
Case studies on GPU usage and data structure design

Jens Breitbart

Research Group Programming Languages / Methodologies
Dept. of Computer Science and Electrical Engineering
Universität Kassel
Kassel, Germany

Supervisor:

Prof. Dr. Claudia Leopold
Prof. Dr. Lutz Wegner

**U N I K A S S E L
V E R S I T Ä T**

My shot for the moon

SELBSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Kassel, den August 7, 2008

.....
(Jens Breitbart)

ABSTRACT

Big improvements in the performance of graphics processing units (GPUs) turned them into a compelling platform for high performance computing. In this thesis, we discuss the usage of NVIDIA's CUDA in two applications – Einstein@Home, a distributed computing software, and OpenSteer, a game-like application. Our work on Einstein@Home demonstrates that CUDA can be integrated into existing applications with minimal changes, even in programs, which have been designed without considering GPU usage. However the existing data structure of Einstein@Home performs poorly when used at the GPU. We demonstrate that using a redesigned data structure improves the performs to about three-times as fast as the original CPU version, even though the code executed at the device is not optimized. We further discuss the design of a novel spatial data structure called dynamic grid, which is optimized for CUDA usage. We measure its performance by integrating it into the Boids scenario of OpenSteer. Our new concept outperforms a uniform grid by a margin of up to 15%, even though the dynamic grid still provides optimization potential.

CONTENTS

CHAPTER 1 – INTRODUCTION	1
1.1 Related work	2
1.2 Structure of this thesis	3
CHAPTER 2 – CUDA	7
2.1 Hardware model	7
2.2 Software model	8
2.3 Performance	11
2.4 Why a GPU is no CPU	12
2.5 Programming the device	13
2.6 Hardware used throughout this thesis	14
CHAPTER 3 – EINSTEIN@HOME BACKGROUND	17
3.1 BOINC	17
3.2 Einstein@Home	19
CHAPTER 4 – EINSTEIN@HOME CUDA IMPLEMENTATION	21
4.1 CUDA integration	21
4.2 Data dependency analysis	25
4.3 CUDA F-statistics implementation	28
4.4 Conclusion	36
CHAPTER 5 – OPENSTEER FUNDAMENTALS	41
5.1 Steering and OpenSteer	41
5.2 OpenSteerDemo	42
CHAPTER 6 – STATIC GRID NEIGHBOR SEARCH	45
6.1 Static grid concept	45
6.2 Host managed static grid	46
6.3 GPU managed static grid	52
CHAPTER 7 – DYNAMIC GRID NEIGHBOR SEARCH	57
7.1 Dynamic grid concept	57
7.2 Three algorithms to create a dynamic grid	58
7.3 Performance	60
7.4 Conclusion	63
CHAPTER 8 – CONCLUSION	65

CHAPTER 1

Introduction

For some years, general-purpose computation on graphics processing units (known as GPGPU) is shaping up to get important whenever the performance requirements of an application cannot be fulfilled by a single CPU. Graphics processing units (GPUs) outperform CPUs in both memory bandwidth and floating-point performance roughly by a factor of 10.

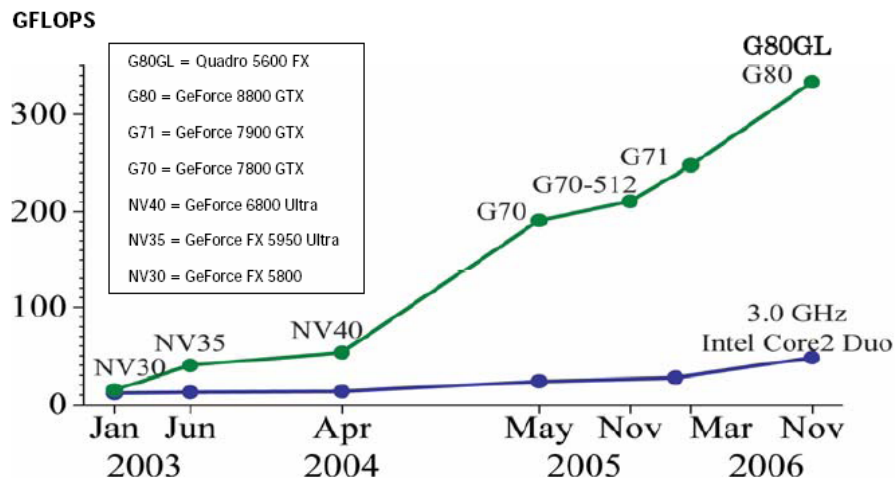


Figure 1.1: A comparison between the floating point performance of the CPU and the GPU [NVI07b].

Over the last two years using GPUs for GPGPU is becoming easier, as different new programming systems are made available on the market for free. We use the system provided by NVIDIA called CUDA throughout this thesis. CUDA exposes the GPU processing power in the C programming language and can be integrated in existing applications with ease.

Our work presented in this thesis is based on two different applications. The first application is the client application of Einstein@Home, a distributed computing project similar to SETI@Home. We integrate CUDA into the Einstein@Home application with the goal to provide a better performance than the generic CPU implementation. Our first approach with the existing data structure failed, due to the different requirements regarding data structure design of both CPU and GPU. A redesigned data structure increased the performances to about the same level of the CPU-based version. A further optimized CUDA-based solution provides more than three times the performance of the original implementation. Our second application is OpenSteer, which is a game-like artificial intelligence simulation. The OpenSteer version we use is based on our previous work [Bre08] and already supports CUDA. We develop two kind of spatial data structures to effectively solve the so called *k-nearest neighbors*' problem and integrate them in OpenSteer to demonstrate their performance. Our new data structures clearly outperform the brute force algorithm previously used by OpenSteer. One data structure is well known from the CPU realm, whereas the other one is a novel design especially created for CUDA usage.

1.1 Related work

The distributed computing project Folding@Home [Fol07] provides a client application supporting GPUs of both AMD and NVIDIA. The Folding@Home client uses BrookGPU, a GPGPU programming system developed by Ian Buck et.al.. BrookGPU provides so called stream processing on GPUs, which uses large arrays to store data and executes the same operation on every data element of the arrays. Details regarding Brook can be found in [BFH⁺04]. At the time of writing there is no other distributed computing project using GPUs.

Spatial data structures are a well known concept and used to effectively solve N-body simulations, collisions detection or the k-nearest neighbors problem. The brute force approach not using any spatial data structures has a complexity of $O(n^2)$ and therefore requires a high amount of processing power. Nyland et.al. describe a highly optimized brute force solution to an N-body simulation using CUDA in [NHP07]. This solution can simulate 16,384 bodies at about 38 simulation steps per second on a GeForce 8800 GTX. We have demonstrated in our previous work [Bre08] that a GPU-based simulation of OpenSteer using a brute force approach can run at about 21 simulations step per second when simulating 16,384 agent on a GeForce 8800 GTS. Nyland et.al. also propose that based on the performance they have demonstrated with their brute force solution, an octree-based implementation should provide about 2 simulation steps per second when simulating 2^{18} bodies. However, they have not implemented any solutions using spatial data structures. NVIDIA provides a particles demo application demonstrating simple collisions, which uses a uniform grid and provides more than 100 frames-per-second with 65,000 particles running on a GeForce 8800 GT. The application uses grid cells of the size of the particles and offloads the sorting required to build up their grid data structure to the GPU. The application is shipping with the beta release of CUDA 2.0 and an overview

can be found in [Gre08].

PSCrowd by Reynolds [Rey06] uses the Cell processor of the Playstation 3 to simulate up to 15,000 agents in a rather simple crowd simulation. Reynolds uses static lattice cells to divide the world in such a way that the cells have a common maximum number of agents. The cells of one agent are scheduled to the SPUs of the Cell processor. Wirz et.al. [WKL08] experimented with different spatial data structures in the same OpenSteer scenario as we do in this thesis. The best performing data structure is the so called cell array with binary search, which has been proposed by Mauch in [Mau03]. In an n-dimensional world, this data structure uses n-1 dimensions to create a grid and the nth dimension to sort the entries of a grid cell. The sorting inside the cells is used to optimize the search inside a cell. This data structure provides a performance increase of about factor 2.5 compared to the brute force approach when both are running at a quad-core CPU system. The simulation can simulate 8,200 agents at 30fps.

1.2 Structure of this thesis

Our thesis is divided into three parts. The first part introduces the GPGPU programming system used throughout this thesis. We use both NVIDIA's CUDA and CuPP [Bre07], which is a framework to ease development (chapter 2). The second part of our thesis gives a general overview of distributed computing and Einstein@Home (chapter 3) and describes our work with the Einstein@Home application (chapter 4). The third part discusses our experience with designing new data structures for OpenSteer. OpenSteer in general and its demo application are described in chapter 5. The next two chapters describe our two different data structure concepts. The so called static grid is described in chapter 6, whereas chapter 7 introduces the dynamic grid.

GPGPU programming system

CHAPTER 2

CUDA

This chapter is based on chapter 2 of our previous work [Bre08], but has been updated to the latest CUDA version and adopted to the work we present in this thesis.

GPGPU development cannot be done without any special programming system. The system we used throughout this thesis consists of CUDA version 1.1, which was released by NVIDIA in November of 2007, and CuPP, which is a framework increasing the usability of CUDA [Bre07]. CUDA is the first GPGPU programming systems offering high-level access to the GPUs developed by NVIDIA. It consists of both hardware and a software model allowing the execution of computations on a GPU in a data-parallel fashion. We describe both models. Afterwards we discuss the performance of CUDA, focusing on the performance-critical parts. We end the CUDA discussion with a comparison between the known CPU programming concept and CUDA. In the end of the chapter we give a brief overview of CuPP. The information we provide in this chapters is based on the CUDA programming handbook [NVI07b], if not explicitly stated otherwise.

2.1 Hardware model

CUDA requires a GPU that is based on NVIDIA's so-called G80 architecture or one of its successors. Unlike CPUs, which are designed for high sequential performance, GPUs are designed to execute a high amount of data-parallel work. This fundamental difference is reflected in both the memory system and the way instructions are executed, as we discuss in the next sections.

GPUs are constructed in a superscalar fashion. A GPU is implemented as an aggregation of multiple so-called *multiprocessors*, which consist of a number of SIMD ALUs. A single ALU is called *processor*. According to the SIMD concept, every processor within a multiprocessor must execute the same instruction at the same time, only the data may vary. We call this build-up, which is shown in figure 2.1, *execution hierarchy*.

Figure 2.1 also shows that each level of the execution hierarchy has a corresponding memory type in what we call the *memory hierarchy*. Each processor has access to local

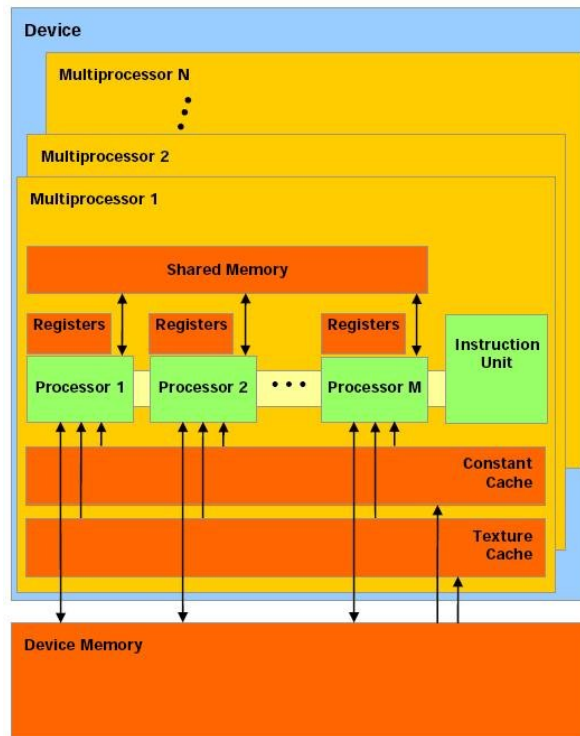


Figure 2.1: The CUDA hardware model: A device consists of a number of multiprocessors, which themselves consist of a number of processors. Correspondingly, there are different types of memory [NVI07b].

32-bit *registers*. On multiprocessor level, so-called *shared memory* is available where all processors of a multiprocessor have read/write access to. Additional *device memory* is available to all processors of the device for read/write access. Furthermore, so-called texture and constant caches are available on every multiprocessor. Both cache types cache special read-only parts of the device memory, called texture and constant memory respectively. Texture memory is not used throughout this thesis.

2.2 Software model

The software model of CUDA offers the GPU as data-parallel coprocessor to the CPU. In the CUDA context, the GPU is called *device*, whereas the CPU is called *host*. The device can only access the memory located on the device itself.

A function executed on the device is called *kernel*. Such a kernel is executed in the single program multiple data (SPMD) model, meaning that a user-configured number of threads execute the same program.

A user-defined number of threads (≤ 512) are batched together in so-called *thread blocks*. All threads within the same block can be synchronized by a barrier-like construct. The different blocks are organized in a *grid*. A grid can consist of up to 2^{32} blocks,

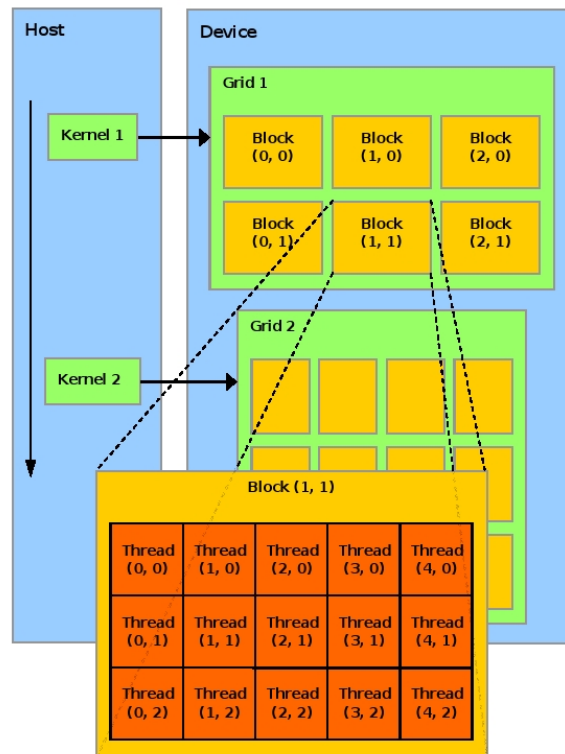


Figure 2.2: The CUDA execution model: A number of threads are batched together in blocks, which are again batched together in a grid [NVI07b].

resulting in a total of 2^{41} threads. It is not possible to synchronize blocks within a grid. If synchronization is required between all threads, the work has to be split into two separate kernels, since multiple kernels are not executed in parallel.

Threads within a thread block can be addressed by 1-, 2- or 3-dimensional indexes. Thread blocks within a grid can be addressed by 1- or 2-dimensional indexes. A thread is thereby unambiguously identified by its block-local thread index and its block index. Figure 2.2 gives a graphical overview of this concept. The addressing scheme used influences only the addressing itself, not the runtime behavior of an application. Therefore 2- or 3-dimensional addressing is mostly used to simplify the mapping of data elements to threads – e.g. see the matrix-vector multiplication provided by NVIDIA [NVI08]. When requiring more than 2^{16} thread blocks, 2-dimensional block-indexes must be used.

A grid is executed on the device by scheduling thread blocks onto the multiprocessors. Thereby, each block is mapped to one multiprocessor. Multiple thread blocks can be mapped onto the same multiprocessor and are then executed concurrently. If multiple blocks are mapped onto a multiprocessor, its resources, such as registers and shared memory, are split among the mapped thread blocks. This limits the amount of thread blocks that can be mapped onto the same multiprocessor. We call *occupancy* the ratio of active threads on a multiprocessor to the maximum number of threads supported by a

software model (memory address space)	hardware model (memory type)	access by	
		device	host
thread local	registers & device	read & write	no
shared	shared	read & write	no
global	device	read & write	read & write
constant	device	read (cached)	read & write

Table 2.1: An overview of the mapping between the hardware model memory types and the software model types including accessibility by device and host.

multiprocessor¹. A block stays on a multiprocessor until it has completed the execution of its kernel.

Thread blocks are split into SIMD groups called *warps* when executed on the device. Every warp contains the same amount of threads. The number of threads within a warp is defined by a hardware-based constant, called *warp size*. Warps are executed by scheduling them on the processors of a multiprocessor, so that a warp is executed in SIMD fashion. On the currently available hardware, the warp size has a value of 32, whereas 8 processors are available in each multiprocessor. Therefore a warp requires at least 4 clock cycles to execute an instruction. Why a factor of 4 was chosen is not known publicly, but it is expected to be used to reduce the required instruction throughput of the processors, since a new instruction is thereby only required every 4th clock cycle.

The grid and block sizes are defined for every kernel invocation and can differ even if the same kernel is executed multiple times. A kernel invocation does not block the host, so the host and the device can execute code in parallel. Most of the time, the device and the host are not synchronized explicitly – even though this is possible. Synchronization is done implicitly when data is read from or written to memory on the device. Device memory can only be accessed by the host if no kernel is active, so accessing device memory blocks the host until no kernel is executed².

The memory model of the CUDA software model differs slightly from the memory hierarchy discussed in section 2.1. Threads on the device have their own local memory address space to work with. Additional threads within the same block have access to a block-local shared memory address space. All threads in the grid have access to global memory address space and read-only access to constant memory address spaces. Accesses done to constant memory are cached at multiprocessor level; the other named memory address spaces are not cached.

The different memory address spaces are implemented as shown in table 2.1. Shared and global memory address space implementation is done by their direct counterpart in the hardware. Local memory address space is implemented by using both registers and device memory automatically allocated by the compiler. Device memory is used when the number of registers exceeds a threshold. Constant memory uses device memory.

¹Current hardware has a fixed limit of 768 threads per multiprocessor.

²With the release of CUDA 1.1, this is no longer true for new GPUs. Memory can be transferred from and to the device while a kernel is active when page-locked memory is used for host memory. The hardware used for this thesis does not provide this functionality.

Instruction	Cost (clock cycles per warp)
FADD, FMUL, FMAD, IADD	4
bitwise operations, compare, min, max	4
reciprocal, reciprocal square root	16
accessing registers	0
accessing shared memory	≥ 4
reading from device memory	400 - 600
reading from constant memory	≥ 0 (cached) 400 - 600 (else)
synchronizing all threads within a block	4 + possible waiting time

Table 2.2: An overview of the instruction costs on the G80 architecture

2.3 Performance

The number of clock cycles required by some instructions, can be seen in table 2.2. To note some uncommon facts:

- Synchronizing all threads within a thread block has almost the same cost as an addition.
- Accessing shared memory or registers comes at almost no cost.
- Reading from device memory costs an order of magnitude more than any other instruction.

The hardware tries to hide the cost of reading from device memory by switching between warps. How effective this can be done by the device, depends on the occupancy of the device. A high occupancy allows the device to hide the cost better, whereas a low occupancy provides almost no chance to hide the cost of reading from device memory. Therefore increasing the occupancy of a kernel also increases its performance, when the performance of the kernel is limited by device memory accesses. Nonetheless reading from device memory is expensive, so reading data from global memory should be minimized, e.g. by manually caching global memory in shared memory. Unlike reading from device memory, writing to device memory requires less clock cycles and should be considered a *fire-and-forget* instruction. The processor does not wait until memory has been written but only forwards the instruction to a special memory writing unit for execution³. Accesses to constant memory are cached. When a cache miss takes place, the cost is identical to an access to global memory. If the accessed data element is cached, the element can be accessed without any cost in most cases.

As said in section 2.2, warps are SIMD blocks requiring all threads within the warp to execute the same instruction. This is problematic when considering any type of control

³Writing to device memory itself is not discussed in [NVI07b], but this is a widely accepted fact by the CUDA developer community, see e.g. <http://forums.nvidia.com/index.php?showtopic=48687>.

flow instructions (`if`, `switch`, `for`, `do`, `while`), since it would require uniform control flow across all threads within a warp. CUDA offers a way to handle this problem automatically, so the developer is not forced to have uniform control flow across multiple threads. Branches are serialized and executed by predication.

When multiple execution paths are to be executed by one warp – meaning the control flow diverges – the execution paths are serialized. The different execution paths are then executed one after another. Therefore serialization increases the number of instructions executed by the warp, which effectively reduces the performance. Code serialization is done by using predication. Every instruction gets prefixed with a predicate that defines if the instruction should be executed⁴. How much branching can affect the performance is almost impossible to predict in a general case. When the warp does not diverge, only the control flow instruction itself is executed.

2.4 Why a GPU is no CPU

GPUs are designed for executing a high amount of data-parallel work – NVIDIA suggests having at least 6,400 threads on the current hardware, whereas 64,000 to 256,000 threads are expected to run on the upcoming hardware generations. Unfortunately, massive parallelism comes at the cost of both missing compute capacities and bad performance at some scenarios.

- Current GPUs are not capable of issuing any function calls in a kernel. This limitation can be overcome by compilers by inlining all function calls – but only if no recursions are used. Recursion can be implemented by manually managing a function stack. But this is only possible if the developer knows the amount of data required by the function stack, since dynamic memory allocation is also not possible. Again, this limitation may be weakened by the developer – possibly by implementing a memory pool⁵ – but currently no work in this area is available to public.
- GPUs are not only designed for a high amount of data-parallel work, but also optimized for work with a high arithmetic intensity. Device memory accesses are more expensive than most calculations. GPUs perform poorly in scenarios with low arithmetic intensity or a low level of parallelism.

These two facts show that unlike currently used CPUs, GPUs are specific and may even be useless at some scenarios. Writing fast GPU program requires

- a high level of arithmetic intensity – meaning much more arithmetic instructions than memory access instructions.

⁴Actually, the predicate does not prevent the instruction from being executed in most cases, but only prevents the result from being written to memory.

⁵Preallocating a high amount of memory and hand it out at runtime to simulate dynamic memory allocation.

- a high degree of parallelism – so all processors of the device can be utilized.
- predictable memory accesses, so the complex memory hierarchy can be utilized as good as possible, e.g. by manually caching memory accesses in shared memory.

The performance of a CPU program is not affected that badly by memory accesses or by a low degree of parallelism. The cost of memory accesses is rather low due to the different architecture of modern CPUs and GPUs – e.g. the usage of caches. Since current multicore CPUs still offer high sequential performance and the number of cores available is small as compared to the number of processors available on a GPU⁶, less independent work is required for good performance.

2.5 Programming the device

In this section we detail how the device is programmed with CUDA. We first give a brief overview of CUDA itself, followed by an introduction to the C++ CUDA framework called CuPP.

CUDA offers a C dialect and three libraries to program the device. The libraries are C libraries and can be used in every program written in C. The Common Runtime library offers new data types, like two-, three- and four-dimensional mathematical vectors, and new mathematical functions. The Host Runtime library mostly provides device and memory management, including functions to allocate and free device memory and call a kernel. The Device Runtime library can only be used by the device, and provides a barrier like synchronization function for synchronizing all threads of a thread block and other device specific functionality. The C dialect of CUDA supports new keywords, which are used to define the memory type a variable is stored in and the hardware a function is execute on. For example, the keyword `__shared__` is prefixed in front of a standard C variable definition to have the defined variable stored in shared memory, functions that can be called from the host and executes on the device (a kernel) is prefixed with `__global__`. A detailed description of how the device can be programmed with CUDA can be found in [NVI07b] or [Bre08].

The CuPP framework builds up on the CUDA libraries and offers some enhanced techniques to ease the development of CUDA applications. We focus on the memory management techniques, a detail description of CuPP can be found in [Bre07]. When programming with CUDA, the developer has to copy memory regions explicitly to and from device memory by using a `memcpy()` like function. CuPP offers a high level approach of memory management by providing both, a ready to use STL like vector data structure called `cupp::vector`, which automatically transfers data from or to the device whenever needed, and an infrastructure to develop data structure with the same functionality. The later is referred to as the *CuPP host-/devicetype bindings*. It allows the developer to specify two independent data structures, of which one gets used at the host (called

⁶The hardware used for this thesis offers 96 processors.

hosttype) and another gets used at the device (called *devicetype*). The developer must provide functions to transform one type into the other.

2.6 Hardware used throughout this thesis

Given below the hardware used for all measurements done throughout this thesis.

- CPU: AMD Athlon 64 3700+
- GPU: GeForce 8800 GTS - 640 MB
- RAM: 3 GB

The GPU offers 12 multiprocessors each offering 8 processors, which results in a total of 96 processors. The GPU is running at 500 MHz, whereas the processors itself are running at 1200 MHz. The CPU is a single core CPU running at 2200 MHz.

Einstein@Home

Einstein@Home Background

Einstein@Home is one of the largest distributed computing projects worldwide. We describe our experience with the Einstein@Home client application in this part of the thesis. At first we introduce BOINC, the distributed computing platform used by Einstein@Home and give an overview of Einstein@Home itself afterwards. In the next chapter we discuss our experience with the development of a CUDA-based Einstein@Home application.

3.1 BOINC

In contrast to the early 90s, the world's processing power is no longer concentrated in supercomputing centers, but spread around the globe in hundreds of millions of computers or game consoles owned by the public. We call using these resources for computations *public-resource computing*. Since the mid-90s public-resource computing projects are emerging for tasks requiring a high amount of processing power, e.g. GIMPS¹ or SETI@Home.

Projects using public-resource computing are mostly run by small academic research groups with limited resources. Participants of these projects are mostly single individuals owning a PC that like to do a *good thing* with unused processing power. The PCs use a wide variety of different operation systems, are not always connected to the Internet and regularly turned off. The participants themselves only receive small “incentives” for supporting a project like screensaver or points, so called *credits* representing the amount of work they have done to support a project.

BOINC (Berkeley Open Infrastructure for Network Computing) is an open source distributed computing platform originally designed by Anderson [And04] to ease the creation of public-resource computing projects. The goal is to allow even researchers with only moderate computer skills to run a project on their own. BOINC is build up as a client-server system. The organization responsible for a project runs multiple servers,

¹The Great Internet Mersenne Prime Search

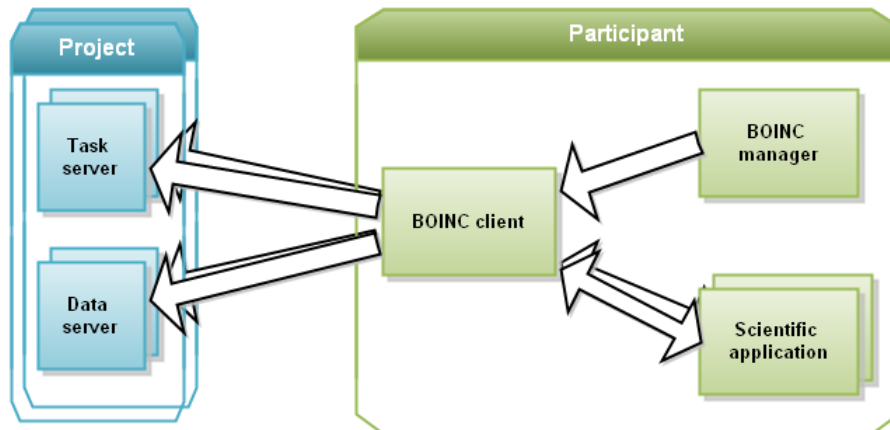


Figure 3.1: A brief overview of the BOINC architecture.

which schedule the work to be done among the participants and stores the calculated results. The participants run at least two different applications on their PC: A scientific application, which calculates results for a project, and the so called *BOINC client*, which manages the scientific application and contacts the project servers whenever needed, e.g. when new work for the scientific application is required. Figure 3.1 gives a brief overview of the BOINC platform. We detail the basic parts next.

Task server The task server schedules the work of a project. In most cases the work is divided in so called *work units* that are send out to a participant. A work unit represents an independent fraction of the overall computation, which can be calculated without communicating to a project server or another participant.

Data server The data server handles result uploads.

BOINC client The BOINC client performs all network communication with the project servers, e.g. downloads of work units, and schedules the scientific applications. An application is scheduled by having a work unit and CPU time associated to it, so it can be executed. The client supports multiple projects.

BOINC manager The BOINC manager is a graphical frontend to the BOINC client. It offers a spreadsheet-type overview of the status of the BOINC client and the scientific applications.

Scientific application The scientific applications are the applications executing the scientific calculations for a specific project. An application is executed by the BOINC client with a low priority, so it only uses free CPU cycles.

This overview of the BOINC platform is not complete, for example the project side of the BOINC platform also offers a validator to check if an uploaded result is correct. The interested reader may find more information in [And04] and [BOI08].

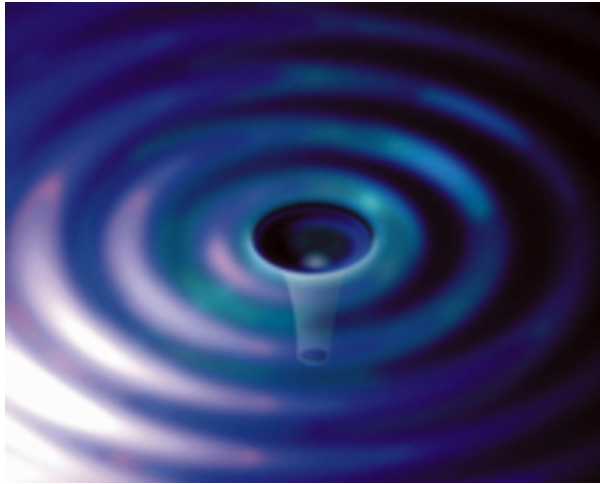


Figure 3.2: An artist rendering of gravitational waves [All07].

3.2 Einstein@Home

Einstein@Home is a distributed computing project using BOINC searching for so called gravitational waves emitted from particular stars (*pulsars*). Several gravitation wave detectors are run under the auspices of the LIGO Scientific Collaboration (LSC) to detect gravitational waves from different sources. Finding a gravitational wave in a recording of a detector requires a high amount of processing power, as the exact waveform is not known. A detailed description of the technique used to find a gravitational wave can be found in [All07].

Einstein@Home had been supported by about 200,000 participants with 470,000 computers since its start. At the time of writing about 80,000 computers are active and provide a 120 TFLOPS of processing power. 120 TFLOPS would be enough to enter the top 10 list of the supercomputers listed at Top500.org [Top07]. To give an example, the system EKA stationed at the Computational Research Laboratories, India offers about 118 TFLOPS and is currently placed at the fourth position at [Top07]².

²The comparison is done by comparing the TFLOPS of Einstein@Home with the R_{max} value generated by the Linpack benchmark. Both values should therefore not be compared directly, but provide a rough estimation of the resources used by Einstein@Home.

CHAPTER 4

Einstein@Home CUDA Implementation

Einstein@Home relies on the resources made available by public, so a GPU-based client could provide the project with currently unused resources. In this chapter we describe our experience with developing such an application. We first introduce the software architecture of the existing CPU-based application and discuss our approach of how we integrated CUDA. The data dependencies of the calculations done by the Einstein@Home application are described afterwards. Based on these results, we develop a first CUDA-based version. Due to an inappropriate data structure design, the performance of this application is slower than the CPU-based one. Throughout section 4.1, we develop different new versions to solve the problems of the previous implementation. The final version is significantly faster than the existing CPU version. We end this chapter with an outlook at possible improvements to the software we developed and problems regarding BOINC and CUDA in general.

4.1 CUDA integration

The Einstein@Home application is built up from four libraries. Two libraries – the GNU Scientific Library [Gou03] and Fastest Fourier Transform in the West [FJ05] – are not used by the code we discuss in this thesis and therefore not described any further. We first give a brief overview of the other two libraries next.

LAL LAL is the abbreviation for the LSC Algorithm Library¹. LAL contains most of the scientific calculations used by the Einstein@Home application, including the so called *F-statistics* we discuss in the sections 4.1.2 and 4.2.

¹To be precise, LSC is again an abbreviation, meaning LIGO Scientific Collaboration, whereas LIGO stands for Laser Interferometer Gravitational Wave Observatory. We refer to the library as LAL.

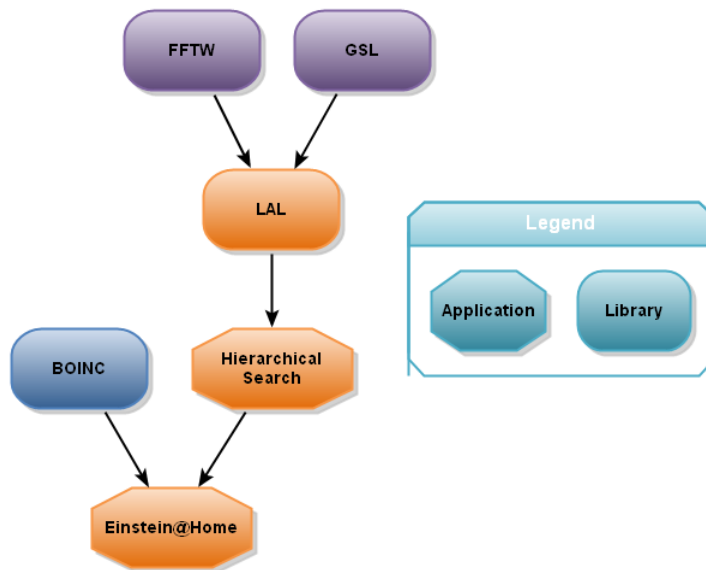


Figure 4.1: Overview of the Einstein@Home application.

BOINC As described in the last chapter, BOINC is the distributed computing platform used by Einstein@Home. The BOINC library is mainly used to handle the communication with the BOINC client.

The source code of the Einstein@Home application itself is part of a collection of data analysis programs called *LALApps*. The application is written in C. LALApps contains a version of the Einstein@Home application called *HierarchicalSearch*, which can be compiled and run without BOINC. Figure 4.1 gives a graphical overview of the software architecture. We describe the CUDA integration based on HierarchicalSearch, as only minor modifications are required to adopt our work to the overall Einstein@Home application².

4.1.1 Design goals

Prior creating a CUDA-based version of HierarchicalSearch, we set two design goals:

- We want to obtain the maintainability of the program for everyone with only minor knowledge of the CUDA programming system.
- The existing source code should be modified as less as possible in the CUDA-based version, so we can transfer changes done to the original version easily.

Our design goals are motivated by the factor that CUDA is a rather new technology and there are not many developers having experience with it. We believe a strict interface

²Actually, only modifications to the build script itself are required to build the Einstein@Home application.

between host code and the code directly interacting with the device (e.g. device memory management or the code executing on the device itself) is the best way to achieve our design goals. A strict interface has the benefit that the original source code only needs to be modified once at specific parts. For example we could add two function calls, of which one executes all device memory management and the second executes the kernel. In theory we can use the host-/devicetype interface provided by CuPP (see section 2.5), but we decided not to use it, since maintaining this code would require advanced C++ skills.

4.1.2 CUDA integration

We believe the best solution to our goals proposed in the last section is to call LALs science functions at the device, so no special device science code must be maintained. However, this is impossible due to the restrictions discussed in section 2.4 and the way LAL is designed. LAL is a library that must be linked with other binaries, but the code of the device functions must be available at compile time. As a result we cannot use the default LAL library. We could solve this problem and use the same functions at both the host and the device by offering all LAL functions as `inline`-functions, but this approach has two major drawbacks. It will not only trigger the problems resulting from inlining all functions³, but also requires a major redesign of LAL. We detail one problem that requires a redesign next, but further ones are expected as well.

LAL maintains a functions stack at runtime, which is used by the LAL error handling mechanism. If a LAL-error occurs, this system terminates the application and prints out both a human readable error message and the functions stack. Offering an identical behavior at the device is impossible at the time of writing, since one thread cannot stop a kernel. If we consider the exact moment when the application is terminated not important – the program could be stopped after all threads have finished the kernel – maintaining a function stack on the device is complex, as delineate in section 2.4. Additional to pure complexity, the performance of such an implementation is expected to be a problem. A function stack must be maintained for every thread, which requires a high amount of device memory. After kernel execution, the function stacks must be transferred to the host and checked to determine if the program must be stopped.

As we cannot reuse the existing code, reimplementing LAL science code for the device cannot be bypassed. Based on the problems outlined above, we decided to not reimplement the LAL error reporting system and even offer no similar functionality⁴. Our code interacting with the device is implemented in a new library called *fstat@CUDA*. Figure 4.2 gives a graphical overview of the new architecture of our Einstein@Home application using CUDA. We now outline the new parts of the architecture, except for CUDA and

³Inlining every function at the host is discouraged from for several reasons, e.g. it results in large binaries and therefore can result in a high number of instruction cache misses. A detailed discussion of this topic can be found in [BM00].

⁴We like to note that the error reporting system would hardly be used by our device code. The error reporting system is mostly used by the memory management, which must be done by the host anyway. We continued to use LALs error reporting system whenever feasible.

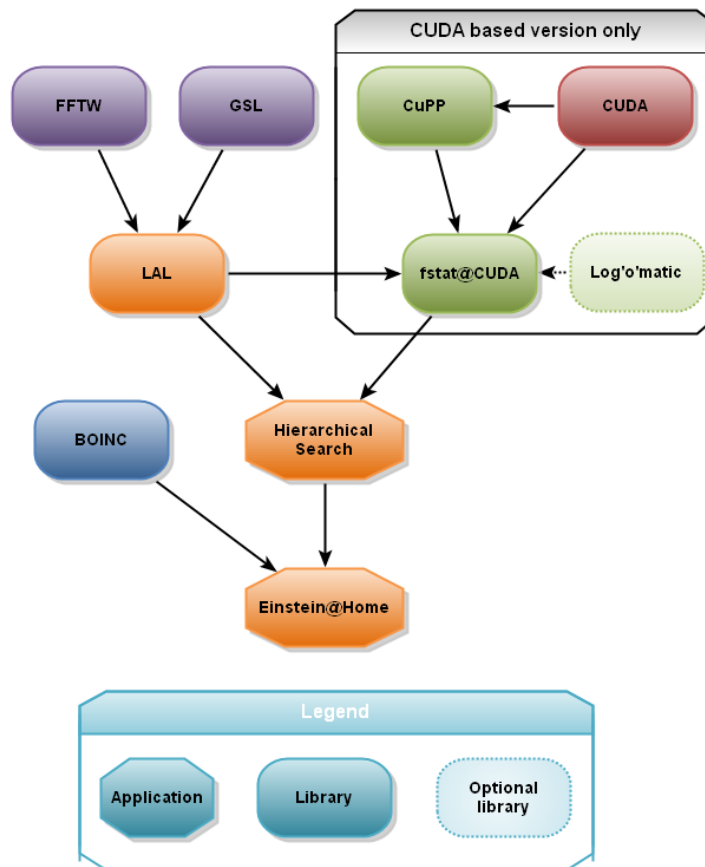


Figure 4.2: Overview of the Einstein@Home application using CUDA.

CuPP, as we have already introduced these two libraries in chapter 2.

log'o'matic Time measurements are done using a self written library called log'o'matic. The library offers an easy to use interface to system specific time functions, similar to the one described by DiLascia [DiL04]. We also provide data structures to collect measurements during a program run. Both data structures and time measurement objects (so called *logs*) are designed to affect the performance of an application as less as possible. The library is currently required to build the application, but since it is only used for time measurements, it could easily be removed from the project. The library itself was only slightly modified as part of the work done for this thesis and is therefore not discussed any further.

fstat@CUDA As said above, the fstat@CUDA library provides all code interacting with the device, including the device science functions and the memory management code⁵. We will detail the development of this library and its functionalities in the

⁵To be precise, this library offers a CUDA F-statistics implementation. See section 4.2 for the reasons why we reimplemented the F-statistics.

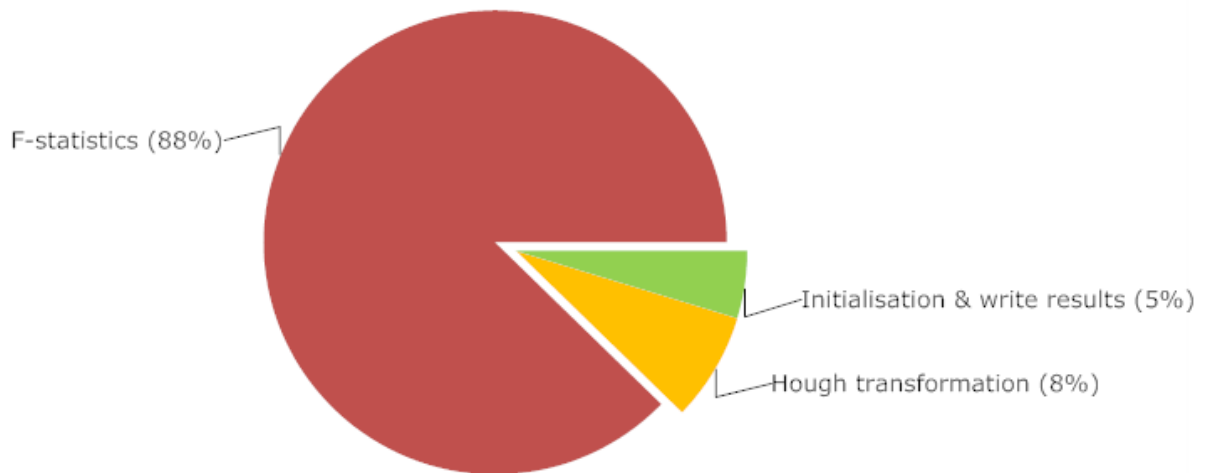


Figure 4.3: Overview of how the runtime is spent by the CPU version.

next section. The library depends on LAL, as it uses LAL data structures to offer the same interface as the existing LAL science functions.

Another major problem with the integration of CUDA is that the calculations of Einstein@Home require double precision to return accurate results, but our hardware only supports single precision. The results calculated with single precision have no scientific value and are therefore of no use for the Einstein@Home project. It could help to rearrange the floating point calculations to get a working single precision version, but as hardware supporting double precision was released in June 2008 we did not work on this problem any further. However, to offer adequate time measurements, we have modified the CPU version to use single precision as well and all comparisons shown in this chapter are done with the modified version.

4.2 Data dependency analysis

In this section we give an overview of the analysis run by the Einstein@Home application, but do not offer a detailed discussion of the science underlying it. We focus our discussion on the data dependencies of the calculation. Data dependencies are important when using CUDA, as we have to work with a high amount of threads on shared memory, for which access cannot be synchronized.

The calculations done by the Einstein@Home application can roughly be divided in two parts. Both parts require about the same amount of runtime regarding a full Einstein@Home work unit. The first part is the so called F-statistics and the second part is a Hough transformation. A full Einstein@Home work unit requires multiple days of CPU time to be calculated at our system, so we decided to use a smaller dataset for all measurements shown in the next sections. Our dataset shows a different runtime behavior, which can be seen in figure 4.3. The Hough transformation requires about

10% of the runtime, where the F-statistics requires about 90%⁶. We therefore decided to concentrate our work the F-statistics, but expect our work to be expressive for a full Einstein@Home work unit as well.

The pseudo code shown in the listings 4.1 and 4.2 gives a rough overview of the parts of the Einstein@Home application we discuss in the next sections. The code is not complete, as the shown functions consist of more than 1000 lines, but we included the required details to see the data dependencies. The code uses `for each` whenever the loop can be carried out in parallel with no or minimal changes to the loop body. `+` is used for all commutative calculations. The parameters of a function denote all values a function depends on, e.g. `f(a, b)` means the result of function `f` only depends on `a` and `b`. The application mostly consists of multiple nested loops, for which all of them loop through different parameters of the data set. The main function (listing 4.1) loops through all sky points and all grid cells for each sky point. Each grid cell again has multiple stacks of input data, which are used to calculate the F-statistics. The F-statistics itself (listing 4.2) is calculated for all frequencies of a frequency band, which differs based on the current stack (see function `ComputeFStatFreqBand`). The frequencies are recorded by two detectors (see function `ComputeFStat`). Every signal found in the recordings is used to calculate the result of the F-statistics (see function `XLALComputeFaFb`).

Listing 4.1: Pseudo code of the HierarchicalSearch application

```

1  int main() {
2      initialisation();
3      // main calculation loop
4      while (skypoint != last_skypoint) {
5          grid_cell = start_grid_cell;
6          while (grid_cell != last_grid_cell) {
7              complex_number f_stat_result[][];
8              int i = 0;
9              for each (stack in stacks) {
10                 // calculate the F-statistics
11                 // for one frequency band
12                 f_stat_result[i++] = ComputeFStatFreqBand(
13                     skypoint, grid_cell, stack);
14             }
15             hough_transformation(f_stat_result);
16             ++grid_cell;
17         }
18         ++skypoint;
19     }
20     write_result_to_file();
}

```

⁶The different runtime behavior is a result from the low star density in our dataset. The F-statistics scales with the size of universe scanned, whereas the Hough transformation works on the found stars.

Listing 4.2: *F*-statistics pseudo code, which is part of LAL

```

1  complex_number[] ComputeFStatFreqBand(skypoint, grid_cell,
    stack) {
2      complex_number result[];
3      int i = 0;
4      // a buffer storing calculation results, which are
5      // constant for the whole frequency band
6      calculation_buffer cfBuffer;
7      for each (frequency in frequency_band(stack) ) {
8          result[i++] = ComputeFStat(skypoint, grid_cell, stack,
                frequency, cfBuffer);
9      }
10     return result;
11 }
12
13 complex_number ComputeFStat(skypoint, grid_cell, stack,
    frequency, cfBuffer) {
14     complex_number result;
15     // check and (if required) update the buffer
16     check_and_update(cfBuffer);
17     for each (detector in detectors) {
18         result += XLALComputeFaFb(skypoint, grid_cell, stack,
                frequency, detector, cfBuffer);
19     }
20     return result;
21 }
22
23 complex_number XLALComputeFaFb(skypoint, grid_cell, stack,
    frequency, detector, cfBuffer) {
24     complex_number result;
25     for each (signal in signals(frequency)) {
26         result += some_calculations(skypoint, grid_cell,
                stack, frequency, detector, signal, cfBuffer);
27     }
28     return normalized(result);
29 }

```

We like to note some important details of the code shown above. The function `ComputeFStatFreqBand` creates a buffer to store values that are constant for one frequency band. It may look uncommon that this buffer is filled in a different function (in our case `ComputeFStat`). There is no reason why the buffer could not be filled in `ComputeFStatFreqBand` in our application, but as these functions are part of the LAL library other programs may exist that benefit of this concept. Figure 4.4 gives an overview of how calculations are performed for one stack. As it can be seen, the number of signals processed in `XLALComputeFaFb` is not constant. It varies based on the frequency, whereas the number of detectors is constant for the complete simulation. The code shown in the

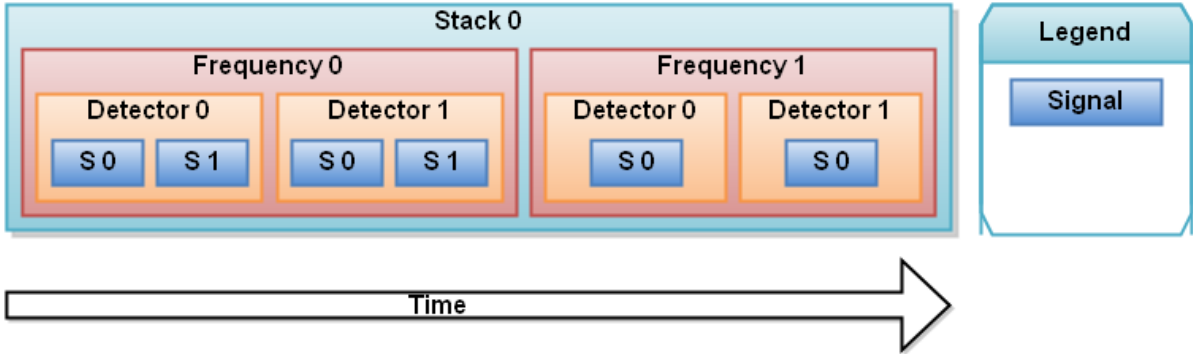


Figure 4.4: Overview of how values are processed per time by the Einstein@Home application for one stack.

listings 4.1 and 4.2 does not include any variable storing the data used during calculation. We discuss these in the next section, but first introduce some variables used throughout the rest of this chapter for cost estimations.

|skypoints| Our dataset uses 45 sky points.

|gridcells| Our dataset has 9 grid cells per sky point.

|stacks| We use 84 stacks to calculate the F-statistics.

|frequencyband| A frequency band consists of about 161 frequencies in our dataset.

|detectors| Currently two detectors are used by Einstein@Home.

|signals| The number of signals in our dataset is about 45.

The naming scheme is inexact, e.g. it does not show the dependencies between *|signals|* and the current frequencies, but since we use the variables only for cost estimation, we expect it be adequate. We further introduce the current *state* of the calculation, as the currently calculated signal, detector, frequency, stack, grid cell and sky point – or more formal speaking: the current state is the tuple of

$$(skypoint, grid_cell, stack, frequency, detector, signal)$$

regarding the variable names used in both listings 4.1 and 4.2. We continue to use these names throughout the rest of this chapter.

4.3 CUDA F-statistics implementation

In this section we describe our experience with the implementation of a CUDA-based F-statistics. We pursue a similar approach as we have in our previous work [Bre08], which includes starting with a single function and then extend the part of the calculation executed at the device. Figure 4.5 shows the performance of the different development steps.

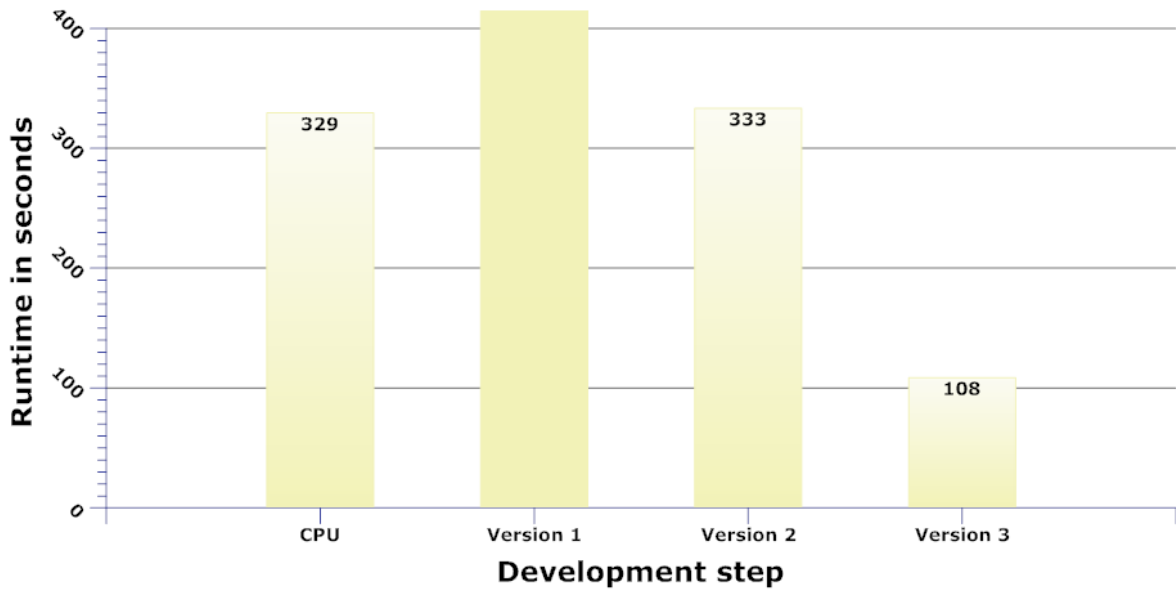


Figure 4.5: Performance overview of the different development versions. We measured one calculation of our example dataset. Version 1 requires about 23,300 seconds,

4.3.1 The first step

We decided to start our work with porting the innermost function of the F-statistics (`XLALComputeFaFb`) to the device. Porting the code to be executed by the device does not require any novel work. Our implementation uses one thread per signal and only one thread block. The number of signals is always less than the maximum number of threads possible in one thread block, so there is no need to use multiple thread blocks. Figure 4.6 gives an overview of how the calculations are performed by this version. We use one thread per signal and therefore we calculate all signals of a detector (and therefore all signals processed by one call of `XLALComputeFaFb`) in parallel. More formal speaking we calculate the states

$$(skypoint, grid_cell, stack, frequency, detector, threadIdx.x)$$

in one kernel call, with `threadIdx.x` being the x-dimension of the block local CUDA thread index. The reduction, which is part of `XLALComputeFaFb` (see listing 4.2, line 26), is implemented by using synchronized access to shared memory, which is possible since all threads are in one thread block. Reductions including multiple thread blocks are hard to achieve with our hardware, as we cannot synchronize threads of different thread blocks nor supports our hardware atomic operations. We did not invest any work in implementing a parallel reduction. The first thread collects all data from the other threads, calculates the result and writes it to global memory. A parallel reductions example is provided by NVIDIA [NVI07a], so we could easily replace our implementation at a later date. The performance of our first version is not limited by the performance of the calculations done at the device, but only by the device memory management done

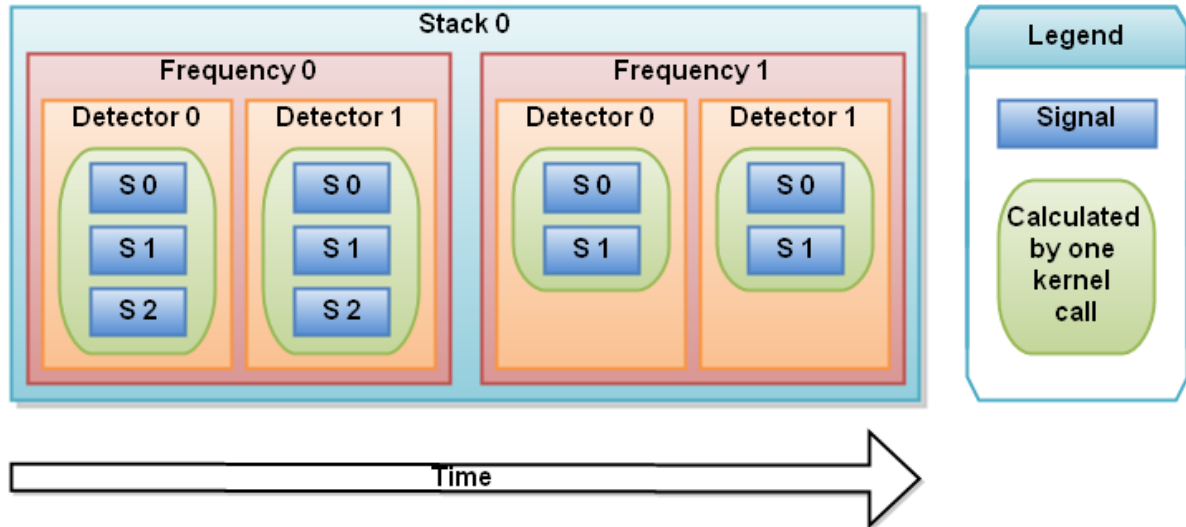


Figure 4.6: Overview of how values are processed per time by the version 1. Values calculated by one kernel call are calculated in parallel.

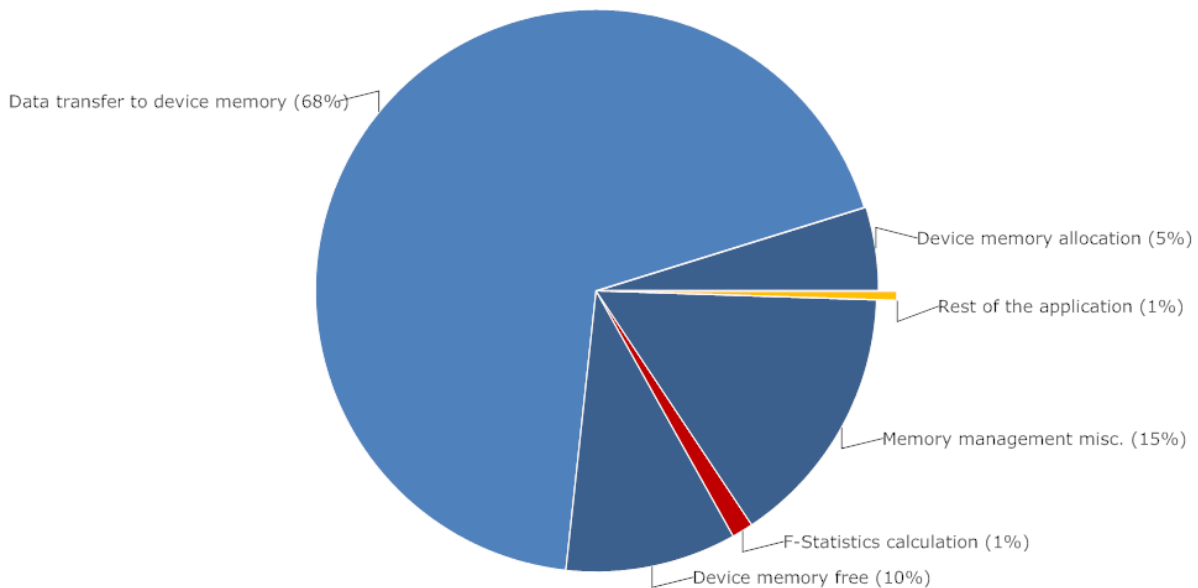


Figure 4.7: Overview of how the runtime is spent by the version 1.

by the host. We refer to this version as *version 1*.

The concept of version 1 was chosen to offer an easy way to start our work and should not provide better performance than the CPU version. Nonetheless we expected at least half the performance of the host-based implementation, but version 1 requires more than 70 times the runtime of the CPU version. Measurements show that almost 95% of the runtime is spent for device memory management and about 70% to copy the data to and from the device. Figure 4.7 gives a graphical overview of how runtime is spent.

Version 1 does not only provide slow performance, but also has a very uncommon problem regarding its memory management. Memory allocations fail irregularly, even though enough free memory is available. We expect this to be a result of memory fragmentation, but NVIDIA offers no access to any data that could proof our assumption and no details of device memory management are discussed in publicly available documentations. Nevertheless our assumption is supported by statements of other CUDA developers having the same problem in similar scenarios⁷.

Analysis of time needed to do the calculations without any memory management shows that the performance of the device is almost identical to that of the CPU version – to be precise, the CPU version requires about 1.05 times the runtime of the device-based one. When considering that we only use one thread block and thereby only one multiprocessor of the device, we can expect better performance of device based implementation compared to a host-based one. This relatively good performance is due to the SIMD nature of the calculations and its high arithmetic intensity. All threads do the same calculation, just on different data and branching is hardly used outside the reduction.

4.3.2 Data structure redesign

As our previous results clearly show that the device memory management is the performance bottleneck, we develop a new data structure in this chapter. In version 1 we only replaced the function `XLALComputeFaFb` and thereby all memory management is also done in this function. By choosing this approach, we must execute the memory management operations $|detectors| * |frequencyband| * |stacks| * |gridcells| * |skypoints|$ -times – except for the values that are constant over the complete calculation. We could move the memory management in one of the functions calling `XLALComputeFaFb`, but this would not reduce the number of memory management operations. The data structure is a conjunction of a high number of rather small memory regions, so a high number of memory management operations is required to copy them one at a time. Figure 4.8 gives an overview of the so called SFT⁸ data, as an example. The high number of independent memory regions is the source of all our memory management problems described in the last section. Performance is diminished, as all memory transfers to or from the device are DMA transfers. DMA transfers are fast at transferring a high amount of data, but perform poorly on a high number of small transfers. Initializing a DMA transfer has a relative high cost, whereas transferring the data itself is cheap. The high amount of memory fragmentation is expected to result from the high amount of small memory areas as well. NVIDIA’s memory management seems not to be designed to perform well at such scenarios.

Our newly developed data structure is designed to directly benefit from the way DMA memory transfer costs are accounted. Despite the requirement to offer a reasonable fast transfer to device memory, no special requirements are set. The data is mostly read once from a known position only depending on the current state of the calculation. The

⁷For example, see <http://forums.nvidia.com/index.php?showtopic=60956>

⁸SFT data contains Fourier transformed recordings of a detector.

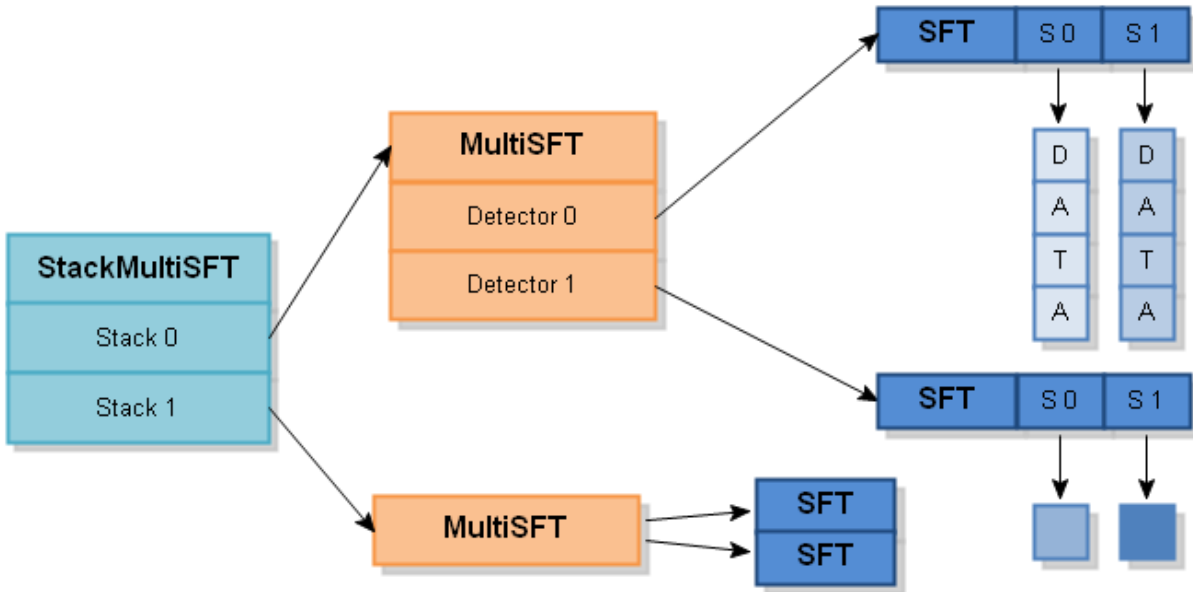


Figure 4.8: Overview of the way SFT data is stored in the CPU version.

requirements of both fast memory transfer and fast access to an element at an arbitrary but known position are satisfied by an array or any data structure that is directly based on it, e.g. vectors. An array is guaranteed to be stored in one contiguous memory block and each element can be directly accessed by using an index, its address or a combination of both – meaning an offset based on an address pointing to one element in the array. Our new data structure is an aggregation of a small number of `cupp::memory1d` objects. A `cupp::memory1d` object represents an array in global memory. The `cupp::memory1d` objects can be grouped in two types. One type stores data used to calculate the result of the F-statistics and is therefore called *data vectors*. The other type stores pointers pointing to elements inside the data vectors – we refer to this type as *offset vectors*, as the pointers stored inside these vectors are used as an offset to access data the way we have just described. The creation of our data structure itself is rather simple. The original small memory regions are all padded into bigger memory regions in an ordered fashion. The elements in the offset vectors are used to identify the old memory regions. Figure 4.9 shows how the data of figure 4.8 is stored in our new data structure.

There are two options of how to integrate the new data structure into our application. The first option is replacing the existing data structure completely and rewriting the memory management. Whereas this approach is expected to offer the best performance, it would also require a rewrite of probably all major parts of the application. A major rewrite is in direct contrast to our design goals laid out in section 4.1.1. The second option is to continue using the old data structure in the main program and only use the new data structure in our code. Thereby we can leave most of the main program untouched, but have to keep both data structures in sync, which should provide less performance than the first option. Nonetheless we decided to choose the second one, as we do not strive for maximal performance but maintainability and our new data structure could be

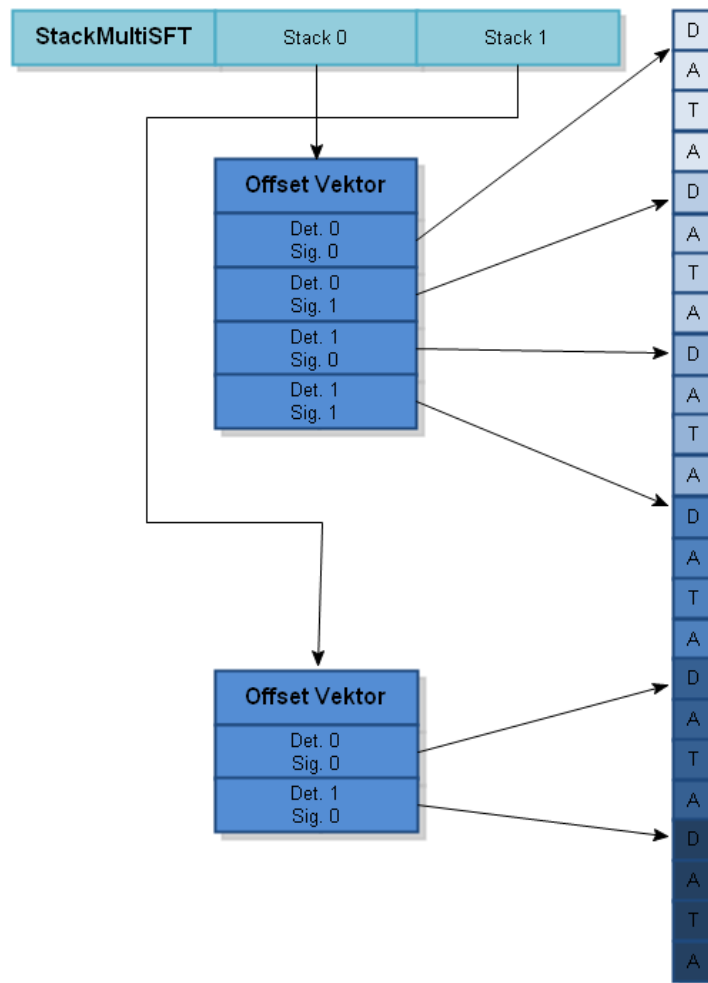


Figure 4.9: Overview of our new designed data structure.

used in the rest of the application at a later date anyway.

To keep our new data structure in sync with the old structure, we added a function called `cuda_prepare`, which is called directly before the grid cell loop (line 6 in listing 4.1). This function copies almost all data to the device – except the data stored in the calculation buffer – and works in three steps:

1. We first copy all data from the old data structure to our new data structure temporarily allocated in host memory. As the same type of data is stored in both structures, we regularly call `memcpy` to copy the data from the small memory blocks into the big one. While we are copying the data, we store the index at which they are located in the new memory block.
2. After we have copied all data into the larger memory block, we copy the memory block to newly allocated device memory.

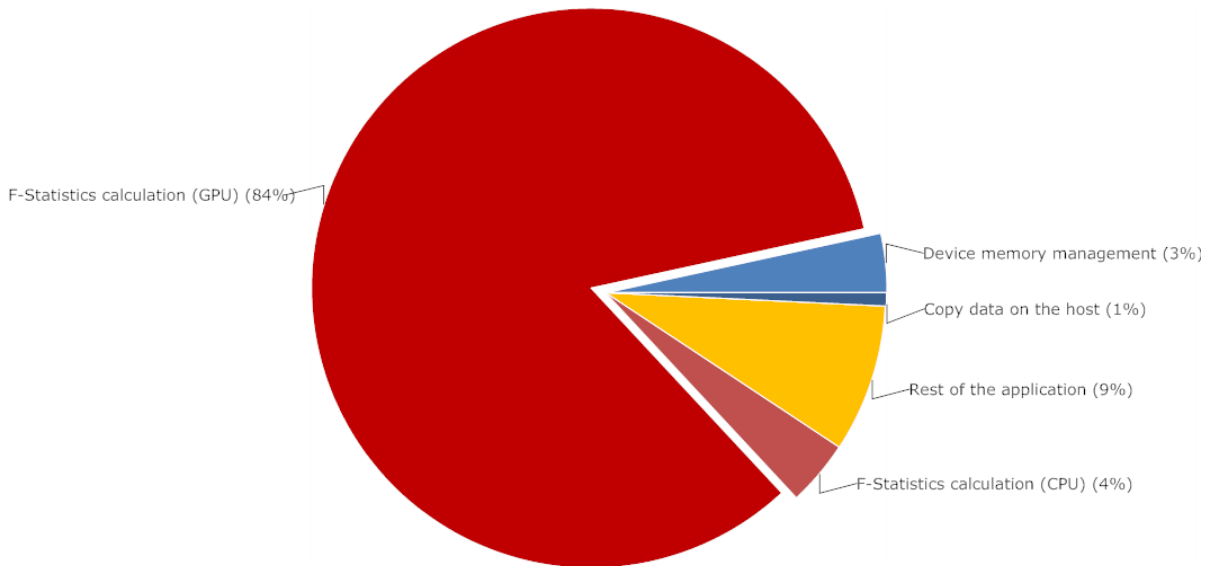


Figure 4.10: Overview of how the runtime is spent by the version 2.

3. We use the indexes stored in the first step and the address of the device memory allocated in the second step to calculate the offset pointers. We store these pointers in the offset vectors described earlier and transfer the data to the device.

As we have just described, the function `cuda_prepare` does not start any calculation on the device, but only creates the data structure. As the data structure must be passed to the kernel, the data structure itself must be made available to the function calling the kernel. We decided not to pass the data structure as a parameter to this function, as it would have changed the signature of the function, which would then differ from the one provided by LAL. Instead we decided to use global variables to store our data structure – even though, this is highly discouraged from in most cases, we believe the benefits outweigh the draw backs in this specific case. To guarantee that the global variable namespace of the application is not *polluted* with our new data structure, we made them available only at file scope. We refer to the HierarchicalSearch version using the new data structure as version 2.

Performance measurements of version 2 show that our new data structure solves our problems described in the section 4.3.1. Memory management is still measurable but no longer the performance bottleneck. Version 2 has about the same performance as the original CPU version, but this comes at no surprise, since the calculations done by the device still require about the same time as that of the CPU version (see section 4.3.1). Figure 4.10 shows, how runtime is spent by this version. As it can be seen, copying the data from the original data structure to our new one hardly requires any time, so we believe working with two data structures is a good choice in this case. The random crashes at allocating device memory seen in version 1 seem to be gone – at least none of our simulation runs done with version 2 or 3 crashed.

4.3.3 Kernel call redesign

As memory management is no longer the performance bottleneck, we decided to change our kernel call concept set in place in the first version to provide better occupancy of the device and thereby improve performance. Up until this version, we repeatedly issue kernel calls with one thread block. We now describe how we replaced these multiple calls with just one kernel call using multiple thread blocks.

Let us first revise the important details of listing 4.2. If we want to calculate the F-statistics for a frequency band, we have to port the functions `ComputeFStatFreqBand` and `ComputeFStat` to the device. `ComputeFStat` contains a reduction of the values calculates by `XLALComputeFaFb` – the function we ported to the device in step one. The results of `ComputeFStat` are written to different memory addresses and afterwards used by the Hough transformation.

As said in section 4.3.1, we use a fairly simple way to calculate the reduction done in `XLALComputeFaFb`, which is only possible because all threads are in the same thread block. Our analysis of the data sets shows that we can continue with this approach. The number of threads required to calculate `XLALComputeFaFb` for all detectors is still small enough to have one thread block calculate them. To ease the readability of our program we use two dimensional thread blocks. The x-dimensions still represents the calculated signal, whereas the y-dimension is now used to represent the detectors. Version 1 calculates the states

$$(skypoint, grid_cell, stack, frequency, detector, threadIdx.x)$$

with one kernel call, whereas our reworked approach calculates

$$(skypoint, grid_cell, stack, frequency, threadIdx.y, threadIdx.x)$$

with one kernel call. Even though this approach uses more threads, it still only uses one thread block per kernel call.

The next step is to use multiple thread blocks to calculate different frequencies – meaning we calculate the states

$$(skypoint, grid_cell, stack, blockIdx.x, threadIdx.y, threadIdx.x)$$

with one kernel call. Figure 4.11 give a graphical overview of the calculations. By concept this can be achieved with relative ease, as the calculations of one frequency are independent from one another and all data required is already transferred to the device. Nonetheless, this concept still results in some administrative work that may not be obvious at first. For example, the number of signals is not constant across all frequencies, but the dimension of all thread blocks is identical. To solve this problem, we scan the dataset for the maximum number of signals found in a frequency of a given frequency band and thereby determine the x-dimension of all thread blocks. This concept results in idle threads for all thread blocks not calculating the maximum number of signals, but our evaluation of the dataset shows only small fractions of idle threads – mostly just 3 threads idle per x-dimension, whereas the maximum found is 10. Another problem resulting from the parallel calculation of multiple frequencies is that the calculation buffer was

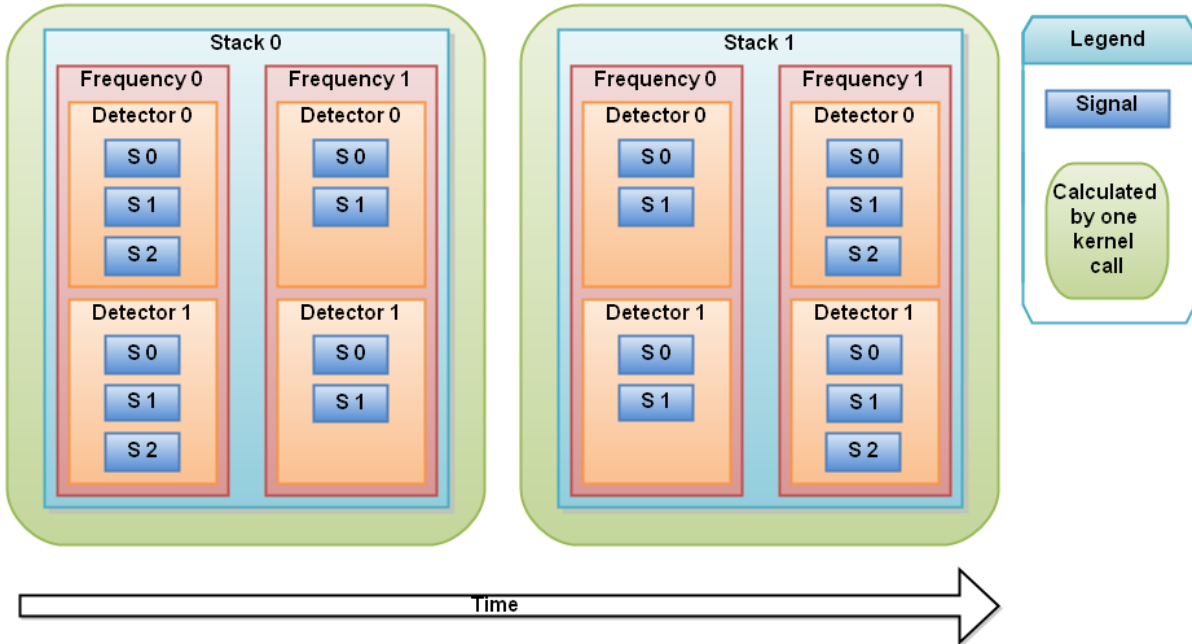


Figure 4.11: Overview of how values are processed per time by version 3. Values calculated by one kernel call are calculated in parallel.

calculated by `ComputeFStat` even though the values are constant for all frequencies. We could have ported these calculations to the device as well and every thread block would recalculate the values stored in the buffer. Recalculating values that are constant offer a complete kernel call $|frequencyband|$ -times reduces the performances, so we decided to continue calculating the buffer on the host and transfer the data to the device. This is our final implementation of the HierarchicalSearch application, which we refer to as *version 3*.

Regarding to the overall performance, this version requires about one third of the runtime to calculate our example dataset compared to the CPU version. When considering only the F-statistics calculations and memory management done by version 3, it requires about one fifths of the time required by the CPU version. Figure 4.12 shows how time is spent by version 3.

4.4 Conclusion

Our work on the Einstein@Home application shows that it is possible to use GPUs to improve the overall performance of applications without the need of a complete redesign. This stays true even if the existing data structures perform poorly when used in conjunction with GPUs. We further demonstrated that with only minimal changes to the application itself – we only modified one function call and added another – most of the application is still maintainable by developers not being aware of anything CUDA related. However, we have to admit that it was not possible to continue using the original science

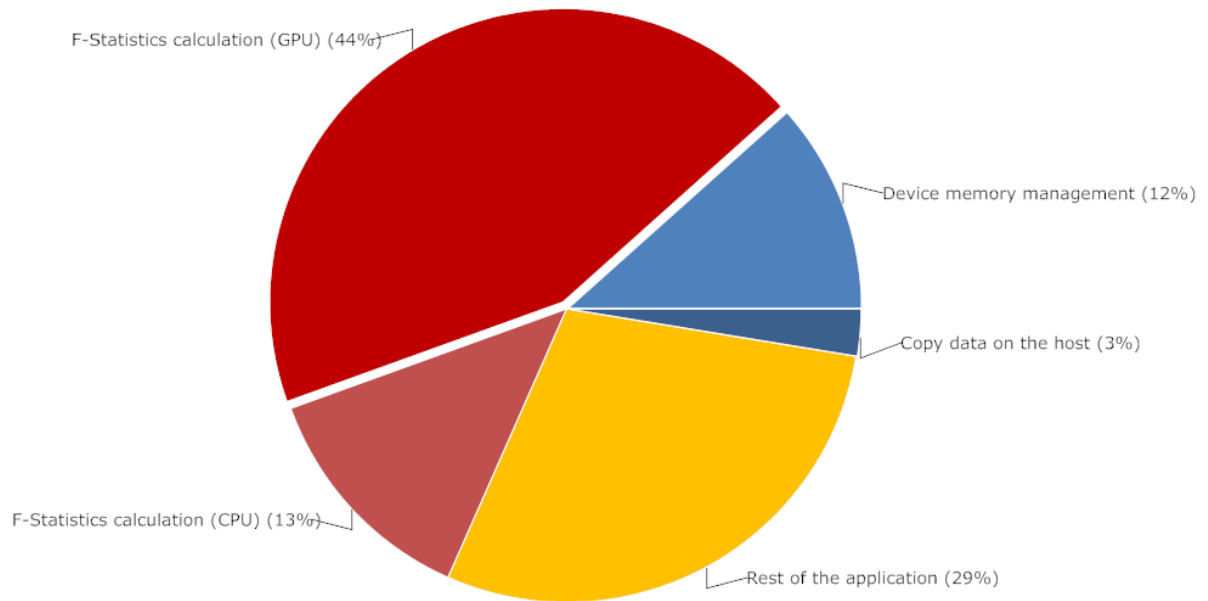


Figure 4.12: Overview of how the runtime is spent by the version 3.

functions. Thereby two different pieces of code exist, which offer the same functionality. Nonetheless we expect that the increase in performance is worth the time needed to be maintaining both codes, whenever the performance increase is required.

Regarding the Einstein@Home project, we have to admit that it cannot directly benefit from our work done as our code only uses single precision. But even if our code would support double precision, our work would still be less attractive due to the low amount of CUDA hardware supporting double precision currently owned by the public. However, we believe the later problems will reduce over time and double precision could be added with relative ease. We also like to note that our approach hardly depends on anything CUDA specific, so it should also work fine on upcoming many-core processors, which are currently in development – e.g. Intel’s so called Larabee architecture [SCS⁺].

The F-statistics code still provides optimization potential, for example it could benefit from using the second dimension of the CUDA grid. We could use it to calculate all frames at the same time – meaning execute

$$(skypoint, grid_cell, blockIdx.y, blockIdx.x, threadIdx.y, threadIdx.x)$$

in one kernel call.

Up until now, we have concentrated all our work on the F-statistics – which only takes about 50% of the runtime – but ignored the Hough transformation. This type of calculation is known to be paralyzeable [HSP⁺07] and a GPU port may perform well. As an alternative, the application could also benefit from the double buffering concept we demonstrated in our previous work [Bre08]. For example, while the device is calculating the F-statistics, the host could calculate a Hough transformation based on the results of the previous F-statistics. Deciding which approach is the best can hardly be decided at

the level of the scientific application and can only be decided when the other parts of the BOINC platform support CUDA as well.

How CUDA should be integrated into BOINC is hard to say without any further research. At the time of writing, highly programmable GPUs are still the minority in the end-user market, but all hardware currently sold by NVIDIA and AMD can be used for GPGPU. The current version of the BOINC client schedules all available CPUs regardless its GPU usage, which is fine as long as no GPU using scientific application is available publicly. As soon as multiple scientific applications are using GPUs, the GPU must be scheduled in some way. We believe scheduling a GPU to an application for the complete runtime is not a good choice in most cases. If an application uses a GPU for the whole time this scheduling is a good choice, but GPUs are not effective in all types of calculations and therefore could idle in parts of the program run. A more dynamic scheduling should provide a better overall performance, e.g. the scientific application could request and free a GPU at runtime. This dynamic scheduling concept should also work fine with multiple GPUs in one system, which may become the standard in the high-end GPU market throughout the next years.

OpenSteer

OpenSteer fundamentals

The so called *k-nearest neighbor* (k-NN) problem appears in a variety of research areas, for example data mining, machine learning and artificial intelligence. The k-nearest neighbor problem is the search for the k-nearest neighbors of a given position in an n-dimensional space. In this part of the thesis we describe our experience with data structure design regarding the k-NN. We concentrate our work on a specific scenario and application, which we introduce in this chapter. However, our solutions presented in the next two chapters are not restricted to the scenario, but apply to the k-NN problem in general.

5.1 Steering and OpenSteer

OpenSteer is a steering library written in C++ by Reynolds in 2002. *Steering* refers to life-like navigation of so called agents. An *agent* is an autonomous character used in games or other interactive media. The navigation of an agent is defined by a so called *steering behavior*, which depends solely on the environment of the agent. Different types of steering behavior exist, e.g. flee from another agent. To demonstrate our new data structures, we use the flocking behavior offered by OpenSteer, which is described by Reynolds in [Rey99]. Roughly speaking, it consists of three basic steering behaviors every agent follows:

Separation: Keep a certain distance to your neighbors.

Cohesion: Move to the center of your neighbors.

Alignment: Move in the same direction with the same speed as your neighbors.

All agents following these behaviors form small groups, in which all agents try to stay together and move in the same direction.

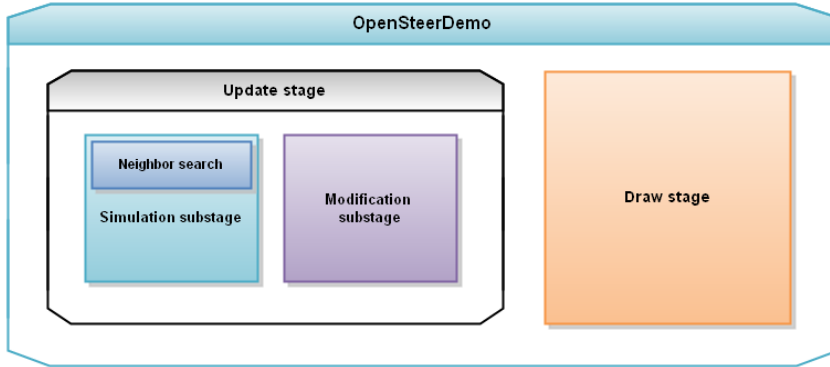


Figure 5.1: An architecture overview of *OpenSteerDemo* [Bre08].

5.2 OpenSteerDemo

OpenSteer offers a demo application, called *OpenSteerDemo*, which we use throughout our work. *OpenSteerDemo* offers different types of plugins demonstrating steering behaviors. The plugin, which demonstrates flocking in a three dimension world, is called *Boids*. *Boids* simulates a group of birds. We use an evolution of the original *OpenSteer Boids* plugin, which is based on our work described in [Bre08]. During our previous work, we developed a GPU-based *Boids* plugin, which does not include obstacles or collusion detection. All agents in our plugin can move freely across a finite spherical world. If one agent leaves the world at one side, it is put back into the world at the diametric opposite of its original position. Our plugin therefore mostly consists of a k-NN and the calculation of the steering behavior described in the last section, so we consider it a good example application to demonstrate our data structures discussed in the next two chapters.

The simulation done by the *Boids* plugin can be divided in two stages. First the states of all agents are calculated and then drawn to the screen. We call the first stage *update stage*, whereas the second one is called *draw stage*. The update stage itself is again divided in two substages. The first substage is called *simulation substage* and includes the steering calculations and the neighbor search. The algorithm used to find the k nearest neighbors of one agent is a fairly simple $O(n)$ algorithm, with n being the number of agents. It is shown in listing 5.1. The *Boids* plugin uses $k = 7$. Since the algorithm is executed for every agent, the problem to find the neighbors of all agents has a complexity of $O(n^2)$. The results of the simulation substage are vectors representing the direction and speed every agent wants to move. These vectors are used in the next substage called *modification substage* to update the position of every agent. The draw stage is executed after the modification substage and draws the new agent positions to screen. Figure 5.1 gives a graphical overview of the architecture of *OpenSteerDemo*.

Our implementation of the *Boids* plugin uses two kernels, each executing one of the substages. Figure 5.2 shows how runtime is spent by this plugin. A detailed description of how the device is used can be found in [Bre08]. During our work presented in this thesis we only modify the simulation substage kernel, as it contains the neighbor search.

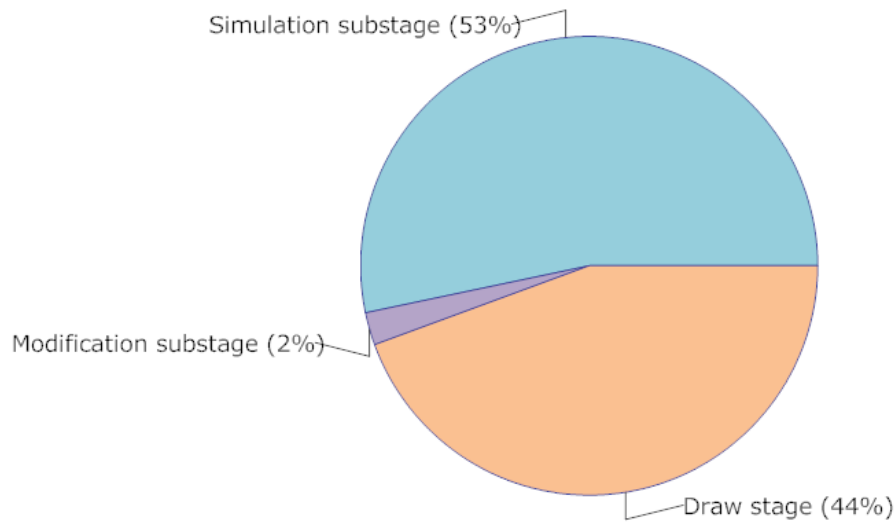


Figure 5.2: Overview of how runtime is spent by the Boids plugin using no spatial data structure.

Listing 5.1: Neighbor search [Bre08]

```

1 Agent [] neighbor_search() {
2   Agent neighbors[k]
3   for each (Agent A in World) {
4     dist := distance(Me, A)
5     if (dist < search radius) {
6       // already the maximum nb of agents found?
7       if (neighbors found < k) {
8         // no
9         add A to neighbors
10      } else {
11        // yes, only store the k nearest
12        B := Agent with the biggest distance in
           neighbors
13        if (distance (Me, B) > distance (Me, A)) {
14          overwrite B in neighbors with A
15        }
16      }
17    }
18  }
19  return neighbors
20 }

```


Static grid neighbor search

A brute force solution to the k-NN or OpenSteers neighbor search is inefficient when simulating a high amount of agents. In this chapter we introduce the *static grid*, our first spatial data structure designed to improve the neighbor search performance. We first outline the overall concept of the data structure and afterwards discuss two variants.

6.1 Static grid concept

A grid subdivides the world into small areas, which we call *cells*. We refer to our grid implementations shown in this chapter as *static grid*. The term static was chosen to distinct this solution to the dynamic grid demonstrated in the next chapter and indicates to way cells are created. The static grid subdivides the world in cubic cells all of them the same size. The number of cells cannot be changed after a grid has been created and is identical for each dimension, so the overall shape of the static grid is a cube as well. The dynamic grid on the other hand creates cells with different sizes dynamically.

In a grid, agents are assigned to cells based on their current position, so one cell contains all the agents that are within its range. A grid can be used to improve the neighbor search performance, as one agent does not need to look at all other agents to find its neighbors, but only at the agents stored in the cells within its search radius. The search inside every cell is done with the brute force algorithm shown in listing 5.1. Listing 6.1 shows the search algorithm when using the static grid.

The performance of this solution depends on how the agents are distributed throughout the world. If all agents are in the same grid cell, the neighbor search performance is identical to the brute force algorithm. Considering the steering behaviors we described in section 5.1, this scenario is unlikely to happen. Our practical experiments show agents are distributed equally throughout the world, so a grid based solution should outperform the brute force approach in most cases.

Listing 6.1: Neighbor search using the static grid

```

1 // get the grid cell the current agent is in
2 const int3 my_grid_cell = grid.get_cell_index(Me.position);
3
4 // how many cell we will check in
5 // each of the six directions
6 const int cpd = floor( r/grid.cell_size + 1 );
7
8 for (int x = -cpd; x<=cpd; ++x) {
9     for (int y = -cpd; y<=cpd; ++y) {
10        for (int z = -cpd; z<=cpd; ++z) {
11
12            const int3 offset = make_int3(x, y, z);
13            const int3 cell = my_grid_index + offset;
14
15            // check if cell is a valid cell, e.g.
16            // cell.x > number_of_cells_per_dimension
17            // is invalid
18            if (!grid.valid_cell (cell)) {
19                continue;
20            }
21
22            // check if the cell is within
23            // the search radius
24            if (!grid.cell_in_range (Me.position, cell)) {
25                continue;
26            }
27
28            const Agent *i    = grid.begin(cell);
29            const Agent *end = grid.end(cell);
30            for (; i < end; ++i) {
31                // brute force neighbor search
32                // for all agents within a cell
33            }
34        }
35    }
36 }

```

6.2 Host managed static grid

Our first static grid implementation uses the host to create the grid and passes it to the device afterwards. We use the CuPP host-/devicetype bindings to work with two different data representations. The creation of the grid is done before the simulation substage is executed and redone for every simulation step – meaning we never update

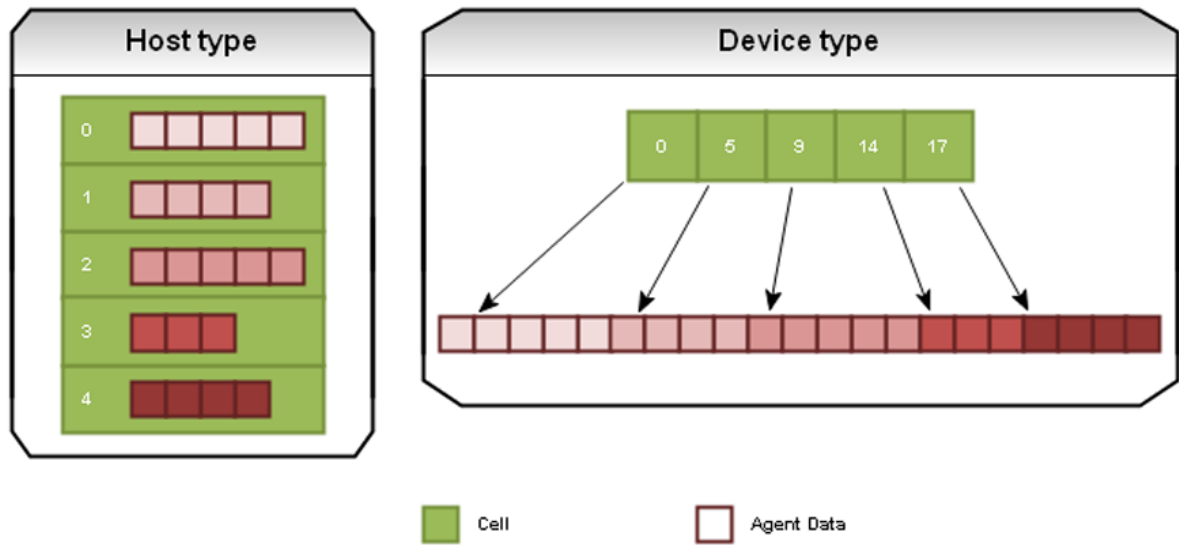


Figure 6.1: Graphical overview of the static grid implementation

the grid, but clear and refill it with data. We choose this way for simplicity, but could hardly think of an efficient way to update an existing grid, as outlined in the next section.

6.2.1 Creation and transfer

The hosttype of the static grid is an aggregation of multiple C++ STL vectors, each vector representing a cell. Cells store references¹ to the agents within the range of a cell. All cell vectors are stored in another vector, so the grid itself is a vector of vectors storing agent references. Figure 6.1 (left side) shows the hosttype. The benefit of this approach is that adding elements to the grid is a $O(1)$ operation. To add an element we must calculate the index of the cell vector and append the element. Appending an element to a C++ STL vector is guaranteed to be a $O(1)$ operation, when no memory reallocation is done. To prevent unneeded memory reallocations, we clear the used C++ STL vectors instead of creating new ones. C++ vectors never free memory already allocated so after the agents are distributed equally throughout the world, the cell vectors hardly need to grow beyond their current size. The index of the cell-vector for an agent currently at `Position` is calculated as shown in listing 6.2.

At first the `x`, `y` and `z` coordinates of a cell are calculated. `Position` varies from `-WorldSize` to `WorldSize` in each dimension, so by increasing each of the coordinates by `WorldSize`, we shift the range to 0 to $2 * \text{WorldSize}$. Further dividing this value by the size of the cells returns the coordinate of the cell. The cell vectors are stored in a one-dimensional vector, so we transform the three-dimensional coordinate into a one dimensional. Creating the hosttype of the static grid for n agents is a $O(n)$ operation, as calculating the index and inserting is a $O(1)$ operation for every agent.

¹The references are implemented by indexes, referring to another vector storing the agents themselves.

Listing 6.2: Calculating the cell-vector index

```

1  int index (Vec3 Position, float WorldSize, float CellSize,
   int CellsPerDimension) {
2      int x = (Position.x + WorldSize) / CellSize;
3      int y = (Position.y + WorldSize) / CellSize;
4      int z = (Position.z + WorldSize) / CellSize;
5      return x +
6          y * CellsPerDimension +
7          z * CellsPerDimension * CellsPerDimension;
8  }

```

As we have demonstrated in section 4.3.2, transferring one large memory block is to be preferred over transferring multiple smaller memory blocks. Based this experience, we designed the devicetype similar to the one used by our Einstein@Home client. The devicetype consists of two independent linear memory blocks. One memory block contains the data of the cell vectors ordered by their index – all data from cell vector 0 is at the start of the memory block, all data from cell vector 1 follows and so on. The other memory block contains the indexes to find the cell vectors within the first memory block – the first value in this memory block is 0, the next is the size of cell vector 0, the following the size of vector 0 plus the size of vector 1 and so on. See figure 6.1 (right side) for a graphical overview of the devicetype. To transfer the hosttype directly to device memory, we would require one memory transfer per cell, whereas the devicetype requires only two memory transfers to transfer all data. Transforming the hosttype into the devicetype is a $O(n)$ operation, as we only have to copy all n agent-references stored in the hosttype into a new continuous memory block. Creating the index memory block is a $O(k)$ operation, with k being the number of grid cells. Creating the devicetype is therefore a $O(n + k)$ operation. Transferring the devicetype to device memory is expected to have the same complexity, but NVIDIA does not detail how memory transfers are executed.

6.2.2 Performance

Our static grid implementation is faster than the brute force approach when using a reasonable amount of grid cells. Figure 6.2 shows the performance of two static grid based simulations. When increasing the number of grid cells, the performance of the simulation also increases. This should come at no surprise, as more grid cells also mean smaller grid cells and thereby less agents within a grid cell. The size of a grid cell can be calculated by

$$CellSize := \frac{2 * WorldSize}{CellsPerDimension}$$

, meaning one grid cell occupies a space of $CellSize^3$.

We can see a huge drop in performance when simulating a world of size 50 with 24^3 grid cells. The performance drop is not directly related to our data structure, but the

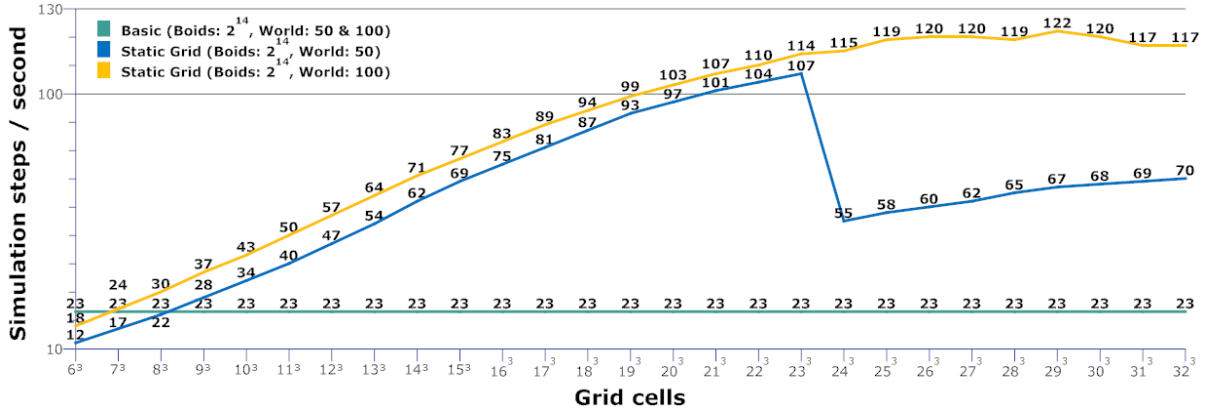


Figure 6.2: Static grid performance with different number of cells.

Boids plugin itself. The simulation uses a neighbor search radius of about 4.24 – the square of the neighbor search radius is 18. When increasing the number of grid cells to 24^3 , the size of one grid cell is smaller than the neighbor search radius, so we have to search through two grid cells (see listing 6.1, `cpd` has the value of 2). When using less than 24^3 grid cells, we only have to search through one cell in every direction. The same performance drop appears when using 48^3 in a world of size 100.

The performance hardly increases when using more than 25^3 grid cells in a world of size 100 and 2^{14} agents. When using such a high amount of grid cells, we have an average of about 1.69 agents per used grid cell². Adding more cells hardly reduce the number of agents checked during the neighbor search, but increases the amount of time required to transform the hosttype into the devicetype. Finding the best performing number of grid cells is hard and we cannot provide an accurate rule of how to find it. Figure 6.3 demonstrates the best performance we could achieve in different settings using the static grid.

Another interesting detail that can be seen in figure 6.2 is the performance of the grid based solution is slower than the basic version when using 6^3 grid cells. This is a result of two important differences. The basic version uses shared memory to cache accesses to global memory, which cannot be done by the static grid based version. In the basic version all agents simulated by one thread block look at the same agents to find their neighbors, but the agents of a thread block in the static grid based version are not guaranteed to have any common data requirements. Agents are put in thread blocks without any order, so the agents located in one grid cell are distributed throughout the CUDA grid and we cannot use shared memory to cache accesses to global memory. The interested reader can find a detailed discussion of the way we used shared memory in the basic version in [Bre08] section 6.2.1. The other difference is that we have to create the grid, which costs performance. Figure 6.4 shows how much time is required by the different parts of the simulation to calculate a given scenario in the best case. Considering

²The world is sphere and our grid is cube, so there are unused grid cells, which lie outside of the simulated world.

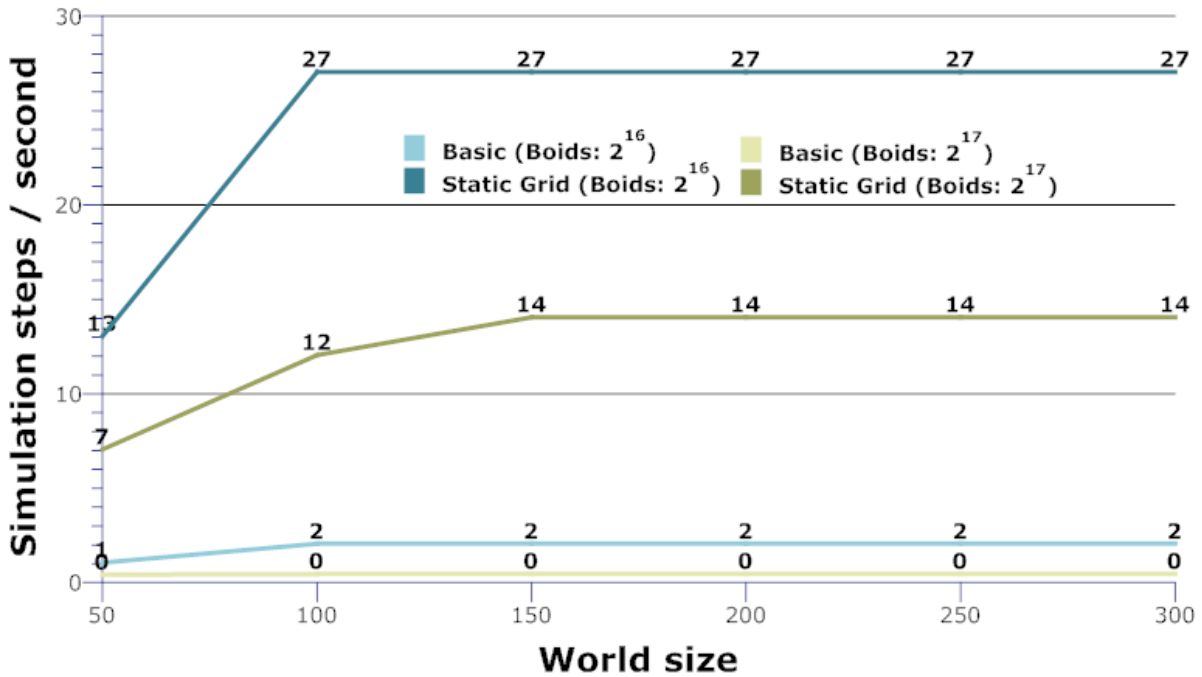


Figure 6.3: The best performance achieved with the static grid implementations.

these measurements, we can see grid management (creation and transfer of the grid) is a factor that should be considered when striving for maximum performance.

In the next two sections of this chapter, we try to further optimize the static grid data structure, whereas the next chapter introduces a data structure that automatically adapts to a given world size and allows the usage of shared memory.

6.2.3 Constant memory support

The static grid version discussed in the last section uses global memory for both memory blocks allocated on the device. We now introduce a modified version of the static grid using constant memory to store the index data. In theory we could use constant memory for both memory blocks, since they are only read by the device and not modified. However, the amount of constant memory available on the device is limited and we cannot allocate enough memory to store a reasonable amount of agent data. When using constant memory only for the index data, the maximum number of cells possible is 25^3 .

The benefit of using constant memory over global memory is that constant memory is cached at every multiprocessor. The first access to an element stored in constant memory has the same costs as an access to global memory. Every following access costs 0 clock cycles per warp in the best case, as shown in table 2.2.

Figure 6.5 demonstrates the performances of the new static grid versions. We can see the performance decreases when using more than 14^3 grid cells and almost no performance increase when using less. The performance decreases as a multiprocessor cannot cache the complete constant memory. One multiprocessor has 8KB of constant cache, but using

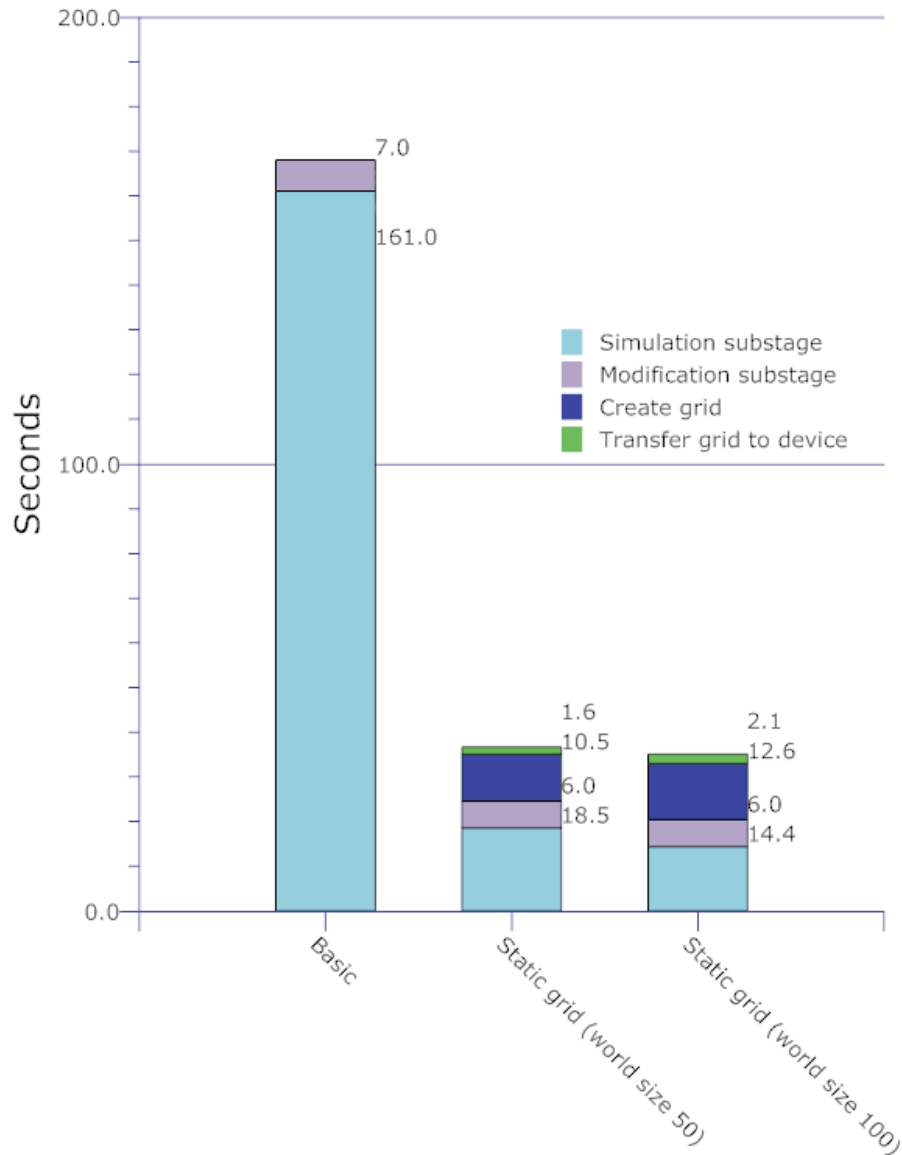


Figure 6.4: Time required to simulate 3600 simulation steps for 16384 agents.

15^3 grid cells already requires about 13KB of index data, so we can expect regular cache misses due to the random access done in the index data. NVIDIA does not provide any details of how cache misses are handled, so we can only speculate cache management imposes overhead, which decreases performance. The negligible performance increase when using less than 14^3 grid cells is a result of the relatively low amount of index data read, compared to the amount of agent data required. We access two elements in the index memory block for every grid cell used during the neighbor search – we read both the beginning and the end of the corresponding data memory block for every grid cell (see listing 6.1). This results in $(2 * cpd + 1)^3 * 2$ data elements read for every thread, with cpd being the number of grid cells needed to be looked at in each dimension. Even

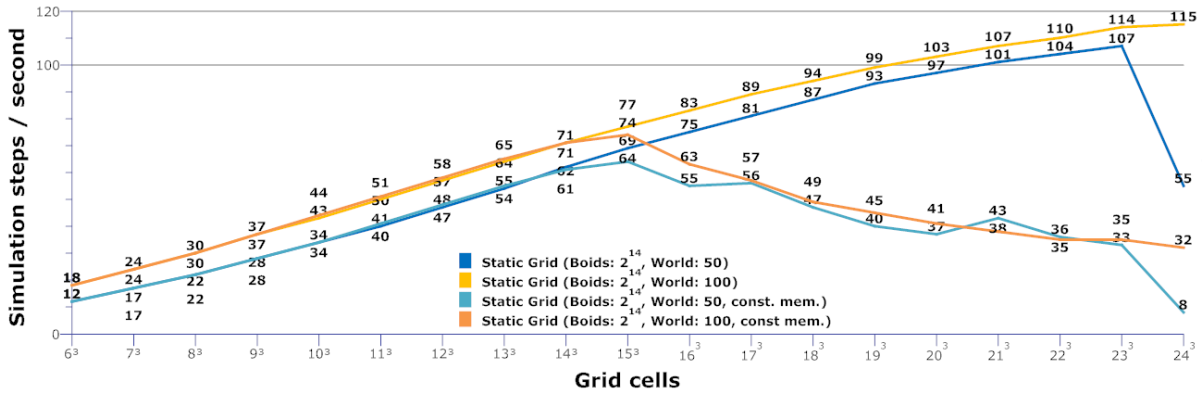


Figure 6.5: Static grid performance including the versions using constant memory.

when simulating a small amount of agents in a large world, we could not achieve more than a 2% increase in performance by using constant memory.

6.3 GPU managed static grid

As discussed in section 6.2.2, grid management is an important performance factor. In this section we present a static grid implementation managed by the GPU. Except that the creation is done by the device, the concept of this solution is identical to one described in 6.2. The goal was to achieve a better performance when creating the static grid, but our results show that the device does not perform well in this case.

6.3.1 Creation

Our new static grid implementation only uses the devicetype, as the host is no longer involved. The creation itself is split in three distinct steps. Each step is implemented in a separate kernel, as synchronization between the steps is required. The first two kernels are used to build up the index structure, whereas the last kernel fills the data memory block. We describe the three steps of our algorithm next. Figure 6.6 gives a graphical example.

Count The count kernel uses one thread per grid cell. Each of these threads looks at all agents and counts the number of agents within its grid cell boundaries. We use shared memory as a cache for agent data, since all threads have to read the same data. The results are written into the index memory block.

Prescan The kernel issues an exclusive scan on the index memory block. Scan is also known as parallel all-prefix-sums done on an array. Scan uses a binary associate operator \oplus with the identity I and an array of n elements as input

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$$

as a result. Our implementation is based on one provided by NVIDIA, which is discussed in [HSO07]. Our kernel executes the all-prefix-sums on the index data using addition as the binary operator, so the index data at position 0 contains a 0, position 1 contains the number of agents to be stored in the 0th cell, and position 2 contains the number of agents to be stored in both the 1st and the 0th cell and so on.

Fill The fill kernel fills the data memory block with the references to the agents. We have one thread per grid cell scan through all agent positions. If the agent is within the cells boundaries, the agents' index is written to the data memory block. The position, to which the values should be written in the data memory block, is based on the values stored in the index data structure and the number of agents already belonging to the cell. We use shared memory as a cache.

6.3.2 Performance

The performance of the GPU managed static grid based plugin is slower compared to its counterpart discussed in section 6.2. Figure 6.7 give an overview of how time is spent when creating the grid. We can see both count and fill kernels require more time than the host-based grid creation. The major problem with both the count and fill kernel is that every thread has to scan through all agents, which results in a high amount of global memory accesses and has a low arithmetic intensity. As we have discussed in section 2.4 a high arithmetic intensity is crucial for good performance at a GPU. We expect that the count kernel can be optimized further. For example one thread could look through a fraction of the agents and count the number of agents belonging to every cell. Reducing the values counted by every thread would provide us with the number of agents belonging to a grid cell. However, we also expect that the fill kernel can hardly be optimized any further on our hardware, as we cannot organize writes to global memory executed by threads in different thread blocks. For example, atomic memory operations could be used to implement a fill kernel in a similar fashion as we have described for the count kernel. As we cannot optimize the fill kernel any further, we decided to stop working on a GPU managed static grid version.

A pure GPU-based solution may not be a good decision, but future work might investigate whether it is useful to offload arithmetic intense parts of data structure creation to the GPU as it is shown in [Gre08]. An optimized version of our GPU managed grid may perform well in other scenarios, e.g. the usage of more grid cells, but we would expect a hybrid-solution to provide still better performance.

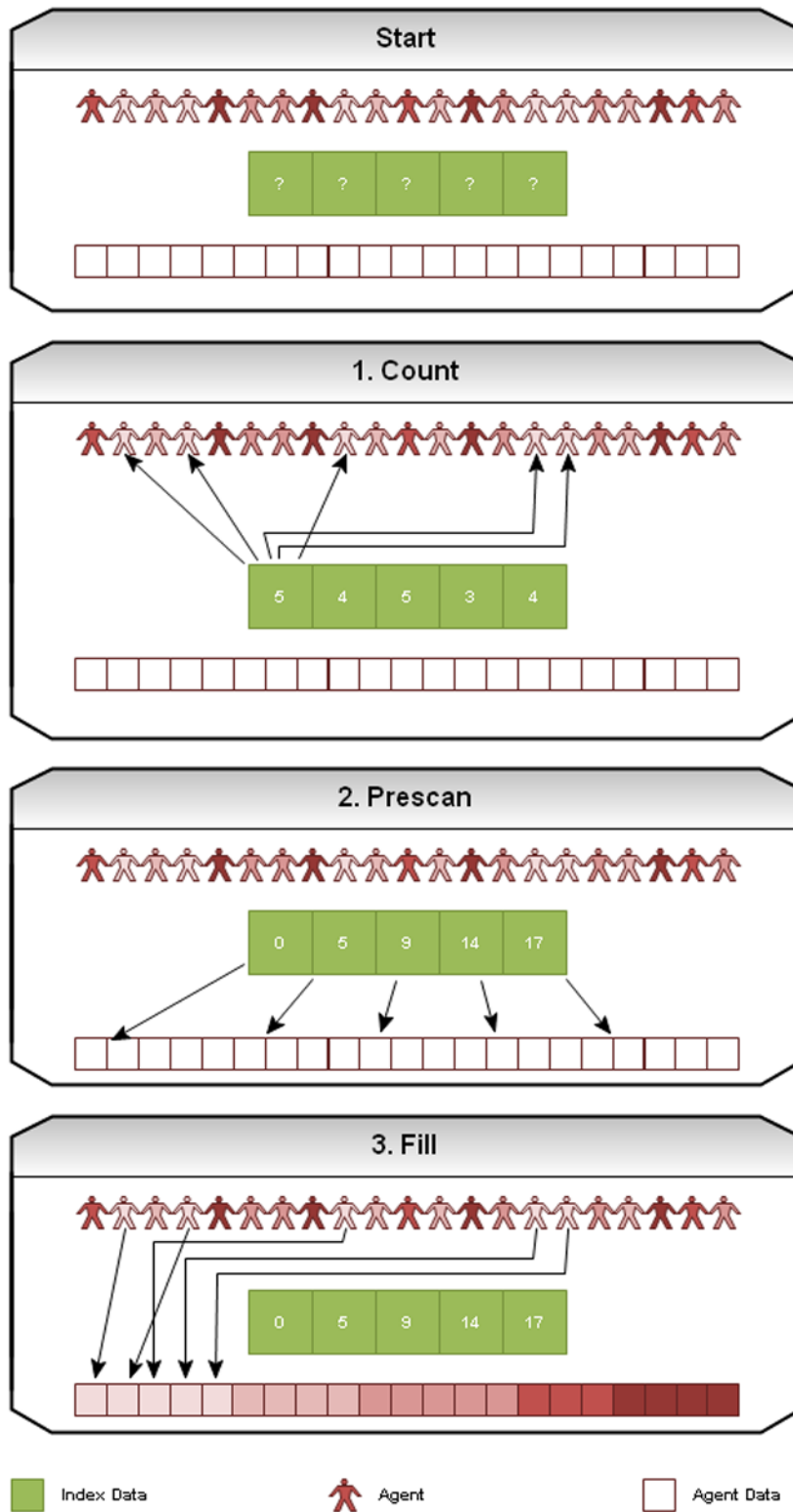


Figure 6.6: Overview of static grid creation algorithm running on the device.

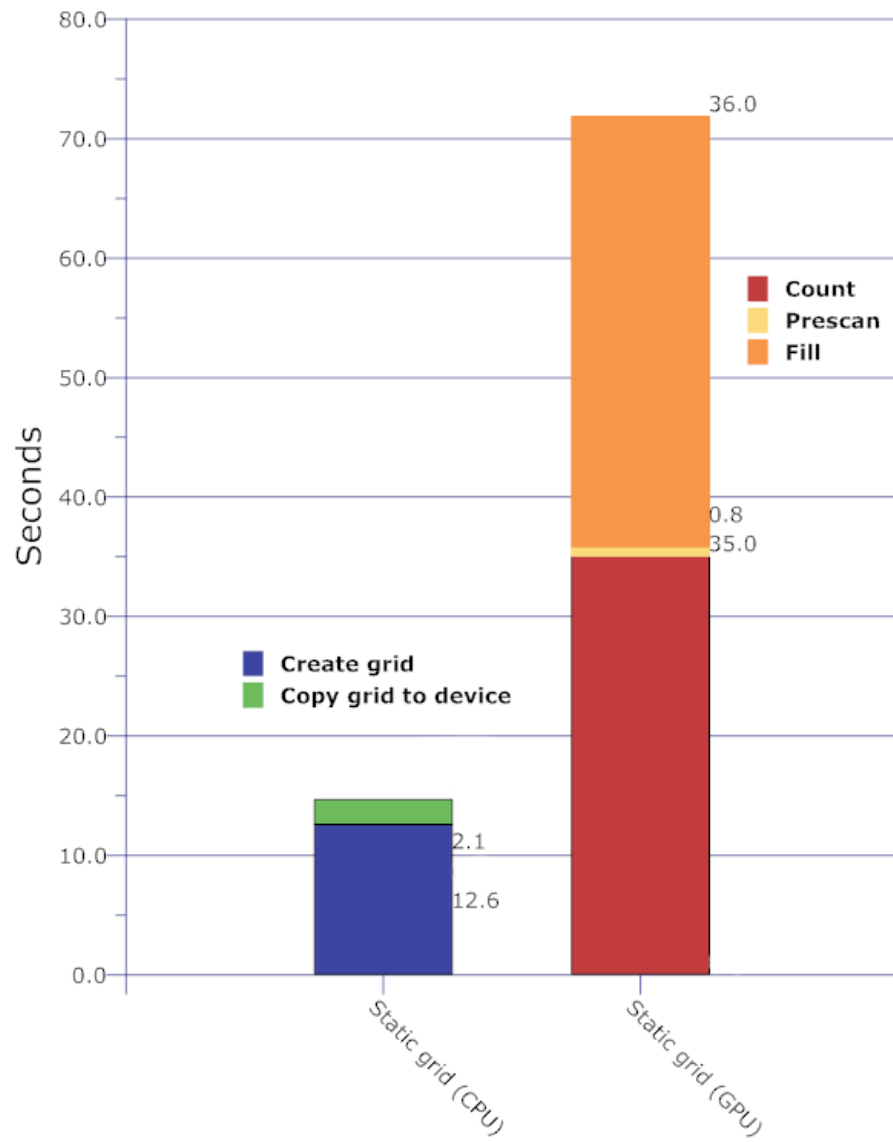


Figure 6.7: Overview of how much time is spent with the creation of the grid.

Dynamic grid neighbor search

Our first approach of a spatial data structure provides a performance increase of up to a factor of 35 compared to the basic version¹, but high performance is only possible when the parameters of the data structure are optimized for the simulated scenario. In this chapter we present a new concept of data structure, which automatically adapts to the simulated world and allows the usage of shared memory on the device to speed up calculations.

7.1 Dynamic grid concept

The plugin using the static grid cannot use shared memory to cache global memory accesses as the agents within one thread block have no common data requirements (see section 6.2.2). We believe with using shared memory to cache accesses to global memory, the performance can be increased.

We now propose a new mapping scheme to solve this issue. We continue to use one thread per agent, but map a group of agents *close*² together to one thread block. A group of agents close together must look at roughly the same agents to find their neighbors, so we can use shared memory to store chunks of agent position data in shared memory. We use this mapping scheme for the simulation kernel and continue to use the old scheme in the modification substage.

Combining the new mapping scheme with our static grid implementation of the last chapters is complex. We could try to map one grid cell to one thread block, but the number of agents in one grid cell varies between 0 and n – with n being the number of agents currently simulated. This variant causes two problems.

- The number of threads per thread block is fixed and limited to a maximum of 512. If we want to simulate n agents with one thread block, we must simulate multiple

¹Settings: 2^{17} agents, world size 200

²The definition of *close* changes with the different data structures implementations presented in the next sections.

agents per thread, which is possible but requires a complete redesign of the kernel.

- One grid cell could contain all agents. If this would be the case, the whole simulation is executed by one multiprocessor, which leads to a poor work balance.

To solve these problems, we introduce a new data structure called *dynamic grid*.

In contrast to the static grid discussed in chapter 6, the dynamic grid relies not on given grid cells, but creates them on the fly. A grid cell of the dynamic grid occupies a cuboidal part of the world. All grid cells can differ in size, but have a common maximum number of agents within its borders. We call the maximum number of agents in one grid cell *max* throughout the rest of this chapter. In our case, *max* is identical to the number of threads per thread block used to execute the simulation kernel. This restriction is required, as we map one grid cell to one thread block. All thread blocks simulating less than *max* agents, have idle threads.

We present three different variants of the algorithm to construct a dynamic grid in the next section, for which all results in different types of dynamic grid. A discussion of the performance of the different types is done afterwards.

7.2 Three algorithms to create a dynamic grid

Our data structures consist of a vector storing all agent positions – called data vector – and additional vectors used to identify cells and the range of agents to be searched by one thread block. The cells and ranges are identified by using pointers pointing elements in the data vector. The algorithms to construct a dynamic grid all consists of the same three basic steps:

1. Fill the data vector.
2. Impose an ordering scheme, which is sufficient enough to create cells, on the data vector.
3. Create the cells and define the grid cells to be searched by the neighbor search.

The first step is identical for all three dynamic grid types. The vector is filled in an unordered fashion with pairs consisting of both the agent position and a reference to the agent itself. Filling the data vector with n entries is a $O(n)$ operation. We now describe the next two algorithm steps for all three dynamic grid types.

7.2.1 Sorting

The first type of the dynamic grid uses the C++ sorting algorithm (`std::sort`) to order all elements of the data vector. `std::sort` is guaranteed to be a $O(n * \log(n))$ operation. The sorting is done based one dimension of the agent position data. We choose the x-dimension for all our measurements, but this is a fairly arbitrary choice. Cells are created by batching *max* number of agent, which are stored continuously in the data

vector, together. The x-dimension of the grid cell is defined by the first and the last agent position within the batch. The y- and z-dimensions are expected to range throughout the whole world. Creating the agent batches is a $O(n)$ operation. This partitioning scheme splits the world in small slices. The number of agents within all grid cells is max except for the grid cell created last, which is occupied by $\leq max$ agents. After the cells are created we scan through all grid cells and check, which grid cells are within the search radius of one another. If a cell is within the search radius the agents of the cell must be considered during the neighbor search. As all grid cells are stored continuously, we only have to store a pointer to the first element and the number of elements to be searched. Identifying the grid cells that are within the range of one another is $O(k^2)$ operation, with k being the number of grid cells. The number of grid cells depends on the number of agents simulated (to be precise, there are $\lceil \frac{n}{max} \rceil$ grid cells), so identifying the grid cells that are within the range of one another is a $O(n^2)$ operation. Creating this dynamic grid type has therefore a overall complexity of $O(n^2)$. We refer to the dynamic grid using sorting as *sorting dynamic grid*.

7.2.2 Ordering in one dimension

Sorting may be the easiest way to create the grid, but also imposes an ordering scheme among the agents within a grid cell. Our device code does not take any benefit from the order within a grid cell. We therefore propose a new algorithm, which does not order agents within the grid cells. The algorithm is similar to Quicksort. It recursively subdivides and partitions the data vector. The partitioning of the agents is always done by the same dimension³. The algorithm stops to subdivide a partition as soon as the number of agents is $\leq max$. Every partition, which is no longer subdivided, is a grid cell. Listing 7.1 shows the algorithm.

Listing 7.1: 1-dimensional ordering

```

1 void order (iterator start, iterator end, float x_left, float
  x_right) {
2   if (end - begin > max) {
3     // calculate the center point of all agents
4     float x_center = calculate_center(begin, end).x;
5     // partition the agents relative to the center
6     iterator center_iterator = partition(begin, end,
      x_center );
7     // center_iterator points to the
8     // first element >= x_center
9     order (begin, center_iterator, x_left, x_center);
10    order (center_iterator, end, x_center, x_right);
11  } else
12    // create cell
13 }

```

³We again arbitrarily chose the x-dimension for all our measurements.

grid implementation	% of used threads	idle threads	active idle threads
1-dim. ordering	59.5%	11152	3201
3-dim. ordering	62.5%	9821	3878

Table 7.1: Number of idle threads when simulating 16384 agents with 128 threads per block.

We refer to the dynamic grid using the algorithm described above as *1-dimensional ordering dynamic grid*. Except for replacing `std::sort` with the algorithm shown in listing 7.1 this dynamic grid implementation is identical to the one described in section 7.2.1 and has therefore also a complexity of $O(n^2)$.

7.2.3 Ordering in three dimensions

Our final implementation of the dynamic grid uses a modified version of the ordering algorithm presented in the section 7.2.2. Instead of ordering all data vector partitions by one dimension, we calculate the dimension for the current partition at runtime. We first calculate the distance from the center point of all agents to the border of the space covered by the current partition. Afterwards we partition alongside the dimension with the minimal distance to the border (and thereby also the maximum distance to the opposite border). This algorithm has the complexity of $O(n * \log(n))$. Practical experiments show that partitioning this way results in less idle threads than dividing the partition the way it would consist of two equally sized spaces in the world. In contrast to the two dynamic grid implementations presented in the last two sections, we do not calculate the range of agents to be searched at the host, but transfer the bounding boxes of the grid cells to the device. The device executed a brute force check on all grid cells to find the ones required at the neighbor search. This is done, since the calculations to find all required grid cells is more complex, as we divide the world in all three dimensions. This brute force check has the complexity of $O(n^2)$, since the number of grid cells depends on the number of agents simulates. Creating this dynamic grid type has therefore an overall complexity of $O(n^2)$. We refer to the dynamic grid type using this algorithm as *3-dimensional ordering dynamic grid*.

7.3 Performance

The three previous described dynamic grid types provide a similar performance, but are slower than the static grid presented in the last chapter. Figure 7.1 shows the scalability of the different dynamic grid implementations compared to the basic version and the best performance we could achieve with our static grid. The reasons for the slow performance of the dynamic grid differ for all three types. Figure 7.2 shows how time is spent by all measured versions.

The sorting dynamic grid is slower than the static grid in both grid creation and execution of the neighbor search. The first should come at no surprise, since we have to sort all agents' positions, whereas the static grid only copies the agents' positions

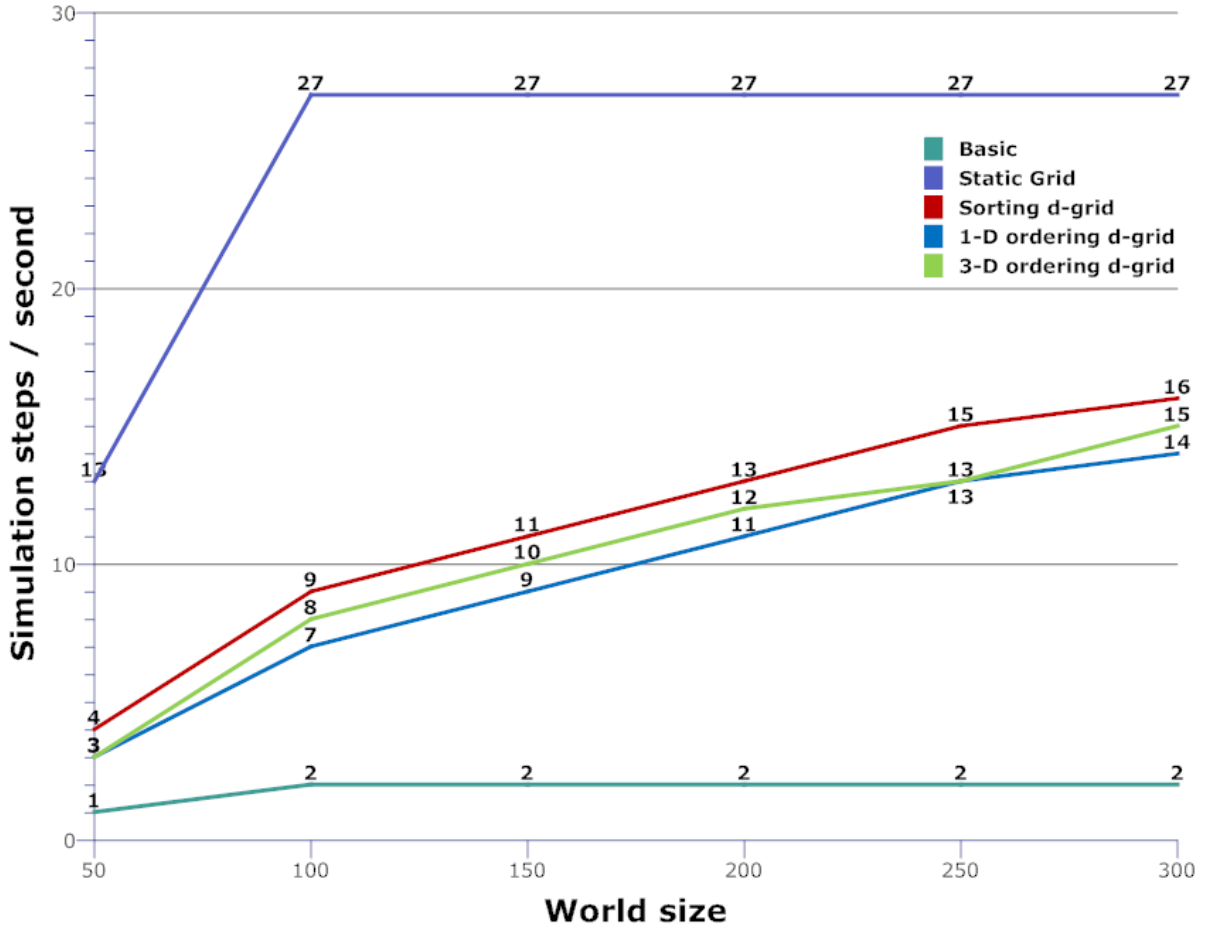


Figure 7.1: Performance overview of all grid implementations when simulating 2^{16} agents.

in a vector. The slow performance in neighbor search is a result of the inauspicious partitioning of the world when using the sorting dynamic grid. Our implementation partitioned the world in slice, which are limited only in x-dimension, whereas the others dimension range throughout the world. Using this partitioning scheme, one thread block must at least search an area of $WorldSize^2 * 2 * NeighborSearchRadius$, whereas the static grid searched $GridCellSize^3 * 9$ in the best case.

The 1-dimensional ordering dynamic grid uses the same way of partitioning the world and thereby suffers from the same problem. As an addition, this implementation generates grid cells with $\leq max$ agents in one cell. Every grid cell having less than max agents generates idle threads on the device, as every thread block has the size of max . Table 7.1 shows the number of idle threads. Most idle threads should hardly occupy any clock cycles, but as the device schedules warps and not single threads, idle threads that are part of a warp with active threads get scheduled to a processor. We call such threads *active idle threads*. Active idle threads directly waste processor cycles, whereas idle threads solely reduce the occupancy of the device.

Considering our three dimensional ordering dynamic grid, the idle threads reduce the

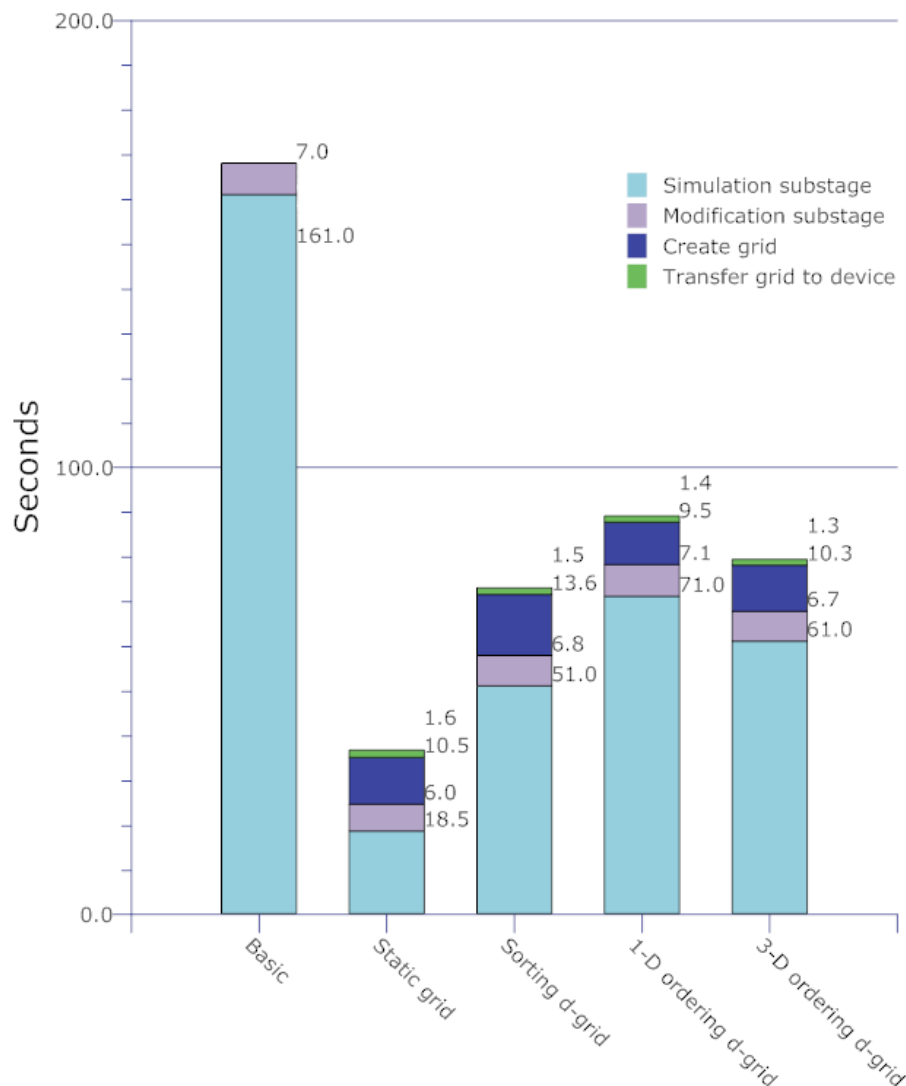


Figure 7.2: Time required to simulate 3600 simulation steps for 16384 agents in world of size 50.

occupancy to about 20%. Figure 7.2 shows that the performance loss due to idle threads is higher than what is gained by the faster grid creating algorithm when comparing the sorting dynamic grid with three dimensional ordering dynamic grid. However, the search algorithm of the three dimensional ordering dynamic grid can be improved. Currently all grid cells in the neighbor search radius are searched during the neighbor search, whereas this is not necessary when enough neighbors are found in the grid cells directly surrounding the simulated grid cell. We implemented a search algorithm expecting all grid cells to have enough neighbors in the surrounding cells. Our experiments show that this algorithm still leads to the correct behavior of the agents and improves performance, as it can be seen in figure 7.3. Further experiments with different sized worlds also show that the performance of the optimized dynamic grid is constant for all world sizes and only

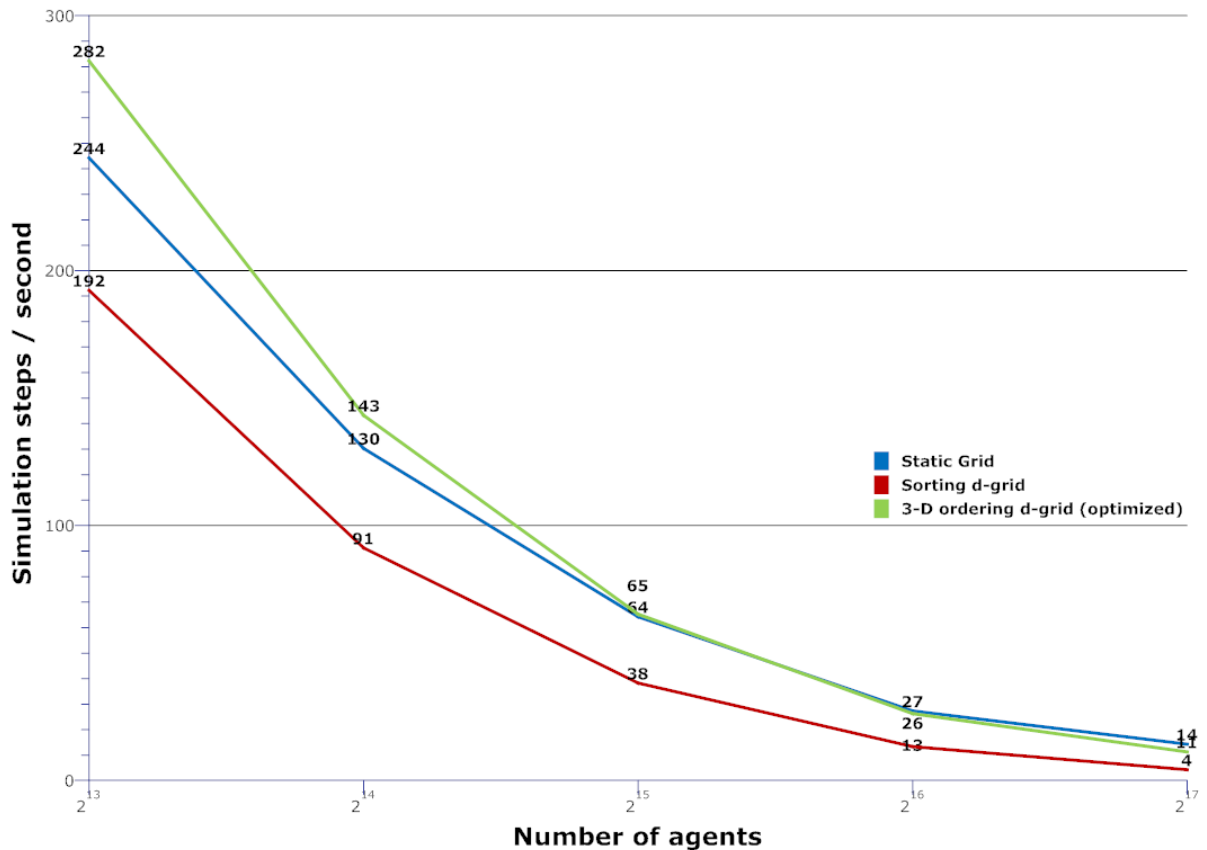


Figure 7.3: Performance overview of all grid implementations with different number of agents. Measurements are done a world size of 200.

changes by the number of agents simulated. We consider the performance improvement demonstrated by the optimized 3-dimensional ordering dynamic grid and its improved usability a success, even though we do not achieve the maximum performance in all cases. When simulating a small amount of agents the optimized dynamic grid provides the best performance – it is 15% faster when simulating 2^{13} agents compared to the static grid – but when we increase the number of agents, the $O(n * \log(n))$ grid creation algorithm becomes the performance bottleneck.

7.4 Conclusion

We have demonstrated in the last two chapters that it is possible to implement spatial data structures to be used with CUDA. Our first approach uses a fairly simple static partitioning scheme of the world, but still offers good performance. However this concept requires manually modification to find the best performing settings and is not perfectly suited for the device, as it does not take any benefit from the CUDA architecture. We designed the dynamic grid to achieve good performance without the need for manual

modifications and took the memory hierarchies of the device into account. Our new solution uses shared memory, but introduced another problem. Creating chunks of work so all threads of a thread block can be utilized is rather complex and should be considered a major design goal, as idle threads can massively impact performance.

The spatial data structures shown in the last two chapters are rather simple, but still provide a high performance increase. Both, the static grid and the optimized 3-dimensional ordering dynamic grid can simulate 2^{18} agents at more than 5 simulation steps per second. This performance is higher than what was proposed by Nyland et.al. [NHP07] for an octree-based solution, even though their numbers are based on a brute force simulation providing almost twice the performance compared to brute force OpenSteer implementation [Bre08]. Reynolds [Rey06] simulates up to 15,000 agents in a two-dimensional world at 60 frames per second using a Playstation 3⁴, whereas we simulate 2^{15} at roughly the same performance. However, these values cannot be compared directly, as Reynolds includes the drawing of the agents in his measurement and only simulates one tenth of the agents in one simulation step. Wirz et.al. [WKL08] worked at the same scenario as we did during this thesis and increased the performance of the neighbor search by a factor of 2.5 with the introduction of a spatial data structure. Introducing a spatial data structure in our GPU-based solution increased the performance of the whole simulation up to 35-times of our GPU-based brute force solution.

Further work to improve the performance of OpenSteer should be done by enhancing the neighbor search algorithm, as it still does not take full benefit from the scenario simulated. For example, neighbors that are behind the simulated agent are not considered when calculating the steering behaviors, since agents cannot *see behind its back*. We could use this to reduce the number of cells considered during the neighbor search and may also decrease the number of agents considered as a neighbor⁵. The data structure creation could also benefit from using multiple CPUs. The algorithm to create an ordering dynamic grid generates up to two independent tasks for every recursive function call. Considering the static grid, we could generate one static grid hosttype object per CPU and merge them when the data is transferred to the device. The dynamic grid concept would also benefit from a three dimensional partitioning scheme generating grid cells containing exactly *max* agents.

⁴The Cell processor of the Playstation 3 provides about 0.6-times the processing power of our GPU.

⁵Agents not within the field of view of another agent are removed from the neighborhood after the neighbor search.

CHAPTER 8

Conclusion

Data structure design is an important factor for CUDA application performance, as we have seen throughout this thesis. The requirements regarding the data structure design for Einstein@Home were fairly simple, as the device is reading from arbitrary but known positions. We experimented with the already existing data structure on the device, but due to the high amount of small memory transfers required to transfer the data structure to device memory, it was impossible to achieve any speedup. This strengthens our expectation, we have already proposed in our previous work [Bre08] that the data structure to be used at the device often differs from the one used at the host. However, our measurements show that as soon as an appropriate data structure is found, high performance improvements by using a GPU are possible. We also assume that the correct data structure is one of the most important factors to get a good performance. There is of course still a high optimization potential for the device code – e.g. the usage of shared memory – but without an appropriate data structure almost no speedup is possible.

Designing an optimal data structure for the device is rather complex, due to the memory hierarchy and the fixed number of threads per thread block for all blocks. Our static grid implementation does not take any CUDA specific details into account and could be implemented the same way for a CPU. However, it performs well and offers the best performance in some cases. Our second data structure has been designed with a focus on the CUDA memory hierarchy, but our attempt to utilize shared memory lead to a new problem. We were unable to bundle the maximum number of agents in one thread block in most cases, so we generated a high amount of idle threads on the device. Idle threads reduced the occupancy of the device and thereby the performance, whenever it is limited by the latency of memory accesses. The low occupancy leads to a rather low performance when using the dynamic grid data structures. We expect that the approach of mapping nearby agents to one thread block to use shared memory is superior compared to the static grid in all cases, but we failed at developing an algorithm providing both a low amount of idle threads and a good world partitioning.

Another way of creating the dynamic grid could take benefit from fact that agents move slowly across the world and batch neighbors of the last simulation step together in

one grid cell. This concept may provide a better performance, but also requires changes to the interface with the device, as we must transfer the neighborhood back to the host, and additional research regarding how to exactly batch agents together.

Our experiments with data structure creation at the device turned out not to be a good option for our case and we do not expect this concept to work well with most other data structures as well. If data structure creation involves a part with high arithmetic intensity, using the device to calculate this part may be a good choice, but executing the memory movements at the device costs a high amount of performance.

As a final note we like to add that regarding the problems outlined above, still some practical problems occur when working with CUDA. We especially missed the ability of directly profiling parts of our application. We were unable to measure the performance of the neighbor search at the GPU prior our work and can only see afterwards that introducing of a spatial data structure improved performance. We could have modified the existing application to get a better guess at the performance cost of the neighbor search – e.g. use one kernel to execute only the neighbor search – however this would still be no direct evidence. For example, when splitting the kernel into two separate kernels, the occupancy of the device may change and the steering calculations cannot be used to hide latency of memory accesses done during neighbor search. We could also not directly proof our assumption that global memory accesses of the static grid harm performance. The dynamic grid, which uses shared memory to cache accesses to global memory, indicates that this assumption is correct, however due to the facts that we can only utilize about 60% of all threads and the rather complex grid creation algorithm, we cannot see a clear performance improvement in all cases.

REFERENCES

- [All07] Bruce Allen. Final report on the S3 analysis. Technical report, LIGO Scientific Collaboration, 28 March 2007.
- [And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
- [BFH⁺04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.
- [BM00] Dov Bulka and David Mayhew. *Efficient C++: performance programming techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [BOI08] Papers and talks on boinc. <http://boinc.berkeley.edu/trac/wiki/BoincPapers>, 2008.
- [Bre07] Jens Breitbart. CuPP website. <http://cupp.gpuified.de>, 2007.
- [Bre08] Jens Breitbart. A framework for easy CUDA integration in C++ applications. Diplomarbeit, University of Kassel, 2008.
- [DiL04] Paul DiLascia. Performance Optimization, Controls versus Components. *MSDN Magazine*, June 2004.
- [FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".
- [Fol07] Folding@Home website. <http://folding.stanford.edu>, 2007.
- [Gou03] Brian Gough, editor. *GNU Scientific Library Reference Manual - Second Edition*. Network Theory Ltd., 2003.

- [Gre08] Simon Green. Particle-based particle based fluid simulation. http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf, Feb 2008.
- [HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, 2007.
- [HSP⁺07] Kun Huang, Joel H. Saltz, Tony C. Pan, Jose Antonio Andrades, Jose Becerra, Antonio Ruiz, and Manuel Ujaldon. The gpu on biomedical image processing for color and phenotype analysis, Dec 2007.
- [Mau03] Sean Patrick Mauch. *Efficient algorithms for solving static hamilton-jacobi equations*. PhD thesis, Pasadena, CA, USA, 2003. Adviser-Dan Meiron.
- [NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31, pages 677–695. Addison Wesley, 2007.
- [NVI07a] NVIDIA Corporation. CUDA parallel reduction. http://developer.download.nvidia.com/compute/cuda/1.1/Website/Data-Parallel_Algorithms.html#reduction, 2007.
- [NVI07b] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.1, 2007.
- [NVI08] NVIDIA Corporation. CUDA matrix multiplication sample. <http://developer.download.nvidia.com/compute/cuda/1.1/Website/samples.html>, 2008.
- [Rey99] Craig W. Reynolds. Steering Behaviors For Autonomous Characters. In *Proc. Game Developer Conference*, pages 763–782, 1999.
- [Rey06] Craig W. Reynolds. Big Fast Crowds on PS3. In *Proc. ACM SIGGRAPH Symp. on Videogames*, 2006.
- [SCS⁺] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. (to appear).
- [Top07] TOP500 List - November 2007. <http://www.top500.org/list/2007/11/100>, November 2007.
- [WKL08] Alexander Wirz, Björn Knafla, and Claudia Leopold. Comparison of Spatial Data Structures in OpenMP-Parallelized Steering. *HPCS 2008*, pages 31–37, 2008.