NTNU
Norwegian University of
Science and Technology

Optimizing & Parallelizing a Large Commercial Code for Modeling Oil-well Networks

Atle Rudshaug

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Since the entry of recent programmable GPUs (Graphics Processing Units), the GPUs are becoming an increasingly interesting platform for more general computations. Much like the math co-processors of the 1980's, GPUs can speed up the CPU by taking care of arallelized compute-heavy calculations.

In this project, we investigate whether a large commercial application that models a network of oil wells, can benefit from off-loading computations to the GPUs. However, to make this project worth-while, the applications first needs to be profiled and optimized. Other modern parallel architectures such as multicore PCs and workstations may also be considered as time permits.

Assignment given: 15. January 2008
Supervisor: Anne Cathrine Elster, IDI

Abstract

In this project, a complex, serial application that models networks of oil wells is analyzed for today's parallel architectures. By heavy use of the profiling tool Valgrind, several serial optimizations are achieved, causing up to a 30-50x speedup on previously dominant sections of the code, on different architectures. Our initial main goal is to parallelize our application for GPGPUs (General Purpose Graphics Processing Units) such as the NVIDIA GeForce 8800GTX. However, our optimized application is shown not to have a high enough computational intensity to be suitable for the GPU platforms, with the data transfer over the PCI-express port showing to be a serious bottleneck.

We then target our applications for another, more common, parallel architecture – the multi-core CPU. Instead of focusing on the low-level hotspots found by the profiler, a new approach is taken. By analyzing the functionality of the application and the problem it is to solve, the high-level structure of the application is identified. A thread pool in combination with a task queue is implemented using PThreads in Linux, which fit the structure of the application. It also supports nested parallel queues, while maintaining all serial dependencies.

However, the sheer size and complexity of the serial application, introduces a lot of problems when trying to go multithreaded. A tight coupling of all parts of the code, introduces several race conditions, creating erroneous results for complex cases. Our focus is hence shifted to developing models to help analyze how suitable applications with traversal of dependence-tree structures, such as our oil well network application is, given benchmarks of the node times.

First, we benchmark the serial execution of each child in the network and predict the overall parallel performance by computing dummy tasks reflecting these times on the same tree structure on two given well networks, a large and a small case. Based on these benchmarks, we then predict the speedup of these two cases, with the assumption of balanced loads on each level in the network. Finally, the minimum amount of time needed to calculate a given network is predicted. Our predictions of low scalability, due to the nature of the oil networks in the test cases, are then shown.

This project thus concludes that the amount of work needed to successfully introduce multithreading in this application might not be worth it, due to all the serial dependencies in the problem the application tries to solve. However, if there are multiple individual networks to be calculated, we suggest using Grid technology to manage multiple individual instances of the application simultaneously. This can be done either by using script files or by adding DRMAA API calls in the application. This, in combination with further serial optimizations, is the way to go for good speedup for these types of applications.

Acknowledgments

First, I would like to thank Associate Professor Dr. Anne C. Elster at IDI-NTNU for supervising this project, and for involving me in three different conferences in the process.

Second, thanks to Yggdrasil AS for providing their large production code and test cases as a basis for this thesis. A special thanks to Asbjørn Sigurdsøn for numerous detailed discussions and help with multithreading problems, Gudbrand Nerby for detailed descriptions of the algorithm and Ole Jacob Velle for providing the test cases.

I would also like to thank my fellow students, especially Rune E. Jensen, for numerous discussions on optimization techniques and other implementation ideas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When the speed barrier was hit, and multi-core technology introduced, serially programmed applications lost their main speed contributor, the increase in CPU frequency. For these applications to be competitive in the future, they have to be adapted to be able to utilize parallel technology.

However, multi-core CPUs is not the only option. The gaming industry's constant craving for more graphical computing power, has spawned a technology capable of immense parallel computation, the GPU. Lately, the science community has been playing with this toy, to speed up their highly computationally intensive tasks.

Another contributor in the era of parallel computation, is the CELL processor. This, unlike the previously mentioned technologies, is a heterogeneous design with one central general-purpose core and eight special-purpose compute cores. It can be looked as a combination of a CPU and a GPU in terms of the combination of general- and special-purpose cores. However, currently the CELL processor is only available in the PlayStation 3 multimedia console, and in expensive server rack units, unlike CPUs and GPUs which can be found in almost every PC or workstation.

1.1 Project Goal

Adapting an existing serial applications to utilize multiple cores, is a common problem in today's industry. Even writing multithreaded code from scratch is challenging without proper tools and training.

The goal of this project is to try to adapt a large and complex, serially coded application, to two of today's parallel architectures, the GPU and multi-core CPUs. The costs of offloading CPU cycles on the GPU shall be identified, while extensive profiling will be used to locate target code. Different parallelization and optimization opportunities will also be discussed. In the same process, possibilities for serial optimizations, prior to parallelization, will be identified and performed.

Finally, the applications scalability and speedup gained by utilizing parallel architectures, shall be identified.

1.2 Our Application

The application used in this project is supplied by Yggdrasil A/S (`http://www.yggdrasil-as.no`). It is a flow-optimization application for petroleum oil-well networks, which can be used to perform network simulations with input from reservoir simulators and flow monitoring of production networks. It uses several equation solvers for calculating pressure loss from the wells to the terminal, and it automatically tweaks actuator input for optimal flow through arbitrary oil-field networks. Transient momentum and temperature calculations are targeted, as well as steady state calculations.

Network simulation can be performed over several years, with variable time step intervals, using minutes to weeks between each time step. This is an inherently serial operation, since the results from each time step is used as input to the next. However, multiple individual pressure system-/network topology configurations can be simulated for each time step, allowing for embarrassingly parallel computations. This operation can add valuable information to which network topology is optimal for each time step, or over time.

The application calculates the pressure loss and phase flows from the reservoir to topside for each network configuration. The oil networks are basically dependence trees, with interdependencies between each level in the tree. This means that all child nodes must be calculated before their parents. When calculating the network serially this is no problem, however, if computed in parallel, care must be taken to preserve the correct order of execution.

For the steady-state calculations, rather simple equation solvers are used to compute the actual pressure and phase flow characteristics in each pipe in the network. The transient momentum and temperature calculations, however, are far more advanced, and might be suited for GPU computing.

1.3 Outline

The rest of the report is structured in the following way:

- Chapter 2 describes the serial optimizations performed in this project.

- Chapter 3 gives an introduction to topics related to parallel computing. It starts off with two important laws related to the maximum parallel speedup possible in an application. Next, an introduction to some new challenges introduced when writing parallel code is presented. A detailed introduction to GPU history, design and how they are programmed is given, followed by an introduction to different tools available for parallelizing applications for shared memory multi-core CPUs.

- Chapter 4 starts with a brief introduction to interesting related work and technology. Next, it describes the main target algorithm in the application, followed by the strategy used to locate target code for parallelization on GPU and multi-core technology. Three models are then described, which all predict potential parallel speedup in the algorithm in different ways. Finally, the test cases are described, followed by a summary of the potential speedup predicted by the models.

- Chapter 5 starts with a brief introduction to available GPU profilers and the hardware used for the parallel implementations. This is followed by a debugging strategy for multi-core implementations. Next, the results of different GPU implementations is presented, and their results are discussed. Finally, different multi-core CPU implementations are described, and their results presented.

- Chapter 6 concludes the work performed in this project, and presents different ideas for future work.

- Appendix A gives a short description of the thread pool and its files. It also gives an example on how to use it and how to include it as a library in an application.

- Appendix B lists all the benchmark related bash scripts used and a ping-pong test for CUDA enabled GPUs.

- Appendix C includes a poster presented at NOTUR 2008, which summarizes this project.

Chapter 2

Serial Optimizations

This project focuses extensively on using profilers to locate inefficiencies and hotspots in an application. This chapter starts off with an introduction to different profilers, and the choice of profiler for this project is presented. Next, the benchmarking method and the hardware used is described.

The target application had not been optimized thoroughly prior to this project and it was seen as necessary to perform some optimizations before introducing parallelism to the application. A detailed description of each optimization performed is described in this chapter.

2.1 Profiling

By profiling the application, hotspots and bottlenecks can be located and targeted for optimizations and parallelization. Finding these areas of an application is difficult without using special tools.

If different parts of a current CPU code are to be moved to a GPU, the current CPU code should be profiled first. This will locate where the performance hits are, and will help see what parts can be done in parallel. Profiling and improving the CPU code, may remove unnecessary overhead before moving parts over to the GPU. If the parts that originally used a lot of CPU-time executing a lot of unnecessary instructions, after a cleanup of the code, these might not be the best parts to target anymore.

There are different tools for profiling CPU code, including the ones below:

- Valgrind
  - Callgrind + KCachegrind
  - Helgrind
  - Memcheck
- Rational Quantify

- Intel VTune + Intel Thread Checker
- AMD CodeAnalyst
- gprof
- TAU

In this project, Valgrind and its tool Callgrind became the preferred profiler. Intel VTune and TAU were tested as well, but they always crashed with unknown errors or segmentation faults during profiling.

2.1.1 Valgrind + KCachegrind

Valgrind is a profiler available in Linux, which can generate profiles for any application. Different tools can be used with Valgrind, such as Callgrind and Memcheck. The output from Callgrind can be quite unreadable and luckily there is a tool, called KCachegrind, which can give a graphical representation of the data. Calling an application with Valgrind is as simple as:

```
valgrind -v –tool=callgrind ./executable -args
```

Using *callgrind* as the tool, will give a good representation of the CPU-time used for each function in the application. The graphical representation by KCachegrind gives an excellent overview and makes it simple to target exact lines of code, by showing CPU-usage for each line inside the source code. This tool will be used extensively throughout this project.

2.2 Benchmarking

Benchmarks will be performed using the scripts in Appendix B. These scripts are used to automatically launch different binaries, a user defined amount of times. Wall clock is used as timing parameter, by using the built in *"date"* application in Linux. Speedup listings are acquired by taking the average of 5 runs for each calculation, which is automatically done by the benchmark scripts.

2.2.1 Hardware Used

Table 2.1: Serial Optimization Benchmark Hardware

Hardware	HP	ASUS W5F Laptop	AMD64
CPU(s)	4 x Intel Xeon X7350	1 x Intel Centrino Duo	AMD64
Freq.	2.93 GHz	1.66 GHz	3500+
Num. Cores	16	2	1
RAM	64 GB	2.5 GB	2GB
OS.	CentOS 5 64-bit	Ubuntu 7.10 32bit	Ubuntu 7.10 64bit
Compiler	gcc 4.1.2	gcc 4.1.2	gcc 4.1.2

2.3 Optimizing and Profiling Our Application

This section describes the serial optimizations performed, and the speedup gained for both the steady-state and transient calculations. Multiple screen shots from KCachegrind are included, showing the effects of each optimization on the profile. We show that there is a lot of potential for serial optimizations, which can result in high speedup.

2.3.1 Steady State Calculation

The program at hand was profiled extensively to locate the critical sections of the code. Figure 2.1, shows a profile of the original code, using a combination of Valgrind and KCachegrind. The final optimized code, results in the profile image shown in Figure 2.2. Here the encircled function calls from the former profile are gone. The other optimizations only alter the percentage usage of the different targets and are not easily observed in the image. Here follows a short description of each optimization:

1. The calls represented by the small squares, inside the function *Pipe::CalcDP*, consists of 3 instances of the following mathematical function inside pvt-tab::propertyFromVect:

   $$x = a + (b - a) \,/\, (c - d) * (p - d)$$

   In one case, this function was called about 1 396 800 000 x 3 times. It was observed that this function was called multiple times with the same c, d and p parameters, thus resulting in multiple unnecessary math operations. The operator "/" is much slower than " $*$ ", and should be called as few times as possible. Thus, the formula was altered to the following:

   $$tmp = 1.0/(c - d) * (p - d) = (p - d)/(c - d)$$

   $$x = a + (b - a) * tmp$$

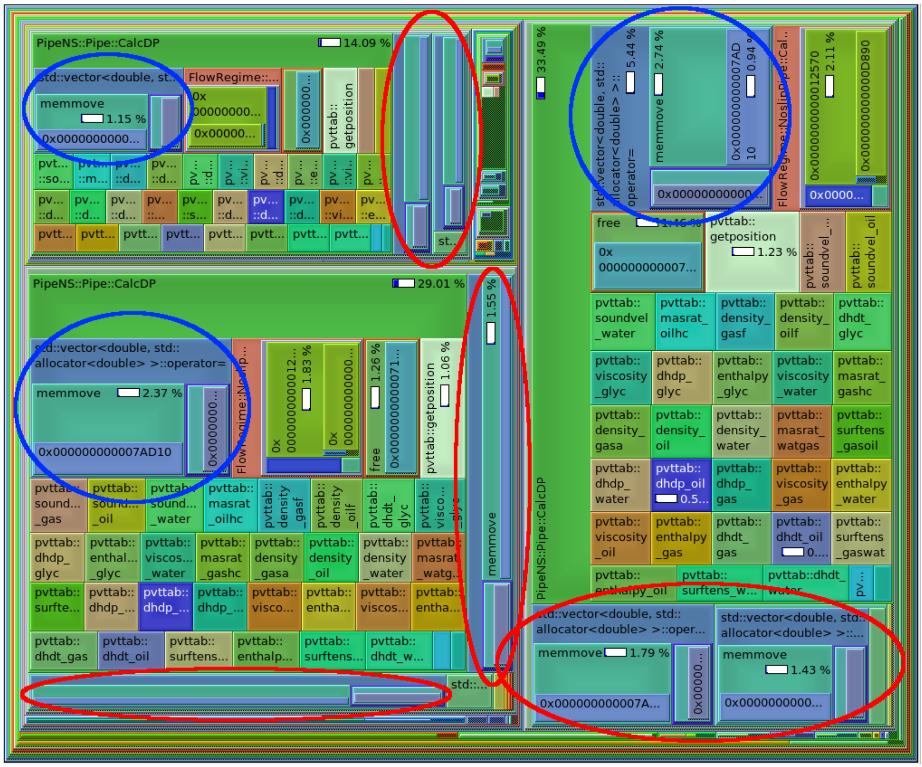


Figure 2.1: Original steady state profile with target with target optimization code encircled

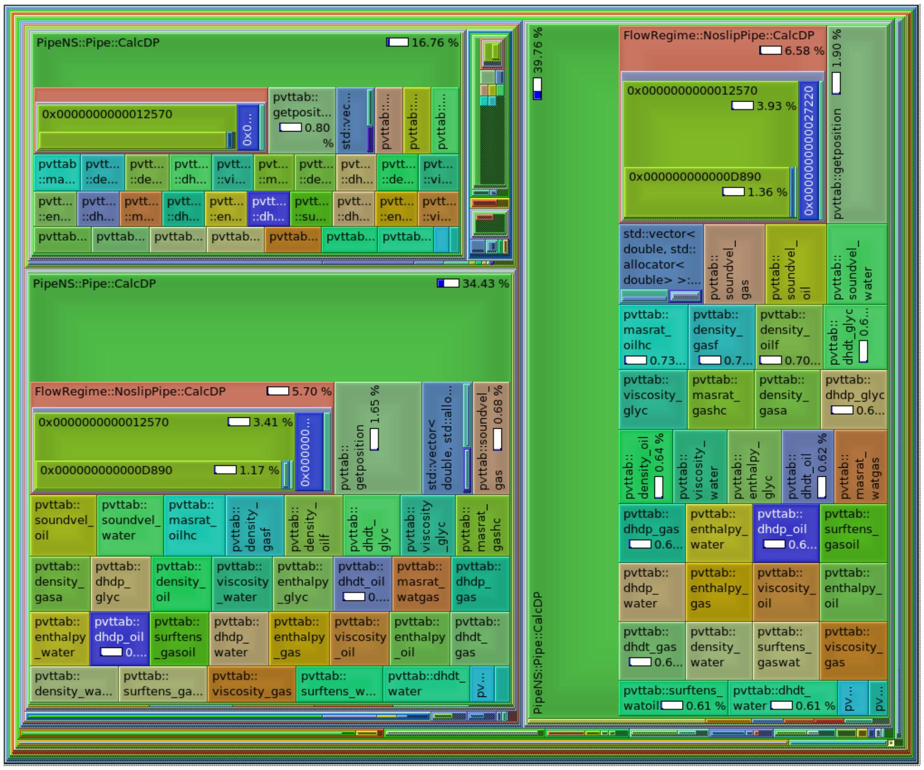


Figure 2.2: Optimized steady state profile. Notice the encircled code from Figure 2.1 is now gone.

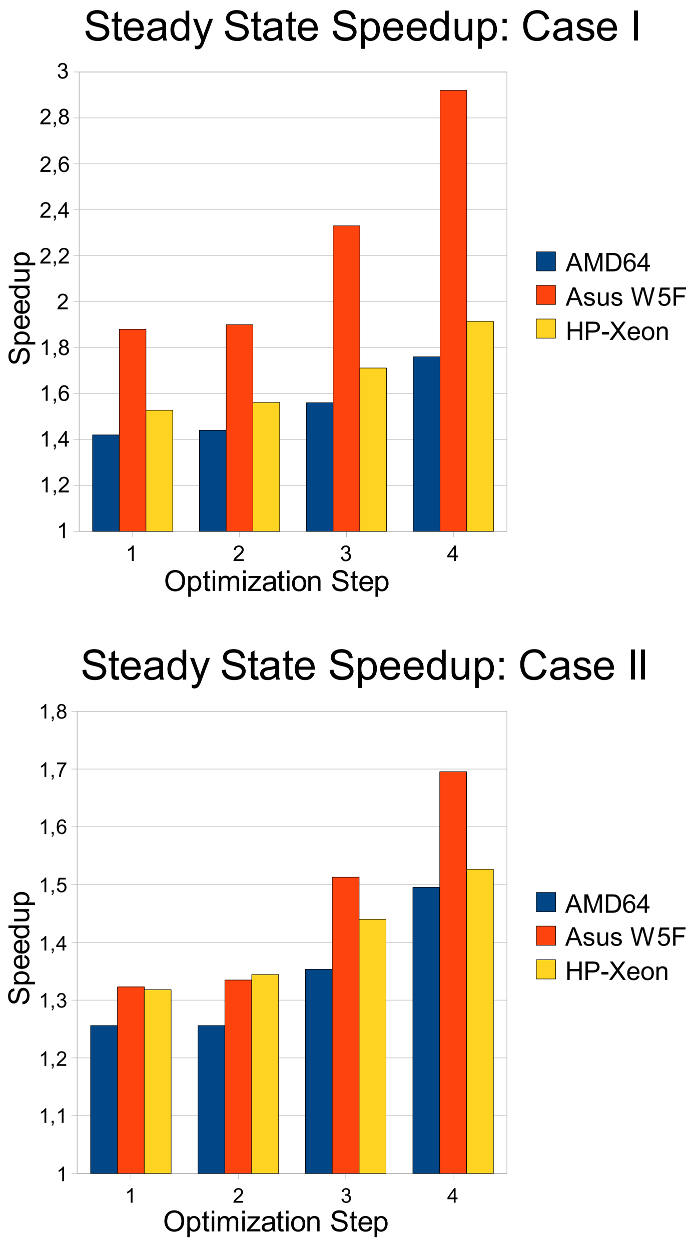
The *tmp* variable is calculated inside the function *pvttab::getPosition,* where the variables *c* and *d* are acquired. The function *pvttab::getPosition* is always executed before the target mathematical functions, to get indices to the correct input variables for each of them. Thus, the pre-calculated *tmp* variable can be reused many times.

2. *pvttab::getPosition,* searches through arrays to find the correct array-positions for the respective input variables, p and t. The array-index before and after the variable's position are saved for later use. A simple linear search algorithm was originally used, which is far from the fastest search algorithm available. However, these arrays can have lengths of only 10 to 20. Thus, a formidable speedup of changing the search algorithm is not expected, however, if these arrays grow in size, a better algorithm than linear search should give better results. The two arrays were also observed to be constant throughout the current case. Thus, a table look up algorithm seemed perfect, but the input variables are floats that have small variations for each call to this function. An exact value is not to be found, only the two positions in the array the input variable is in between. It was also observed that the input variables could be outside the scope of the array, resulting in the function returning the extremal indices; 0 for lower values and length-1 for higher. Two simple if-tests could, in many cases, eliminate the search altogether, by directly testing the two extremal values and setting the indices accordingly. Thus, getPosition was altered to first test the extremal values, then to use a the simple binary search algorithm when needed. Compared to the last optimization, this reduced the CPU usage of the *getPosition* function by only 0.75%.

3. The original supplied code has been written for readability and understandability, and has not been optimized i any way. Many places, different "in" and "out" objects of the same type are used for readability. Many of these objects contain multiple arrays, resulting in many copies of arrays between objects. Local "tmp_array" variables were also used, which resulted in many unnecessary array copies. It was observed that this could be avoided by getting references to the arrays and objects, instead of first copying them to local tmp-variables, then altering them and ultimately copying them back to the object's array. By checking the "in" and "out" objects for equality, correct execution could be maintained when directly updating one of the objects instead of copying all the data back and forth. This removed the calls encircled in red in Figure 2.1.

4. An optimization similar to the former, was performed on the calls encircled in blue in Figure 2.1, thus removing these unnecessary computations from the program execution. Figure 2.2 shows the profile after the above optimizations were performed, resulting in even more speedup in the computations.

Figure 2.3, shows to speedup gained in each optimization step of the steady state's calculation. It shows an interesting result, which is the higher speedup



Steady State Speedup: Case I

Steady State Speedup: Case II

Figure 2.3: Steady state speedup on different architectures

Listing 2.1: Return by reference removed large portions of Figure 2.6

```
const WallSpec GetWallSpec()const { return _pwspec;}

//****************altered to:***********************//

const WallSpec& GetWallSpec()const { return _pwspec;}
```

gained on the slower laptop computer, compared to the other architectures. This might have something to do with memory bus saturation, where the slower CPU frequency of the laptop might, to a lesser extent, saturate its memory bandwidth. It's important to keep in mind that the total run time of the calculations were much shorter on the other two computers.

During the project period, the application provided by Yggdrasil AS went through some major modifications, including some design alterations and bug fixes. Some of these updates resulted in very different run times compared to the initial revision, which shows how important it is not to use too much effort optimizing small, narrow parts of non-fixed code, at least without close cooperation between developers. Thus, a single revision was chosen for the duration of the project, to keep the run times consistent.

2.3.2 Transient Temperature Calculation

In this calculation, a transient temperature calculation is performed in addition to the original steady state flow calculation. The transient part is not in production code yet, but will be in the future. Figure 2.4, shows that it is highly unoptimized. More than 83% of the CPU-time is used for database storage. For each step in the calculation, a data frame is written to a SQLite database file. The database function used is designed for the steady state calculation which has totally different database needs. A new data storage design is needed for the transient calculation.

An alternative function was implemented, where the data is saved directly to a binary file during computation. When the calculations are done, the data can be retrieved from the binary file and written to the database. This makes it possible to focus on other optimizations other than the database in the rest of this project. Figure 2.5, shows the profile when saving to a binary file instead of the database. The speedup can be seen in Figure 2.11, optimization step i).

Figure 2.6, is a profile of the total CPU usage of the temperature transient calculation. As seen in the profile, creating and destructing instances of *WallSpec*, uses 2.43% of the CPU-time. This might be too little of what should be focused on initially when optimizing, but removing this was as simple as returning the WallSpec object by reference instead of copying it, as shown in Listing 2.1.

The std::vector:...:operator= was removed by using pointers and references

Listing 2.2: Many areas of the code was changed like this, using pointers and references to arrays

```
vector<double> kwal;
...
kwal = ppipe->GetWallSpec().GetSpecK();
...
kwalP1 = kwal;
...

//****************altered to:***********************//

vector<double> *p_kwal;
...
p_kwal = &ppipe->GetWallSpec().GetSpecK();
...
p_kwalP1 = p_kwal;
...
const vector<double> & kwal = *p_kwal;
...
```

to the vectors, instead of copying the arrays to new vectors in memory. There were multiple instances of similar code, as shown in Figure 2.2, which copied the arrays between different vector variables. By using pointers instead, only pointers are swapped instead of arrays copied. The arrays were intensively used further down in the code, and by getting a reference to the array, no further code had to be altered. The result can be seen in Figure 2.7.

Initially, the temperature calculation seemed perfect for GPU implementation, since calculations could be done on multiple pipe-segments in parallel. However, as Figure 2.7 shows, the optimized version uses only about 1.5% of the CPU time for this particular case. *Pipe::CalcDP*, from the steady state flow calculation, is dominating the total runtime. Thus, even though the target function is able to utilize the GPU's parallelism, its minimal impact on the runtime does not justify maintaining a GPU code for it.

2.3.3 Full Transient Calculation

This calculation consists of both transient flow calculations and transient temperature calculations. This profile ends up being quite different from the combined steady state flow calculation and transient temperature calculation, from the previous section. The optimization of the temperature calculation from Section 2.3.2 applies here as well. The red circle in Figure 2.8 was removed in the previous section, and the total CPU-time used by the temperature calculation drops from 71.42% to 24.20%, which can be seen in Figure 2.9. The speedup

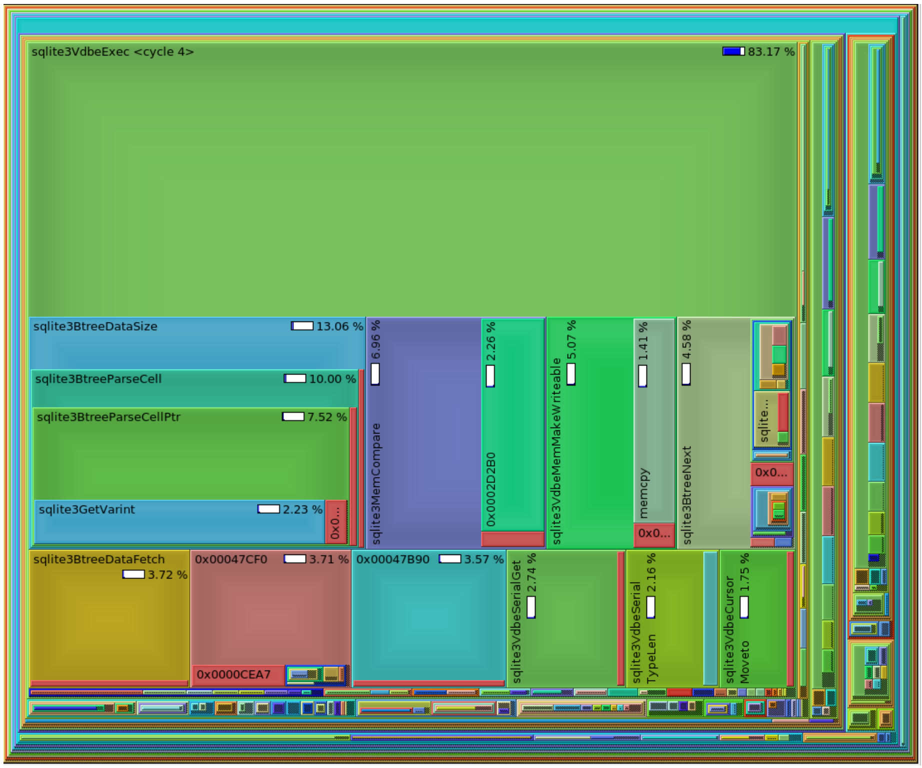


Figure 2.4: Original transient calculation, with original database storage. Notice 83.17 CPU time used by the SQL storage routine, totally dominating the overall performance of the application.

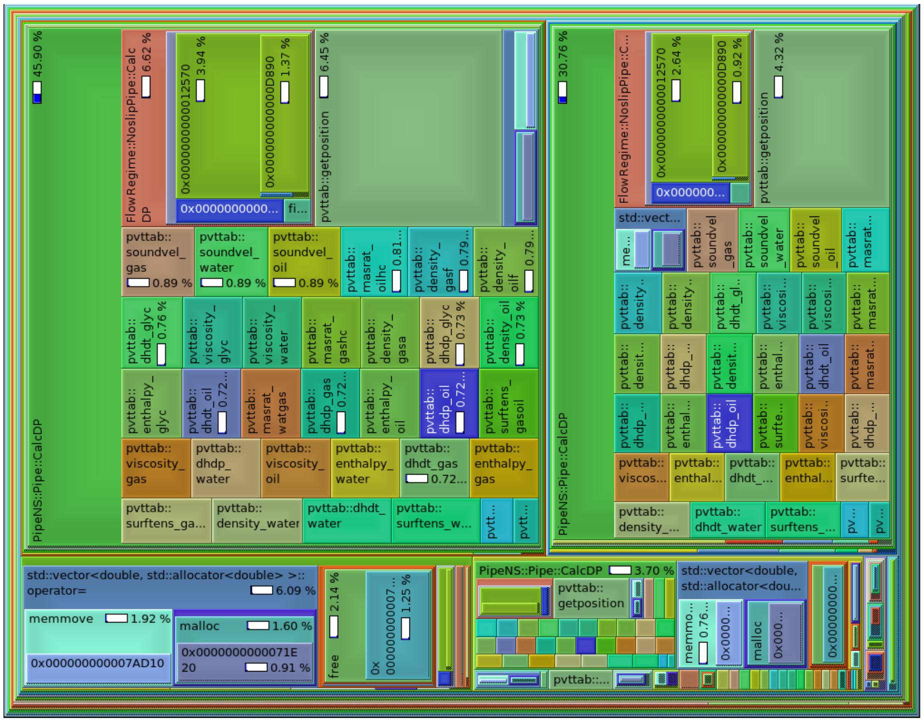


Figure 2.5: Transient temperature- with steady state flow calculation, with binary file data storage.
Notice the previous SQL storage routine is replaced by the binary write routine which now is so fast it no longer shows in the profile. Now, the steady state calculation is dominant instead.

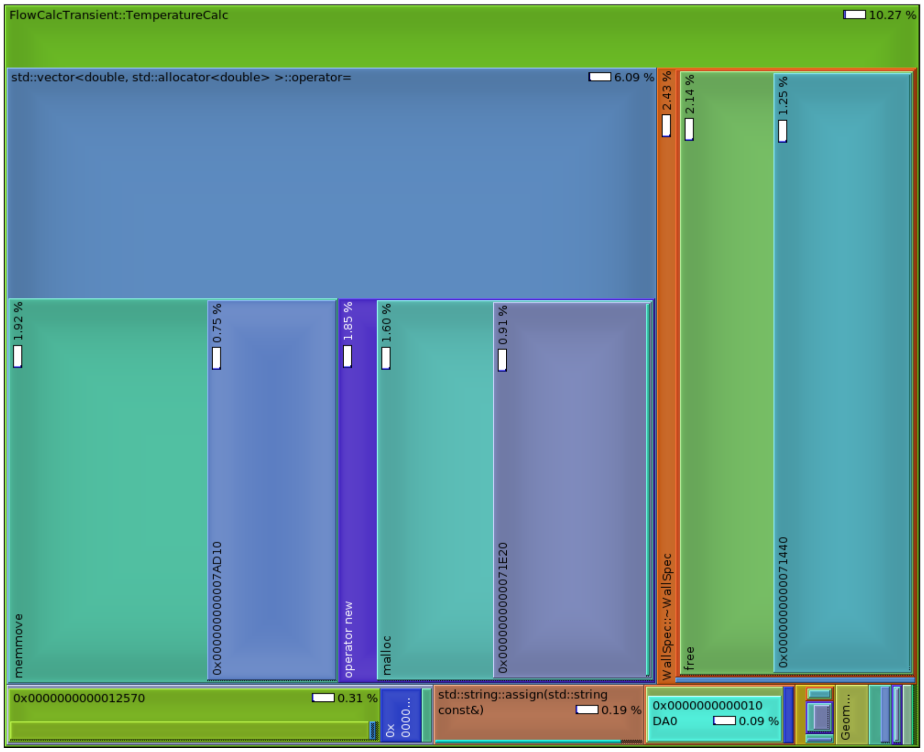


Figure 2.6: Cutout from 2.5. The temperature calculation takes 10.27% CPU time now.

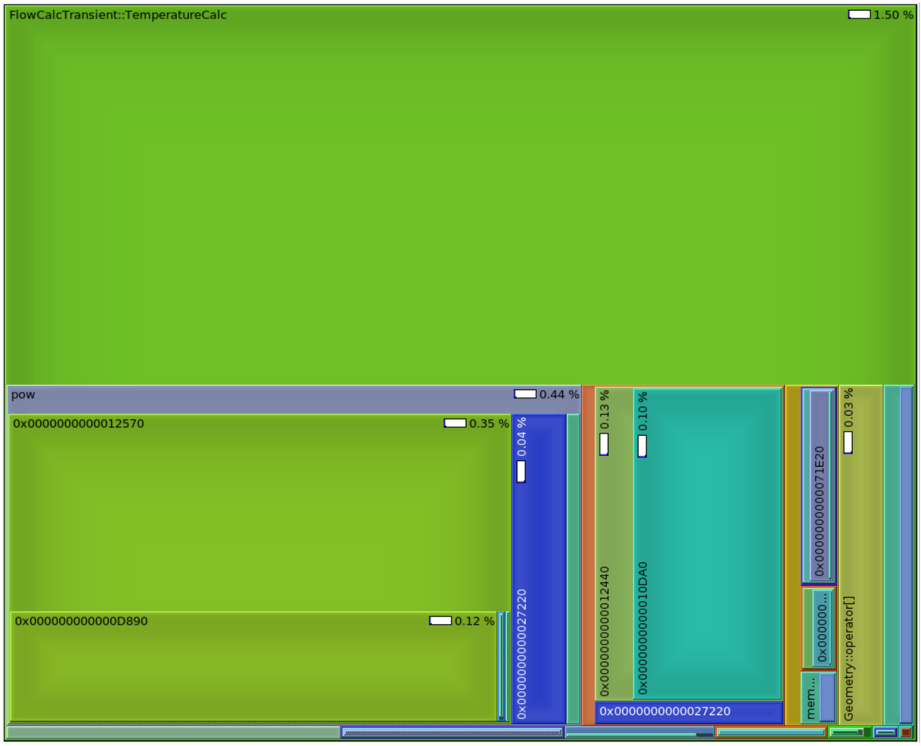


Figure 2.7: Optimized version of profile from Figure 2.6. The temperature calculation now takes 1.50% of the total CPU time.

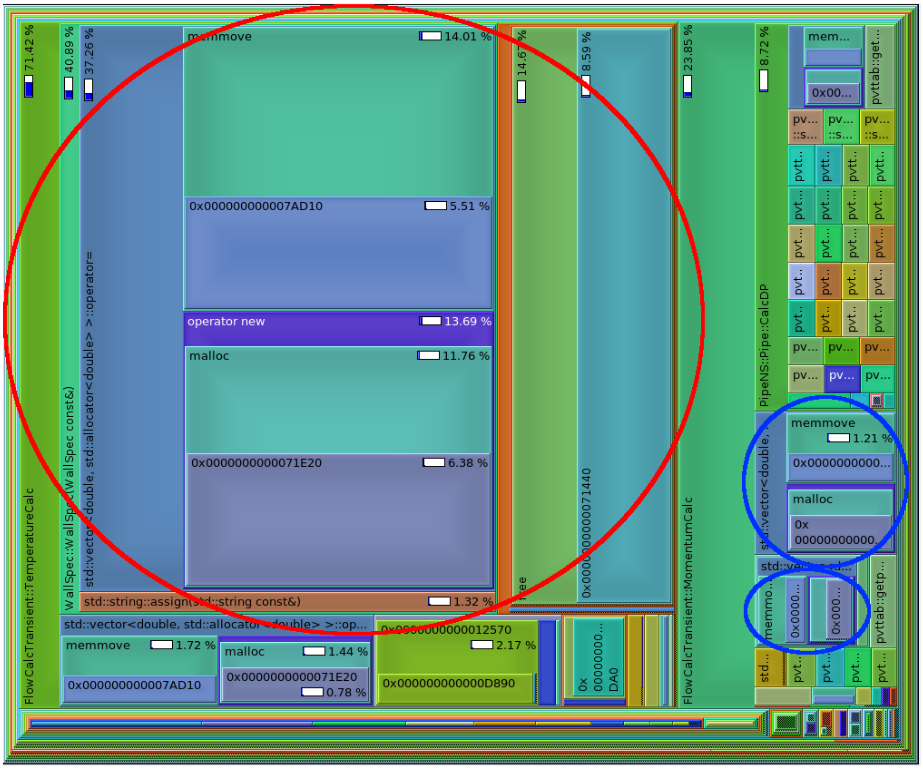


Figure 2.8: Un-Optimized full transient calculation, with binary file storage. Encircled functions are targets.

can be seen in Figure 2.11, optimization step ii).

The blue circles, seen in both Figures, represent the same function call and became new targets after the temperature optimization. Removing these should, in theory, remove about 11% of the momentum calculation's CPU-time, increasing the temperature calculation's CPU-time accordingly. Thus, speedup may be gained by offloading the temperature calculation to the GPU in this case, especially if both the momentum and temperature calculations can be performed simultaneously. Finally, however, the temperature calculation only takes about 30% of the CPU time after this last optimization, which is probably too low for a GPU implementation. The speedup of the last optimization can be seen in Figure 2.11, optimization step iii). This last optimization included altering large portions of the momentum calculation to use pointers and references to arrays and objects instead of doing unnecessary data copies.

There is another important matter at hand, which is the highly unoptimized database storage routine. This has to be reimplemented somehow. An idea is to use a dedicated database thread, which gets its workload by other threads. Thus, when the database thread is busy saving data, other threads/cores or the

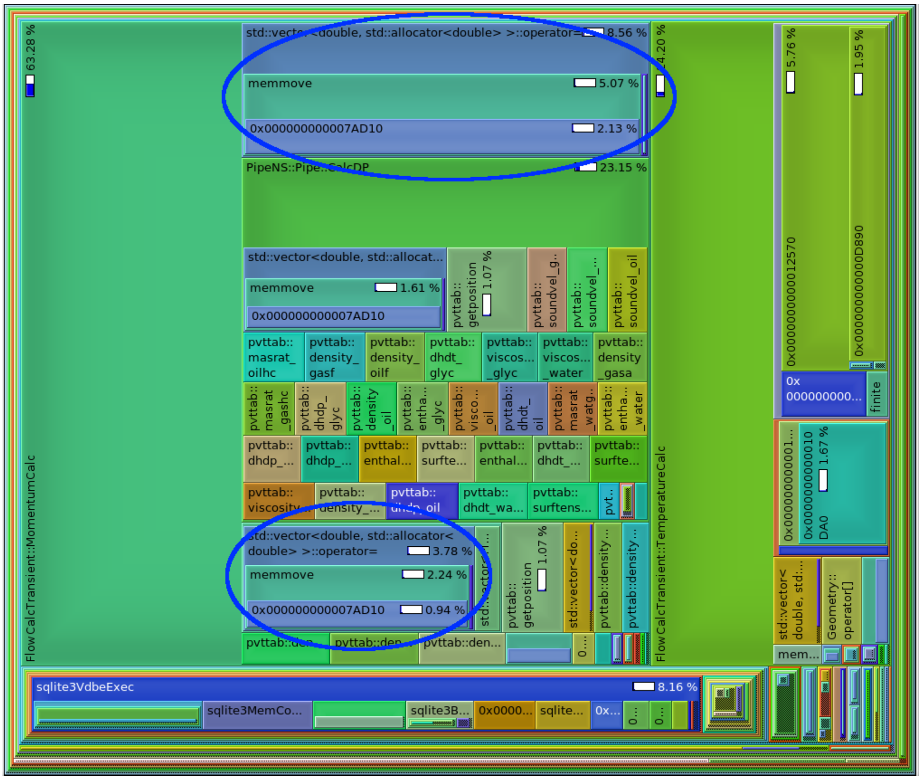


Figure 2.9: Effect of temperature calculation optimization. The large circle in Figure 2.8 is gone.

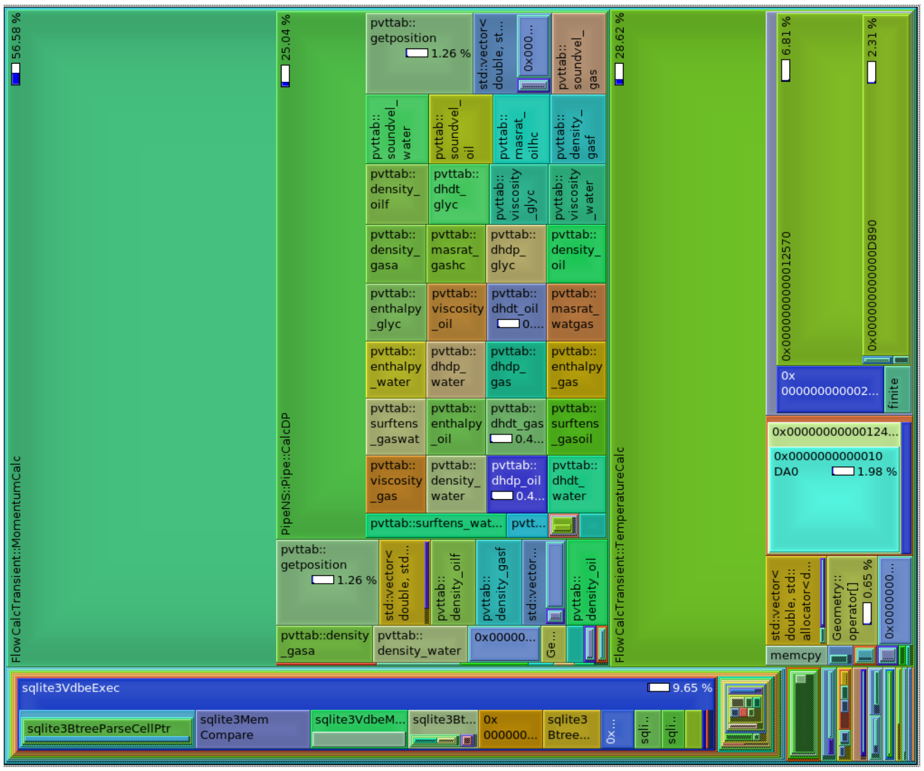
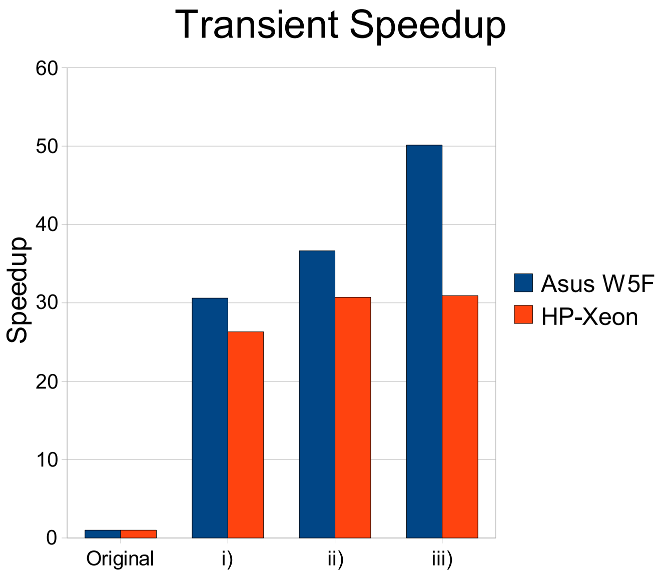


Figure 2.10: Effect of momentum calculation optimization. All circles in Figures 2.8 and 2.9 are now gone, since these routines are now optimized so much that they no longer have significant performance impact.



i)	Change from unoptimized SQL to binary file storage
ii)	Removing array and object copies by using references and pointers in the temperature calculation
iii)	Removing array and object copies by using references and pointers in the momentum calculation

Figure 2.11: Speedup achieved, on two different architectures, by optimizing the transient computations

GPU can do computations, feeding the database thread with new data.

Chapter 3

Parallel Programming and Architectures

In this chapter, an introduction to parallel computing is given, starting with two important laws about maximum theoretical speedup. The next section gives a thorough introduction to previous and current GPU technologies. Finally, different technologies for programming multi-core technology are described.

3.1 Parallel Computing Theory

3.1.1 Amdahl's Law

Amdahl's law [1] states the theoretical limit of a parallel application's scalability, based on the amount of sequential operations in an application. In the following formula, P is the parallel portion of the process and N is the number of processors used.

$$\frac{1}{(1-P)+\frac{P}{N}}$$

As an example, if P=90%, which means that 90% of the process is parallelizable, using an infinite amount of processors and neglecting parallel overhead, gives a maximum speedup of 10. Thus, minimizing the serial portion by adding more parallelism is actually more important than adding additional processor cores.

3.1.2 Gustafson's Law

Gustafson's law [1] differs from Amdahl's law in that linear speedup may still be gained, even though Amdahl's law predicts otherwise. Gustafson's law takes into mind the possibility of increasing the problem size when more processors are available, resulting in constant runtime of the application. Amdahl's law assumes that the problem size stays the same. Amdahl's law also assumes that the serial algorithm is the fastest one available, however, this may not always be the case. Multi-core architectures may also give extra speedup, because of the extra cache for each core holding more of the problem data.

In the following formula, also known as scaled speedup, N is the number of processor cores and s is the ratio of the time spent in the serial part versus the total execution time:

$$N + (1 - N) * s$$

3.1.3 Adding Parallelism to a Serial Application

Adding parallelism in an application can be quite hard and time-consuming. Thus, to get the most out of your work, it's important to find the most time-consuming parts of code and parallelize those first. The next step is dividing the time-consuming code into individual tasks, if possible. This can either be functional decomposition, splitting multiple function calls in to independent groups, or data decomposition, splitting a large amounts of data into independent groups, and assigning a thread to each group [6].

3.1.4 Challenges When Parallelizing

Several issues emerge when going from serial to multithreaded code, that were a non-issue in the former. One serious issue is data-race conditions. This occurs when multiple threads access or update the same memory location at the same time, which may result in erroneous results [1].

Creating and killing threads introduces some overhead to the program execution as well. It is important that this overhead does not exceed the work given to each thread. If there is not enough work to be split among the threads, the program execution may actually become slower than the serial version.

The programmer must also take care when handling critical sections in a code, that has to be executed in a specific order. Barriers and locking variables can be used to synchronize threads before and after critical sections, to make sure the order of execution is correct.

When using multiple processors, it is important that each processor is given the same amount of work. This is known as load balancing. If some threads have twice as much work as others, the other threads will finish, leaving processors idle while waiting for the threads with larger workload to finish.

When working with multi-core processors, a new cache issue may occur called false sharing. Multi-core processors can have multiple caches that may become out of sync. False sharing may happen when two cores are working on neighboring memory locations. If the neighboring data values are stored in the same cache line, the memory system may mark the cache line invalid for one core, which may not be the case for another. This will result in cache line flushing, invalidating the cache system. This should be avoided by introducing strides between data for different cores, which will result in data being buffered in different cache sectors on the CPU [1, 6].

3.2 Introduction to GPUs

In pre-GPU times, graphics hardware consisted of multiple cards with multiple chips working together for outputting graphics on the screen. One could, for example, combine a 2D accelerator with a 3D accelerator expansion card. 3DFX's Voodoo2 cards had three chips on each card, and allowed for two 3D accelerators combined. In the era of the AGP (Accelerated Graphics Port), multiple accelerators was no longer possible, due to the limitation of only having one AGP port. Today, however, multiple PCI-express ports allow for multiple GPUs again, using NVIDIA's SLI and ATI/AMD's Crossfire technology.

In the mid 1990's, the graphics hardware evolved into considerably cheaper one-chip designs, combining 2D and 3D acceleration in the same chip. Eventually, more and more of the graphics pipeline steps were moved from the CPU to the GPU, as well as opening for more configuration and programmability. First generation GPUs, relieved the CPU of having to update individual pixels on screen, however, the CPU still had to do all the vertex transformations. Second generation GPUs offloaded the CPU by also doing the 3D vertex transformation and lighting operations, and allowing some configurability. Third generation GPUs offered vertex programmability and more pixel configuration than before, although still not truly programmable. Not until the fourth generation, released in 2002, did GPUs provide fully programmable vertex and pixel shaders. The first fully programmable GPUs included NVIDIA's GeForce FX and ATI's Radeon 9700 [5].

The GPU's main purpose has always been massive graphics calculations for video games on PCs and consoles. However, when the fourth generation GPUs opened for full programmability, higher level programming APIs were developed for easier access to the tremendous power hidden inside them. Until then, the small amount of programmability and configurability of the GPUs, had to be done through low-level assembly language. With the new high-level languages, programming the GPU became more attractive to less hardcore programmers. It also caught the interest of the HPC community, wanting to utilize the GPU's massive parallel performance for scientific computing.

3.2.1 GPU Compared to CPU

The GPU is a special-purpose processor, specially designed for processing massive amounts of graphical data, using massive parallelism. Unlike CPUs, it devotes its hardware to computation instead of communication and administration [19]. The most recent high-end GPUs, have a total of 128 special purpose stream processors, compared to four cores of high-end CPUs. Because of its special design, the GPU is not suitable for general-purpose tasks, such as running the operating system or word processors [5].

The CPU on the other hand, is a general-purpose processor able to execute all kinds of different applications written in general purpose languages, such as C++ and Java. This generality comes at a cost. Many of the CPU's transistors are used for administrative tasks, such as memory prediction and task switching, leaving fewer transistors for actual computations.

CPUs have SIMD MMX and SSE extensions, but these are often unused directly by the programmers. Newer GPUs have gone from MIMD to SIMD design, where multiple threads execute the same instruction on different data elements. Is necessary to exploit the SIMD design of GPUs as much as possible, if any speedup is to gained.

This difference in design requires different techniques to exploit the GPU to the max, which is discussed in the following sections.

3.2.2 Previous GPU Architectures

Previous GPU architectures split their computation units into two groups, namely vertex and pixel shaders [8]. For instance on a GeForce 7 series, there were 8 vertex shaders and 24 pixel shaders [27]. If running heavy geometrical calculations, the vertex shaders would be saturated while the pixel shaders would do nothing, thus wasting 24 calculation units. This would result in wasted computation cycles by not utilizing the total power of the GPU.

Earlier architectures from both ATI and NVIDIA used a MIMD processor design. This allowed computations on four components per pixel at a time, for example RGBA color values or XYZW coordinates. For instance, MUL and ADD instructions could be executed simultaneously on these four values [26].

3.2.3 Unified Architecture

The latest trend is the unified architecture. This design features one single type of shader unit, capable of both vertex- and pixel shader operations. This yields better utilization of the computing power, since all the shader units can be used for either vertex or fragment operations, according to current needs. GPUs from both ATI and NVIDIA are now using the unified architecture.

NVIDIA's G80 GeForce 8 and Tesla Cards

The GeForce 8, NVIDIA's current flagship series, at the time this was written, is a total redesign from the former 6 and 7 series. While the older series had separate vertex and pixel shaders, the 8 series has a unified architecture, where the pixel shader engines has been extended to support vertex shader capabilities. The instruction set has been changed from vector MIMD to fully scalar SIMD. In certain cases, this reorganization may give a 25% increase in performance, using the same amount of resources [26]. The Tesla is a product targeted at the GPGPU community, sporting 1.5 GB of RAM compared to the mainstream card's 768 MB.

The high-end 8 series cards, have a total of 128 shader units running at 1.35 GHz [15]. The shaders units are grouped into 16 multiprocessors with 8 shader units each. Each multiprocessor has access to 16K shared memory, which allows threads within a thread block to share data between themselves. This is good news for GPGPU programmers, since data can be shared between threads with very little latency.

GPUs have highly efficient thread context switching, which is virtually free in terms of overhead. Thus, unlike CPUs, GPUs prosper when thousands of threads are running at the same time. This is actually recommended, since this can mask memory latency by having some threads do computations while others wait for memory transfers from global memory [16].

3.2.4 Programming the GPU

Programming GPUs, requires detailed knowledge of the hardware architecture to fully exploit their special design [8]. Data parallelism is extremely important, since the GPU is optimized for independent computations. However, the new memory hierarchy with shared memory, allows fast sharing of data between threads in same multiprocessor.

It is important to have algorithms with high arithmetic intensity. This means having a high ratio between the number of arithmetic operations and the number of words transferred. Because of the high memory transfer latency through the PCI express port, it is important to transfer as little as possible between the host and the GPU during computation [16].

Memory transfer latency between global GPU memory and shared memory, should be masked by employing more threads than there are hardware calculation units. This will keep the GPU's calculation units continuously fed with new data. However, the amount of threads to create also depends on the problem to be solved. If there is not enough work, the extra threads created will not do anything useful [16].

Solving linear equations often employ a high arithmetic intensity, doing computations on large matrices and vectors (streams of data). The shared memory allows for fast sharing of data between threads, such as when using five-point stencils to solve differential equations.

Current GPUs do not support double precision values, and some even have deviations from the IEEE 754 standard when it comes to single precision floating-point values[7]. In the future, however, double precision values are said to be supported.

Branching

Branching was first introduced on the GeForce 6 series [9, 26]. However, care should be taken to avoid divergent branches. GPUs are designed to work on groups of pixels, executing identical instructions on all pixels simultaneously. Thus, the GPU cannot execute instructions in multiple branches simultaneously. If different threads in a group is to execute different branches, the execution of the different branches will be serialized. This can give an enormous performance hit.

There are different branching mechanisms available for different scenarios. For branches with a small number of instructions, both branches can be evaluated and only the correct branch write its results. This is called predication. For larger, more complex branches, other methods should be used, such as static branch resolution, pre-computation and Z-Cull techniques [9].

However, the G80 architecture only has a branching granularity of 32 objects per clock, also known as a warp, where the former architectures, using Shader Model 3.0, had a granularity of at least 800 objects per clock [3]. Although still a factor to be considered, the G80 minimizes the branching penalties compared to earlier designs.

GPGPU Before

Previous GPGPU applications had to be adapted specifically to the GPU pipeline, by splitting the application into fragment- and pixel shader code. The fragment shader could read data unlimited times in a kernel, but only write once at the end. The computational domain for GPUs were in the form of texture coordinates, which are similar to indices in an array on a CPU. Each fragment was given a set of texture-coordinates by the rasterizer, which were linearly interpolated between the input vertices. The fragment processor manipulated the data in its texture-coordinate, passed the result to the vertex processor, which in turn controlled the output range of the computation. Each computation was invoked by drawing geometry [4]. The texture unit was thought of as read-only interface, while Render-to-Texture was thought of as write-only interface. For better utilization of the hardware, manually distributing specific workloads amongst the rasterizer, fragment- and vertex processor was necessary.

Initially, a fragment program had no scatter instructions (writing to memory from computed address, a[i] = x), because a fragment shader had no texture write operation. One could convert a scatter to gather operation by multiple passes over the data. However, one could use the vertex shader, instead of fragment shader, to scatter values using point rendering. While the fragment processors had direct access to texture memory, the vertex processors had to go through the rasterizer and fragment shader to access memory. With the GeForce6800, VTF instructions gave the vertex processor direct access to texture memory as well [8].

GPGPU Now

With introduction of the unified architecture, it is no longer necessary to split a GPGPU application into vertex and fragment programs. Now, the shader units have become more general purpose than before, supporting both vertex and fragment operations.

With the new memory hierarchy, data can be quickly shared between threads within the same execution unit. All stream processors have access to the global memory [16]. Synchronization between threads can be performed local on a multiprocessor, to make sure all threads have their correct data before starting the computations. However, synchronization is not supported between multiprocessors.

CUDA [16] is a new high-level GPGPU API introduced by NVIDIA. CUDA is very similar to the C programming language, only with some extra GPU specific functions and variables. Unlike Cg, CUDA is only supported by NVIDIA GPUs.

Using this API, the programmer can split the threads into a grid of blocks in multiple dimensions. Each block of threads is guaranteed to execute on the same multiprocessor, thus easily allowing for quickly sharing data between the threads within the same block. To share data between different blocks, it is necessary to transfer data to the global memory first. This is a much slower operation and should be avoided, if possible.

Code that runs on the GPU is called a kernel. These kernels are started from the host, after the necessary data has been transferred from the host to the GPU memory. The kernel calls are asynchronous, which means that the CPU can continue its work, while the GPU works on the kernel code. However, the memory transfers to and from the GPU are blocking operations.

ATI's alternative is called CTM (Close-To-Metal), which is a low level specification of the GPU instructions. They leave it to the open source community to develop APIs for their GPUs. Brook+ is an alternative to CUDA for programming ATI GPUs [29].

Data Types

Previous GPUs only supported 16 to 32-bit floating point values. The NVIDIA FX and 6 Series supports both formats, according to the IEEE-754 standard. However, the competing ATI products, Radeon 9800 and X800, only supports a non-compliant 24-bit format. Integers were not supported at all, and had to be represented by shorter range floating point values [8]. However, the latest GPUs support integer values, and the latest FireStream GPUs from AMD, support double precision floating point numbers as well. NVIDIA's Tesla cards are to support double precision in the future.

GPU Languages

Multiple languages are available for programming GPUs [13]:

- Cg: Offers a C-like syntax for creating GPGPU applications for split shader architectures. A Cg compiler is used to compile fragment and vertex processor instructions.

- Microsoft HLSL: Basically Microsoft's implementation of Cg. The difference is that Microsoft has added some extra functions for calling DirectX via the shader programs, thus not adding anything to GPGPU applications, except making them platform dependent to Windows and DirectX.

- OpenGL SL: This is also similar to Cg, and was included in OpenGL 2.0.
- CUDA [16]: This is the new vendor specific API for programming NVIDIA cards. It is very simple to use, with its C-like syntax and straightforward thread-domain decomposition support. However, for full utilization of a GPU, important factors, such as memory alignment, must be handled by the programmer. This requires skilled programmers and intricate knowledge of the specific hardware.
- Brook+: An extension to the Brook stream programming language, developed at Stanford University, for use with ATI GPUs.
- RapidMind: The RapidMind platform [20] is interesting, because of its one code fits all paradigm. The programmer simply uses RapidMind's data types and program structure in combination with regular C++ code, and libraries takes care of the underlying architectures (e.g. Cell, GPU, multi-core CPU). There is no need to alter the source code when changing platforms. It is especially interesting in combination with the Cell processor, which is quite hard to program using SPU and PPU intrinsics. We tried to contact RapidMind for an academic license to their platform, but they did not offer this nor single commercial licenses. Thus, we were not able to use it this project.

3.3 Programming Multi-Core CPUs

Multi-core CPUs were introduced to the market when heat dissipation and power consumption became a limiting factor when pushing the clock speeds further. Multiple slower cores were a cheaper solution, which generated less heat and needed less power than increasing the speed of a single core [24].

The production size technology is currently at 45 nm, which opens for an incredible amount of transistors on a small area. Hence, the number of cores will grow in the future, as the production size gets smaller and smaller.

For software developers, this means that investments have to be made in multithreading their applications, to keep up with the current computer architecture development. Luckily, there are multiple tools are available to make the transition from serial to parallel code easier. In the following sections, an introduction to different tools will be introduced, and pros and cons of each are discussed.

3.3.1 C++ Parallel Programming APIs

Compiler auto parallelization

The Intel compiler has auto-parallelization capabilities, which is an easy way to add parallelism to an application. Adding the *-parallel* flag to the compilation, will make the compiler look for parallelizable loops and try to parallelize them automatically, using OpenMP. However, auto-parallelization is not magic, and

Listing 3.1: Creating and joining threads using Pthreads

```
int main ()
{
 pthread_t thread;
 int input = 10;
 ...
 pthread_create(&thread, NULL,
                threadFunctionName, (void*)&input);
 ...
 pthread_join(thread, NULL);
}

void* threadFunctionName(void* in)
{
 ...do work...
}
```

if the compiler for some reason thinks a loop should not be parallelized, even if it is parallelizable, it will not try to do so.

Adding the flag *-par-report3* to the compiler, will make it print report information about the loops it analyzes. The programmer can use this information to find which loops were not parallelized and why, and try parallelize them manually.

POSIX Threads

POSIX threads is a portable raw threads standard for consistent programming of multi-threaded applications, across different operating systems. It gives the programmer more control over the threads than OpenMP, but in the same time leaves things to the programmer which OpenMP does automatically, such as thread creation and load balancing. Thread synchronization is done using mutexes, which can be locked and unlocked by the programmer to protect critical sections of code from other threads. Condition variables are also available for the programmer to suspend threads in certain cases. The suspended thread can be signaled by other threads to wake up and continue execution when a specific condition applies [14].

There are different implementations of the POSIX standard, such as Pthreads in Linux and Windows threads. The Boost library includes a thread wrapper, that provides a highly portable raw threads interface [21].

Listing 3.1 shows how to create and join threads using the Pthreads API.

Listing 3.2: OpenMP parallel for

```
#pragma omp parallel for
for(int i = 0; i < n; ++i)
  a[i] += i;

#pragma omp parallel for
for(int i = 1; i < n; ++i)
  a[i] = a[i] + a[i-1];
```

OpenMP

OpenMP is a portable API which offers an easy approach to parallelization. It provides different pragmas and function calls, leaving the threading details to the compiler. Thus, the programmer does not have to handle load balancing, synchronization and creating and destroying threads, and can focus on re-designing and locating parallelizable algorithms instead [1][18].

Adding OpenMP parallelism to existing serial code is as easy as enclosing existing code in simple pragma statements. However, care must still be taken when there are interdependencies between data and when working with large, advanced algorithms. Since OpenMP hides the parallel logic behind the scenes, the programmer does not have the same amount of control as with Pthreads. This restriction may, in some cases, force the programmer to use Pthreads instead. However, combining Pthreads and OpenMP is fully supported, and can be used as a tool to add parallelism to advanced applications.

Another excellent property of OpenMP is that the serial code is kept intact. If a compiler does not support OpenMP, the pragma statements are ignored and the application is compiled serially. Although most compilers today support OpenMP, this adds an higher level of code portability.

The next version of OpenMP, v3.0, will have support for a new *taskq* and *task pragma* [18]. A similar construct has been available in the Intel compiler for some time. This new pragma adds a work-queuing execution model to an application, allowing to parallelize recursive algorithms, dynamic-tree search and while loops [23]. The work-queuing model allows for fast task switching between threads, using a thread pool, instead of creating and destroying threads per task. The master thread enqueues tasks in the work queue, while the other threads dequeue tasks in the other end until it is empty. Nested task-queuing is also possible, where a task's thread becomes the master thread for any new taskq block it encounters inside its current task.

Listing 3.2 shows two for-loops parallelized by OpenMP. The first loop will be parallelized correctly, however, the second has flow dependencies between elements in the loop. A limitation in OpenMP is that it does not analyze code correctness [10], thus the second loop will generate wrong results, which will not be detected by the compiler. This means that the programmer must carefully design code, with good data decomposition, to avoid problems like this.

Intel Threading Building Blocks (TBB)

Intel TBB is a portable library adding scalable parallelism to standard C++. It focuses on splitting the work into tasks, leaving the threading details, such as synchronization, to the library [21]. It includes concurrent containers, such as queues, vectors and hash maps, and provides interfaces for scalable memory allocation.

3.3.2 Parallel Programming Tools

Debugging parallel code can be quite hard without tools to help you. Detecting race conditions and deadlocks in parallel code can be close to impossible without debugging tools. Different tools are available for debugging parallel code, such as Intel's VTune with Intel Thread Checker, Tau, Valgrind and GDB.

Chapter 4

Multi-Core & GPU Models

In this chapter, different models used in this project are described. It starts with a description of related work and models, followed by a description of the target algorithm. The strategy for introducing GPU computing in the application is then described, followed by a theoretical multi-core performance and implementation model. Finally, three performance models are introduced. First, a model using serial timings to predict parallel speedup is presented, followed by two theoretical performance models.

4.1 Related Models

In the current era of multi-core architectures, a lot of development and research has been done not only on homogeneous architectures, but also on heterogeneous execution models. Both Intel with Nehalem and AMD with Fusion are focusing more and more on heterogeneous chip designs, scheduled to arrive in 2009. The already available Cell processor is another example of a heterogeneous design with huge performance capabilities. GPGPU computing, where CPU cycles are offloaded on GPUs, is another example. However, parallel programming introduces many caveats such as race conditions and data locality problems, and requires a new way of thinking. For homogeneous multi-core platforms, OpenMP has been a standard for handling parallel execution for a long time. However, for heterogeneous platforms, new tools and extensions are needed to utilize the new architectures.

EXOCHI [28] is an attempt to tightly couple specialized accelerator cores and general purpose CPU cores, by representing heterogeneous accelerators as ISA-based MIMD resources in a shared virtual memory heterogeneous multi-threaded execution model. It extends the OpenMP pragma for multithreading and allows for accelerator-specific in-line assembly in a C/C++ environment, creating an IA32 look-and-feel in the programming environment. Prototyping was performed on an Intel Core 2 Duo in combination with an Intel GMA X3000, and speedups between 1.41x to 10.97x was gained using this execution model.

Korch and Rauber [12] have implemented different types of task pools, in both C and Java, and performed a thorough comparison of the different implementations. They saw that the choice of implementation for a task pool, can have great impact on its performance. They concluded that dynamic task stealing, using a private and a public queue for each thread, provides the best scalability. Using this design, the bottleneck of having multiple threads accessing a single, global queue is eliminated. They also implemented a memory manager, which allowed the task pool to reuse memory for new tasks, instead of allocating and de-allocating tasks in memory for each calculation.

Several thread pool implementations are described in various literature [1, 14], and different libraries, under various licenses, include working thread pool implementations. One thread pool is built on top of the C++ Boost library, and is a work in progress [25]. Intel also provide a working solution in their Threading Building Blocks library [21].

Aliaga et al. [2] used a task pool strategy to create a parallel preconditioner based on ILUPACK. By first using the Multilevel Nested Dissection algorithm on the initial sparse coefficient matrix, they were able to extract balanced elimination trees with a high level of concurrency. By manipulating this elimination tree, they were able to organize all tasks in a tree like structure, while preserving the dependencies between them. Their tree structure is similar to the tree structure in our application, which means that our application could be parallelized in a similar way.

Kessler and Löwe [11] have proposed a framework for performance aware functions, which automatically selects the optimal implementation variant, based on component meta-code and information about the current hardware. The choice is performed dynamically during runtime, based on the current problem size and the number of available processors to each respective component. A simplified version of this framework, which could choose between serial or parallel execution dynamically, would be interesting for our application.

4.2 The Algorithm

The algorithm's main purpose is to optimize flow by calculating phase flow from the wells, through to network and up to the terminal. By manipulating different parameters in the network, different targets, such as min/max pressure, will be maintained if possible.

Our application can be used, in combination with an oilfield simulator, to stimulate the development of an oilfield over time. Our application's job is to assure optimal flow through the network of oil-wells, by collecting sensor information and controlling different actuators that in turn affect the flow in the pipes. The simulation over time is necessarily a serial operation, since the results from the previous time steps is used as input into the next. Thus, it is not possible to parallelize this operation. However, our application allows the user to simulate multiple network topologies in each time step, to find the optimal topology over time. These different topologies are fully independent of each other and can therefore be computed in parallel. This is the highest level of parallelism possible in this application.

The second highest level of parallelism, is parallel traversal of the oilfield networks. However, these networks have interdependencies between each level in the tree structure they compose. All child nodes must be calculated before their parent nodes, since the results from the children are used as input to the parent.

The third level of parallelism is similar to the previous, only in that smaller parts of the tree is recalculated when certain parameters are reached. These smaller parts are the applicable child nodes of the current node in the previous parallel level. This third level will yield the lowest speed up of the three, since in certain cases there might only be one or two child nodes to be calculated. This may cause the setup and management of the parallel computation to add significant overhead compared to the amount of calculations to be performed. It may even cause slowdown. Figure 4.1 gives a graphical representation of the algorithm.

4.3 GPU Model

Our initial focus was on seeing if GPUs could be used to speed up the calculation, by using profilers are used to locate hotspots in the application. While looking for hotspots, it is important to keep in mind to GPU's high level of parallelism. If the hotspots located cannot be split up into a significant amount of parallel operations, the GPU might end up being severely underutilized.

So, what are the costs of offloading CPU cycles to the GPU? The main factor is data locality. The latency of transferring data from the host, through the PCI express port, via the GPU's global memory and finally to the GPU's shared memory, severely exceeds transferring data between RAM and the CPU. Thus, it is important to look for parts of the code which transfer as little data as possible between the host and the GPU, compared to the number of computational instructions performed on the data.

The theoretical transfer rate of the PCI express port is 4GB/s i each direction. DDR2-1300 (PC2-10400) RAM modules, have a bandwidth of 10.4 GB/s, and for comparison, the CUDA SDK bandwidth test returns a bandwidth of ~86 GB/s between the GPU (GeForce 8800 GTX) and its global memory. The global memory bandwidth on the GPU is over 20 times faster than the transfer rate over the PCI express port, and eight times faster than the main memory bandwidth on the host. The two last bandwidths are theoretical, and may not be the actual numbers in a real-life benchmark.

When the data has arrived in GPU's global memory, it is extremely important to utilize the shared memory, since they offer one cycle access to a value, compared to the global memory's 2-300 cycle transfer latency [17].

Another important factor is the data-type support offered by different GPUs. Older GPUs only supported proprietary floating-point values. Integer values were not supported, and floating-point values had to be used instead. This

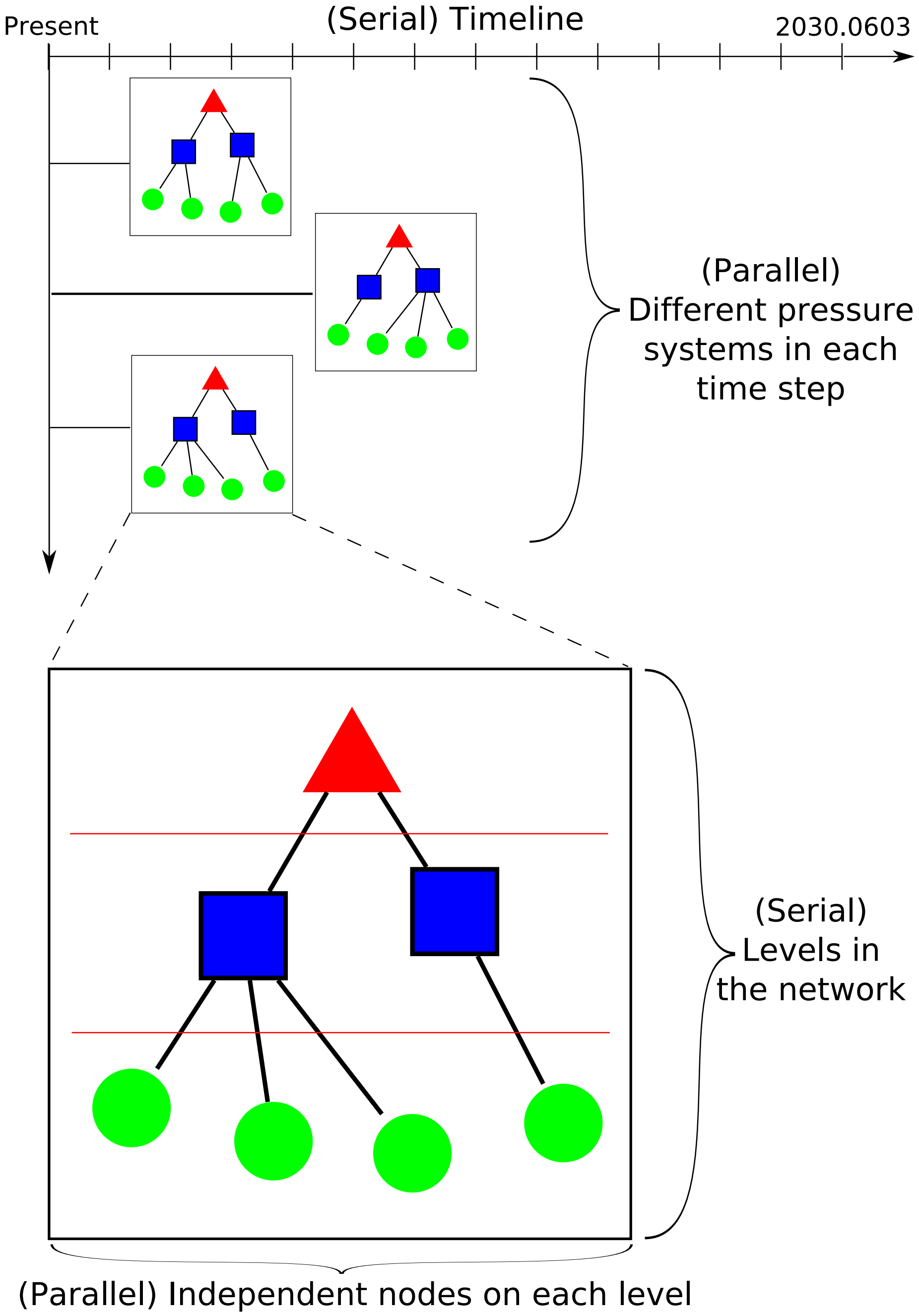


Figure 4.1: Algorithm overview

could induce problems with addressing, because of the floating point precision. Newer GPUs, however, support integer values as well as IEEE 754 standard floating point values. Special GPGPU's, such as the ATI FireStream, support double position values as well. All this must be considered when targeting the hotspots in an application [4].

Recursion is not directly supported by GPUs either, however, this can be made possible by implementing a stack, say, in CUDA. All in all, GPUs are happiest when doing the same instructions on enormous amounts of data at the same time, without transferring anything between two are from the host. Using these criteria, together with the profiler, we will try to find suitable code to put on the GPU to offload the CPU.

4.4 Multi-Core CPU

Using multi-core CPUs allows for higher level parallelism than by using special-purpose GPU technology. Thus, parallelizing the application using multi-core technology will be attempted from top to bottom, instead of bottom to top, as in the GPU attempt.

The overhead introduced with multithreading may in some cases cause calculations to be slower in parallel than in serial. Thus, a method for suspending threads is needed to minimize threading overhead. A model over serial versus parallel traversal of the trees can be used to predict if the target case will benefit from parallel execution.

4.4.1 Parallel Implementation Model

The order of execution of the initial target, the second level of parallelism, in serial mode, is maintained in a one-dimensional vector and traversed using a regular for-loop. Since there is only one thread traversing the tree in serial mode, the only important traversal issue in this mode is building the traversal order vector correctly. However, when using multiple threads, care must be taken to make sure all child tasks are completely finished before a thread can execute the parent. Parallelizing the tree traversal while maintaining the dependencies, requires low-level control over the order of execution, thus using a simple OpenMP parallel-for is not possible.

Another important property in our application is that each calculation is fairly small, but executed millions of times during a simulation. Thus, creating and destroying threads for each calculation will add a lot of unnecessary overhead. To avoid this, a thread pool using suspended threads will be developed, in combination with a task queue. By creating a base task and using polymorphism, the task queue can easily be used with different kinds of tasks.

A task queue can be used for all three levels of parallelism mentioned in Section 4.4. The tasks in the highest level of parallelism, namely the different pressure systems, are totally independent of each other. Thus, there is no need to maintain interdependencies between these. However, since the second level of parallelism is inside the first, and the third inside the second, the task pool must be able to handle nested parallelism. This can be handled by allowing the use of multiple task queues. By pushing nested queues on a FILO stack, threads will pop tasks from the innermost queues first, thus maintaining the correct order of execution. When a task queue is empty, it is popped from the queue stack, revealing the higher-level queues again.

For the second and third levels of parallelism, each pipe will be defined as a task which a single thread can execute. To maintain the dependencies between parent and child tasks, each task must know when its children are done. This will be implemented by using a child counter in each task which is initialized with the number of children for this task. Each task will have a pointer to its parent, and will decrement its parent's child counter when completed. The value of the counter will be returned to the thread which executed the child task, and the thread that gets zero in return, will execute the parent task next. This counter will be accessed by multiple threads, thus it is a critical section which must be protected by a mutex. This task tree must be created prior to execution, to define the interdependencies correctly and to ensure correct execution order.

The simplest form of task queue will be implemented, namely using a single global queue, to see if a task queue is a feasible parallelization solution. For the parallel tree traversal, only the leaf nodes will be put in the queue, while the remaining tasks will be accessed through each task's parent pointer when all child tasks are finished. Thus, only the leaf nodes will get the performance hit of using a single global queue.

4.4.2 Practical Performance Model

This model uses a hands on approach to try to determine which part of an application would best benefit from parallelization, without actually parallelizing it. By doing detailed timings of the serial execution and building replica tasks, which are doing dummy CPU-intensive work for the same amount of time as it did serially, one should be able to determine if it's worthwhile parallelizing the timed code.

A replica network will be created using timings from the serial executions. Each task in the replica network will be set to sum a variable for the same amount of time the task used serially. When a task reaches its execution time, it will simply exit, and the thread running it will get a new task from the queue. Although these tasks will not behave exactly like the tasks in the actual application, this should give a basic idea of the possible speedup to be gained by parallelizing the selected code.

Timers

The first thing to do when doing detailed timings of code, is to find a decent timer. Three important characteristics for timers is its overhead of use, its precision and finally, its correctness. A simple benchmark was performed between

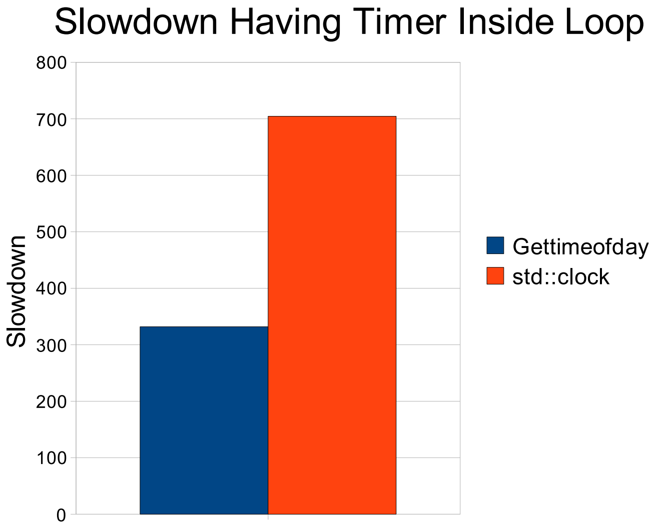


Figure 4.2: Timer slowdown

two timers, namely std::clock and gettimeofday, determining the overhead and granularity of the two. Determining their correctness is more difficult on a microsecond level, and the timer was chosen from the first two criteria. The chosen timer will be used for timing the replica tasks as well, which should make its correctness irrelevant relative to the real time used.

The timers were tested with a simple for loop, summing an integer value for 9999999 iterations. The base case had the timers outside the loop only, and the overhead test had the timers inside the loop as well. Figure 4.2, shows the overhead of having the respective timers inside the loop, and that std:.clock has the most overhead of the two. The benchmark also showed that gettimeofday had the highest precision, as can be seen in Listing 4.1.

4.4.3 Theoretical Performance Model 1

In this section, a performance model of the serial and parallel computation of an arbitrary network configuration, is developed. The algorithms used to solve the nonlinear equations are highly irregular, thus making it hard to model them in detail. Hence, several simplifications will be made. A similar tree structured problem can be found in [2].

The main simplification is the predicted calculation time of each pipe. Only the number of segments in a pipe will be used to predict its calculation time. Also, the application might perform an undefined number of traversals of the tree in each time step, altering the input to available actuators in each traversal, to try to match the target pressures defined by the users. Since it is impossible to know the number of times this will happen before the calculation, this will

Listing 4.1: Output from timer benchmark

```
9999999 iterations in loop for all benchmarks:

—— Timing sum+=1 outside loop using std::clock
STD outside loop timed: 0.020000 seconds
—— Timing sum+=1 outside loop using gettimeofday
TIMEVAL outside loop timed: 0.026519 seconds

—— Timing sum+=1 outside and inside loop using std::clock
STD inside loop timed: 6.860000 seconds
STD outside loop timed: 14.090000 seconds
—— Timing sum+=1 outside and inside loop using gettimeofday
TIMEVAL inside loop timed: 4.287836 seconds
TIMEVAL outside loop timed: 8.803945 seconds
```

not be included in the formulas. However, this will be the same for the serial and parallel model. The model also assumes a fairly balanced work load across each level in the network, since it will use the average time to compute one pipe on each level and it assumes that the overhead of distributing work to threads is negligible. It will use the timing output described in Section 4.4.2 for the calculations.

A model for serial execution follows:

$$T_s = \sum_{i=0}^{N} nSeg_i * T_{seg}$$

Where N is the total number pipes in the tree, $nSeg_j$ is the number of segments in the ith pipe and T_{seg} is the average time to calculate one segment.

The following formula will be used to predict the parallel execution time of an arbitrary network. Because of the interdependencies between the levels in the tree, it is necessary to involve the height of the tree in the calculation. Each higher level in the tree will have fewer pipes than the levels below, thus the higher one gets in the tree, the fewer threads will be able to do work in parallel.

However, another important property of the parallel traversal, is that it is not necessary to compute all nodes in a level before going to the next. Once all child nodes of a specific node are finished, this parent node can be computed in parallel with nodes from lower levels. Thus, once finished with one level, most nodes in the level above might be finished already, even nodes in higher levels as well. The model assumes all nodes in a level is completed before going to the next.

A model for parallel execution follows:

$$T_p = \sum_{i=0}^{h} T_i * R_i$$

where h is the height of the three.

The next formula sums the time taken to calculate the total number of segments in a level in the tree, and averaging them over the total number of pipes on the same level. This gives the average amount of time it takes to calculate one pipe on this level.

$$T_i = \frac{\sum_{j=0}^{n_i} nSeg_j * T_{seg_j}}{n_i}$$

Where n_i is the number of pipes on the current level, $nSeg_j$ is the number of segments in the jth pipe and T_{seg_j} is the average time to calculate one segment on the jth level. The last variable must be timed using a benchmark to get the correct value on each platform.

The last formula calculates the ratio between the number of pipes on one level and the number of processors/threads used in the calculation. Since only one thread can be assigned to a pipe at a time, the number of threads utilized cannot be greater than the number of pipes on the current level. Thus, the ratio cannot be smaller than one.

$$R_i = \frac{n_i}{p} \quad , p \leq n_i$$

Where p is the number of processors, which cannot be higher than the number of pipes, n_i, on the current level.

4.4.4 Theoretical Performance Model 2

This section describes a simple model which finds the minimum time needed to calculate a network in parallel, while maintaining the serial dependencies. It assumes perfect conditions with unlimited threads and no scheduling overhead. This model will also use the timing output described in Section 4.4.2.

This model can be summarized as follows:

1. Obtain timing results from calculating network serially
2. Build execution tree of tasks with this timing info
3. Traverse this tree recursively from root to find most expensive path. This path will be the best parallel performance possible, since all other paths will take less time.

The model calculates the minimum time needed, by finding the most expensive path from the root node, through the network and down to the leaf node level. Since the model assumes unlimited threads, all other paths in the network will be finished before this path reaches the root node. Thus, it will not be possible to calculate the network faster than this model predicts, while still maintaining the serial dependencies.

The model is calculated by recursively traversing the network from the root node down to the leaf nodes. Each node in the tree calculates which child node

Listing 4.2: Model 2 calculation

```
//---------------------------------------------------------
// Calculates the minimum amount of time needed
// to calculate a whole tree in parallel,
// by finding most expensive path from root to leaf
//---------------------------------------------------------
double findMostExpensivePath(SleepTask* task)
{
    double max = 0;
    for(int i = 0; i < task->getChildren().size(); i++)
    {
        double time = findMostExpensivePath((SleepTask*)
            task->getChildren()[i]);
        max = (time > max) ? time : max;
    }
    return max + task->sleep_usec;
}
```

is most expensive, by using the timing data from Section 4.4.2, and returns its own time plus the time of its most expensive child. Thus, the most expensive calculation time from the leaf node level up to the root is summed and returned. See Listing 4.2 for the code.

4.4.5 Model Results

Three different test cases, two for steady-state and one for transient calculations, are used in the benchmarks. The small steady-state case (Case I) consists of four wells and three transport lines while the large case (Case II) varies its amount of wells between 60 and 80 at different times steps. The transient case (Case III) is the same size as Case I. The main focus is on the steady-state cases I and II, since the transient calculations is still in the alpha stage.

Case II also has a lot of events specified, which are triggered at specific times during the calculation. These events might affect the whole network in different ways, such as closing of wells or changes in different targets throughout the network. It also has three root nodes, since there are three platforms and three transport lines to shore. One of these transport lines is an order of magnitude more expensive to calculate and the two others, which might degrade parallel performance.

Results

Detailed timings were performed of the serial network traversal of Case I and Case II. The execution time of each branch in the network was saved to a file during serial execution, using the function "gettimeofday". Listing 4.3, shows

Listing 4.3: Practical model's timing output of the serial execution of the network traversal

```
# Column descriptions
# Node name;line(0)/well(1);Parent name;Num. segments;
   Time in usec.

A-14H;1;TM_OS_2;38;2542
A-13H;1;TM_OS_2;36;2399
TM_OS_2;0;FCP;20001;374457
A-12HT2;1;TM_OS_1;40;261
A-11AH;1;TM_OS_1;37;2154
TM_OS_1;0;FCP;20001;332785
FCP;0;0;1002;7845
----
A-14H;1;TM_OS_2;38;237
A-13H;1;TM_OS_2;36;184
TM_OS_2;0;FCP;20001;248254
A-12HT2;1;TM_OS_1;40;258
A-11AH;1;TM_OS_1;37;180
TM_OS_1;0;FCP;20001;164595
FCP;0;0;1002;5999
----
 (...)
```

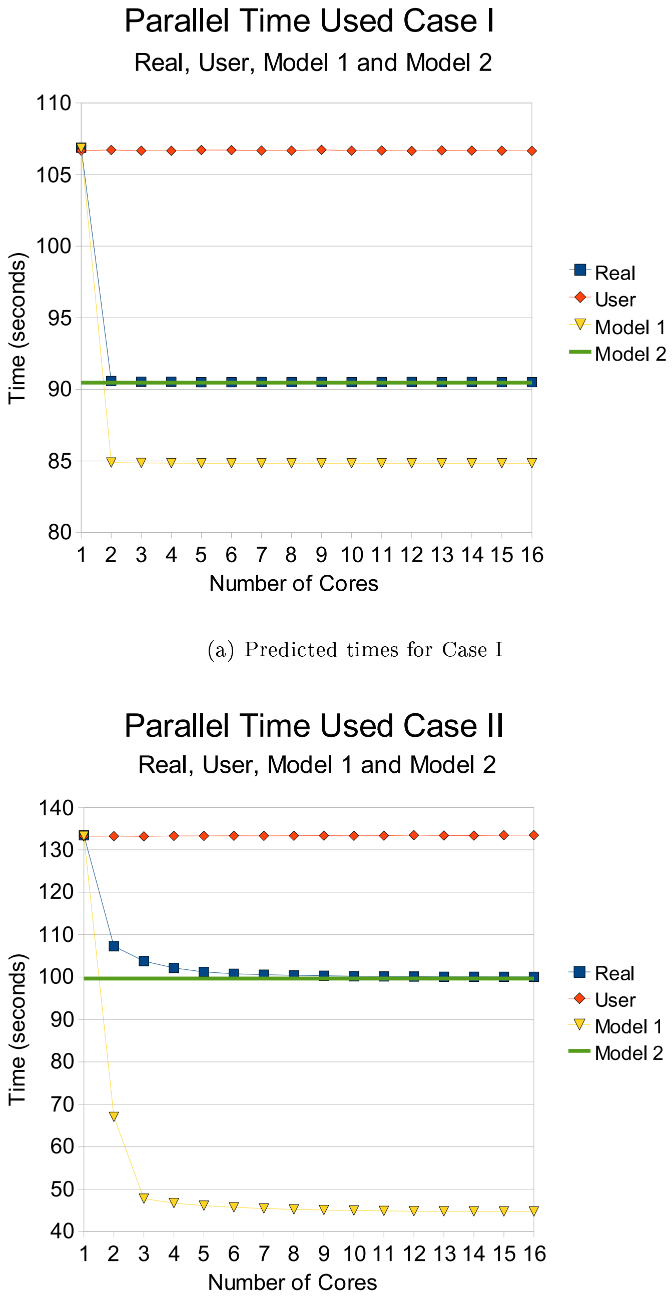
the output from the serial timings inserted into the target code in the application. The fields, from the left, are: pipe name, if it is a well (1) or a transport line (0), its parent's name, its number of segments and finally the calculation time used, in microseconds. This data was then parsed by a test application in the thread pool implementation, which used both models to predict the parallel time used.

Figure 4.3, shows the results produced by the different models. Real and User is the output from the time function, in Linux, for the practical model. Real is the wall clock time and User is the combined time used by the application, which includes the time used by all threads and other overhead. It is observed that the User time is stable as the number of cores increase, which shows that the thread pool implementation scales well, without introducing a lot of extra overhead per new thread.

It is also observed that the Real time used matches Model 2 very well. For Case I, the theoretical best time is reached already at two cores, which is due to the low amount of parallelism for this case. For Case II, the theoretical best is reached at around nine cores due to its higher level of parallelism. However, even though there are more tasks available to do in parallel, it does not scale over nine cores due to the low level of parallelism in the highest levels in the network and their order of magnitude longer computation time.

Model 1 is able to model speedup using arbitrary number of cores, however, it does not match the others very well. This model's assumption that all tasks on each level are fairly well balanced, does not necessarily match to real case scenarios. If one task on a level is much more expensive than the others, this task's computation time will be spread out over all tasks on that level. This will in turn make multiple threads work for a shorter amount of time, instead of having some threads working for a very short time, while one thread keeps working for a long time after the others are done and waiting. This is what happens in both cases, and especially for Case II, where one of the three top-level nodes is an order of magnitude more expensive than the others, thus, an artificially high speed up is predicted. Thus, care must be taken when making assumptions in a model, such that decisions are not made on the wrong basis.

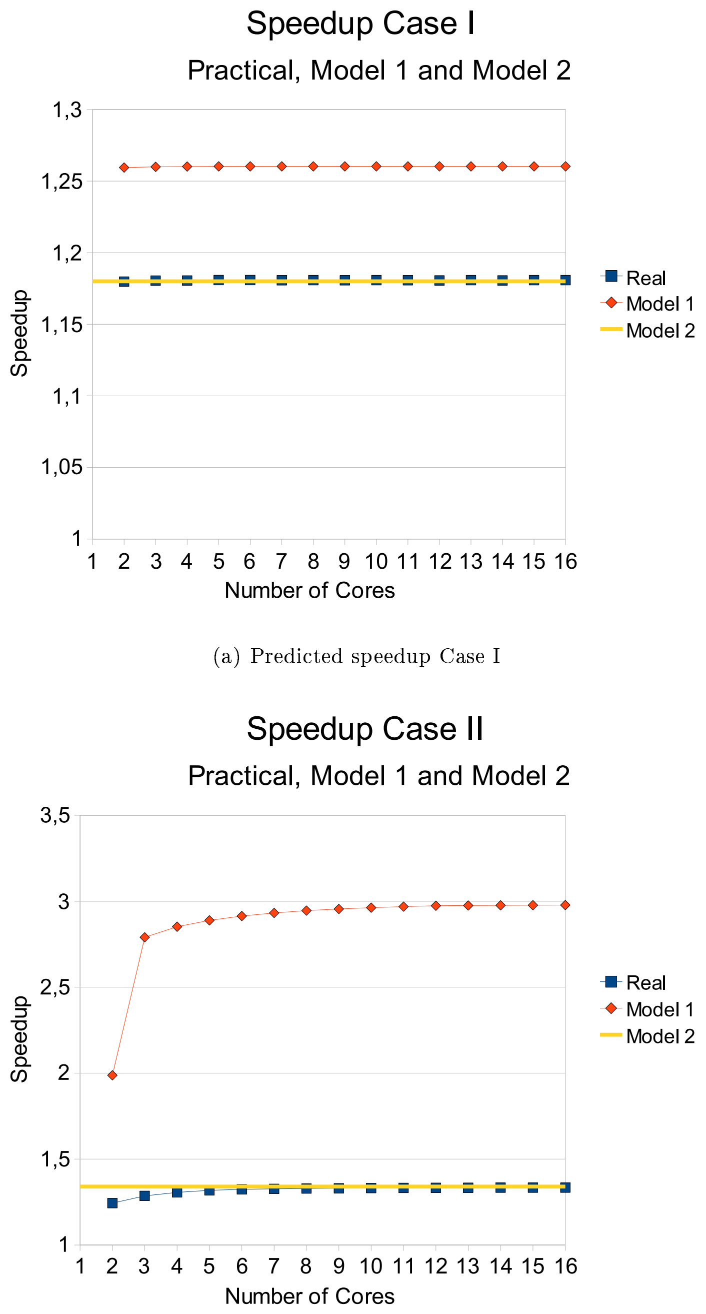
By using three different models and analyzing the problem cases in accordance with the assumptions made, we conclude that the practical timing model and Model 2 give reliable results, while Model 1's predictions are too ambitious for the current test cases.



(a) Predicted times for Case I

(b) Predicted times for Case II

Figure 4.3: Time predicted by all models, for Case I and Case II



(a) Predicted speedup Case I

(b) Predicted speedup Case II

Figure 4.4: Speedup predicted by all models, for Case I and Case II

Chapter 5

Parallel Implementations

This chapter describes the parallel implementations performed in this project. First, short introduction to different profilers and given, followed by a listing of the hardware used in the benchmarks. Next, debugging issues introduced in parallel code is discussed, followed by a descriptions of the GPU implementations and the problems encountered when trying to offload CPU-cycles on the GPU. The last section gives a detailed description of the multi-core CPU implementation and its results.

5.1 Profiling

Special GPU profilers are available which can show how the GPU is utilized. However, these were not initially intended for GPGPU applications and will therefore not be used in this project. Care must be taken when using general CPU profilers on code with calls to GPUs. Blocking and non-blocking calls between CPU and GPU may give erroneous profiles [30].

Here is a list of GPU profilers:

- NVIDIA's NVPerfHUD
- AMD's GPU PerfStudio
- AMD's GPU ShaderAnalyzer

5.1.1 Quick Load Balancing Test for GPU Code

A quick way to see if the GPU computations are memory bandwidth limited, is to change the frequency of either the core or the memory to see if the time used changes. If the timing results are the same after down-clocking or overclocking the core, it means that memory is the bottleneck [30]. On earlier split shader designs, adding more work to either the fragment or vertex shaders, could help determine if either are overloaded. If the timings are the same after adding more work, the respective shader processor is underutilized and the new computations are for free.

5.2 Hardware Used

Table 5.1, shows the hardware used for the CUDA implementations. Initially, a rather outdated computer was used with CUDA, since there was already a GeForce 8800GTX installed in it. The P4 machine was soon replaced by the another AMD64 machine, for reasoned explained in Section 2.3.1. For CPU and multi-core implementations, the hardware listed in Table 5.2 was used for testing.

Table 5.1: CUDA Hardware

Hardware	P4	AMD64
CPU	P4	AMD64
Freq.	3.2 GHz	3500+
Extras	HT	64-bit
RAM	1GB	2GB
MotherB.	MSI 945PL Neo v1.0	ASUS A8E-V Deluxe
GPU	GeForce 8800GTX	GeForce 8800GTX
Driver	169.09	169.09
CUDA Toolkit	1.1	1.1
OS.	Linux Ubuntu 7.10 32bit	Linux Ubuntu 7.10 64bit
Compiler	gcc 4.1.2	gcc 4.1.2

Table 5.2: Multi-core Hardware

Hardware	HP	ASUS W5F Laptop
CPU(s)	4 x Intel Xeon X7350	1 x Intel Centrino Duo
Freq.	2.93 GHz	1.66 GHz
Num. Cores	16	2
RAM	64 GB	2.5 GB
OS.	CentOS 5 64-bit	Linux Ubuntu 7.10 32bit
Compiler	gcc 4.1.2	gcc 4.1.2

5.3 Debugging

Multithreaded code, because of its undefined order of execution, introduces some new interesting bugs which can be daunting to address, known as deadlocks and race conditions. Deadlocks occur when a thread is waiting on a signal from another thread or waiting for a locking variable to be unlocked by another thread, without this ever happening. This will cause the program to hang indefinitely.

Race Conditions

Race conditions occur when multiple threads are accessing a shared resource or memory location at the same time. If one thread writes to this location while another thread reads from the same location, a race condition occurs and the end result is undefined. However, the application may pass such critical sections with no harm done, which makes these bugs harder to detect. Duplicating these bugs may also be a problem, since they may not occur in an orderly fashion, because of arbitrary threading conditions or other events in the operating system. Even worse, debugging using a debugger may introduce certain synchronizing events, which hides the bugs altogether.

Even introducing too many printfs may mask this bugs in certain cases. The order of appearance of these statements are random onscreen, making it harder to debug using this technique. However, printf statements are still used for debugging multithreaded code. Information such as thread ID and the function the printf statements is called from should be printed. Pthreads do not have a thread ID in the specification, but one could use the thread handle's address, which is returned from the call to pthread_create [14]. In our implementation, a thread class will be created containing this handle as well as an integer value ID.

Calls to mutex and condition variables should be surrounded with a printfs when trying to locate a deadlock or a race condition. This makes it easier to see if a certain call to mutex or a condition variable actually happened. It is convenient to conditionally compile these debug printf statements into the code by defining a DEBUG symbol, since these debug statements can come in handy at a later time.

Debugging Tools

In this project, a combination of debuggers will be used. First of all, printf statements will be placed in strategic locations in the code. Our second weapon will be a thread enabled debugger, GDB, already available in Linux. There is also a convenient graphical user interface, DDD, available. The third, and most advanced, is Valgrind's tools Helgrind and Memcheck, available in version 3.3.0 and above. Helgrind is a multithreading debugger, which locates possible race conditions during runtime. Memcheck can be used to locate memory leaks in the code. Profiling is executed like this[1]:

valgrind -v –tool=helgrind ./executable -args

[1] More options can be found on the Valgrind web page, valgrind.org.

```
valgrind -v –tool=memcheck ./executable -args
```

5.4 Parallelizing the Application for GPU

The following sections describes the different parts of the code that was implemented on the GPU, and the resulting speedup/slowdown.

The steady-state calculation, uses an integral method for calculating pressure-loss from the start to the end of a pipe. Each section has to be calculated serially, since the current section needs the result from the former as input. This indicates that to introduce parallelism, multiple pipes have to be calculated in parallel. Because of the applications complexity, this will be extremely hard to implement on a GPU. Thus, some attempts at lower lever parallelism were attempted, to see what the impact of introducing GPU computations would be.

The transient calculation, however, can do calculations on multiple pipe-segments simultaneously, thus should be much more suitable on a GPU. It also consists of a single function, with no tree-traversal, external function calls or administration logic, thus should be low-level enough to fit for GPU computation.

5.4.1 CUDA Benchmarks

Both the included bandwidth test and a simple ping-pong test was performed on both CUDA machines. It was observed that none of the machines were able to reach close to maximum transfer rates between the CPU (Host) and the GPU (Device). With newer motherboards and CPUs, the bandwidth between the Host and Device has been reported, on GPU forums, to push close to the theoretical maximum of 4 GB/s over a PCIe 16x port. Due to restricted resources, this has not been confirmed in this project.

Table 5.3: CUDA benchmark results, pinned memory

Test\Machine	P4	AMD64
Bandwidth: Host->Device	~980 MB/s	~1.5 GB/s
Bandwidth: Device->Host	~650 MB/s	~670 MB/s
Ping Pong Test: 32 floats	~25 000 iter/s	~31 000 iter/s

5.4.2 Steady State Calculation

Figure 2.1, shows that the only candidate code for GPU computations is *Pipe::CalcDP*. It is the function with the largest impact on CPU-time as well as having multiple values that can be calculated in parallel. It includes 93% of the program's runtime. Profiling also shows that this particular function is called 43 573 093 times for 100 calculation steps. Benchmarks, using wall-clock as parameter, states that this setup uses an average of 61 seconds to complete. Thus, this function is called ~700 000 times/second. This means that transferring data from the CPU to the GPU and back again, must be possible to do at least 700 000 times/second, to gain any speedup.

A simple ping-pong program, shown in Appendix B.2, was written to see if this is possible. Tests show that sending 32 single-precision values to and from the CPU and GPU, without any calls to a kernel, peaked at around 25 000 times/second for the P4 Neo chipset and at about 31 000 on the AMD64 Via chipset. This is not even close to what is needed to gain speedup in this case.

Thus, in theory, this calculation is not suited for GPUs. It's computational intensity is too low, and the target functions are called too often. To see just how unsuitable it is, a few implementations were attempted and their speedup calculated. A short description of each implementation follows:

1. Each function, represented by the small squares inside *CalcDP* seen in Figure 2.2, calls the same mathematical calculation with different input data. Hence, it was seen as a candidate for parallelization and moved from the CPU to the GPU. Different instances of these functions are called from other places of the code as well, which will also utilize this code. The implemented function takes 16 input variables and returns a single float. It executes only one single thread in each call, thus highly unoptimal for GPUs. This resulted in extreme slowdown of 508.08. The CPU code used 61 seconds, while the GPU implementation used 29977 seconds! The fact that the implementation only executes a single thread on the GPU speaks for itself.

2. After the obvious failure of the first attempt, more code was added to introduce parallelism on the GPU. A function getDensity calls six instances of the function described in the previous point. On the GPU, these six instances are called simultaneously. However, the amount of instructions needed to create different arrays of input data for the GPU, supersedes the amount of computations needed by the CPU to do the computations itself. The former implementation did not need these arrays to be computed, since it only sends 16 variables in the argument list. Despite this, a speedup of 1.095 was observed compared to the former implementation. However, a slowdown of 463.95, compared to the CPU version, is still observed.

3. The third stage was to implement all 32 calculations on the GPU, including the ones in getDensity. This implementation consists of a large function in C++, that gathers the necessary data from the different input matrices and sends it to the GPU. The GPU uses 32 threads to calculate all values simultaneously, before the results are transferred back to the CPU. A speedup of 11.68 was observed, compared to the first implementation. This shows how extremely important it is to utilize the GPU's parallelism. However, 32 threads are not even close to the number of threads needed to fully utilize a GPU and a slowdown of 43.49 is still observed compared to

the CPU version. The main bottleneck, is that the data cannot be moved fast enough between the CPU and the GPU, as mentioned in Section 2.3.1. There is also a lot of necessary data gathering and administration overhead on the CPU, before the data is sent to the GPU for computation.

4. There are other areas of the code which calls some of the same functions represented by the squares mentioned in point 1, but none that show up in the profile. As a last attempt, only the part in PipeNS::Pipe::CalcDP is run on the GPU, since this is the only sufficient place where all 32 variables are needed. The rest is run on the CPU. This shows the impact only on the part really showing in the profile. A speedup of 1.75 was observed, compared to the former implementation, which concludes that there is nothing to gain by offloading CPU-cycles on the GPU for this particular calculation. A slowdown of 24.71 compared to the full CPU implementation, is still observed.

All these attempts confirms the conclusion in Section 2.3.1, that this calculation is not suitable to run on the GPU. A lot more code has to be moved to the GPU to be able to gain any speedup. The primary element that has to be eliminated, is the transfer of data between the CPU and the GPU. It can not be done fast enough for this part of the code, thus the necessary input data would have to be generated on the GPU. This will not be tested in this project, since the possible gain is assumed to be too low compared to other parts of the code. A better parallelization for this particular code, would be to use multithreading on the CPU on a higher level of the code instead.

5.4.3 Transient Calculation

In this calculation, multiple parts of a pipe can be calculated simultaneously. A pipe can consist from tenth to thousands of sections, each that can be calculated in parallel. There are two parts in this calculation. A momentum calculation followed by temperature calculation. The latter calculates how the temperature spreads out of the pipe walls and into the ground. It has no external function calls, as the momentum calculation does. Since these calculations are done one after the another on the CPU, it should be possible to execute the temperature calculation on the GPU, while the CPU executes the momentum calculation. However, after the serial optimizations from Section 2.3.3, the target temperature calculation function only occupies about 30% of the total CPU-time. This, in combination with the function's complexity, does not justify maintaining GPU code for it, thus it will not be implemented in this project.

5.5 Parallelizing the Application for Multi-Core

Parallelizing the application for multi-core CPUs, was soon seen as a more feasible solution than utilizing a GPU. The application has several possible

Listing 5.1: Pipe::CalcDP: OpenMP Sections

```
pvt.getposition(p,t);
#pragma omp parallel sections
{
 #pragma omp section
 {
  pvt.Getdensity(p,t, roa, roof, roga, rogf, rowl, rowv, roglyc);
  pvt.Getmasratios(p,t, mratghci,mratohci, mratwgac);
  pvt.Getviscosity(p,t, viso, viswl, visg,visglyc);
 }
 #pragma omp section
 {
  pvt.GetdHdP(p,t, dhdpo, dhdpw, dhdpg,dhdpglyc);
  pvt.GetdHdT(p,t, dhdto, dhdtw, dhdtg,dhdtglyc);
  pvt.Getentalpy(p,t, ho, hw, hg,hglyc);
  pvt.Getsoundvel(p,t, svelo, svelw, svelg,svelglyc);
 }
}
```

high-level parallelization opportunities, which is unsuitable for GPUs due to its special purpose design.

When testing the thread pool, the Bash utility Time will be used. This can give an indication of the scalability of the thread pool, by calculating both the real and the user time used by an application. The script B.4 in Appendix B.1 converts the output, from the Time utility, into seconds.

5.5.1 Parallelizing by looking at the profile

Looking at the profile image in Figure 2.2, the first parallel option that comes in mind is the same as was tried on the GPU, namely *Pipe::CalcDP*. By using OpenMP sections, the code in listing 5.1 was altered to add parallel execution of the 32 *pvttab::(...)* functions inside *Pipe::CalcDP*. However, this actually caused a slowdown compared to the serial version, which tells us that the overhead of managing threads for these computations is too large to gain any speedup by executing them in parallel. It is not a scalable solution either, since there are only two sections. Adding a section to each function call was even slower. Thus, it is necessary to go to a higher level for parallelization to gain speedup in this application.

This shows that profiling alone is not always enough to find the correct targets for optimization in an application. Deeper knowledge of the respective algorithms and applications is also necessary.

5.5.2 Thread pool Implementation

Initially, Intel's *taskq pragma*, also to be available in the OpenMP 3.0 specification, was investigated to see if it could solve our problem with little programming effort. However, no option to easily maintain the interdependencies between the tasks was found, thus it would result in incorrect results.

A task pool was created using the Pthreads library available in Linux instead. Using Pthreads, controlling the interdependencies between the tasks was possible, using parent pointers and a child counter protected by mutexes. The same tools were used to control the thread pool. A thread class was developed which supports suspending the threads between computations instead of destroying them. When there's no more work available, the threads will register themselves in a ready queue and put themselves in suspend mode. The main thread resumes worker threads by getting a task from the queue and assigning it to a ready thread. It then signals the worker thread to start executing its new task. When the main thread has assigned a task to each worker thread, it starts emptying its queue alongside the other threads until it's empty. There is also a barrier for the main thread, which makes sure all tasks are done before the main thread continues.

Each worker thread will get new tasks from the queue themselves, until there is no more work left. Since only the leaf nodes are placed in the queue, each thread checks its current task's parent task, to see if its current task is the last child. If it is, the thread will execute the parent task before getting a new task from the task queue. The last thread to finish its task in a queue, signals the main thread for this queue to continue.

Nested Parallel Queues

Another important feature of the thread pool, is support for nested parallel queues. This means that a task in a queue may itself create a queue of its own and register it in the thread handler, in a FILO manner. Thus, the most recent queue to be registered will be emptied first, by all available threads. This feature requires the main thread to join the work of emptying its own queue, or else all threads would eventually end up waiting at the barrier mentioned above.

Debugging

The thread pool implementation was debugged extensively by using a small and simple dummy task tree. It is extremely important to protect all critical sections with mutexes to avoid race conditions. Each execution may slightly differ from each other, which in turn may result in deadlocks or other errors arbitrarily, if the code has not been made 100% thread safe.

Using printfs to debug parallel code is quite hard, since the order of the printouts are arbitrary and may differ in every execution. In Linux, GDB can debug multithreaded code by allowing the user to step through the code, following every thread's execution. DDD is a graphical user interface for GDB. By using GDB in combination with Valgrind's tool Helgrind, for multithreaded

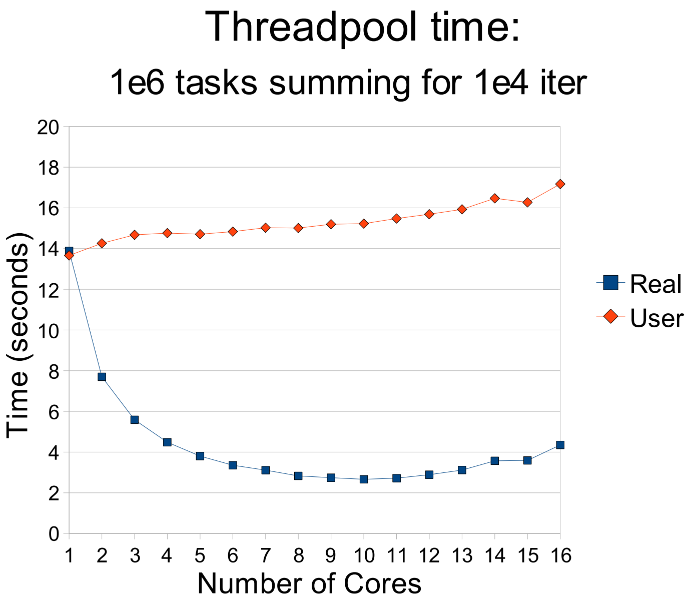


Figure 5.1: Thread pool test: Small sized tasks

debugging, and Memcheck, for memory leak detection, all race conditions and memory leaks were addressed and fixed. Without these tools, this would have taken much longer. Intel's VTune and TAU were also tried, but they always crashed without giving any profiling results.

The thread pool implementation was compiled as a dynamic library and linked into the application. An example can be found in Appendix A.

Results

Figures 5.1, 5.2, 5.3 and 5.4, shows the scalability of the thread pool up to 16 cores, for very short tasks, and the effect of having larger tasks. For short tasks, the overhead of managing more threads reduces the efficiency of the calculation, when using more than 10 threads. Adding more work to the tasks gives better scalability, as expected.

The first attempt at parallelizing, focused on the third level, mentioned in Section 4.2. This quickly showed that the overhead of creating a task tree and maintaining the threads, is too large compared to the amount of computations performed. The main problem is that it's not possible to reuse the task tree once created, since it may change in each iteration.

The second attempt focused on the second level, namely the traversal of the whole three. This time it's possible to reuse the tree for multiple iterations, thus minimizing overhead for this operation. However, the number of iterations

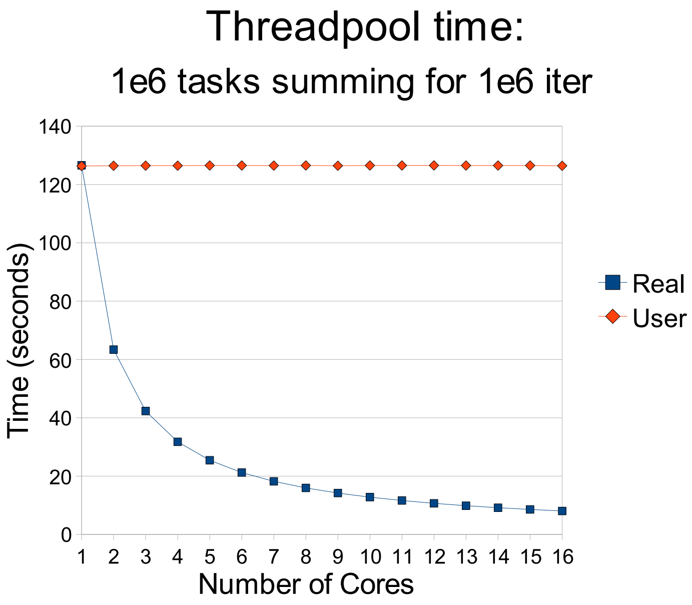


Figure 5.2: Thread pool test: Medium sized tasks

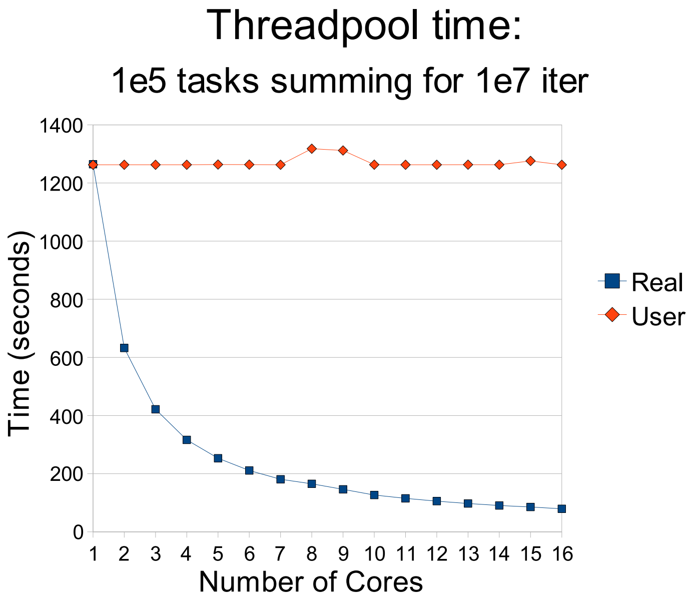


Figure 5.3: Thread pool test: Large sized tasks

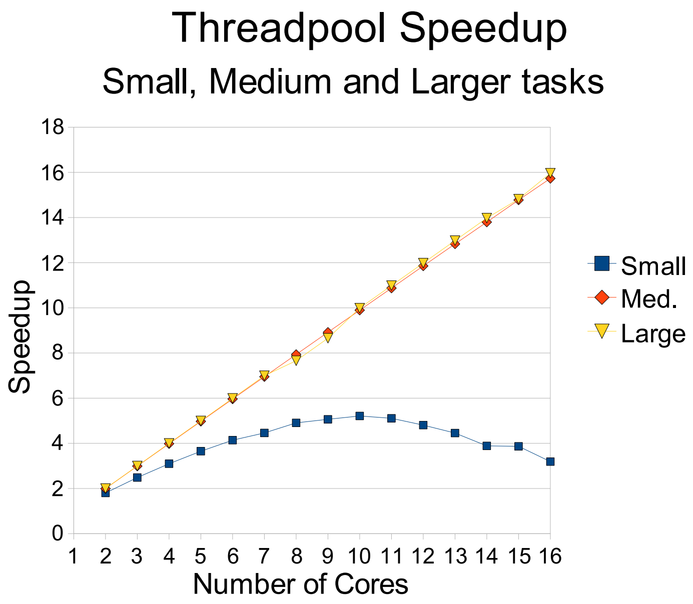


Figure 5.4: Thread pool test: Speedup of Small, Med. and Large tasks

is unknown and might even be a single iteration.

Although the application, in theory, as a lot of independent computations, the developer's focus has been on serial execution. Thus, problems occurred when introducing multiple threads, mostly due to race conditions. The developers have made an effort to minimize memory consumption, by having different parts of the code pointing to the same shared memory locations. This is fine for serial execution, however, when introducing multiple threads, an enormous amount of possible race conditions were detected by the Valgrind profiler.

A large amount of time and effort was used trying to remove all potential race conditions, by making multiple copies of shared data, making functions static and cleaning up different parts of the code. Still, a lot of potential race conditions were detected by the profiler and a solution for these was not found within the time frame of this project, even with the help from the developers. Despite this, for Case I, correct results were produced in some cases, while others not, by altering the number of threads used. However, this is not acceptable. For Case II, the complexity of the event system triggered during the execution, resulted in erroneous results for every parallel traversal of the network.

Due to the race condition problems, there was no attempt in trying to parallelize the highest level, namely multiple pressure systems in parallel. Another way to parallelize this level, is to simply execute multiple instances of the application at the same time, using different input files. This can be streamlined by using Grid technology, were script files can be used to enqueue different jobs (e.g. different pressure systems) [22]. The Grid middleware will then assign a job to each core in a network of computers.

Chapter 6

Conclusion and Future Work

In this project, we have optimized a large commercial, serially coded application for computing optimal flow through oil-well networks. Then, we saw how it would adapt to two of today's parallel architectures, the GPU and multi-core CPUs.

We also created one practical and two theoretical parallel performance prediction models for traversing the oil-well networks in parallel. For different test cases, we were able to predict the maximum speedup theoretically possible to gain, by traversing the network in parallel.

Serial Optimizations

Initially, the application was thoroughly profiled and analyzed, using Valgrind, to find hotspots in the code, and to get an overview of the application. Prior to this project, not much time had been spent in code optimization, thus, this became the first goal.

In this project, we showed that for this application, in its current state, there was more to gain by optimizing the algorithms and changing storage strategies, than by adding parallelism.

Multiple serial optimizations were performed, gaining between 1.5x-3x speedup for the steady state calculation, and 30x-50x speedup for the transient calculation, on different architectures. The latter was achieved by switching from an unoptimal database storage to a fast binary file storage during the computation.

GPU

By using profilers to locate hotspots in the application, target code for GPU was located. However, the goal to offload CPU cycles to the GPU, for the steady-state calculation, was in the early stages of the projects seen as unfit, due to the lack of computational intensity. The transient calculation was at the time of this project too experimental to convert to GPU code and after our serial optimizations gained 30x-50x speedup, the target temperature calculation only took about 30% of the total CPU time in the Valgrind profile. It was hence concluded that, with only using 30% of the CPU-time, implementing and maintaining a GPU code for the transient calculation, would not give adequate speedup compared to the effort needed to do so. Large parts of the application must be redesigned and rewritten to work on a GPU, with the most important factor being minimizing data transfer over the PCI express port.

Multi-Core

Realizing that GPU technology was likely unfit for this application, we moved our focus to multi-core CPU technology. By instead doing a top down approach to find suitable parallel parts of the application, the choice of parallelization technique was a thread pool implementation in combination with a task queue, which could be well adapted to the structure of the application. By using a thread pool, the threads can be reused for different tasks during the computation, saving the overhead of creating and destroying threads for each small computation. However, introducing multiple threads also introduced the main problem in multithreading, namely race conditions. In both test cases, the possible race conditions (more than 600), found by Valgrind, resulted in erroneous results for the multithreaded executions. A lot of work was put into fixing these errors, with the help from the developers of the application. However, not all could be addressed within the time-frame of this project.

The final part of the project, took a more theoretical approach in trying to determine if parallelization is worthwhile for these types of applications, on given sets of test data, and to figure out what which parts of the application is suited for parallelization. Detailed timings of the serial execution were used to create parallel replica calculations. Using three models, we saw that the parallel traversal of the networks did not scale well beyond 2-4 threads. The practical model showed low overhead caused by the thread pool, compared to the other models, which suggests that the shared task queue is not the bottleneck here, although a better thread pool implementation would be using distributed queues with task stealing [12].

A maximum speedup of only 1.18 was predicted for the small case, and 1.34 for the large case, both with unlimited processors available. The small test case did not gain any more speedup due to the small amount of wells, and the large test case did not scale well, due to high workload imbalance between the top and the bottom layers of the network, and the serial dependence in between.

Summary

From the results of this project, the recommended strategy for further speeding up the given application, is focusing on serial optimizations of the core algorithms and storage routines. A lot of wasted cycles can be saved, by removing unnecessary data copying and simplifying or changing database storage routines with other, simpler data storage methods. Our binary file storage showed a tremendous speedup compared to the database routine in use now, and by using this binary file as a buffer, a thread could be used for storing this data in a database, while another thread is calculating as usual.

Our models also predicted low scalability and gain in speed, by multithreading only the network traversal part of the application. A lot of work has to be done to remove the remaining possible race conditions and, for any decent scalability at all, multiple networks must be calculated in parallel using the thread pool.

6.1 Future Work

In this section, different suggestions for future work is presented, starting with recommendations for this application:

- The first priority for this code should be a thorough cleanup of the main algorithms. Simple things such as removing unnecessary variables and breaking extremely long functions into smaller ones, will make the code easier to read and manage. By making each function smaller and less complex to analyze, compilers may also be able to do optimizations of its own more successfully. Breaking the code into smaller functions, might also reveal other portions of the code which can be used in the thread pool, in addition to portion targeted in this project.
- There's still a lot of potential for serial optimizations in this code, which should be addressed with close cooperation with the developers to maintain correct results.
- A suggestion for parallel execution of the application, is using Grid technology to enqueue multiple instances of the application, with different input data. This can either be done using script files, which defines different jobs to be executed, or the DRMAA API which gives access to Grid middleware from within an application. The former has already been tried for this application in previous work [22], with success. Grid middlewares are able to deliver work throughout a network of computers. Each core can be defined as a worker, thus utilizing multi-core technology without multithreading the application. This, however, requires multiple instances of the application being queued by the user, which again requires the user to have multiple individual cases to be computed simultaneously.

Other ideas for future work:

1. Further development of the models: The models created in this project, only models the traversal of the network by itself. However, there might be more potential speedup to be gained by utilizing the support for nested parallel queues in the thread pool. Extending the models to include multiple networks in parallel, could reveal if there is enough potential speedup to be gained to justify the amount of work needed to remove all race conditions and managing the added complexity of multithreaded code.

2. Verifying the models against similar dependence-tree problems: The models created in this project should, with minor modifications, be usable in similar applications to predict potential speedup by adding parallelism, such as the preconditioner problem in [2].

3. During this thesis project, NVIDIA introduced two new GPU series, the 9 series and the G260/G280 series. The latter was announced during the last week of my work on this thesis. The most important updates from the 9 series is a 512 bit memory bus and 240 shaders compared to 128 shaders. Testing applications more suitable for GPU, such as the mandelbrot set or 3D ultra sound applications, would be very interesting with these new GPUs.

Bibliography

[1] Shameem Akhter and Jason Roberts. *Multi-Core Programming. Increasing Performance through Software Multi-threading.* Intel Press, first edition, April 2006.

[2] José I. Aliaga, Matthias Bollhöfer, Alberto F. Martín, and Enrique S. Quintana-Ortí. Parallelization of multilevel preconditioners constructed from inverse-based ilus on shared-memory multiprocessors. In *Proceedings of the International Conference ParCo 2007: Parallel computing: Architectures, Algorithms and Applications*, pages 287–294. NIC-Directors, 2007.

[3] Beyond3D. Nvidia g80: Architecture and gpu analysis. `http://www.beyond3d.com/content/reviews/1/1`, 2007. [Online; accessed 08.02.08].

[4] Ian Buck. *Taking the Plunge into GPU Computing*, volume 1 of *GPU Gems 2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 32, pages 509–519. Addison-Wesley, Upper Saddle River, NJ, first edition, March 2005.

[5] Randima Fernando and Mark J. Kilgard. *Introduction*, volume 1 of *The Cg Tutorial, The Definite Guide to Programmable Real-Time Graphics*, chapter 1, pages 1–35. Addison-Wesley, Upper Saddle River, NJ, first edition, February 2003.

[6] M. Gillespie and C. Breshears(Intel Corp.). Achieving threading success. `www.intel.com/cd/ids/developer/asmo-na/eng/212806.htm`, 2005. [Online; accessed 02.04.08].

[7] gpgpu.org GPU floating point. `http://www.gpgpu.org/w/index.php/FAQ#Where_can_I_get_information_about_GPU_floating_point_precision.3F_Is_it_IEEE-compliant.3F`. [Online; accessed 04.05.08].

[8] Mark Harris. *Mapping Computational Concepts to GPUs*, volume 1 of *GPU Gems 2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 31, pages 493–508. Addison-Wesley, Upper Saddle River, NJ, first edition, March 2005.

[9] Mark Harris and Ian Buck. *GPU Flow-Control Idioms*, volume 1 of *GPU Gems 2 Programming Techniques for High-Performance Graphics and*

General-Purpose Computation, chapter 34, pages 547–555. Addison-Wesley, Upper Saddle River, NJ, first edition, March 2005.

[10] Andrew Binstock (Intel). Choosing between openmp* and explicit threading methods. `http://softwarecommunity.intel.com/articles/eng/1677.htm`, 2008. [Online; accessed 02.04.08].

[11] Christoph Kessler and Welf Löwe. A framework for performance-aware composition of explicitly parallel components. In *Proceedings of the International Conference ParCo 2007: Parallel computing: Architectures, Algorithms and Applications*, pages 227–234. NIC-Directors, 2007.

[12] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1):1–47, 2003.

[13] Leif Christian Larsen. Utilizing gpus on cluster computers. Technical report, NTNU, Norway, 2006.

[14] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly and Associates Inc., 101 Morris Street, Sebastopol, CA, first edition, September 1996.

[15] NVIDIA. `http://www.nvidia.com`. [Online; accessed 20.01.08].

[16] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide v1.1*. NVIDIA, November 2007.

[17] Lectures Univ. of Illinois. `http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html`. [Online; accessed 04.02.08].

[18] openmp.org. `http://www.openmp.org`. [Online; accessed 26.03.08].

[19] John Owens. *Streaming Architectures and Technology Trends*, volume 1 of *GPU Gems 2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 29, pages 457–470. Addison-Wesley, Upper Saddle River, NJ, first edition, March 2005.

[20] RAPIDMIND. `http://www.rapidmind.net`. [Online; accessed 25.03.08].

[21] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, July 2007.

[22] Atle Rudshaug. Grid technologies for task parallelization of short jobs, 2007.

[23] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen. Compiler support of the workqueuing execution model for intel smp architectures. In *Proceedings, EWOMP, Rome*, 2002.

[24] Craig Szydlowski. Multithreaded technology and multicore processors. preparing yourself for next-generation cpus. In *Dr. Dobb's*, May 2005.

[25] threadpool.sourceforge.net. `http://threadpool.sourceforge.net/`. [Online; accessed 16.04.08].

[26] Damien Triolet. Nvidia geforce 8800 gtx and 8800 gts. `http://www.behardware.com/articles/644-1/nvidia-geforce-8800-gtx-8800-gts.html`, 2007. [Online; accessed 05.02.08].

[27] Damien Triolet and Marc Prieur. Nvidia geforce 7800 gtx. `http://www.behardware.com/articles/574-1/nvidia-geforce-7800-gtx.html`, 2005. [Online; accessed 05.02.08].

[28] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

[29] Wikipedia. `http://en.wikipedia.org/wiki/Close_to_Metal`. [Online; accessed 16.06.08].

[30] Cliff Woolley. *GPU Program Optimization*, volume 1 of *GPU Gems 2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 35, pages 557–571. Addison-Wesley, Upper Saddle River, NJ, first edition, March 2005.

Appendix A

Thread Pool Overview

- ThreadHandler: Creates and manages threads and distributes work to available threads from a list of task queues.
- CThread: A thread wrapper which supports suspending threads. When a thread is signaled from the thread handler, the thread resumes execution until all task queues are emptied. When there is no more work, it puts itself in suspend mode and registers itself as available.
- TaskQueue: Holds a queue of tasks and condition variables used for synchronizing the parallel queue traversal with the main thread.
- CTask: A base class for tasks, with pointers to parent tasks to support tree structured workloads. Override this class to make arbitrary tasks.
- ThreadTimer: Has functions for starting and stopping timers, and saving timings in arrays.
- main: Includes a lot of test code and examples of thread pool usage.

A.1 Using the Thread Pool

The thread pool implementation can be compiled as a library and linked into any application as in Listing A.4. For tasks with interdependencies, use parent pointers between tasks to keep the correct order of execution. Only leaf tasks must be added to the queue, since the parent tasks will be accessed through the parent pointers from their children. Thus, it is necessary to register at least one root task in the queue to maintain synchronization with the main thread. When all root tasks are finished, the main thread is signaled to continue.

Listing A.3, A.1 and A.2, shows a simple example of how to use the thread pool.

Listing A.1: Thread pool build test task tree example

```
// Recursively builds a test case tree
CTask* buildTaskTree(TaskQueue & taskQueue, CTask* parent_task, int depth)
{
    if(g_counter <= (1 << depth))
        return 0;

    TestTask* task = new TestTask;
    task->setId(g_counter--);
    task->setParent(parent_task);
    task->startValue = 100;
    task->iter = 100000;

    if (0 == parent_task)
    {
        taskQueue.addRootTask(task);
        task->setIsRootTask(true);
    }
    else
        parent_task->addChild(task);

    bool isLeafNode = true;
    for(size_t i = 0; i < 3; i++)
    {
        if(0 != buildTaskTree(taskQueue, task, depth+1))
        {
            task->setChildCount(task->getChildCounter()+1);
            isLeafNode = false;
        }
    }
    if(true == isLeafNode)
        taskQueue.push(task);

    return task;
}
```

Listing A.2: Thread pool task example

```
// A test task that overrides the threadpool task class
struct TestTask : public CTask
{
    int iter;
    int startValue;

    //Override work
    void Work()
    {
        do
        {
            startValue++;
            iter--;
        }while(iter > 0);
    }
};
```

Listing A.3: Thread pool usage example

```
int main(int argc, char **argv)
{
 // Register number of threads to use in threadpool
 int numThreads = atoi(argv[1]);
 g_counter = atoi(argv[2]);
 ThreadHandler::Instance()->addThreads(numThreads);
 TaskQueue taskQueue;

 // Some function for building a tree structured queue
 buildTaskTree(taskQueue, 0, 0);

 // Distribute work to threads
 ThreadHandler::Instance()->distributeWork(&taskQueue);

 // Clean up threads
 delete ThreadHandler::Instance();
}
```

Listing A.4: Including the thread pool into your application

```
#Add the following to Makefile for threadpool support
INCLUDES = -I/path/to/threadpool/headers
LDLIBS = -lThreadPool
```

Appendix B

Benchmark Scripts and Code

B.1 Bash Scripts

The following bash scripts were used to automatically execute different binaries and collect average run times for each execution.

Listing B.1: Script for automatic benchmarking

```
##############################################################
# ./runMultiBenchmark.sh <num_bench_pr_binary> <start> <stop> <binary_>
# Executes binary files $1 times from start=$2 to stop=$3
# Collects and writes average time used to a resultfile
##############################################################
#!/bin/bash

resfile=benchmarks/timingResults.`date +%Y%m%d-%H%M%S`
for filename in $( ls $4* ); do
        for ((i=0;i<$1;i++)) ; do
                echo "Running dagocbench on" $filename $i
                ./runDagocBenchmark.sh $filename $2 $3
        done

        echo "bench_$2_$3-"$filename >> $resfile
        ./getTimeFromBenchFiles.sh benchmarks/bench_$2_$3-$filename >> $resfile
        echo "******************************" >> $resfile

done
```

Listing B.2: Script for executing and timing a steady state binary

```
############################################################
# ./runDagocBenchmark.sh <binary> <start> <stop>
# Executes binary file $1 from start=$2 to stop=$3
# Writes application output and time used to a resultfile
############################################################
#!/bin/bash

unique=‘date +%Y%m%d-%H%M%S‘
echo "unique: " $unique
res=benchmarks/bench_$2_$3-$1-$unique.txt
echo "ResultFile: " $res

s0=$(date +%s)

./$1 -c -start=$2 -stop=$3 ~/ygg/dagocproject/setups/TEST.sup >> $res

s1=$(date +%s)

let s=$s1-$s0
echo "Start:" $2 >> $res
echo "End:" $3 >> $res
echo "Executable:" $1 >> $res
echo "Total time:" $s "seconds" >> $res
```

Listing B.3: Script for getting average time for multiple benchmarks

```
############################################################
# ./getTimeFromBenchFiles.sh <bench_file>
# Collects and writes average time used to a resultfile
############################################################
#!/bin/bash
t=0
num=0
for i in $( ls $1* ); do
        echo "File: " $i
        tmp=$(tail -n 1 $i | grep -oE [0-9]+)
        echo "Time used: " $tmp
        echo "
            ------------------------------------------------------------
            "
        let t=t+tmp
        let num=num+1
done
echo "Total time for" $1 ":" $t
echo "Total files:" $num
let tot=t/num
echo "***AVERAGE TIME***" $tot
```

Listing B.4: Script for converting time output to seconds

```
############################################################
# ./convert_timings.sh <bash time output>
# Converts real and user time output from
# bash time function into seconds
############################################################
#!/bin/bash
echo "REAL:"
for i in $( grep "real" $1 | cut -d" " -f5); do
 min=$(echo $i | cut -d"m" -f1)
 sek=$(echo $i | cut -d"m" -f2 | cut -d"s" -f1)
 echo "scale=4; ($min*60)+$sek" | bc | sed 's/\./,/'
done

echo "USER:"
for i in $( grep "user" $1 | cut -d" " -f5); do
 min=$(echo $i | cut -d"m" -f1)
 sek=$(echo $i | cut -d"m" -f2 | cut -d"s" -f1)
 echo "scale=4; ($min*60)+$sek" | bc | sed 's/\./,/'
done
```

B.2 CUDA Ping-Pong Test

The Ping-Pong test is used to check how many times pr. second data can be transferred between the Host and the Device.

Listing B.5: CUDA Ping-Pong main.cpp

```cpp
#include <iostream>
#include <stdio.h>

extern "C" float run_pingpong_host(int* data, int numElements, int timeToRunInMs);

int main(int argc, char** argv)
{
    int numElem = 32;
    int seconds = 10;
    if(argc == 3)
    {
        numElem = atoi(argv[1]);
        seconds = atoi(argv[2]);
    }

    printf("PingPong %d elements for %d seconds\n", numElem, seconds);
    pingpongtest(numElem, seconds*1000);

    return 0;
}

void pingpongtest(int numElements, int timeToRunInMs)
{
    int data[numElements];
    float iterPrSec = 0.0f;
    int value = numElements * timeToRunInMs;

    //Run pingpong test
    iterPrSec = run_pingpong_host(data, numElements, timeToRunInMs);

    printf("Value in: %d\n", value);
    printf("Values out:\n");
    for(int i = 0; i < numElements; ++i)
    {
        if(i%10 == 0 && i != 0)
            printf("\n");
        printf("%d\t", data[i]);
    }
    printf("\nIterations pr. second: %.2f\n", iterPrSec);
}
```

Listing B.6: CUDA Ping-Pong Host

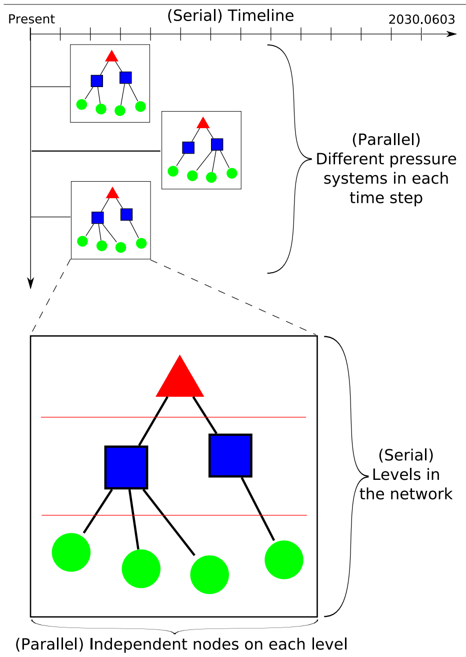
```
#include <stdlib.h>
#include <stdio.h>
#include <cutil.h>
#include <cuda.h>
#include <kernel.cu>

extern "C" float run_pingpong(int* data_slow, int numElements, int timeToRunInMs)
{
        //Get a CUDA device
        if(0 != getCudaDevice())
                return -1.0f;

        //Data pointers
        int *h_data, *d_data;

        //Calculate memory size to allocate
        const unsigned int mem_size = numElements*sizeof(int);

        //Alloc faster page-locked host memory
        cudaMallocHost((void**)&h_data, mem_size);

        //Alloc device memory
        cudaMalloc((void**)&d_data, mem_size);

        //Copy data from slower pageable memory to page-locked memory
        for(int i = 0; i < numElements; ++i)
        {
            h_data[i] = data_slow[i];
        }

        //Create a timer
        unsigned int timer = 0;
        float elapsedTimeInMs = 0.0f;
        cutCreateTimer(&timer);
        //Iteration counter
        unsigned int numIterations = 0;

        //Copy data from Host to Device then back again for a user defined time duration
        while(elapsedTimeInMs < timeToRunInMs)
        {
            //Start CUDA timer
            cutStartTimer(timer);

            //Copy data from host to device
            cudaMemcpy(d_data, h_data, mem_size, cudaMemcpyHostToDevice);

            //Call kernel here
            //pingpong_kernel<<<1,1>>>(d_data);

            //Copy data from device to host
            cudaMemcpy(h_data, d_data, mem_size, cudaMemcpyDeviceToHost);

            //Increment number of iterations
            ++numIterations;

            //Stop timer and get current time used
            cutStopTimer(timer);
            elapsedTimeInMs = cutGetTimerValue(timer);
        }

        //Free data
        cudaFree(d_data);
        cudaFreeHost(h_data);

        //Return number of iterations pr. second
        return (float) numIterations/(elapsedTimeInMs/1000);
}
```

Appendix C

Poster Presented at NOTUR 2008

Analyzing and Optimizing an Oil Well Network Code for Today's Parallel Architectures

Atle Rudshaug, IDI-NTNU, rudshaug@stud.idi.ntnu.no
Supervisor: Anne C. Elster, IDI-NTNU, elster@idi.ntnu.no


Present
(Serial) Timeline
2030.0603
(Parallel) Different pressure systems in each time step
(Serial) Levels in the network
(Parallel) Independent nodes on each level

The Application (METTE by Yggdrasil AS):

- Figure to the left identifies possible parallelism in the application
- Calculates pressure loss in oil well networks (tree structure)
- Controls actuators from sensor information for optimal flow
- Sample cases:
  - Small – order 10 wells/lines
  - Large – order 80 well/lines
  - Also varying time steps and distances (Seconds - weeks; 10m – 500km)
- Only serial application available

Valgrind Used for Serial Optimizations:

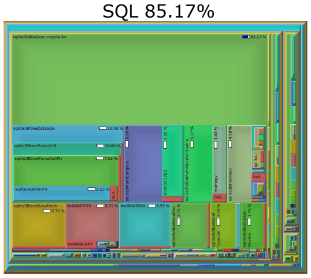
- Cachegrind
- Callgrind
  - KCachegrind (GUI)
- Helgrind: Thread debugger
- Memcheck: Detects memory-management problems
- (Both Intel VTune and TAU failed to give any results)

Hardware Used:

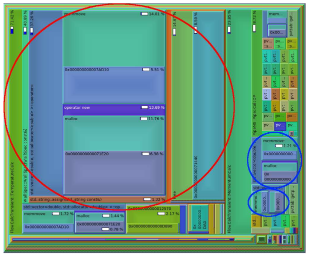
- AMD64 3500+, 512kb L2, 2GB RAM
- Asus Centrino Duo 1.66GHz, 2MB L2, 2.5GB RAM
- HP 4xQuad Xeon 2.93GHz, 4MB L2, 64GB RAM

Step 1: Optimize Serial Code

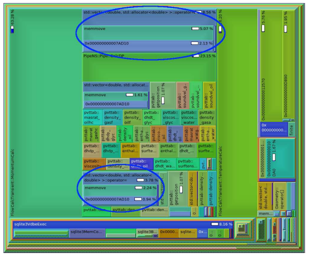
Original profile:
SQL 85.17%



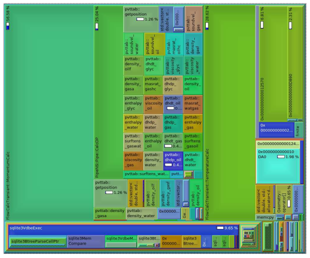
i) Temperature calc. 71.42%
Momentum calc. 23.85%

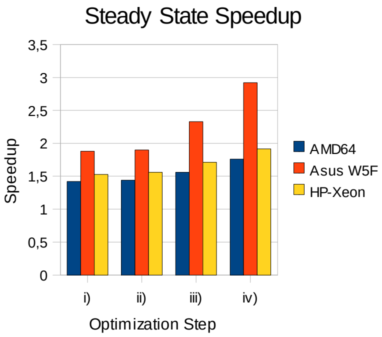


ii) Temperature calc. 24.20%
Momentum calc. 63.28%

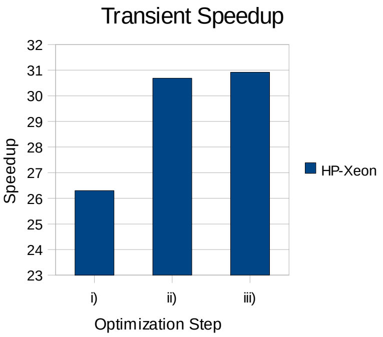


iii) Temperature calc. 28.62%
Momentum calc. 56.58%





~3x speedup gained on Asus
NB! Calculation time faster on the other architectures



~31x speedup gained

NTNU
Norwegian University of
Science and Technology

Step 2: Parallelize Serial Code

Original Goal:

- Parallelize network traversal using thread pool and task queue
- Problem: Profiler detected multiple possible race conditions when running multiple threads!
- Large amount of work to fix this problem

New Goal:

- Model/Predict parallelization to:
  - ○ Determine if parallelizing is worth-while
  - ○ Estimate which part of code worthy of parallelization

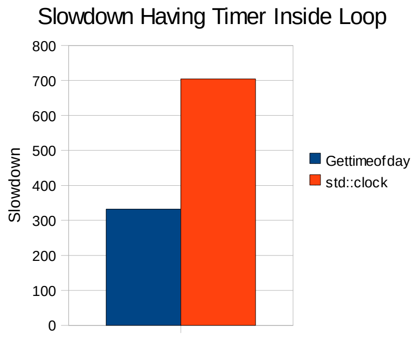
Modeling Steps:

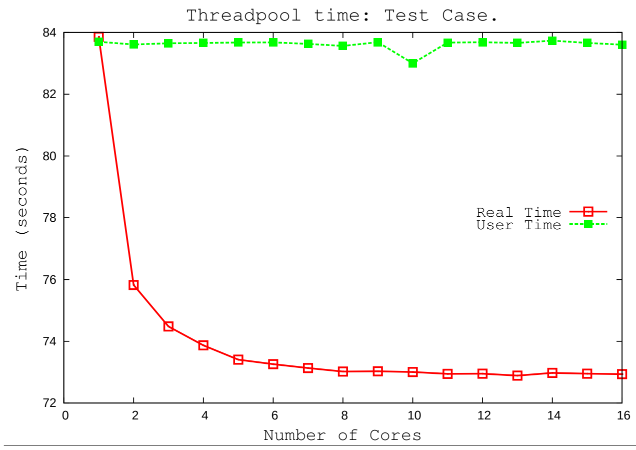
- Perform detailed timings of serial code
- Model network of tasks working/sleeping for same amount of time as individual task in serial code
- Compare traversal of tree with one thread vs. multiple threads (1-16 threads tested)

Different timers, different granularity and overhead!

9999999 iterations in loop for all benchmarks:

```
--- Timing sum+=1 outside loop using std::clock
STD outside loop timed: 0.020000 seconds
--- Timing sum+=1 outside loop using gettimeofday
TIMEVAL outside loop timed: 0.026519 seconds

--- Timing sum+=1 outside and inside loop using std::clock
STD inside loop timed: 6.860000 seconds
STD outside loop timed: 14.090000 seconds
--- Timing sum+=1 outside and inside loop using gettimeofday
TIMEVAL inside loop timed: 4.287836 seconds
TIMEVAL outside loop timed: 8.803945 seconds
```





Results

Our results show minimal gain parallelizing tested part of code, since individual nodes on top level takes much longer than sum of nodes on bottom levels, "serializing" code.

However, significant speedup were gained through analysis and optimizations!

TO DO: Check run-time & model for running multiple networks in parallel.

Also tried off-loading work to GPU:

(NVidia GForce 8800GTX using CUDA)

Experienced ~500x to ~43x slowdown due to data transfer overhead and processor under-utilization.

Acknowledgments:
We would like to thank Yggdrasil AS for providing a test application and a test case,
and for many detailed discussions.