coccoliths Using Finite Difference Methods'. MA thesis. Norwegian University of Science and Technology (NTNU), 2017.

[24] Paul Kocher et al. 'Spectre Attacks: Exploiting Speculative Execution'. In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[25] Mireille Levy. *Parabolic equation methods for electromagnetic wave propagation*. 45. IET, 2000.

[26] Moritz Lipp et al. 'Meltdown'. In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[27] *NVIDIA CUDA Runtime API*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 01/08/2018).

[28] *NVIDIA CUDA Toolkit Release Notes (CUDA 9.1)*. URL: https://docs.nvidia.com/cuda/archive/9.1/cuda-toolkit-release-notes/index.html (visited on 25/07/2018).

[29] *NVIDIA CUDA Toolkit Release Notes (CUDA 9.2)*. URL: https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html (visited on 25/07/2018).

[30] *NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built*. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 02/08/2018).

[31] *nvidia/cuda Docker Repository*. URL: https://hub.docker.com/r/nvidia/cuda/ (visited on 23/07/2018).

[32] *nvidia/cuda-ppc64le Docker Repository*. URL: https://hub.docker.com/r/nvidia/cuda-ppc64le/ (visited on 23/07/2018).

[33] Nate Oh. *PCI-SIG Finalizes and Releases PCIe 4.0, Version 1 Specification: 2x PCIe Bandwidth and More*. 26th Oct. 2017. URL: https://www.anandtech.com/show/11967/pcisig-finalizes-and-releasees-pcie-40-spec (visited on 03/08/2018).

[34] J. D. Owens et al. 'GPU Computing'. In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[35] *Releases - Ubuntu Wiki*. URL: https://wiki.ubuntu.com/Releases (visited on 22/07/2018).

[36] G. Ruhl et al. 'IA-32 Processor with a Wide-Voltage-Operating Range in 32-nm CMOS'. In: *IEEE Micro* 33.2 (Mar. 2013), pp. 28–36. ISSN: 0272-1732. DOI: 10.1109/MM.2013.8.

[37] Bahaa EA Saleh, Malvin Carl Teich and Bahaa E Saleh. *Fundamentals of photonics*. Vol. 22. Wiley New York, 1991. DOI: 10.1002/0471213748.

[38] Ryan Smith. *NVIDIA Announces Tesla P100 Accelerator - Pascal GP100 Power for HPC*. 5th Apr. 2016. URL: https://www.anandtech.com/show/10222/nvidia-announces-tesla-p100-accelerator-pascal-power-for-hpc (visited on 24/07/2018).

[39] Ryan Smith and Nate Oh. *The NVIDIA Titan V Preview - Titanomachy: War of the Titans*. 20th Dec. 2017. URL: https://www.anandtech.com/show/12170/nvidia-titan-v-preview-titanomachy (visited on 24/07/2018).

[40] *TESLA K40 GPU ACCELERATOR*. Nov. 2013. URL: http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf (visited on 24/07/2018).

[41] S. E. Thompson. 'Power, cost and circuit IP reuse: The real limiter to Moore's Law over the next 10 years'. In: *Proceedings of 2010 International Symposium on VLSI Technology, System and Application*. Apr. 2010, pp. 88–89. DOI: 10.1109/VTSA.2010.5488936.

[42] Scott Vetter, Alexandre Bicas Caldeira and Volker Haug. *IBM Power System S822LC for High Performance Computing Introduction and Technical Overview*. 26th Oct. 2016. URL: http://www.redbooks.ibm.com/abstracts/redp5405.html?Open (visited on 24/07/2018).

Appendices

Appendix A

Source Code

The git repositories used in this project are shown in Table A.1. Note that the docker repository has the PaXPro and VISA repositories as git submodules with relative URLs. This means that they must have the same parent in the URL (in the case of GitHub, this means that the same user must host all three repositories).

Table A.1: Git repositories used for this project.

Repository	URL
PaXPro	https://github.com/acelster/paxpro-ntnu.git/
VISA[1]	https://github.com/acelster/visa-ntnu.git/
Docker	https://github.com/acelster/paxpro-docker.git/

[1]VISA is a dependency of PaXPro used for visualization in some solvers.