ABSTRACT

Today's Graphic Processing Units (GPUs) are not only good for gaming and graphics processing their highly parallel structure is predestined for a range of complex algorithms. They offer a tremendous memory bandwidth and computational power. Contrary to Central Processing Units (CPUs), GPUs are accelerating quickly and advancing at incredible rates in terms of absolute transistor count. Implementing a massively parallel, unified shader design, its flexibility and programmability makes the GPU an attractive platform for general purpose computation. Recent improvements in its programmability, especially high level languages (like C or C++), GPUs have attracted developers to exploit the computational power of the hardware for general purpose computing.

Several GPU programming interfaces and Application Programming Interfaces (APIs) represent a graphics centric programming model to developers that is exported by a device driver and tuned for real time graphics and games. Porting non-graphics applications to graphics hardware means developing against the graphics programming model. Not only the difficulties of the unusual graphics centric programming model but also limitations of the hardware makes development of non-graphics applications a tedious task.

Therefore NVIDIA Corporation developed the Common Unified Device Architecture (CUDA) that is a fundamentally new computing architecture that simplifies software development by using the standard C language. Using CUDA this thesis will show on the basis of an massively parallel application in which extent GPUs are suitable for general purpose computation. Special attention is paid to performance, computational concepts, efficient data structures and program optimization.

The result of this work is the demonstration of feasibility of General Purpose Computation on GPUs (GPGPU). It will show that GPUs are capable of accelerating specific applications by an order of magnitude.

This work will represent a general guideline for suggestions and hints as well as drawbacks and obstacles when porting applications to GPUs.

ACKNOWLEDGMENTS

Put acknowledgments here.

TABLE OF CONTENTS

1 INTRODUCTION

In recent years GPUs have moved from fixed pipeline processors to fully programmable processors. This evolvement has attracted many developers to do GPGPUs. The GPUs have devoted there silicon (transistors) for computing engines rather than for control engines like caches, branch prediction, coherency protocols compared to traditional CPUs. This incredible computing power made algorithms with an high arithmetic density run by an order of magnitude faster than on CPUs. Speedups of 100×–400× faster than the CPU were stunning but only a few people understand why such speedups are possible and why only a couple of algorithm can attain such speedups.

Many developers in these days are faced with multicore processors and have to implement or extend existing algorithms to take full advantage of the processing power of such cores. The CPU manufacturers are facing fundamental problems when increasing performance only by frequency. In former times higher frequency meant higher performance but a paradigm shift took place now the new way is going multicore which means higher performance. Moore's law says that for every 2 years the number of transistors on a chip will double. The transistors will be dedicated to more cores rather than for bigger chips as in the past. What does it mean to developers? They have to think in parallel, not only for two or four cores but rather for 16 or 32 cores. They have to assure that there code is scaling over many cores over many generations of CPU chips. There are several parallel programming languages and middleware to help developers to program in parallel but a quasi standard has not been established.

There are several programming platforms that are favored in certain fields respectively system leverls. On thread or loop-level there is OpenMP a programming interface for parallelization on shared-memory systems. Going a level higher to distributed systems that exchange messages during parallel execution there is Message Passing Interface (MPI) which is a quasi standard for message passing. There are further libraries but none of them is really suited for heterogenous computing systems.

Heterogeneous computing systems are different than common computing systems. For developers it is crucial to understand the underlying

architecture and programming models. They have to find the right tool (API, middleware, ...) to exploit parallelizm. Pitfalls and drawbacks often arise when studying the new system and developers have to adapt and deal with it. Furthermore its not only the use of the right tools its more the parallel thinking which is new to most of them. Parallel thinking and spotting concurrency in algorithms is the key for acceleration. Those aspects have to be considered when trying to do GPGPU.

As stated before there are several programming models respectively APIs availabe for programming the GPU. Most of them are based on a graphics API and are exporting a C++ interface to the developer abstracting the underlying hardware. The biggest problem with such APIs is that there are a lot of them and every has its pros and cons. Thats why the focus will be on APIs directly supplied by the manufacturers of the GPUs.

At the time of writing this thesis there were several myths about GPGPU. Developers are buffled when they here they have to implement the algorithm with a graphics API or it is not possible to do scatter operations to the memory. There exist further myths that discourages developers from computing on GPUs that should be demystified.

New technologies have a higher learning curve than existing and its always the question if its feasible and if its even possible to do general purpose computing on GPUs.

The starting point will be a review of what has been done followed by a feasibility study which will demystify the myths and cover major obstacles and drawbacks when doing computations on a GPU. Further experiments will show and reflect the computational capabilites especially in the view of general purpose programs.

The GPU compared to the CPU is very powerful and inexpensive. It has a much higher memory bandwidth and computational throughput. Not only that it is fast it is accelerating quickly. Recent improvements made the GPU as well flexible and programmable. But there are limitations and difficulties. It is a not only the new programming model or architecture the massively parallel nature of a GPU discomforts many developers.

The thesis will therefore start with a background chapter to show the relevance of the topic. After establishing the knowledge the reader will get a first introduction into the architecture and programming model of GPUs. Equipped with the knowledge a first algorithm will be ported to

the GPU for experiments and a first hands on.

This preparation will be helpful when getting to the main part where the actual algorithm is presented. An extensive analysis followed by the design and implementation of the algorithm will complete the theoretical part.

The following chapter will then go deeper into hardware specific stuff where the algorithm will be highly optimized for the GPU. The optimization will be verified in a subsequent chapter where the performance and scalability of the algorithm will be presented.

Last but not least a summary and future work will top the thesis off.

1.1 AIMS & OBJECTIVES

This thesis will cover all the topics to understand the architecture, programming model, drawback and pitfalls when doing GPGPU. The GPU is a highly parallel processor with thousands of threads and a peak performance of several hundred Giga FLoating point Operations Per Seconds (GFLOPS).

By the means of an application which will be ported to the GPU the general workflow will be shown and various procedure models examined. It will be presented that often traditional software engineering principles do not apply to high performance, parallel computing. For this work a NVIDIA GPU will be used together with CUDA that is a extension to C for parallel programming of GPUs.

The remainder of the thesis will give some in depth background to the topic and expose with programming models and the architecture of GPUs. Furthermore the development of a parallel implementation of an algorithm will be examined step by step. In this context software analysis and design principles will be shown that fit to parallel programming. A feasibility study will cover major obstacles and show how to avoid them.

Finally an application for segmenting an image will be implemented and presented in all aspects to the reader.

1.2 INITIAL TIME PLAN

For this thesis the following milestones were defined and listed below. Each milestone will have a short description which tasks have to be

accomplished to complete a mile stone. With every milestone a chapter respectively a section of the master thesis will be written which corresponds to the current carried out milestone. So the development of the milestones goes hand in hand with editing of the thesis.

M1: RELATED WORK IDENTIFIED (2/14/2009) The next step to take is to identify related work, conducting the literature survey. This survey should show the demand of GPGPU and used algorithms. Furhtermore it will help to understand the architecture and programming models. After gaining enough knowledge about the field its time to get the hands on the GPU.

M2: ALGORITHMIC VIEW UNDERSTOOD (2/28/2009) Equipped with the knowledge from the latter milestone it is time to understand how the architecture works by examining algorithms already mapped to the GPU. This step is crucial as it shows how algorithms map to and work on the GPU and a good understanding of the environment can help later on for choosing the right algorithm for porting, analysis, design and implementation.

M3: FEASIBILITY STUDY (3/14/2009) The feasiblity study will be used for the first steps into GPGPU. A low featured ray tracer will be used for examining what the hurdles and obstacles are when developing for the GPU. The knowledge will be used for choosing the right algorithm.

M4: APPLICATION, ALGORITHM CHOOSEN (3/21/2009) After completion of this milestone the algorithm is chosen to be ported to the GPU. So the next milestones will be concentrated only on this algorithm and not more generaly.

M5: ANALYSIS, SOLUTION PROPOSED (4/4/2009) Here a solution will be proposed and outstanding issues will be identified. There are several myths about GPGPU which will be stated here as well, because there are affecting the design of the application and cannot be discarded.

M6: EXPERIMENTS (4/14/2009) To exclude all eventualities a set of experiments will be undertaken. This will as well solve all myths.

M7: DESIGN, APPLICATION PORTED (5/14/2009) This milestone will cover the design of the application with all constraints, obstacles and hurdles that are involved with GPGPU. In this task the application will be implemented and run on the GPU. A good implementation will need several iterations were several aspects will be tuned: coalesced memory access, load balancing and many more.

M8: DISCUSSION (5/22/2009) Another important part is the discussion of the results and methodologies applied which will be done in this milestone.

M9: MASTER THESIS EDITED (5/30/2009) Here the thesis will be recaped and briefly outlined. Conclusions will be drawn and future work presented. A critical summary and conclusion will be provided as well.

2 LITERATURE REVIEW

There is a lot of literature that is describing and showing GPGPU. The literature of old algorithms ported to the GPU focuses how to exploit a graphics API to do general purpose computing. With the rise of Software Development Kits (SDKs) specially for GPGPU more and more literature emerges with variuos algorithms ported to the GPU.

The aim of this chapter is to provide, through selective reference to some of the literature, a clearer understanding why GPGPU is a topic nowadays, the miscellaneous programming environments and lastly the wide field of GPU computing. Several implementations of algorithms on the GPU are presented and evaluated. The chapter is divided in several parts. The subject of the first part is a general view about the move to parallel processing and obstacles and problems that arise when doing parallel programming. Building up on the first part, several programming environments that enable GPGPU are presented and discussed. Succeding a part that describes the algorithms that are suited for computing on the GPU. The remainder of this chapters shows some already implemented algorithms to the GPU and shows the wide field of application.

2.1 GENERAL PURPOSE COMPUTING ON GRAPHICS HARDWARE

The first thing to answer is why should someone do GPGPU anyway. For the most people CPUs are just enough. They do not demand on high computational power and on a high bandwidth. Still there is paradigm shift taking place and this can not be neglected. As stated in [39] and in [40] CPUs manufacturers are facing problems which they cannnot overcome just by increasing the frequency. The problems are known as *The Memory Wall*[36], *The Frequency Wall* and *The Power Wall*. The only way seen by CPUs makers is currently to go multicore. Intel e.g. went multicore with there new *Core Microarchitecture* for consumer products and even a GPU replacement, *Larrabee* [46]. International Business Machines Corporation (IBM), Toshiba and Sony developed the *Cell Broadband Architecture* a 9 core chip [42]. Sun Microsystems developed the *Niagara* CPU a multi-core general purpose processor. It has eight

in-order cores, each of them capable of executing four simultaneous threads [25]. A further processor from Sun is the *Sun Rock* a 16-core 32-thread plus 32-scout-thread CPU developed for high throughput and high single-thread performance [53].

The GPUS went years ago to multicore and multithreading compared to CPUS. The GPUS are maybe the kind of processor where CPUS are heading to in terms of multithreading and raw performance. Forecasts project that every two years the amount of cores can double. The multicore approach may be the answer to the problems stated above but this yields to another thing the *parallel programming problem* [41]. User will only benefit from this growth if software can make use of all the cores and software can only benefit from it if developers can leverage concurrency in software. Developers have to look at their code to identify computatinal intensive operations and identify how those operations can benefit from concurrency. Sutter[51] stated that ''The free lunch is over'' , developers cannot count only anymore on CPU throughput they have to get their hands on concurrent programming requirements, pitfalls, styles and idioms [51].

2.2 PARALLEL PROGRAMMING & THINKING

Many developers learned about the single-threaded von neumann model and are not familiar with parallel code which is subject to errors such as deadlocks and livelocks, race conditions and many more. Parallel programming is difficult and there are several paradigms to make life easier for developers.

On of these is the data-parallel paradigm. Where one is not trying to assign diffferent subtasks to separate cores rather assigning an individual data element to a separate core for processing [7]. The three-dimensional (3D) rendering, an embarassingly data-parallel problem, has driven the GPU evolution which makes the GPU a perfect target for data-parallel code. There are several fine-grained or data-parallel programming environments that leverage the GPU and other parallel architectures for general purpose computing.

Programming Libraries & Languages

Beyond graphics APIs like Cg, High Level Shading Langauage (HLSL), OpenGL or DirectX several libraries and laguages were introduced to make GPGPU possible. Many of them are based on the graphic APIs but have backends to many other libraries.

They have one thing in common, they enable stream computing on the GPU. The first langauge is Brook[6] for GPU it extends American National Standards Institute (ANSI) C with concepts of stream programming. Brook stream are nothing else than arrays on which the computation is done in parallel. The program that is doing some operations on streams is called kernel. At run-time the kernels and streams are automatically mapped to the fragment processer and to texture memory.

A similar concept follows Sh [34]. Sh is embeded in C++ and allows to program GPUs for graphical and general purpose computations. The research was commercialized and RapidMind Inc. was formed. RapidMind delivered with its programming environement not only support for GPUs but also for other massively parallel processors like the Cell. Recently Intel aquired RapidMind to merge it into there Intel Ct Technology.

For scientific visualization there is Scout [35], it allows run-time mapping of mathematical operations over data sets to the GPU, for general purpose computing it is rather limited as it focus is on visualization of data sets.

Another library is Glift [28] which is a template library for a wide range of data structures. It opreates wiht C++, Cg and OpenGL development environments.

All the above libraries and languages are all based on a graphics API. The functions are all graphics centric and not all are suited for GPGPU. Since these functions have to accomodate to the graphics pipeline not all hardware features are directly exposed to the developer they are encapsulated and additionally the graphics pipeline puts up some constraints to the programming flow [39].

Therefore ATI Technologies Inc. (ATI) and NVIDIA Corporation (NVIDIA) developed programming envrionments specially for GPGPU. From ATI came Close To Metal (CTM) a low level programming interface exposing previously unavailable low level functionality , direct access to the native instruction set and memory, for GPGPU. ATI stand point

was that the community should contribute the high level API and tools. NVIDIA has gone a different way, they offered a complete programming environment (SDK) for GPGPU namely known as CUDA [5] with many tools for programming and debugging.

Recently ATI released as well a full featured programming environment, the Stream SDK. Nevertheless as of writing of the thesis this SDK was not available and hence not considered.

The focus of this work will be on CUDA. CUDA is one of the environments which is not based on a graphics library and officialy released by NVIDIA for their GPUs. CUDA is a minimal extension to C and C++ programming languages. The technique employed by CUDA is Single Program Multiple Data (SPMD). Tasks are split up and run simultaneously on multiple threads (cores) with different input [38].

2.3 EMBARRASSINGLY PARALLEL ALGORITHMS

Before even digging into the wide field of algorithms and difficult problems in high performance computing one has to understand the hardware architecture of the GPUs to make a decision if an algorithm can be mapped on GPUs. A pretty good overview over the NVIDIA GPU gives the NVIDIA CUDA programming guide [48].

A little bit outdatet but still of interest is the GeForce 6 Series GPU Architecture [24] which gives an overview how the GPU fits into the whole system, what a fragment processor, vertex processor or what textures are. To have a even deeper look into GPUs the article [16] is highly recommended.

2.3.1 *Computations which Map Well to GPUs*

It is important to understand that GPUs are good at running computer graphics and algorithms which mimic or have the attributes of computer graphics in terms of data parallelism and data independence. Not only that similar computations are applied to streams of many data elements (vertices, fragments, ...) but also the computation of each element is completely or almost completely independent [21]. Such types of algorithms are often called embarassingly parallel algorithms where subtasks rarely or never communicate to each other.

Another important fact for a algorithm is the *arithmetic intensity*. The arithmetic intensity is the ratio of computation to bandwidth or formally:

$$\textit{arithmetic intensity} = \textit{operations} / \textit{words transferred}.$$

This fact is important because the increase of computational throughput is faster than the memory throughput which leads to the problem known as *The Memory Wall*. The GPU memory systems are architected to deliver high bandwidth, rather than low-latency, data access. As such computations that benefit most of the GPUs have a high arithmetic intensity [21]. The next sections will represent some algorithms which are ported completely or to some extent to the GPUs.

2.3.2 *Raytracing*

Ray Tracing [2] is an embarrassingly parallel algorithm which could fit well to GPUs. A work about raytracing on massively parallel computers [27] is already done. There were several implementations already done for the GPU e.g.

2.3.3 *Photon Mapping*

Raytracing has a local illumination model. To generate more realistic effects like caustics, diffuse / glossy indirect illumination and more a more sophisticated model has to be used. Global illumination like *Photon Mapping* [23] can create all the effects that raytracing cannot. There are implementations of photon mapping on GPUs [44] which are developed with graphics apis and not with CUDA. Anyhow since photon mapping is heavily using a kd-tree it would be a major effort to develop an efficient data structure which has the same functionality as a kd-tree. Nevertheless *Photon Mapping* is a embarassingly parallel algorithm that could fit very well to the GPU.

2.3.4 *Multiple Precision Arithmetic*

Another interesting field which demands high computational power is number theory. The most common/known application is asymmetric cryptography. To decipher messages that are cyphered with an asymmetric algorithm one needs superior computational power. An overview

over common algorithms gives [26]. All algorithms have one thing in common they need a multiple precision library to represent numbers with 200 and more digits. A good overview gives [52].

The idea could be to implement some of the factoring algorithms to the GPU. The onlything needed is the multiple precision library. In [26] the GNU Multiple Precision (GMP) library was used to implement the algorithms. So the multiple precision arithmetic is another candidate for algorithms that can run on the GPU. There are several implementations but they are rather limited and implement a specific algorithm with multiple precision but not a generic multiple precision library [58, 1].

2.3.5 *High Dynamic Range*

Image processing is always a candidate for embarassingly parallel algorithms. A pretty new algorithm to enhance images is tone mapping [33]. Tone mapping is the compression of dynamics in High Dynamic Range (HDR) pictures. There are several algorithms for tone mapping: Mantiuk [32], Reinhard [45], Durand [15], Fattal [17] and many more. All of this algorithms are present in the pfs-tools library written by Krawczyk [33]. As this document was written Krawczyk was already implementing a part of the pfs-tools on GPU but not publishing it.

2.3.6 *Genetic Algorithms*

Parallel genetic algorithms are usually implemented on parallel machines but fine-grained parallel genetic algorithms can be mapped to GPUs [56]. In [29] its shown what kind of genetic algorithm map well to GPUs and how the work and communication is handled. It is pretty easy to implement simple genetic algorithms on the GPU but with increasing complexity one has to consider many more things: load balancing, communication pattern, dynamic memory allocation, resolving of recursion .... Another paper [55] compares genetic algorithms implemented on CPUs and GPUs and shows that the latter is much more effective than the former. All genetic algorithms have one thing in common the core algorithm. Once effectively implemented on the GPU the core algorithm is extended with definitions like the population, selection, recombination and mutation to solve a specific problem. The skill here is to choose

the right definitions and not more the efficient implementation of the core algorithm.

2.3.7 *Chaos Theory*

Another interesting field is chaos theory. Especially the visualization of chaos. The maybe most famous visualization of chaos is *The Fractal Flames Algorithm*. Fractal flames are a member of iterated function system class of fractals created by Scott Draves [13]. He uses a rather complicated set of functions in the system to generate stunning visualization of the iterative process of the system. The next release will have support for the GPU. Thats why it was firstly discarded but the research about chaos led to other interesting papers respectively books.

There is no need to use approx. 21 function for a system to generate visually appealing pictures of chaos. Sprott showed in [50] that even with very simple functions one can create patterns in chaos. This patterns are called *Strange Attractors* and are the visualization of chaotic behaviour. There are created by iterating a simple equation some million times.

Pickover shows in his book [43] how to create patterns from a variety of sources. He shows how to create nice looking patterns from fourier analysis, acoustic, chemistry and many more. Anyhow all of these equations or differential systems have on thing in common no matter how complicated the system is the core algorithm is to iterate a specific equation with correct input numbers to create chaos. The algorithm is heavily computational bound which makes it a good candidate for porting to the GPU.

2.3.8 *Image Processing*

Many image processing algorithms are embarrassingly parallel as one can calculate filters on idividual pixels without considering the neigbouring pixels. Bruyns and Feldman showed by example how image processing is done on the GPU by a canonical example, limb finding [4]. Another paper shows how to do convolution, diffusion, wavelet or tone mapping [37]. Furthermore there are several universities that deal with GPU image processing and publishing algorithms and reports.

2.4 SUMMARY & CONCLUSION

The GPU is intruding into more and more fields in scientific, mathematical, financial and many more fields. This is happening because people are demanding more and more computational power and the GPU can satisfy this demand. As it was shown in the previous chapters the GPU is used in many ways and there is a high demand on exploiting the GPU even farther. With the development of new native programming environments (CUDA, Stream SDK) even more developer will be attracted to use this new architecture.

But all of this reports showed that with the old tools the programming is and was error prune and difficult. Only some specialist that understood the graphics part in extense and the algorithm to be ported could implement a efficient solution.

This is where this thesis will attach. Taking the new programming envrionments and triying to port a sample application to the GPU.

3 PARALLEL PROCESSING WITH GPUS

In recent years more and more parallel hardware appeared on the computing market with different architecture specialized for different problems. One of the recent and most discussed architectures is the GPU. The major player in the field are ATI and the NVIDIA corporation that both offer SDKs and GPUs for general purpose computing. Beside GPUs there are a lot other companies that offer parallel hardware architectures listed in Chapter 2. Following a brief introduction to parallel architectures important to specify and identify the sort of parallelism offered by GPUs and other hardware.

3.1 PARALLEL ARCHITECTURES

There are several ways how operations in a system can be executed in parallel.

- Single Instruction Single Data (SISD) This is the common von Neumann model, which is a system which exploits no parallelism.
- Single Instruction Multiple Data (SIMD) operations inside a functional unit where one operation is simultaneously applied to a vector of 2–16 components. Many modern CPUs have now SIMD units but every manufacturer is naming them differently (e.g. Altivec, Vector Multimedia eXtension (VMX), Matrix Math Extensions (MMX), Streaming SIMD Extensions (SSE)).
- Multiple Instruction Multiple Data (MIMD) threads, cores, processors each executing on its own data. Many parallel architectures can be mapped onto the MIMD architecture.

  There is as well the Multiple Instruction Single Data (MISD) classification which is a uncommon architecture but can be used e.g. in fault-tolerance systems where different units generate a result on the same data and they must all agree on the result. For common parallel architectures its not used. By far the most used classification for parallel systems is MIMD that can be further divided into two more classifications.
- SPMD threads within a core where one program is running on the core and different threads are executing with different data. A combination

of the SIMD and SPMD model is used on modern GPUs.

- Multiple Programs Multiple Data (MPMD) threads executing a different program on different data. The MPMD threads reside either within a single core (hardware threads, hyper-threading) or in distinct cores of a processor or lastly in different processors.

The above classification is based upon the number of concurrent instruction and data streams available in a system. The classification comes from Flynn in [18]. Today there is still no satisfactory characterization of the different types of parallel systems. Flynn's taxonomy was a first step, today there are much more characterizations that refine the previous mentioned.

To characterize the GPU one needs two classifications. First of all in each GPU there are threads that are grouped together which are executing one instruction on different data. These SIMD groups are further grouped to a SPMD system where all of these, there can be many of them, are executing the same program. On the basis of NVIDIAs Tesla architecture a more detailed look at a GPU will be given in the next sections.

3.2 THE TESLA ARCHITECTURE

The Tesla architecture announced 2007 and developed by NVIDIA is the first GPU highly specialised for raster operations and more important for general purpose computing. Formerly GPUs had fixed-function pipelines and separate processing units with no ability for programmability to the vertex and fragment stages of the pipeline. In recent years manufacturers of GPUs added more and more programmability to the different stages of the pipeline and at the same time general purpose computing capabilities [31]. Furthermore manufacturers introduced the Unified Shader Model (USM) that unifies the processing units allowing for better utilisation of GPU resources. The resources needed by different shaders varies greatly and the unified design can overcome this issue by balancing the load among vertex, fragment and geometry functionality [8].

The Figure 3.1 shows the Tesla architectures. As mentioned above the new GPU architectures are a radical departure from traditional GPU design. The Tesla 8 Series has 16 multiprocessors. Each multiprocessor

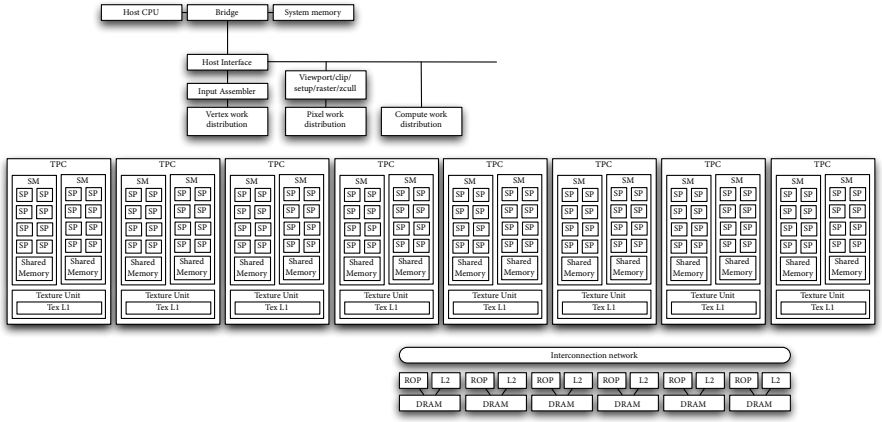


FIGURE 3.1 Tesla Architecture [NOT FINISHED YET]

is composed of 8 streaming processors, 128 processors in total. Each streaming multiprocessor has 16 Kilobyte (KB) of shared memory a L1 cache attached and has access to a texture unit. A streaming processor consists of a scalar Arithmetic Logic Unit (ALU) and performs floating point operations. 32 streaming processor build up a SIMD unit in NVIDIA terms called warp, in that one instruction is executed. The Tesla C870 has 1.5 Gigabyte (GB) Graphics Double Data Rate v3 (GDDR3) graphics memory, 519 GFLOPS of peak performance and 77 GB/s peak memory bandwidth. Many massively data-parallel algorithms can be run sufficiently on this specialized architecture [8]. Programming for this architecture is done with CUDA the C language extension which will be covered in the next section.

THE MAINSTREAM GPU There are two solutions for GPU computing from NVIDIA. Firstly the *Tesla 8 Series* which is specifically designed for high-performance computing. The *Tesla 8 Series* is available as an Peripheral Component Interconnect Express (PCI-E) add-in card, desk-side computing system and as a one rack unit (1U) computing system. All this systems have no display port and hence they can exclusively be used as a computing nodes.

The second solution are the mainstream *GeForce 8 Series* card that is mainly used for gaming. The architecture of these both solutions is the same (USM) they differentiate only in the magnitude of some per-

formance characteristics (memory size, peak memory bandwidth, peak GFLOP). The advantage of a mainstream GPU is that the visualization can be done on the same card and the low price.

For the purpose of this thesis a *GeForce 8800 GTX 512* card will be used. The main performance characteristics are shown in Table 3.1 and Table 3.2. For reference the *Tesla C870* card is shown as well.

Card	Mem. Size [MB]	BW [GB/s]	Clock [MHz]	Bus Width [bit]
Tesla C870	1500	77	1600	384
8800 GTS	512	62.1	1940	256

TABLE 3.1 Memory performance characteristics for Tesla and GeForce

The most significant difference between the two cards is the peak bandwidth. There are several algorithms which rely on a high bandwidth which would increase the run time on as *8800 GTS* card even if it has a higher peak floating point operations per second as seen in Table 3.2.

Card	Stream Processors	GFlops	Clock	TDP [Watt]
Tesla C870	128	519	1536	170
8800 GTS	128	624	1625	135

TABLE 3.2 Computing performance characteristics for Tesla and GeForce

Every card of the *8 Series* can be programmed with CUDA an API for GPU computing. What CUDA is and how to use it will be explained in the next section.

3.3 COMMON UNIFIED DEVICE ARCHITECTURE (CUDA)

In 2007, NVIDIA introduced an extended ANSI C programming model and software environment the compute unified device architecture. The reason why CUDA was born is that parallelism is increasing rapidly with Moore's law[1] and the challenge is to develop parallel application software that scales transparently with the number of processor cores. The main

1 Processor count is doubling every 18 - 24 months

goals when CUDA was developed were that it scales to hundreds of cores, thousands of parallel threads and it allows heterogeneous computing (CPU + GPU). All this considerations led to the result that CUDA runs on any number of processors without recompiling an the parallelism applies to both CPUs and GPUs [5].

CUDA is C with minimal extensions and defines a programming and memory model. There are three key abstractions in CUDA, a hierarchy of thread groups, shared memories and barrier synchronization [48] that are exposed to the developer. CUDA uses extensive multithreading where threads express fine-grained instruction, data and thread parallelism that are grouped into thread blocks which express coarse grained data and task parallelism. The developer has to rethink about his algorithms to be aggressively parallel. The problem has to be split into independent coarse sub-problems and at finer level into fine-grained sub-problems that can be solved cooperatively [48].

CUDA has been accepted be many developers which can be seen by the huge amount of already developed software and contributions to the CUDA zone [2].

A brief look shows the CUDA computing sweet spots[5]

- High arithmetic intensity (Dense linera algebra, PDEs, n-body, finite difference, ...)
- High bandwidth (Sequencing (virus scanning, genomics), sorting, database, ...)
- Visual computing: (Graphics, image processing, tomography, machine vision, ...)
- Computational modeling, science, engineering, finance, ...

This is just a small snapshot of algorithms that can be used with CUDA on GPUs. For a more extensive list and speedups compared to high-end CPUs see the webpage of NVIDIA CUDA zone.

3.4 CUDA PROGRAMMING MODEL

The CUDA programming model exposes the graphics processor as a highly multithreaded coprocessor. The GPU is viewed as a compute

2 http://www.nvidia.com/cuda

device to the host that has its own device memory and runs many threads in parallel.

Applications are accelerated by executing data parallel portions of the algorithm on the GPU as *kernels* which run in parallel on many threads. There are some major differences between CPU and GPU threads. A GPU needs thousands of threads for full efficiency where CPUs only need a few of them. GPU threads have a very little creation overhead and are extremely lightweight compared to CPU threads.

THREAD BATCHING A kernel is executed as a grid of thread blocks where data memory space is shared by all threads. A thread block is a batch of threads that can cooperate with each other by synchronizing there execution[3] and efficiently share data through the low latency shared memory. Two threads from two different blocks can cooperate with atomic functions through the global memory. The identification of a thread is accomplished through block and thread ids which are assigned to each thread at creation time.

BLOCK AND THREAD IDS Every thread and block has a unique id. As a result of this each thread can decide whata data to work on. For every block there is a assigned id in one-dimensional (1D) or two-dimensional (2D) layout. Thread ids can be accessed either with 1D, 2D or 3D coordinates similar to multidimensional arrays. It simplifies memory addressing when processing multidimensional data(e.g image processing, matrix multiplication or solving partial differential equations on volumes). The data can reside in several levels of the device memory.

DEVICE MEMORY SPACE The memory space is a hierarchy of several memory types that can be accessed per thread, block, grid and the host. The threads have access to all memory levels beginning with the read-write (RW) registers, local, shared, global, read-only (RO) texture and constant memory. The grids have only access to global, constant and texture memory. Whereas the CPU can RW global, constant and texture memories.

Global, constant and texture memory have long latency accesses. They

3 For hazard free shared memory access

reside off-chip where registers local[4] and shared memory resides on-chip. The global, constant and texture memory are mainly used for communication of RW data between host and device where the contents is visible to all threads. As mentioned above texture and constant memory can be written by the host where constants and data are initialized.

3.5 A SIMPLE EXAMPLE

This simple example will show the structure of an CUDA program. The executing kernel will do some easy calculations on the data provided, load the data into shared memory and write back the results to the global memory.

A CUDA program has a specific structure where the major parts are described in this paragraph. The first thing to do is to initialize the device and some auxiliary variables. The Listing 3.1 shows the initalization.

```
1 CUT_DEVICE_INIT(argc, argv);
2
3 uint32_t num_threads = 32;
4 uint32_t mem_size = sizeof(float) * num_threads;
```

LISTING 3.1 Hardware initalization

Since the GPU is attached to the PCI-E bus the host has no direct access to the global, constant and texture memory and has to transfer the data back and forth with the Direct Memory Access (DMA) engine of the device. This is accomplished through the CUDA API calls that initiate the transfer. Before any transfer can be done one has to allocate memory on the host and on the device for input and output data. This is shown in listing Listing 3.2.

```
1 // allocate host memory
2 float* h_idata = (float*) malloc(mem_size);
3 // allocate device memory
4 float* d_idata; cudaMalloc((void**) &d_idata, mem_size);
5 // allocate device memory for result
6 float* d_odata; cudaMalloc((void**) &d_odata, mem_size);
```

4 Not true for older GPUs chips where local data is spilled out to global memory

```
7  // allocate mem for the result on host side
8  float* h_odata = (float*) malloc( mem_size);
9  // copy host memory to device
10 cudaMemcpy(d_idata, h_idata, mem_size, cudaMemcpyHostToDevice);
```

LISTING 3.2 Data transfer of data

After setting up the input data the setup execution parameters are defined that are used to startup the kernel. The grid(1, 1, 1) statement defines a multi-dimensional array of grids $x = 1, y = 1, z = 1$ whereas the threads(num_threads, 1, 1) defines a multi-dimensional array of threads $x = num\_threads, y = 1, z = 1$ which are actually 1D arrays. Listing 3.3 shows the call of the kernel with its input and output data.

```
1 // setup execution parameters
2 dim3  grid( 1, 1, 1);
3 dim3  threads(num_threads, 1, 1);
4
5 // execute the kernel
6 kernel<<< grid, threads, mem_size >>>(d_idata, d_odata);
```

LISTING 3.3 Execution of the Kernel

Attention must be paid to the both statements that define the array for the grid and the thread block. The size of the array or the amount of threads dependent on three factors, (1) register usage, (2) shared and (3) local memory usage per thread. Accumulated per grid they are not allowed to pass a certain size. Calculations how the existing ressources are used can easily be calculated with the *CUDA occupancy calculator*. The performance of the application depends on how good/bad the existing resources are exploited.

If everything went well the host can copy the data from device memory to host memory and check, visualize or store the calculated values. Listing 3.4 shows the last steps before exiting the program.

```
1 // check if kernel execution generated and error
2 CUT_CHECK_ERROR("Kernel execution failed");
3
4 // copy result from device to host
```

```
5  cudaMemcpy(h_odata, d_odata, sizeof( float) * num_threads,
6             cudaMemcpyDeviceToHost);
7
8  // cleanup memory free(x), free(y), free(z) ...
9  CUT_EXIT(argc, argv);
```

LISTING 3.4 Retrieving of the Results

The previous listings showed the host code and how to launch a kernel on the device. The Listing 3.5 shows the device code portion. There are several qualifiers that define which function is compiled for which processing unit. The __global__ qualifier specifies that this function is run on the device and hence compiled for the GPU where the __host__ qualifier specifies that this function is only run on the host and not on the device. There are more function specifiers that can be looked up in [48].

For data there are as well qualifiers where one can specify where the data is located, either in constant, global or shared memory. In Listing 3.5 the __shared__ qualifier is used. The device will use the shared memory to preload the data for faster access.

```
1  #include <stdio.h>
2
3  #define SDATA(index) CUT_BANK_CHECKER(sdata, index)
4  // Simple test kernel for device functionality
5  __global__ void kernel( float* g_idata, float* g_odata)
6  {
7    // shared memory, the size is determined by the host application
8    extern  __shared__  float sdata[];
9
10   // access thread id
11   const unsigned int tid = threadIdx.x;
12   // access number of threads in this block
13   const unsigned int num_threads = blockDim.x;
14
15   // read in input data from global memory
16   // use the bank checker macro to check for bank conflicts
17   // during host emulation
18   SDATA(tid) = g_idata[tid];
```

```
19    __syncthreads();
20
21    // perform some computations
22    SDATA(tid) = (float) num_threads * SDATA( tid);
23    __syncthreads();
24
25    // write data to global memory
26    g_odata[tid] = SDATA(tid);
27  }
```

LISTING 3.5 CUDA device code

After loading the data the kernel just multiplies the thread-id with the number of threads and saves the result back to global memory where the host can pick up the result.

3.6 PORTING STRATEGY FOR GPUS

This section is a high-level overview of the porting process to not only the GPU it can be rather seen as general porting process to parallel hardware. It includes strategies that help through the porting process. It breaks down the porting process into a series of discrete steps.

The steps included here are not the only path through the porting process because the porting process is flexible and has to be adapted slightly for every hardware.

In general, one will follow these steps when porting an algorithm to the GPU

- Algorithm complexity study. Profile the code, find the most compute intensive parts analyze the parts of the algorithm which could be parallellized.
- Data layout/locality and Data flow analysis. Look for independent task or independent data to exploit task or data paralellism.
- Experimental partitioning and mapping of the algorithm and program structure to the architecture.
- Develop CPU control, CPU scalar/multicore code
- Develop CPU control, partitioned GPU scalar code (Communication, synchronization, latency handling)

- Transform GPU scalar code to GPUthreaded code, multi GPU code. Exploit all avaible cores and devices on the system.
- Re-balance the computation / data movement.
- Other optimization considerations (load balancing, bottlenecks...)

4 FEASIBILITY STUDY

Entering a new field in computing without enough exeperience on the platform it is always advisable to get a better knowledge about the hardware ans software before designing anything without proper knowledge. A feasibility study can give an overview if a project is technically feasible and a way to gain some knowledge about the complete eco system. Furthermore the study can answer first questions and help demistify myths.

There are several myths about GPGPU which will be examined and explained Listed below some myths in no particular order.

1. GPU programs are written with a graphics API and layered on top of graphics
2. GPUs can only do a gather and no scatter memory access
3. GPUs are power-inefficient
4. GPUs don't do real floating point math
5. GPUs in average one can only exploit about 10% of the peak performance for general purpose computing
6. GPUs are very wide SIMD machines on which branching is impossible, with 4-wide vector registers

The most interesting myth here is that in average one can only exploit about 10% of the peak performance for general purpose computing, which in turn would lead to that GPUs are power-inefficient in average. Another thing to keep an eye on is the statement that GPUs do not do real floating point math which would exclude many communities (High Performance Computing (HPC), Physics, Astronomy, …) from spending time in doing GPGPU.

Many HPC, Physics, Astronomy,... benchmarks and algorithms rely on floating point math

Therefore the first step before choosing an algorithm or put further energy into research of GPUs is to make the feasibility study. The study will examine the myths and show solutions to the problems respectively statements. Furthermore it will show whether the technology, software eco system exists for building applications on GPUs and how difficult it will be to build.

By the means of a ray tracer the study will show which steps have to be undertaken to gain high performance with an parallel algorithm and show which architectural points have to be considered when designing an application.

4.1 AN OVERVIEW OF RAY TRACING

Ray tracing is a technique for realistic image synthesis. Ray tracing as the name says traces light rays generated from an imaginary camera to their points of origin. Ray tracing is based upon a physical, mathematical model behind light, which facilitates to render photo realistic images.

Ray tracing can be seen as an extension to ray casting. Therefore its easier to understand ray tracing if the base concept of ray casting are understood. The next section will introduce ray casting [1].

4.2 RAY CASTING

Ray casting was first introduced by Appel [2] he developed some techniques for a shading machine for rendering of solids.

First of all it is important to understand the concept of rays. A ray is the path of a particle of light (photon) extending from the eye into the scene [19]. The path is a thin, straight line used to model a beam of light. Each ray can be seen as a *feeler* that reaches the scene and finds out which objects are visible and what color the object has at a specific point. Rays are the fundamental element of any ray tracer.

The ray scans or rasters the scene that is why a ray is often called a feeler

The representation of light on the screen is organized in so called pixels. A pixel is a point sample not a little geometric square. This misconception is widespread and it is an issue that strikes right at the root of correct image computing and the ability to correctly integrate the discrete and the continuous [49].

The color of a given pixel is the color of the light that passes from the object, through the associated pixel into the eye [22].

For each pixel on the screen a ray is cast from the eye through the pixel into the virtual world. Then for each object it is checked if the ray intersects any of them. If there are several objects in a scene intersected by the ray the shortest distance to the intersection point is the one that

1 For an in depth description of ray tracing on a parallel machine see Krnjajic [27]

is visible to the eye. All other intersection are behind the nearest object and not visible respectively occluded by the visible object. The color at that point is the accumulated contribution of intensities radiated from all light sources. This color is given to the pixel through which the ray passed. Ray casting does not consider light reflected or transmitted by other objects in contrast to ray tracing.

Ray casters and ray tracers spent most of their time calculating intersections of rays with different objects. Whitted [54] estimates that anywhere from 75% to over 95% of rendering time is spent in intersection tests. For example an image with 640 pixel width and 480 pixels height, for a total of 307200 pixels, with a medium complex scene with 100 objects results in 30720000 intersection tests. There are well documented ray tracing accelerating techniques not only to decrease the number of intersection tests per ray but also to decrease the number of rays.

In the study the standard ray casting algorithm will be used only, which means that every ray will be intersected with every object to have a high arithmetic density and not to be limited by the memory bandwidth.

Further experiments to extend the basic ray casting algorithm to ray tracing will be discarded this will only add more complexity to the algorithm and will not help in solving the essential problems.

4.3 PERFORMANCE TUNING

After the initial port of the ray tracer to CUDA, the first goal was to make it run on the GPU, it was time to fine tune the run configuration. For any CUDA application it is crucial to take a look at register, shared memory and local memory usage. The performance of the application depends on how good/bad the existing resources are exploited.

The examination of the ray traced showed that the initial port used 48 registers, 28 bytes shared memory and 96 byte constant memory per thread. As shown in Section 3.5 the number of launched threads, blocks and grids is dependent on the three factors mentioned above.

Each Streaming Multiprocessor (SM) has 8192 registers which means one could have 8192/48 = 170 threads to fully exploit the resources 8 blocks should be launched so 170/8 = 21 threads per block which is by far too low. It is easily spotted that the application is limited by registers per SM. These calculations can easily be done with the *CUDA occupancy*

calculator supplied with the SDK.

After fiddling around with compiler switches especially the switch that limits the used registers *-maxrregcount=x* the application used only 14 register which meant we could have a blocksize of 8, 8x8 = 64 threads per block . When reducing the amount of registers used, one has to consider that registers which are additionally needed are allocated from the local memory rather than the register file. Local memory is located in the global memory which has a high latency (200–400 cycles) and registers often referenced have a big impact on performance. The application can become memory bound.

It is always good to have a thread count which is a multiple of 32. A warp consits of 32 threads and the scheduler can easier schedule the threads

The Table 4.1 shows the execution configuration used for the sample runs.

blockSize.x	blockSize.y	gridSize.x	gridSize.y	Regs./Thread
8	8	96	96	14

TABLE 4.1 Execution configuration.

To summarize, just by compiling the application and fiddling with the run parameters one can achieve, in this case, a speedup of about 8 times. See Table 4.2 for several runs varying object count and image size.

Image	Objects	CPU GFlops	GPU GFlops	Speedup
768	338	0.37	3.17	8.54
768	450	0.37	3.22	8.6
768	840	0.37	3.23	8.74
512	338	0.37	3.19	8.53
512	450	0.37	3.18	8.5
512	840	0.37	3.18	8.64
256	338	0.37	3.03	8.09
256	450	0.37	3.03	8.44
256	840	0.37	2.95	7.96

TABLE 4.2 Comparison between CPU and GPU

As it can be seen in Table 4.2 the application runs 8 times faster but when looking at the GFLOPS values there are far beyond that what a

GPU could achieve[2]. The application is only using 0.5% of the peak performance. All performance values are gathered with the NVIDIA Performance Profiler availabe as well with the SDK.

The reasons for this uneffective usage of processing power can be seen in 4.3a and Table 4.4. The application is heavily memory bound. The consequence of reducing the register usage is heavy access to local memory, e.g. there are over 328000000 stores issued to the memory. Furthermore the code is using many branches which is actually reflected by the value 22578032.

Load	Store
63294179	328180560

(A) Local memory

Sum	Divergent
22578032	26834

(B) Branches

TABLE 4.3 Local memory and branches

Another point is the access to global memory. The GPU is able to do coalesced reads and writes when possible otherwise the access is serialized. The application was not ported with coalesced memory reads and writes in mind and hence the application is loading almost everything incoherent. See Table 4.4 for the values.

Load Incoherent	Load Coherent	Store Incoherent	Store Coherent
1115657863	67961	2506752	0

TABLE 4.4 Global memory loads and stores.

The ray tracer could be extended or optimized in many ways but it will be leaved as is. The main point of the feasibility study was to get a feeling for developing on GPUs and to spot major drawbacks and obstacles. Keeping the lessons learned here in mind the development of the main application will be easier. The following section will demystify some but not all myths stated in the beginning.

2 The used GPU is a G92 chip with peak 620 GFLOPS

4.4 DEMYSTIFING THE MYTHS

There were several myths about GPGPU see Chapter 4 that are simply wrong or need to be proven. Beginning with Item 1 this statement is simply wrong. Using CUDA or the ATI SDK *Close To Metal* no developer is anymore forced to use graphics APIs. Furthermore the abandoned graphics API for GPGPUS makes it possible to gather and scatter to the memory, which resolves myth Item 2.

The GPUS are able to do real floating point math. The myth Item 4 comes from a time where floats were only 16 or 24 bit and had no support for *NaN*, *Rounding to zero* and so on. The situation changed completely with the recent development and the switch to the *Unified Device Architecture* where SIMD processing was discarded in favour of more single independent threads.

The last to myths Item 3 and Item 5 are hard to demystify. If one has a algorithm which fits excellently to the GPU the full power of the GPU can be exploited and hence it is power efficient. It depends heavily on the algorithm. The feasibility study has shown that one can achieve easily a speedup but considering the inefficient use of the ressources and the power dissipation it could be a slow down investing so much power for so little return.

5 MEAN SHIFT

In low level computer vision tasks like filtering, segmentation or edge detection, the analysis of data is often not done on the original images. Features like colors are rather projected into a feature space where they can be more easily analyzed. The analysis of the feature space can find interesting attributes of the image like edges or segments.

The feature space has to be smoothed before analysis. Feature spaces originate from real images therefore they are composed of several components from different distributions. The basic approach of a mixture model, a mixture model is a probabilistic model for density estimation using a mixture distribution, is not efficient enough to estimate the density satisfactorily of such complex, arbitrarily comprised densities. The discontinuity preserving smoothing is therefore accomplished with kernel based density estimators. Kernel based density estimators are making no assumptions about densities and hence can estimate arbitrary densities.

The maxima of a feature space correspond to the searched components like the edges of an image. Gradient based methods of feature space analysis are using gradients of the probability density function to find the maxima. Such methods are complex because they need among other things a estimation of the probability of density.

Mean shift is an alternative to the gradient based methods as it is easier to calculate then to estimate the probability of density and then to calculate the gradient. The mean shift vector points to the same direction as the gradient of gradient based methods. Furthermore the *mean shift* vector has a adaptive size and is non parametric. There is no need to supply a step size compared to the other methods. Mean shift a robust approach toward feature space analysis was originally introduced by Comaniciu and Meer [10].

5.1 DENSITY ESTIMATION

In probability and statistics it is known that for different tasks there exist more or less suitable features. In Duda et al. [14] are giving an example of a classification of two different fish types. Features like the length and

brightness are there observed that fit for the task. It is of course possible to find more descriptive features for the fish like the amount of finns but in image processing it would be very expensive and difficult to count such feature. In image analysis specially in real time applications it is important to find suitable features for the task, but also features which are easily visually identified. Color observations are because of their simplicity and for the eye easily to gather important features. The color features can have components from the Red Green Blue (RGB) or gray value color space. Furthermore there are several other color spaces that could be observed like the Hue Saturation Value (HSV) or the L* u* v* (LUV) color space with one luminance and two chromatic components. The LUV color space is often used in computer graphics because of its attribute to be a perceptually uniform color space.

With a finite set of observations follows a finite feature space. The main point of *mean shift* is to find the maxima in the feature space. The maxima of a feature space are all important for *mean shift* applications (filtering, segmentation, …) as the distributions or discontinuities map to clusters or edges of the image.

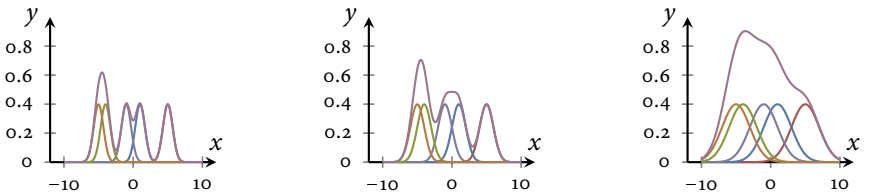
The *mean shift* method is based on the gradient method. For the gradient estimation a function is upon estimations of discrete observations in the feature space. For this kernel density estimators are used also known as *Parzen Window* method.

5.2 KERNEL DENSITY ESTIMATION

Kernel density estimation is a method to estimate an unkown density distribution with finite observations of a point in the sample space. The result of such a procedure is a probability density function that describes the density of probability at each point in the sample space. To estimate the density of a point $x = x_1, \ldots, x_d, \ldots, x_D \in \mathbb{R}^D$ in a D dimensionl feature space, N observations x_1^N with $x_N \in \mathbb{R}^D$ within a search window that is centered around point x have to be observed. The search window with radius h is the bandwidth of the used kernel. The probability density in point x is the mean of probability densities that are centered in the N observations x_1^N.

The effect of different bandwidth parameters h (search window radius) is shown in Figure 5.1. The example shows a kernel density estimation

with five observations $x = 5, 1, -1, -4, -5$ and a gaussian kernel. The total density estimation is the sum of each kernel at a observation, here shown for three bandwidths. With bigger bandwidth h the density estimation becomes smoother.



(A) $h = 0.7$ (B) $h = 1.0$ (C) $h = 2.0$

FIGURE 5.1 Effect of bandwidth selection

Kernel density estimation is a non parametric method, although some parameters exists like the search window radius. Non parametric methods are making no assumptions about the density of probabilities. The strenght of such methods is that they are not limited to just one probability but they can deal with arbitrary coupled/joined probabilities. With an unfinite number of observations the non parametric methods can reconstruct the density of the original probabilities.

To derive the kernel density estimator in point $x \in \mathbb{R}^D$ first of all some definitions have to be made. Let x be a random variable and N observations x_1^N with $x_n \in \mathbb{R}^D$ given. The kernel density estimator $\hat{f}(x)$ in a point $x \in \mathbb{R}^D$, with a kernel $K(x)$ and a $D \times D$ bandwith matrix $\mathbf{H}$ is

$$\hat{f}(x) = \frac{1}{N} \sum_{n=1}^{N} K_{\mathbf{H}}(x - x_n) \tag{5.1}$$

where

$$K_{\mathbf{H}}(x) = \frac{1}{\sqrt{|\mathbf{H}|}} K\left(\frac{x - x_n}{h}\right) \tag{5.2}$$

Since a full parametrized matrix $\mathbf{H}$ would lead to very complex estimates only a single bandwidth parameter, the window radius will be

regarded. With the simplification $\mathbf{H} = h^2\mathbf{I}$ the Equation 5.1 can be written as

$$\hat{f}(x) = \frac{1}{Nh^D} \sum_{n=1}^{N} K\left(\frac{x - x_i}{h}\right). \tag{5.3}$$

The kernel density estimator is valid for several kernels and successive considerations will deal with several kernel the Equation 5.3 will be formated into a more generic form. For this transformation the definition of a kernel and profile of the kernel is needed. The following definition of a kernel and its profile is from Cheng [9]. The norm $\|x\|$ of x is a non negative number so that $\|x\|^2 = \sum_{d=1}^{D} |x_d|^2$. A $K : \mathbb{R}^D \to \mathbb{R}$ is a known as a kernel, when there is a function $k : [0, \infty] \to \mathbb{R}$ the profile of the kernel, so that

$$K(x) = c_{k,d} k(\|x\|^2) \tag{5.4}$$

where K is radial symmetric, where k is non negativ, not increasingly and piecewise continuous with $\int_0^\infty k(r)dr < \infty$. $c_{k,D}$ is a positive normalization constant so that $K(x)$ integrates to 1.

Now the kernel density estimator from Equation 5.3 can be transformed into a new equation. The two indices K and h are representing which kernel and which radius are used for the density estimator. With the profile notation k, where Equation 5.3 is inserted into Equation 5.2, the Equation 5.3 is transformed to

$$\hat{f}_{h,K}(x) = \frac{c_{k,D}}{Nh^d} \sum_{n=1}^{N} k\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \tag{5.5}$$

5.3 KERNEL AND THEIR PROPERTIES

The following section will introduce three univariate profiles and their associated multivariate radial symmetric kernels.

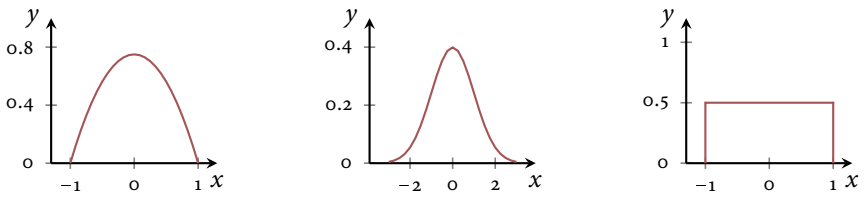
From the Epanechnikov profile

$$k_E x = \begin{cases} 1 - x & 0 \le x \le 1 \\ 0 & x > 1 \end{cases}, x \in \mathbb{R} \tag{5.6}$$

follows a radial symmetric kernel

$$K_E(x) = \begin{cases} \frac{1}{2}c_D^{-1}(D+2)(1-\|x\|^2) & \|x\| \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R}^D \tag{5.7}$$

where c_D is the Volume of the D dimensional globe. The epanechnikov kernel is used often as it minimizes the Mean Integrated Squared Error (MISE) [11]. The Figure 5.2a shows the epanechnikov kernel. The dervative of the kernel is a uniform profile.



(A) Epanechnikov (B) Gaussian (C) Uniform

FIGURE 5.2 Kernel density estimators

From the normal profile

$$k_N(x) = exp\left(-\frac{1}{2}x\right) \text{ where } x \geq 0, x \in \mathbb{R} \tag{5.8}$$

follows the normal kernel

$$K_N(x) = (2\pi)^{-D/2} exp\left(-\frac{1}{2}\|x\|^2\right), x \in \mathbb{R}. \tag{5.9}$$

The normal distribution as every other kernel with infinite support, is often capped for finite support. Finite support is important for the convergence. Capping the normal kernel can be accomplished by multiplying it by a uniform kernel where the inner part of the normal kernel is cut out and weighted with 1 and the outer part is set to 0. The derivative of the normal profile is again a normal profile.

From the uniform profile

$$k_U(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R} \tag{5.10}$$

follows the uniform kernel

$$K_U(x) = \begin{cases} 1 & \|x\| \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R}^D \tag{5.11}$$

which is a hyper unit ball in the origin.

Assuming that a derivative of a profile $k(x)$ exists for all $x \in [0, \infty)$, follows a new profile $g(x)$. Now a new kernel $G(x)$ can be defined as

$$G(x) = c_{g,D} g(\|x\|^2) \tag{5.12}$$

where $c_{g,D}$ a normalizing constant and $K(x)$ the *shadow kernel* of $G(x)$. The term *shadow kernel* was introduced in the context of mean shift in [9]. The mean shift vector of a kernel points to the same direction as the gradient of the shadow kernel (See **??**).

5.4 MEAN SHIFT

In gradient based methods first the gradient s calculated and then the kernel is shifted by a vector with a specific length in direction of a maximum of the probability. The magnitude/length that is the step size of the vector has to be chosen. The problem of such gradient based methods is the choice of a suitable step size. The run time of such algorithms depends heavily on the right choice of the step size. If the step size is too large the algorithm diverges and choosing a too small step size the algorithm becomes very slow. Convergence is only guaranteed for infinitesimal step sizes. There are several complex procedures for finding the right step size, see Comaniciu and Meer [10].

In the case of mean shift there are no additional procedures needed to choose the step size. The magnitude/length of the mean shift vector is the step size which is adaptive regarding the local gradient of the density of probability. Because of this adaptive nature the mean shift algorithm converges (Proof see Comaniciu and Meer [10]).

The advantage of the mean shift method contrary to gradient based methods is that the step size has not to be chosen by hand and the gradient has not to be calculated. It can be shown that the mean shift vector is pointing to the same direction as the gradient and that it moves along the gradient to the maxima that can be seen in **??**. Hence it is

sufficient to calculate the more efficient mean shift vector rather than the gradient.

Given are N D-dimensional observer feature vectors x_1^N with $x_n \in \mathbb{R}^D$ and a kernel G at the point $x = x_1, \ldots, x_d, \ldots, x_D \in \mathbb{R}^D$ in the feature space with search window radius h is

$$m_{h,G}(x) = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \tag{5.13}$$

the D-dimensional *mean shift* vector. The N observations are weighted by the means of kernel G, summed and normalized with the total sum. The *shift* is the difference between the weighted *mean* and x, thus the name *mean shift*.

As pointed out in Section 5.2 the main objective of density estimation is to efficiently find the maxima of a distribution in feature space. The maxima of a function f are located at the positions where the gradient $\nabla f(x) = 0$. With the above stated attribute that the *mean shift* vector always moves along the direction of the gradient, it is a elegant solution for finding the maxima of a distribution without estimating the density.

5.4.1 *Density Gradient Estimation*

The usage of a differentiable kernel allows one to write the density gradient estimator as the gradient of the density estimator

$$\hat{\nabla} f_{h,K}(x) \equiv \nabla \hat{f}_{h,K}(x) = \frac{2c_{k,D}}{Nh^{D+2}} \sum_{n=1}^{N} (x - x_n) k'\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \tag{5.14}$$

where the inner part and a part of the prefactor originate from the differentiation of $k\left(\left\|\frac{x-x_n}{h}\right\|\right)$

$$\begin{aligned} \frac{\delta}{\delta x} k\left(\left\|\frac{x - x_n}{h}\right\|^2\right) &= \left(\left\|\frac{x - x_n}{h}\right\|\right)' k'\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \\ &= 2\,(x - x_n)\left(\frac{1}{h}\right) k'\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \\ &= \frac{2}{h^2}(x - x_n) k'\left(\left\|\frac{x - x_n}{h}\right\|^2\right). \end{aligned} \tag{5.15}$$

Using $g(x) = -k'(x)$ and with Equation 5.12 a new kernel $G(x)$ with profile $g(x)$ can be defined. Transforming Equation 5.14 with the new profile $g(x)$ the gradient of the density estimator becomes

$$\hat{\nabla} f_{h,K}(x) = \frac{2c_{k,D}}{Nh^{D+2}} \sum_{i=1}^{N} (x_n - x)\, g\left(\left\| \frac{x - x_n}{h} \right\|^2\right) \tag{5.16a}$$

$$= \frac{2c_{k,D}}{Nh^{D+2}} \left[\sum_{n=1}^{N} g\left(\left\| \frac{x - x_n}{h} \right\|^2\right)\right] \left[\frac{\sum_{i=1}^{N} x_i g\left(\left\| \frac{x-x_n}{h} \right\|^2\right)}{\sum_{n=1}^{N} g\left(\left\| \frac{x-x_n}{h} \right\|^2\right)} - x\right]. \tag{5.16b}$$

The first term of Equation 5.16b conforms with the density estimator $\hat{f}_{h,G}(x)$ for kernel G (compare with Equation 5.5 for kernel K) except for a factor whereas the second term is the difference between the center of the, with kernel G weighted center of observation and the center x of the kernel window which conforms to the *mean shift* vector from Equation 5.13. To localize the maxima with *mean shift*, the maxima are the roots of the gradient, it firstly has to be shown that the *mean shift* vektor is moving along the direction of the gradient. Inserting $\hat{f}_{h,G}(x)$ and $m_{h,G}(x)$ into Equation 5.16b follows

$$\hat{\nabla} f_{h,K}(x) = \frac{2c_{k,D}}{h^2 c_{g,D}} \hat{f}_{h,G}(x) m_{h,G}(x) \tag{5.17}$$

transformed to $m_{h,G}(x)$ follows

$$m_{h,G}(x) = \frac{1}{2} h^2 c \frac{\hat{\nabla} f_{h,K}(x)}{\hat{f}_h, G(x)}, \text{ whereas } c = \frac{c_{g,D}}{c_{k,D}}. \tag{5.18}$$

The denominator of Equation 5.18 is the normalizing factor that originates from the density estimator with kernel G in x and the numerator is the gradient density estimator with kernel K. In fact the *mean shift* vector is proportional to the gradient which means it is adaptive. Kernel K is the shadow kernel of kernel G. The term shadow kernel was firstly introduced by Cheng [9].

Let be

$$mi_{h,K}(x) = \frac{\sum_{i=1}^{n} x_i k\left(\left\| \frac{x-x_i}{h} \right\|^2\right)}{\sum_{i=1}^{n} k\left(\left\| \frac{x-x_i}{h} \right\|^2\right)} - x \tag{5.19}$$

the D-dimensional mean of the observations x_1^N from $\mathbb{R}^D$ weighted with kernel K and a window radius h. Then is K the shadow kernel to kernel G if the *mean shift* vector with kernel G

$$m_{h,G}(x) = mi_{h,G}(x) - x = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \tag{5.20}$$

lies in the gradient density estimator direction with kernel K

$$\hat{f}_{h,K}(x) = \frac{c_{k,D}}{Nh^d} \sum_{n=1}^{N} k\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \tag{5.21}$$

In the following sections gradients will not more be considered as it was shown in Equation 5.18 that the *mean shift* vector moves along the same direction as the gradient. Instead of estimating a density with a kernel density estimator with kernel K and then calculating the gradient now one can achieve the same solution with the *mean shift* and the differentiation K' of kernel K. The next section will continue with the actual *mean shift* method.

5.4.2 *Mean Shift Method*

The *mean shift* vector moves in direction of the maximal slope of the density, it defines a path to the maximum. The *mean shift* method is described by the following iterations:

1. Choose a window radius h_n for the kernel density estimator in Equation 5.5
2. Choose a start position $y_1 \in \mathbb{R}^D$ for the kernel window
3. Calculate the *mean shift* vector $m_{h,G}(y_j)$ and shift the kernel window G
4. Repeat 3. until convergence

ALGORITHM 5.1 Mean Shift Method

The first step of the algorithm is to choose the window radius or the bandwidth h_n of the kernel in the observation x_n. The bandwidth can be

chosen adaptively or can be fixed. With a fixed bandwidth the density estimator in Equation 5.5 works with identical scaled kernels in every observation or is adapted in every observation. In literature there are several ways described to perform a adaptive Mean Shift (see e.g. Li1 and Klette [30]).

The second step is to choose a start position $y_1 \in \mathbb{R}^D$ in the feature space where to start the mean shift algorithm.

The third step involves shifting the kernel window with the mean shift vector $m_{h,G}(y_j)$. The new position $y_{j+1} \in \mathbb{R}^D$ of the search window in the feature space is calculated with:

$$y_{j+1} = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} \quad j = 1, 2, \ldots \tag{5.22}$$

where $y_j \in \mathbb{R}^D$ the old position of the kernel window is. The algorithm is convergent and if a stationary point is also a convergence point, the point is moved by a small random vector and the algorithm is applied again to the shifted point. This way one can guarantee that the found point is really a convergence point. The convergence attribute and the small trick stated above are described by Comaniciu and Meer in [10].

5.5 FILTERING & SEGMENTATION

The primary use of *mean shift* is filtering (smoothing) and segmentation. A color image can be seen as a 2-dimensional matrix $I \times J$ with N 3-dimensional vectors. The pixels in the image x_n, $n = 1, \ldots, N$ consist of a spatial part $x_n^r = (i, j) \in I \times J$ and a part with a color range $x_n^f = (r, g, b)$ so $x_n = (x_n^r, x_n^f) \in \mathbb{N}^5$ for $n = 1, \ldots, N$. Other color spaces like the LUV color space could be used as well. The euclidean metric is assumed for both spaces.

When applying *mean shift* for filtering and segmentation applications like in Comaniciu and Meer [10] a joined feature space is used for the spatial and range components of a pixel. The color space used in filtering and segmentation is the LUV color space because of its feature of linear mapping. In a joint D-dimensional feature space ($D = 5$ color images, $D = 3$, gray tone images) the different attributes of each space have to be equalized by normalization. Therefore is the joint kernel written as a

product of two kernels with window radius h_r for the spatial part and h_f for the color range part

$$K_{h_r,h_f}(x) = \frac{C}{h_r^2 h_f^p} k\left(\left\|\frac{x^r}{h_r}\right\|^2\right) k\left(\left\|\frac{x^f}{h_f}\right\|^2\right) \tag{5.23}$$

where p is the color dimension of the image, $p = 1$ gray tone and $p = 3$ color. An example of an gray tone image with its feature space is shown in **??**. With the parameter $h = (h_r, h_f)$ the window radius one can specify the size of the kernel and thereby specifiy the resolution of the maximum search.

5.5.1 *Mean Shift Filtering*

Smoothing or filtering with mean shift or bilateral filters have the advantage that discontinuity like edges are preserved. Applying a simple smoothing a weighted average of the neighbors in both space and in color range are considered which systematically excludes pixels across the discontinuity from consideration.

(A) Original (B) Filtered

FIGURE 5.3 Mean shift filtering with parameters $(h_r, h_f) = (6.5, 7)$ applied to a color image

The pixels $x_n, n = 1, \ldots, N$ of the image converge toward their local

density maximum applying the filtering Algorithm 5.2.

Data: $x_n = (x_n^r, x_n^f), n = 1, \ldots, N$ the D-dimensional RGB pixels
Data: $c_n = (c_n^r, c_n^f), n = 1, \ldots, N$ the D-dimensional LUV pixels
Data: $z_n = (z_n^r, z_n^f), n = 1, \ldots, N$ the D-dimensional filtered pixels
Data: $o_n = (o_n^r, o_n^f), n = 1, \ldots, N$ the D-dimensional output pixels

for $n = 1, \ldots, N$ do
  convert x_n from RGB to LUV color space
  $c_n = x_n$
end

for $n = 1, \ldots, N$ do
  initialize $m = 1$ and $y_n = c_n$
  while *not converged* do
    calculate $y_{n,m+1}$ according to $y_{j+i} = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}$
  end
  assign $z_n = (x_n^r, y_{conv}^f)$
end

for $n = 1, \ldots, N$ do
  convert z_n from LUV to RGB color space
  $o_n = z_n$
end

ALGORITHM 5.2 Mean Shift Filtering

All pixels that converge to the same um are lying in the basin of attraction of this maximum. The filtered image points $z_n, n = 1, \ldots, N$ keep there spatial coordinates but obtain the color values off the convergence point. The convergence points are found by moving the kernel window with *mean shift* in direction of maximum slope of the spatial range feature space. The sample Figure 5.3 was smoothed with the algorithm of Algorithm 5.2. A uniform kernel with $h = (6.5, 7)$ was used.

5.5.2 *Mean Shift Segmentation*

Image segmentation is a method to partition an image into homogeneous regions. Searched are regions with similar colors like a wall, lawn and clothes. The found areas are associated with the same color values. The segmentation can be seen as a strong smoothing where the edges are preserved.

The segmentation algorithm is a extension of the *mean shift* filtering algorithm. After applying the filter and all convergence points are found, clusters are build out of them. All convergence points that are closer then h_r in the spatial domain and that are closer then h_f in the range domain are grouped together. In the end all points are labeled after their cluster assignment.

(A) Original (B) Segmented (C) Region boundaries

FIGURE 5.4 Mean shift segmentation with parameters $(h_r, h_f, M) = (6.5, 7, 20)$ applied to a color image

The Algorithm was applied on the image Figure 5.4a. A uniform kernel with a kernel radius of $h_r = 6.5$ and $h_f = 7$ was used. M is a parameter for the last step of the algorithm where regions where the pixel count is smaller than M are purged. M defines the smallest significant feature size. Figure 5.4 shows the result of a segmentation. After the image segmentation follows a edge detection by examining the cluster regions Figure 5.4c.

In filtering as well as in segmentation algorithm a fixed sized window size is used. In Comaniciu and Meer [10] ist is noted that for segmenation the algorithm is not heavily dependent on the choice of the kernel

parameters where as there are application where the window size matters. The segmentation part of the Algorithm 5.3 has no significant impact

on the run time.

Data: $x_n = (x_n^r, x_n^f), n = 1, \ldots, N$ the D-dimensional RGB pixels
Data: $c_n = (c_n^r, c_n^f), n = 1, \ldots, N$ the D-dimensional LUV pixels
Data: $z_n = (z_n^r, z_n^f), n = 1, \ldots, N$ the D-dimensional filtered pixels
Data: $o_n = (o_n^r, o_n^f), n = 1, \ldots, N$ the D-dimensional output pixels

for $n = 1, \ldots, N$ do
  convert x_n from RGB to LUV color space
  $c_n = x_n$
end

for $n = 1, \ldots, N$ do
  initialize $m = 1$ and $y_n = c_n$
  while *not converged* do
    calculate $y_{n,m+1}$ according to $y_{j+i} = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}$
  end
  assign $z_n = (x_n^r, y_{conv}^f)$
end

for $n = 1, \ldots, N$ do
  delineate in the joint domain the clusters $\{C_p\}_{p=1,\ldots,P}$ by grouping together all z_n which are closer than h_s in the spatial domain and h_r in the range domain.
end

for $n = 1, \ldots, N$ do
  assign $L_n = \{p | z_i \in C_p\}$
end

eliminate spatial regions containing less than M pixels

for $n = 1, \ldots, N$ do
  convert z_n from LUV to RGB color space
  $o_n = z_n$
end

ALGORITHM 5.3 Mean shift segmentation

6 MEAN SHIFT ALGORITHM ANALYSIS

Before any porting can be started the developer has to find out if there are multiple activities or tasks which can run simultaneously which expose exploitable concurrency. The developer has to find concurrency either by decomposing the data or tasks. By this decomposition one wants to solve bigger problems in less time as several processing units can solve different parts of the problem.

But before any analysis is started one has to know if the problem is large enough and if the resulting speedup justifies all the effort that is expended on making a parallel version out of it. In case of mean shift which is used for several things like filtering, segmentation, pattern recognition and real time tracking one can deduce that for bigger images or many images the computation time climbs fast as the dsize or the number of images rises. To have a clue how the mean shift algorithm behaves with big pictures several run times where recorded. For the analysis of the mean shift algorithm a ready to use Edge Detection and Image SegmentatiON System (EDISON) was used that was profiled and modified and parallelized for the purpose of this thesis.

The EDISON [1] system which was developed by the authors Comaniciu and Meer of [10] offers functionality to filter, segment and detect edges in images.

The Figure 6.1 shows results of CPU run times of the EDISON application dependant on image size. The vertical axis shows the run time in seconds and the horizontal axis shows the side length in pixels of the quadratic Figure 5.3a. Considering the numbers in the result one can see that the run times grow linear to the image sizes. The mean shift algorithm has linear complexity, hence its complexity can be written as $O(n)$. Doubling the side length of the quadratic picture the run time increase by a factor of 4. See e.g. the run times for side length: (l=256, t=9) and (l=512, t=36).

But this is obvious, as stated in Chapter 5 for each pixel of the image the mean shift vector has to be calculated, and if one increases the number of pixels by a number n we have to deal with n times longer run time. Such consideration have to be done in the run-up to have a clue to which

1 http://www.caip.rutgers.edu/riul/research/code/EDISON/index.html

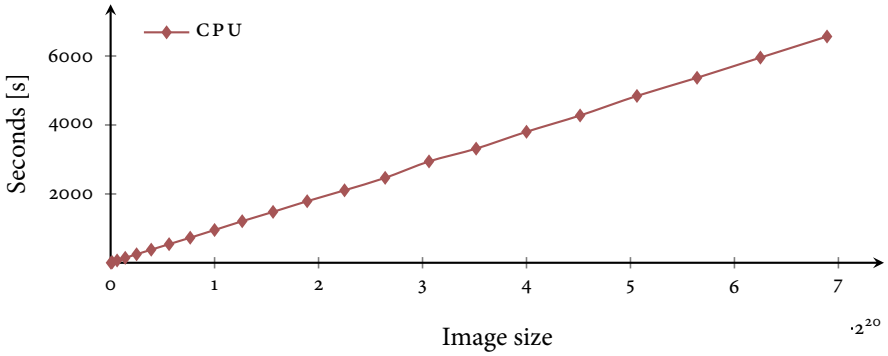


FIGURE 6.1 CPU run time depending on the image size

point the algorithm can be accelerated. The problem has to be well understood.

The next important step is to find parts which can be offloaded to an accelerator as the GPU. A first good point for such parts is to make a profiling of the application to find out the most computationally intensive parts. It makes no sense to parallelize functions which contribute only 1% to the run time.

6.1 PROFILING THE ORIGINAL CODE

A good start point is to take a profiler und generate a profile of the functions, recording their run time and call history. In this case a statistical profiler is used which operates by sampling. The samples are taken from the hardware performance counters which every modern CPU has builtin. OProfile[2] a system wide profiler for Linux was used to examine the run time of the EDISON program. The Table 6.1 shows the run time analysis of EDISON.

Before taking a closer look at the table, the columns have to be explained. The column *Total* shows how long the function plus their child function were executing. Whereas the column *Self* shows how long the function spent executing itself without the execution time of their child functions. The column *Symbol* shows the function that was executed.

Now having a look at the table there is one function which is executing

2 http://oprofile.sourceforge.net/news/

Self (%)	Total (%)	Symbol
0.00	100.00	Segment()
0.00	100.00	meanShift(int)
0.00	99.30	Filter(...)
0.40	99.10	NonOptimizedFilter(...)
0.60	98.80	LatticeMSVector(double*, double*)
98.20	98.20	uniformLSearch(double*, double*)
0.00	0.40	FuseRegions(float, int)
0.10	0.30	BuildRAM()
0.00	0.20	TransitiveClosure()

TABLE 6.1 EDISON run time analysis

98.2% of the run time, the function *uniformLSearch(double*, double*)*. In this function the algorithm tries to find feature points that fall into the search window with radius h (see Section 5.4). This is the starting point as it is the most computationally intensive and the focus for parallelization. There is a way to calculate to which extent the particular function can be parallelized and which speedup one can expect. Speedup is the ratio of run time when executing a program on a single processr to the run time when using n processors. Given T_1 the run time of a program on a single processeor and T_n the run time of the same program on n processors then

$$S(n) = \frac{T_1}{T_n} \tag{6.1}$$

is a measure for the speedup. One familiar law to calculate the how much speedup can be obtained through paralellism is Amdahl's law.

6.2 AMDAHL'S LAW

Amdahl's law says supposing that 80% of a computation can be parallelized and 20% can't, then even if the 80% that is parallel were run on an infinite number of processors the highest speedup that one can achieve is 5. Generally speaking if a fraction p of a computation can be run in parallel and the rest must run serially, Amdahl's law upper bounds the speedup by $1/(1-p)$.

The speedup of a application according to Amdahl is given by

$$S_{max}(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{6.2}$$

where 1 is the execution unit time of the old computation, $(1-p)$ the inherently serial part and p/n the parallel part divided by n the number of processing units. When $n \to \infty$ then the execution time approaches the time for executing the sequential program fraction. So no matter how many processors one adds to the system it will at least execute as long as the sequential program fraction, which is an upper bound of speedup. A important addition is that the sequential program fraction or serial processing percentage is relative to the overall execution time using a single processor. It is independent of the number n of processors Shi [47].

Assuming that the mean shift filtering can be parallelized with any number of processing units one can calculate the maximum speedup achieveable according to Amdahl. Taking the parallel processing percentage from Table 4.2 for the filtering step:

$$p = 99,3\% = 0.993p$$

we get

$$S_{max}(n) = \frac{1}{(1-0.993)} = \frac{1}{0.007} = 142.857$$

Applying the calculation with $n = 128$, the used GPU has 128 processors one can acheive a speedup of:

$$S_{max}(128) = \frac{1}{(1-0.993) + \frac{0.993}{128}} = \frac{1}{0.007 + 0.0077578125} = 67.76$$

One can see that even for very small serial processing percentage the speedup is not very high. Therefore Gustafson proposed a alternative formulation where the fractions are now dependent on n. He assumed that for larger problem, the fraction of a program to parallelization increases. Thats why Gustafson's law is often referred as a scaled speedup measure and Amdahl's as a non scaled speedup measure. Given the

serial s' and paralllel time p' a single processor would require $s' + p' \times n$ time to finish the execution. The speedup is then given by:

$$S_{max}(n) = \frac{s' + p' \times n}{s' + p'} = n + (1 - n) \times s' \tag{6.3}$$

Applying the calculation with $n = 128$, the used GPU has 128 processors one can acheive a speedup according to Gustafson of:

$$S_{max}(128) = 128 + (1 - 128) \times 0.007 = 127.11$$

It looks like that Amdahl's & Gustafson's law are in contradiction but looking more precise on both definitions one can see that both laws employ different definitions for the fraction of the serial and parallel exectuion times. Amdahl uses non scaled percentage and Gustafson scaled percentage. Mathematicaly they are equal just two different formulations. The scaled percentage can be transformed to a non scaled perecentage where Gustafson's law gives the same results as Amdahl's law [47].

In summary one can expect more than a 100 fold speedup when parallelizing the filter step of the mean shift algorithm. The next steps will focus on analyzing how the mean shift filter can be decomposed to take advantage of multiple processors.

6.3 DATA & TASK PARALLELISM

There are two ways to decompose an algorithm to take advantage of parallelism. The first way is to decompose the algorithm into several tasks that can run independently. Each task is performing independently different calculation on the data. This characteristic is known as task parallelism. To know if two task can run independent one can take Bernstein conditions and evaluate them.

Let P_i and P_j be two tasks of a program P. For P_i let I_i be the input and O_i the output data and for P_j let I_j be the input and O_j be the output data, then P_i and P_j are independent if they satisfy the following conditions [47]:

$$I_j \cap O_i = \varnothing \tag{6.4a}$$

$$I_i \cap O_j = \varnothing \tag{6.4b}$$

$$O_i \cap O_j = \varnothing \tag{6.4c}$$

The first two formulations Equation 6.4a and Equation 6.4b are stating that the input of one task has to be independent from the other task. Otherwise one task would have to wait for the output of the other task to continue. Furthermore not only the inputs vs. the outputs have to be independent but also the outputs of each task have to be independent. If the above conditions cannot be applied to a program P than it is likewise that race conditions and poor performance can arise through e.g. excessive synchronization overhead.

6.3.1 *Task Parallelism*

Equipped with the knowledge how to identify independent parallel tasks, the mean shift segmentation algorithm Algorithm 5.3 can now be examined. Having a look at the algorithm steps one can easily see that every task is violating every condition stated above. The filtering step input data is dependent on the RGB to LUV conversion output data. The segmentation step input data is dependent on the filtering output data. So it does not matter how big an image is or if it is a color or grayscale image the mean shift segmentation algorithm will always perform all from each other dependent calculations as described in Algorithm 5.3. The longest path of dependent calculations is the critical path. For the mean shift algorithm there exists no shorter path.

In summary one can discard the idea of task parallelism for the mean shift segmentation algorithm. Thats why the focus of parallelization is now on data parallelism.

6.3.2 *Data Parallelism*

The second way of decomposing an algorithm is data decomposition. The data is decomposed into chunks on which similar operations are being applied in such a way that the different chunks can be operated on concurrently. The focus in data parallelism is on the data structures which define the problem and not at the tasks.

Focusing on data and having a look at the most computational intensive part (the filtering step) one can see that each pixel of an image can be calculated independently. The input and output pixels are independent and furthermore it doesn't matter at which pixel one starts, the filtering

step is deterministic, starting from the same input will lead to the same output.

Another important aspect is how often the subtasks (where each subtask is calculating one pixel) have to synchronize or communicate. An algorithm exhibits fine-grained parallelism if the subtasks have to communicate many times, coarse-grained parallelism if the subtasks do not communicate many times. In this case the algorithm even exhibits embarrassingly parallel parallelism. The subtasks never have to communicate or synchronize. Each subtask takes one or a chunk of pixels applies the filtering steps and writes back the result. The intermediate steps where the mean shift vector is moved over the spatial space are also independent for each pixel.

The approach in parallelizing the mean shift algorithm is to use a data decomposition where each subtask is filtering a chunk or a pixel of the entire image. Depending on the number of processing units one can choose an appropriate chunk size to exploit every unit.

If the algorithm would only exhibit fine-grained parallelism the next steps would be to identify groups to simplify the job of managing dependencies and have a look at the ordering to satisfy constraints among tasks. Luckily here one has only to deal with a embarrassingly parallel algorithm and can move to the next step, identify how data is accessed and shared among subtasks.

6.4 DATA FLOW

It is important to understand that inefficient data access can lead to very poor performance on every processing unit especially on a GPU. Inefficient data access leads to an algorithm that is memory bound which means it doesn't matter how much computational power the processing unit has it will be bound to the memory performance (See Chapter 2 for *arithmetic intensity*). Therefore its crucial to understand the data and optimize it for access. If done incorrectly tasks may get invalid data or it could lead to excessive synchronization overhead.

In Section 6.3.1 it was shown that the filtering step is dependent on the color conversion step where the output data of the color conversion step becomes the input data of the filtering step. Since this is done sequentially there is no need to take care of some synchronization. This

scheme is the same for the filtering and segmentation step. Since the filtering step is done in parallel and the segmentation needs the output data form the filtering step a barrier after the filtering step is needed.

Now to the interesting part the parallel filtering step. For the following considerations the viewpoint will be a single processing unit. It is firstly enough to consider only one unit and one pixel at a time since the GPU exhibits a SPMD programming model (see **??**) and secondly the mean shift algorithm is embarrassingly parallel. It is later easy to broaden the examination to a cluster of processing units and chunks of pixels if its needed.

A processing unit firstly reads the corresponding pixel $x_n = (x_n^r, x_n^f)$ from the image. Then it moves the mean shift vector to the next pixel on the path to the convergence point $z_n = (x_n^r, y_{conv}^f)$ by finding the feature vector which falls into the search window. This new pixel is the basis for the next iteration—calculation of the mean shift vector. On average of about 15–20 iterations the mean shift vector converges to the mean at the convergence point [57]. The whole path where the mean shift vector is moving on is unpredictable. The vector can move over the whole image or a few pixels, this is an important fact to consider when mapping the algorithm to the GPU because random access to the memory can decrease the memory performance significantly. At last the convergence point is saved to the $z_n = (z_n^r, z_n^f)$ data structure.

The segmentation step as stated above needs the data for the filtering step. The only parallel portion of the algorithm is the filtering step so no further considerations will be made.

6.5 SUMMARY

So far the problem has been extensively analyzed. The most compute intensive parts have been identified and a perliminary speedup has been calculated according to Gustafon's law. Furthermore the problem has been decomposed into tasks that can execute concurrently using task respectively the data decomposition. Additionally to the decomposition the dependencies between processing units has been identified.

The items mentioned above will guide the design of the algorithm in the next phase. Therefore it is important to have clearly understood the algorithm and have done the decomposition in the right way.

The decomposition has to be suitable for the target platform. Since the GPU is a SPMD architecture, which means one program the mean shift algorithm is operating on many data the pixels. Above it was shown that the way of data decomposition is preferrable because (1) the pixels can be mapped 1:1 to a processing unit on the GPU and (2) there are enough pixel to keep the GPU busy. Furthermore the GPU is a number cruncher and algorithms doing heavy math or calculations can profit from the sheer computational power.

The decomposition has to be flexible, efficent and simple. A flexible setup means that the amount of independent task can vary and the configuration how they are arranged is as well selectable. This means it can be adapted to from one to many processing units, the load can evely be distributed among the processing units which leads to good efficiency and scalability and the decomposition can be parameterized in terms of units and problem size. Additionally weak dependency in the decomposition makes load balancing easier. Simple in this case means minimizing the synchronization and communication overhead through the decomposition, which is in this case non-existant.

This chapter exposed the concurrency in the mean shift algorithm that can now be taken into the next phase, designing the algorithm.

7 MEAN SHIFT ALGORITHM DESIGN

The previous chapter explained and showed what is gonna be done to the algorihtm to have it running on a GPU. In this chapter it will be explained how all the proposals are realized and how everything is put together. The result is an scalable and efficient realization of the mean shift algorithm on the GPU.

The next step involves the structuring of the algorithm to refine the design going further into detail and move it closer to a program that can run on the GPU. The main concern here is how to map the concurrency identified in the previous chapters to the processing units of a GPU. Usually one wouldn't go to much into hardware details in this phase because the design should be flexible und portable, but without considering any hardware details of the GPU the algorithm is unlikely to run efficient. Therefore each design decision made is based upon observance of hardware and programming model (CUDA) features. The strategy taken here is start from high level elements of the programming model and go ever further into detail until the point is reached where the algorithm can be implemented.

7.1 PROGRAMMING MODEL

In Section 3.4 it was explained that applications are accelerated by executing data parallel portions of the algorithm on the GPU as *kernels* which run in parallel on many threads exposing the SPMD nature of a GPU. Additionally in Section 6.3.2 the parallel portion of the algorithm was identified and hence the filtering step will be implemented as a CUDA kernel where it operates on different data simultaneously. There is a strong dependency between the steps of the alogrithm shown in Section 6.3 that have to be considered here as well. Before and after the filtering a color conversion from RGB to LUV and back is needed. These steps are not very compute intensive but why are they considered here?

An example is taken to have some numbers to juggle with. Supposing the algorithm is split in 3 parts and the main part is executing 100 seconds and the two other parts are executing 2 seconds. Additionally supposing that the main part can be accelerated by a factor of 100 which means

the main part takes now 1 second, the most computational intensive part is not more the main part but rather the remaining two parts. That means no matter how much faster the main part will be the program will need always at least 4 seconds + accelerated main part to execute. The solution to this problem would be to accelerate the two remaining parts as well and thats what will be done for the mean shift algorithm. The two color space conversion steps will be as well offloaded to the GPU. With bigger and bigger images the CPU will need more and more time to convert from one color space to another.

All the other portions will be run on the CPU because there are not(-yet) a significant part of the run-time and are mostly postprocessing steps for the actual algorithm. The CPU will prepare, provide and gather the data and results from the GPU.

7.2 THREAD BATCHING

Fine grained, data parallel threads are the key concept of CUDA. Several of these threads are grouped together to a thread block. Thread blocks are again grouped to so called grids and kernel is as a grid of thread blocks where data memory space is shared by all threads. The grid and the thread block can be organized either as a 1D,2D or as a 3D array, dependent on the problem structure adjusting to the state of the original data. This means that e.g. if the problem is 3D one could use the 3D configuration or as in the mean shift algorithm case where the problem is a an 2D image than obviuosly one should use the 2D array configuration.

Therefore the kernel used will be a 2D grid of 2D thread blocks. The grid represents the whole image and the thread blocks are the image chunks that are updated in parallel where a single thread is working on a single pixel. The exposed parallelity is directly mapped to the problem.

For a efficient execution several x,y combinations for the 2D array of the thread blocks should be evaluated. The grid size in x and in y is fixed since that are the dimensions of the analyzed image.

Since the algorithm is embarassingly parallel there is no need to look for communication and sychronization mechanisms in CUDA.

7.3 DEVICE MEMORY SPACE

The memory space is a hierarchy of several memory types that can be accessed per thread, block, grid and the host. The first most important memory is the global memory that is the main data exchange place between the CPU and GPU. For the further considerations other memory types are not taken into account because (1) there are implementation specific and (2) using different memories for the algorithm would not change the original nature of organization and decomposition. They can be later used to increase performance and data throughput.

Coming back to global memory it is mainly used to read in the input data and store the results which can be later either visualized using a graphics library or transferred back to the CPU where it can be further processed.

Therefore the CPU will run all preprocessing steps to the image and transfer it to the global memory on the GPU. After the filtering step where the GPU is reading continuosly from the global memory for each iteration of the filtering step the CPU will fetch the data and postprocess it. Further interaction between CPU and GPU are non-existent.

7.4 WARPS

On the GPU 32 threads build up a SIMD unit in NVIDIA terms called warp, in that the same instruction is executed at a time but on different data where on a CPU the threads are scheduled independently.

The consequence out of this is that if there is a branch the execution of the two branch paths, when both are executed, are serialized. The only branch that can happen in the mean shift algorithm is that several threads have found there basin of attraction and other are still calculating. In this case there will no serialization happen rather there are threads that are idle.

This branching or dividing of a warp into idle and active threads is impossible to circumvent. Because each image has different densities and the locations of the densities vary there have to be threads that are idle, those e.g. that are close to a density maximum and finish the calculation in few iterations, and threads that are active, those that were further apart of the density maximum and need a higher number of

iterations. This means threads in a warp that are gone to idle state will stay idle and active threads will do their calculations no rearranging of threads or reassignment of data will be done.

In summary it can be said that each thread is executing the same program. The execution time of each thread highly depends on the location of the thread in the grid and the location of the density maximums. The different execution times are visible directly in the final iteration count.

7.5 PROGRAM FLOW

The point of view will be single thread since all threads are doing the same. After the CPU has loaded the picture it preprocesses it by converting the RGB pixel values to LUV. After the conversion the CPU transfers via DMA the image to the GPU.

A thread in the grid having a unique id can now fetch the corresponding pixel according to the id. Equipped with the color data the thread can now calculate the mean shift vector until convergence. While calculating the thread will access various pixel in the image that are read-only in the global memory. The found convergence point is saved to a new buffer in the global memory.

Finally the CPU fetches the convergence points, converges the image from the LUV color space to the RGB space. The postprocessing step involves the segmentation and pruning of regions smaller than a specified variable. The result is a segmented image.

7.6 SUMMARY

After an extensive analysis it was no problem to put the single parts together to form a program. The design phase usually does not consider any hardware but in this case where an efficient algorithm should be mapped to the GPU it is crucial to know the architecture and the features the hardware offers. Furthermore it is important to have a look at the data. A good design follows from good data design. In this case a data decomposition was choosen and hence the program had to accomodate to this circumstance. The data decomposition was not choosen arbitrarily it was choosen with the ulterior motive that the data fits $1:1$ to the hardware features.

In summary, data is more important than code, because crappy data design leads to low bandwidth usage and the computational units starve because the data is not in place (memory bound). Writing a good algorithm means to understand the data.

8 IMPLEMENTATION

The EDISON system that was already used in Chapter 6 for analysis was the base for the implementation respectively the porting process. EDISON is build around the mean shift method and offers functionality to filter, segment images and edge detection. The complete software is rather complex having a scripting engine, Graphical User Interface (GUI) and different versions of the filtering step, where each step is introducing some technique to accelerate the filtering. Furthermore the filtering was construed to analyze feature spaces with arbitrarily dimensions, different kernels, weight maps and so on.

Since the focus of this work is on image segmentation only the filtering functions and some segmentation helper functions were taken from EDISON as reference. The image loading, saving, pre-, postprocessing and the filtering part, which is the most compute intensive part with 99.1% (from Table 6.1) of the run-time, were completely rewritten whereas the clustering, labeling and elimination of spatial regions (see Algorithm 5.3) with in sum 0.9% of the run-time were adopted without change.

As mentioned before there are three algorithms of the filtering step. For the purpose of this work the standard mean shift algorithm was choosen because (1) it is the most compute intensive and (2) it has the best quality output. The other two versions of the algorithm induce a quality loss and use structures to accelerate the algorithm that are not well suited for the GPU. Additionally the compute power of a GPU is more visible with the first version.

8.1 REFERENCE IMPLEMENTATION

The first step was to do a reference implementation on the CPU. The CPU programming framework has better debugging and logging facilites than a GPU. It is not possible to do a system call like printf, fopen on a GPU which makes debugging hard and logging rather impossible.

The idea behind the reference implementation was to have a running code which is doing the segmentation in the *right* way. The advantage of this is that when porting to the GPU there is always a reference to which

the results of a GPU run can be compared too. This way it is assured that the algorithm is doing the right thing. It will be more crucial when optimization techniques were used. It has to be verified that they are not breaking the algorithm. How this reference implementation was applied can be seen in **??**. Another positive effect is that now there is a implementation against the GPU, where the run-times of each implementation can be compared and speedups calculated.

The rest of this chapter is splitted into two parts. The first part describes the host part, its that part which the CPU is executing and the second part describes the device part where the CUDA kernels were run.

8.2 THE HOST PART

The host part takes care of preparing and data proivisioning. The first thing to do is to load the image into a apropriate format into the memory. Remembering the lessons learned from the feasibilitiy study in Chapter 4 about global memory access the loaded image should be arranged in such a way that threads getting the data can do that in a coalesced manner.

Luckily the CUDA helper functions library ''cutil'' has image loading function that can align the data in memory for efficient access. The functions in question is cutLoadPPM4ub() which reads the Portable Pixel Map (PPM) file which is in RGB (0–255) format and pads a zero to the fourth byte. The effect of this padding is that coalescence and better performance is achieved by accessing 1, 2 or 4 consecutive memory locations simultanously (see Chapter 9 for more optimizations).

Another thing to consider is that the execution configuration should be configurable and further parameters that are crucial for performance like the number of GPUs, thread configuration, device selection, ...are implemented as commandline parameters.

The CPU is responsible for calling the CUDA kernels. In this implementation there are three of them, the first one is a kernel that is converting from RGB to LUV color space, the second is the filtering kernel and the last one is converting from LUV back to RGB. The color space conversion kernels will not be considered further because there are rather simple and can be looked up in any good computer graphics book. The atttention is on the filtering kernel that is explained in the device part (Section 8.3).

Before any kernels are started several input and output buffers were allocated in the CPU and GPU memory space. Alignement of the buffers is crucial for high bandwidth transfers thats why the allocating functions from CUDA are aligning the buffers dependent on the data type per default. Right before the kernel execution the input data is transferred to the GPU buffers and after execution the results are transferrred back to the CPU via the GPU DMA engine. This doesnt mean that for each kernel the CPU is transferring the data back and forth, this is done only one time. After the first kernel finishes execution the data can reside in the memory of the GPU for the next kernel. Memory stays consistent across kernel calls and the subsequent kernel needs the preprocessed data from the previous kernel. This coupling and strong dependency of the kernels or steps was already described in Section 6.3 and effected the design Chapter 7. When the last finishes execution the host can fetch the results for saving and result verification.

Progress & Result Verification

As mentioned in Section 8.1 there exists a CPU and a GPU implementation of the algorithm. When executing the program it is possible to supply a parameter which triggers the result verification. The images saved by the CPU and GPU are compared pixel by pixel. For this comparison the great tools from ImageMagick a software suite to create, edit, and compose bitmap images, are used.

The compare program can mathematically and visually annotate the difference between an image and its reconstruction. The tool emphasizes pixels that are different between the CPU and GPU version with a semi-transparent red overlay, whereas a white overlay de-emphasizes pixels that are the same between the two versions. An example of such an comparison can be seen in Figure 8.1.

This way it was assured that any modifications or optimizations were done correctly and not changing the natural behaviour of the algorithm.

8.3 THE DEVICE PART

For the device part only the filtering step will be further described. As said before the color conversion is rather simple and can be looked

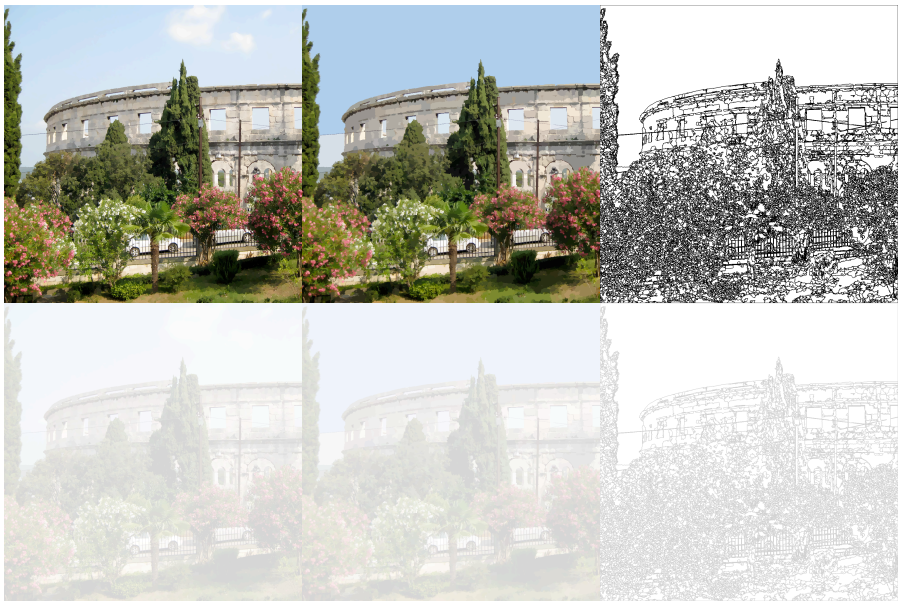


FIGURE 8.1 Result verification. The first row shows the reference pictures computed by the CPU, (1) the filtered, (2) the segmented image and (3) the edges. Below the annotations made by ImageMagick. In this case every pixel computed by the GPU matches the reference

up in any good computer graphics book. It can be used with minor modifications on the GPU.

The kernel is a single source-code image that runs on each thread on the GPU. The first thing that is done in the kernel is to obtain a unique identifier. The identifier is obtained with special functions calls from the hardware. This unique identifier allows different threads to make different decisions during kernel execution. The assignment of the unique id to the hardware threads is deterministic. Thread zero is always the first hardware thread, one can be for sure that actions which have to be done by a certain thread are really done by the corresponding thread in the hardware.

The unique id is as well important for data distribution. In Section 7.2 it was shown that one thread is responsible for a single pixel. So there are so many threads as there are pixel in the image where the threads are numbered from 0–(number of pixel - 1). Every thread having its unique

id which is at the same time the position in the array of pixel can now easily fetch the pixel from global memory for computation.

Each thread takes the LUV values and calculates the mean shift vector until convergence. When a thread finishes execution it saves the convergence point to a second buffer to the same place in the array (unique id) where the thread fetched the input pixel.

For the first shot no other memories where used to improve performance. The first implementation was just to make it work. The next chapter will digg deeper into the hardware and optimize the code to run efficiently. The performance and scalability is evaluated in a subsequente chapter.

8.4 SUMMARY

After an extensive analysis and a good design the effort to implement the algorihtm was not so big. The most difficult part was the implementation of the mean shift method in the reference implementation. It had to be verified that the algorithm is working and doing the right stuff.

After that it was crucial to get in touch with CUDA and all dependent libraries and to know what function is exactly doing and how to use it.

Merging the reference implementation with CUDA was pretty easy as CUDA is just a extension to C and only some keywords where introduced that have to be used in the right way.

The programming flow of first moving data to the GPU, processing and then getting the data back to the host was already known, from implementing a raytracer on the Cell processor. Pretty all accelerators have a similar programming flow.

Luckily there was the reference implementation which was used for result verification not only for the filtering part but as well for the other two kernels where a bug was easily spotted in form of red pixels thanks to the great tools of ImageMagick.

9 OPTIMIZATION STRATEGIES

The following chapter will present optimization strategies which were used to accelerate the mean shift image segmentation. The presented strategies are not only valid for CUDA, they can be applied to many parallel machines which are build after the shared memory or SPMD model. The chapter is divided in several sections. Each section will present an optimization. The are major and minor optimization that are examined where mostly major optimization give higher speedup whereas minor optimization give lower speedup.

Many of the optimization are hardware centric, which means that one has to have a good knowledge about the architecture to understand which steps can be undertaken to accelerate the algorithm and which where even possible. Each section will cover the architectural specialty which led to perform the specific optimization. The first optimization of the next section is a general rule when trying to accelerate an algorithm, offload compute intensive parts. The succeeding sections respectively optimizations are in chronological order and where applied in that order, identifiable by the ever increasing speedups.

9.1 TEST & BENCHMARK CONFIGURATION

For measuring the run-time the CUDA timers are used. They have a resolution of approximately half a microsecond. The timings are measured on the GPU, and hence they are operating system independent. Further benchmark metrics like bandwidth, coalesced or uncoalesced access to global,shared and local memory or time spent in device functions were reported by cudaprof a profiler for CUDA programs. OProfile was used on the host side for profiling the program.

For the measurement a image with a dimension of 256×256 pixel was used and with the following parameters: sigmaS = 7, sigmaR = 6.5 and minRegion = 20 (minRegion is used for pruning regions smaller than 20 pixel).

9.2 OFFLOAD COMPUTE INTENSIVE PARTS

As shown in Chapter 6 before any analysis is started one has to know if the problem is large enough and if the resulting speedup justifies all the effort that is expended on making a parallel version out of it. This way the most compute intensive parts are identified and these functions are candidates for offloading.

In this case the most compute intensive part is 98.2% of the complete run-time and hence a valuable part to offload. All other parts have no significant part of the run-time but when optimising it can happen that these other part grow in terms of run-time and the before most compute intensive part with a very high percentage drops to a low percentage and hence the percentages of all other parts increase and become very compute intensive.

In other cases the run-time is often distributed among several parts and in this case it should be tried to offload as much as possible of the computational parts to an accelerator as the GPU. In the first place the focus will be on the filter part that is executing in total 99.1% of the run-time.

For the following considerations about speedup the Equation 6.1 will be used to calculate the speedup.

In **??** it was shown how to design and to implement the algorithm to offload the important parts to the GPU and the first naive implementation of the mean shift filter in CUDA resulted in a speedup of

$$S = \frac{T_{CPU}}{T_{GPU}} = \frac{9616,77\,\text{ms}}{1932\,\text{ms}} = 4,97762$$

times faster than the CPU version. The reference time was generated from EDISON. It reports timings for filtering and segmentation. The first implementation was just a naive implementation without regarding the hardware restrictions and without exhausting the hardware potential. In Section 6.4 it was shown that in every iteration of calculating the mean shift vector the algorithm fetches color values for every pixel contained in the search window. The search window is defined by sigmaS the spatial window and sigmaR the range window. For now, only sigmaS is important since that value spans the spatial window over the pixel (see Listing B.1).

9.3 GLOBAL MEMORY & COALESCING

In Section 6.4 it was shown that the search window is moving unpredictable over the image the main memory is accessed randomly. The issue that raises up is that only a fraction of the bandwidth is utilized. Therefore it is advisable to clubb or coalesce the access to the global memory.

The GPU can coalesce global memory reads and wirtes in as few as one transaction, or two transactions in the case of 128-bit words when certain access requirements are met. These requirements are e.g. that a specific access pattern is realized, the address of a 64-byte segment is 64-byte aligned and further more. For a complete list see Chapter 3 of [12]. This requirements are typical for high speed memory not only valid for GPUs rather for many parallel architectures like the Cell or the Niagara CPU.

To satisfy one requirement, the access pattern, it is advisable to use the *float4* data type. It ensures that data is accessed on consecutive memory locations and hence it can be grouped together. After rewriting the algorithm the new speedup was

$$S = \frac{9616,77\,\text{ms}}{1454\,\text{ms}} = 6,614$$

The speedup was really not very satisfying but the GPU has another mechanism which could be used to speedup the algorithm. The input image is read only and the results are written to another structure. Furthermore read pixels from the image are reused in the calculation when the search window is shifted and the access is rather random.

What now proves advantageous is the ability of CUDA to use texture memory for computing. Texture memory can be more faster than device memory because it is first of all read only and it uses a cache for faster access. The image data stored in a texture is optimised in that way that locality is exploited. The texture is perfect match for the algorithm as random access do not harm the bandwidth so much compared to global memory and pixels read in for one search window can be reused for the next search window as they are cached.

Switching to *textures* (glossary textures) yield to a huge speedup of

$$S = \frac{9616,77\,\text{ms}}{250\,\text{ms}} = 38,46708$$

Texture memory is great for algorithms were random access to memory occurs and data read in are used more often than once. Where it is predictable how much and how (access pattern) the data is read in global memory in conjunction with shared memory can be advantageous. But as said at the beginning of this section certain requirements have to be met to achieve a high bandwidth.

9.4 DIVISION INSTRUCTION OPTIMIZATION

Every hardware instruction is taking a specific amount of clock cycles to complete. For single-precision floating-point arithmetic instructions like add, multiply or multiply-add take 4 clock cycle to complete. Whereas the single-precision floating-point division takes about 36 clock cycles, which is 9× slower than the common arithmetic instructions. Therefore divisions should be avoided as much as possible to increase instruction throughput.

Having a look at Algorithm 5.2 it can be seen that in the calculation of the mean shift vector divisions occur. The denominator is the size of the search window. In this implementation the size of the search window is not changing and hence a constant. This circumstance can be exploited and the reciprocal can be calculated. The reciprocal can then be substituted by the division and denominator to produce faster multiplications.

For example if it looks like this

```
1 // Determine if inside search window
2 // Calculate distance squared of sub-space s
3 float dl = (luv.x - yj_2) / sigmaR;
4 float du = (luv.y - yj_3) / sigmaR;
5 float dv = (luv.z - yj_4) / sigmaR;
```

LISTING 9.1 Expensive divison

one can calculate rsigmaR = 1.0f/sigmfaR in advance and everywhere where a division by sigmaR occurs replace it into a multiplication. The resulting code is then

```
1 // Determine if inside search window
```

```
2 // Calculate distance squared of sub-space s
3 float dl = (luv.x - yj_2) * rsigmaR;
4 float du = (luv.y - yj_3) * rsigmaR;
5 float dv = (luv.z - yj_4) * rsigmaR;
```

LISTING 9.2 Division turned into fast multiplication

With this technique the kernel code is free of divisions except only one that cannot be eliminated since the value is changing throughout the calculations. This optimization yielded a speedup of

$$S = \frac{9616,77\,\text{ms}}{193\,\text{ms}} = 49,82782$$

9.5 EXECUTION CONFIGURATIONS

It is important to check the best execution configuration. Each execution configuration has its benefit. Some can exploit the bandwidth to the global memory some can benefit from the cache of the texture. Furthermore it is important to keep the hardware busy to hide memory latencies. Busy means that one should design the application to use threads and blocks so that the hardware can balance the work across the multiprocessors.

A good execution configuration can hide latency arising from register dependencies , provide optimal computing efficiency and facilitate coalescing to global memory. Not every execution configuration is possible because ressources are limited and too many threads can eat up one ressource where the other ressource is not used. The ressource meant here are the shared, local memory and the register usage. See the CUDA programming guide [48] for a in depth description about the effects of execution configuration.

Several test showed that the best execution configuration for this algorithm is 2×64 threads and the new speedup resulting out of this new configuration was

$$S = \frac{9616,77\,\text{ms}}{171\,\text{ms}} = 56,23842$$

Another positive effect of a configuration that is multiple of the warp size is that the scheduler can work more efficiently because a warp consist out of 32 threads.

9.6 NATIVE DATA TYPES

The CUDA Programming Guide [48] highly recommends the use of the *float* type for single-precision floating-point code. Additionally it is recommended to use the single-precision floating-point mathematical functions. One should use the $float$ data type where ever possible. The GPUs are highly optimized for floating point calculations. After converting all integer calculations to floating point operations another speedup was achieved.

$$S = \frac{9616,77\,\text{ms}}{165\,\text{ms}} = 58,28345$$

On current GPU hardware integer operations can be very expensive in terms of clock cycles. The situation can change with future GPU architecture check against *Fermi* the new architecture from NVIDIA **??**.

9.7 AVOID BRANCH DIVERGENCE

On a architecture with such high throughput of calculations per cycle it is preferred to calculate values rather then generating them through if and $else$ statements. Further problems arise when control flow depends on the thread id which means that they are divided into groups that are executing different execution paths. In this case the execution gets serialised. This can happen when a thread has found a density maximum and all the other threads are still calculating. When the active threads finish the calculation they converge back to the same execution path as the idle threads and they finish together.

Therefore the remaining if and $else$ statements where arranged in such a way so that a thread is exiting early from the loop and avoiding unnecessary calculations. This leads of course to higher branch divergence but the execution time is lower.

The speedup achieved from this optimisation was

$$S = \frac{9616,77\,\text{ms}}{115\,\text{ms}} = 83,62408$$

9.8 SHARED MEMORY

The optimization guides state that one should use shared memory to avoid redundant transfers from global memory. Shared memory is much faster than global and local memory assumed that there are no bank conflicts. The GPU shared memory is divided into so called banks for simultaenous access. If two threads are accessing the same bank the access is serialized and hence can be very slow if not only two but far more threads are accessing the same bank [12].

In Section 6.4 it was shown that the search window is moving unpredictable over the image and hence preloading of data into the shared memory is not possible. Furthermore if two or more search windows from different threads overlap it is obvious that the access to the pixel would lead to bank conflicts. After many tryouts this optimization was discarded.

9.9 KNOW THE ALGORITHM

This section is not related to the hardware more related to the software part of the algorithm. After all the other optimizations done it was obvious that there was not more room for many optimizations that target the hardware. The focus went on to the algorithm and its peculiarity when filtering the image.

The idea was to examine how many iterations in average the algorithm needs to filter the image. In the original code of EDISON there is variable named iterationCount which limits the iteration count to 100 per pixel. The comment in the source code says *"Iteration count is for speed up purposes only – it does not have any theoretical importance"* that brought up a question, why it is needed anyway when the mean shift algorithm converges [10]. For this the source code was modified in that way that for each pixel the iteration count was reported. A precise look a the generated numbers showed that some pixel have iteration counts of 100.

To have an clue if its worth to continue the research into this direction several tests were done. Looking at the iteration count it sticks out that most of the pixel are finished after about 10 iterations, it was strange to see that several pixel were iterating till 100.

The approach taken was to limit the iteration count not to 100 but

to 10 and 50. Setting the limit to 10 and executing the CPU and GPU version the speedup was

$$S = \frac{7369\,\text{ms}}{29\,\text{ms}} = 83,62408$$

Setting the limit to 50 and executing the CPU and GPU version again the speedup was

$$S = \frac{10373\,\text{ms}}{82\,\text{ms}} = 126,5$$

In the case where the limit is 10 the GPU could achieve a speedup of 254,10344× because all multiprocessors are busy then and are not idle as in the case when the limit is 100 and just a few threads are calculating and the rest idles. The algorithm is as fast as the slowest thread which is iterating until 100.

A candidate with iteration count of 100 is picked up from the iteration list and examined further. For this the magnitude squared (the length) of the mean shift vector is analysed over the iterations. When the mean shift vector approaches the convergence point the magnitude becomes smaller and smaller until it falls under a small number. In this implementation this small number is $\epsilon = 0.01$. Then it can be said that the mean shift vector has reached the basin of attraction.

Attractor

Dynamical systems evolve to a specific set called an attractor after many iterations. Dynamical systems are described either by differential equations or by difference equations. A difference equation is a specific type of recurrence relation where each term of a sequence is defined as a function of the preceding terms.

Such an attractor can be a point (fixed point), curve (limit cycle), manifold or a complicated set with a fractal structure known as a strange attractor, see Sprott [50] for impressive pictures of strange attractors.

But not only dynamical systems can lead to such an attractor, furthermore the effect of quantization, the limited precision of the float data type can lead to an attractor.

The above terminology comes from chaos theory where the behaviour of dynamical and iterative functions systems are described. In Section 2.3.7 several books and articles were presented for further information.

The chaos theory was mentioned because it can happen very often without knowledge that some algorithms are iterating and iterating and the stopping criterion is never reached. In the mean shift algorithm case the upper bound was set to 100 without mentioning why. If the algorithm converges an that is proven, why do we have a upper limit anyhow? Because in this case there are attractors as well. All the densities where the mean shift vector moves to are attractors, fixed points. Small perturbations of the vector lead to the same density.

But there are also attractors where the mean shift vector is moving to and it is not the final density.

```
1  10992 - 0.00000 - 1
2  10992 - 2.52029 - 2
3  10992 - 2.52029 - 3
4  10992 - 2.52029 - 4
5  10992 - 2.52029 - 5
6    ...
7  10992 - 2.52029 - 100
```

LISTING 9.3 Magnitude squared of pixel 10992

Examining the sequence of the magnitude squared for pixel 10992 in Listing 9.3 one can easily see that after some iterations the magnitude is fixed to 2.52029. The algorithm is stuck at this value and will not move away from this attractor. An attractor with a single value is a fixed point.

At the beginning of the section it was shown that if the iteration count of each pixel is dropped to a lower level the performance increases. Further examination showed that several pixel have attractors that were limit cycles. Which means that the calculation of the magnitude squared is periodic. The period in this case is from a period-2 up to a period-8 limit cycle.

For visual feedback the source image in Figure 5.4a was taken and for each pixel the iteration count was visualized. The Figure 9.1 shows from blue to red, where red values represent high iteration count, the

iterations for each pixel in a 128×128 big picture.

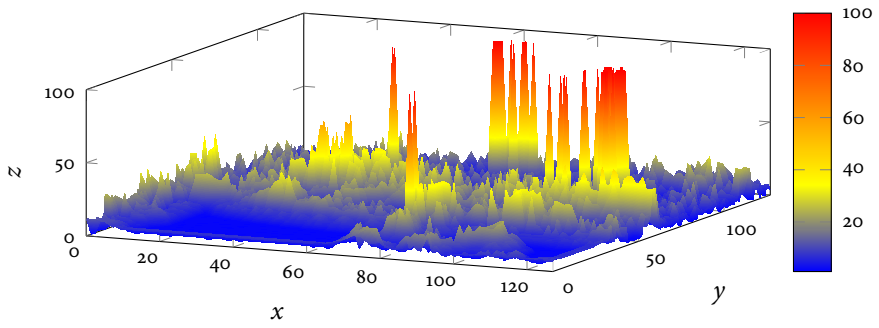


FIGURE 9.1 Iteration count for each pixel

It can be seen that there a several pixel that ware iterating to 100. To circumvent this problem a simple limit cycle detection algorithm was implemented to prevent that several threads are blocking the termination of the algorithm.

In the first place a period-4 limit cycle detection algorithm was implented where a vector was saved with the last four values and compared to the new calculated value. If any value from the vector matches the new value the thread terminates and finishes the computation.

The period-4 limit cycle detection led to a speedup of

$$S = \frac{9616,77\,\text{ms}}{99\,\text{ms}} = 97,13909$$

Extending the limit cycle detection to period-8 limit cycle detection a speedup of

$$S = \frac{9616,77\,\text{ms}}{87\,\text{ms}} = 110,53758$$

was achieved. Surely one cannot be sure to have limit cycles with a higher period but the first problem is the register pressure. Because the compiler has to keep the saved values of the previous iterations in the registers for comparison the registers cannot be used for the other calculations. The CUDA compiler tries to keep as much as possible in the registers. Secondly comparisons are not very fast as computation and having to compare 16 or even 32 values makes no sense because than

the prevention of limit cycles would cost more performance than just iterating to 100. The following series of Figure 9.2,Figure 9.3, Figure 9.4 show visually the effect of limit cycle detection an how all iterations to 100 vanish.

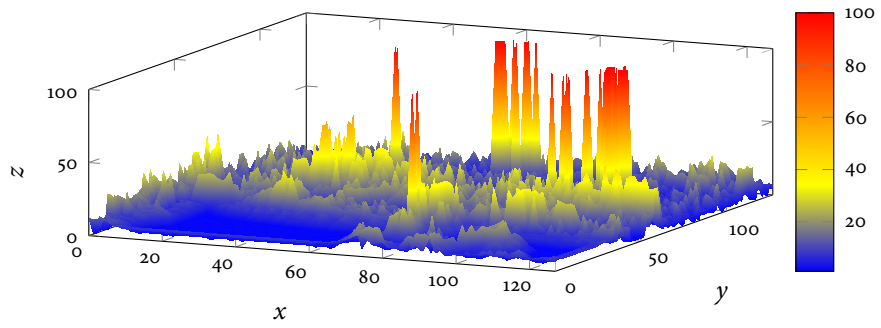


FIGURE 9.2 Period-0 limit cycle detection

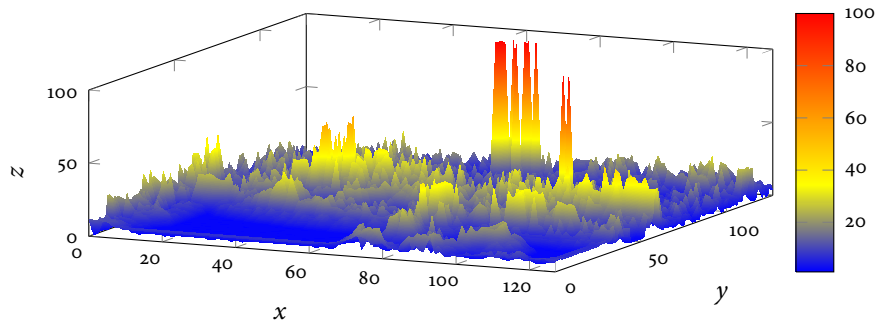


FIGURE 9.3 Period-4 limit cycle detection

9.10 UNROLLING LOOPS AND MULTIPLICATIONS

The last improvement that was made is to unroll loops and complicated multiplications. The idea behind this is that the compiler has more chances to schedule the instructions and reorder them in a proper way. This is important because there are some instructions, e.g. fmadd, that are executing two floating point operations rather than just one floating

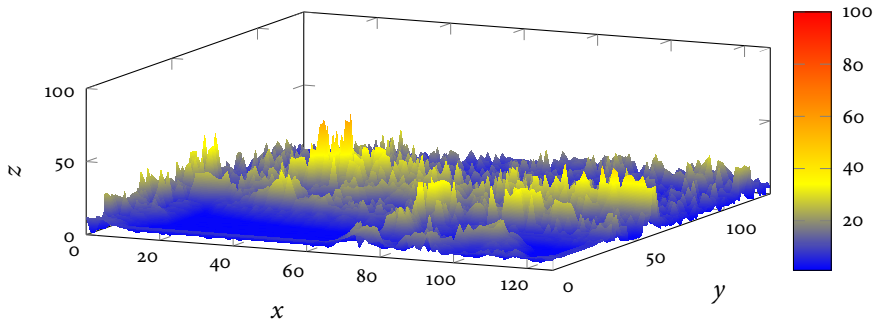


FIGURE 9.4 Period-8 limit cycle detection

point operation per 4 clock cycles and unrolled multiplications are more *clear* to the compiler.

After unrolling the multiplications and loops a final speedup of

$$S = \frac{9616,77\,\text{ms}}{73\,\text{ms}} = 131,73657$$

was achieved.

9.11 SUMMARY

The naive implementation has already led to an speedup of 5×. But putting some knowledge and energy into the optimization one can acheive a speedup of 131,73657×. The optimization was not only done on hardware but also on the algorithm side. It is not enough to know some programming langauge very good the hardware on which its run is more important. No programming langauge will expose all hardware features directly to the developer, the developer has to know about the hardware and its features. Furthermore it is crucial to understand the data and how it is accessed. The biggest speedup was achieved when going from global memory to texture memory. The access of data fits perfectly the texture memory and hence the algorithm can take full advantage of it.

After all the optimizations and a recent speedup it is now important to look after the performance and scalablility of the algorithm. The next chapter is going to tackle this.

10 PERFORMANCE & SCALABILITY

To evaluate the performance and the scalability of the implemented mean shift algorithm several benchmarks will be made. Firstly the algorithm is tested against different image sizes. After that a multiple GPU version is evaluated and tested with the same image sizes as in the previous test. Lastly the hardware parameters of the GPU like the core clock and memory clock are manipulated to get a clue if the algorithm is memory or computational bound. In the case that the algorithm is memory bound one should try to decrease the communication and if the algorithm is computational bound a restructuring of the algorithm could help here. All in all its interesting to see how an algorithm behaves at different circumstances. In general all runs were performed 20 times in a row and the mean was taken as the final result.

10.1 VARYING THE IMAGE SIZE

The first benchmark varies the image size from 128 × 128 pixels to 2688 × 2688 pixels. The image side length is incremented by 128 pixels. The Figure 10.1 shows the GPU runtime and the speedup compared to the CPU run time. The CPU run time was not included in Figure 10.1 because the range of CPU values is 100× larger and the GPU values would not be visible. Its interesting to see that the speedup is increasing faster at

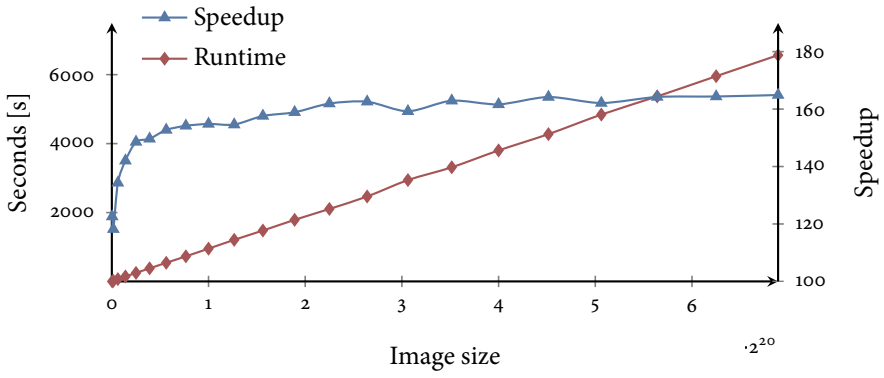


FIGURE 10.1 GPU runtime and speedup dependent on the image size

the beginning and than to level off at about 160. This is no surprise because every software respectively hardware has a ramp-up phase. This ramp-up phase, where no computation is performed, often involves allocation and initialization of data and hardware and additionally on this platform the movement of data from the host to the GPU buffers. With smaller image sizes the time for the ramp-up phase is a significant amount of the complete run-time. With bigger and bigger image sizes the amount of the ramp-up phase to the complete run-time becomes smaller and hence the speedup increases.

10.1.1 *Linearity*

Another fact to consider is how the algorithm is scaling with increasing image sizes. In Chapter 6 it was shown that the algorithm exposes linear complexity $O(n)$ and it is interesting to see how the implemented algorithm behaves in terms of linearity. An algorithm is linear when the two metrics for measuring the performance here the image size and the run-time are proportional. The image size is proportional to the run-time if they have a constant ratio. If one variable increases by an factor x than the proportional variable increases by that factor too.

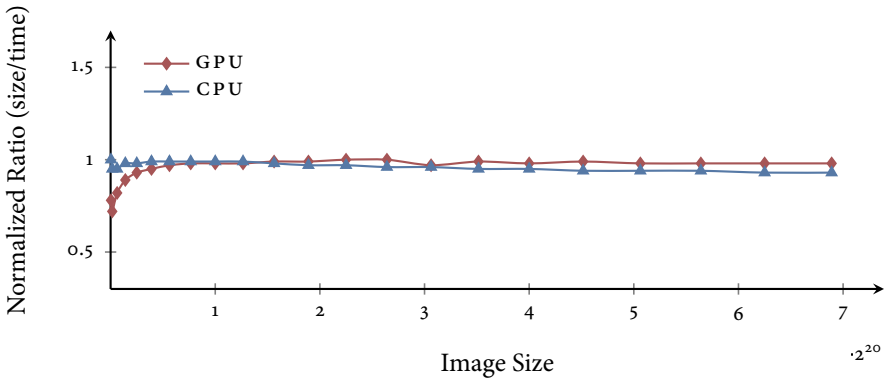


FIGURE 10.2 Normalized size/time ratios

The Figure 10.2 shows the normalized image size per time ratios in the interval $[0,1]$. As it can be seen the ratio for the CPU and the GPU are increasing similar to the speedup because of the before mentioned ramp-up phase. The ratio increases until it reaches the right endpoint of

the interval. If the image size is doubled the run-time is doubled (image size $\propto$ run-time) which means the algorithm scales linearly.

10.2 MULTIPLE GPUS

The previous section analyzed how the implemented algorithm scales over the image size. Another important attribute is how the algorithm scales if another GPU is added for computation. A second identical GPU was used to generate the results shown in Figure 10.3.

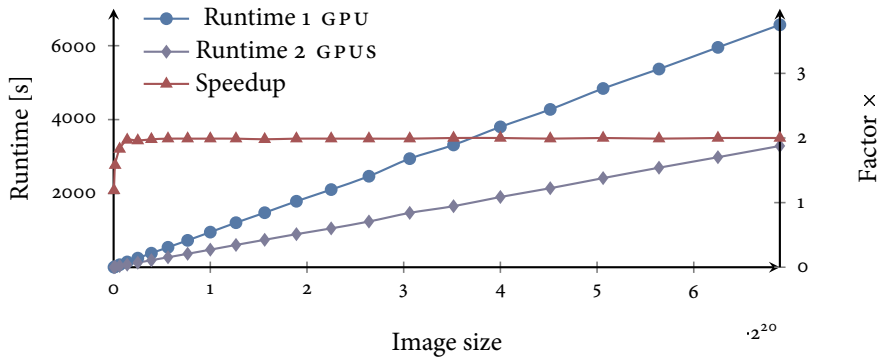


FIGURE 10.3 Multiple GPU(s) runtime and speedup dependent on the image size

As stated in the previous sections the ramp-up phase is responsible that the algorithm is not scaling with small image sizes. Its again clearly identifiable in the Figure 10.3. It's safe to assume that the multiple GPU run-times are proportional to the image size as in Figure 10.2 seen for one GPU since the run-times are 2× faster but the ratio will nevertheless still be a constant. For the precise numbers see **??**. This section showed that the algorithm scales perfectly with multiple devices and it would be no surprise to see a speedup of 3× if a third GPU were added to the system. Additionally the Figure 10.4 shows the speedups of 1 GPU and 2 GPUS compared to the CPU run-times.

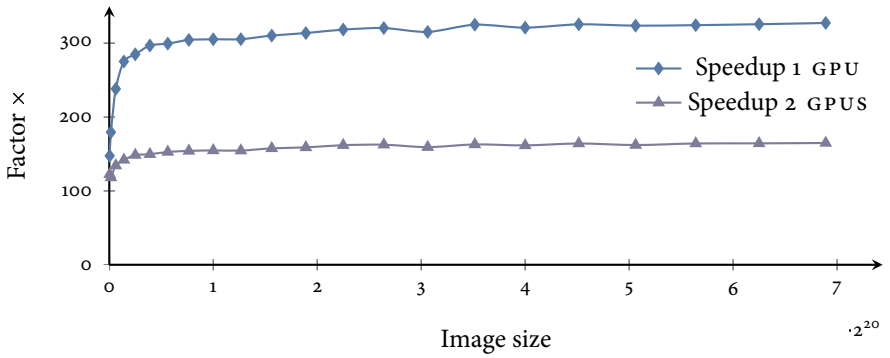


FIGURE 10.4 GPU speedups compared to the CPU run-times over image size

10.3 OVERCLOCKING THE GPU

The GPU has three clocks that can be over-clocked to achieve higher performance. The first is the core clocks, the second clock is the memory clock and the third clock is the shader clock where the shader clock typically moves synchronically with the core as they work on a set ratio. For example the used *GeForce 8800 GTS 512* features a 650 Megahertz (MHZ) core, a 1620 MHZ shader and a 970 MHZ memory clock. This means this particular graphics card uses a core to shader clock multiplier of 2.5×. Increasing the core clock means increasing the shader clock at the same time. Therefore only the core and memory clock will be considered for the experiments.

The reason why someone would over-clock a GPU is at first hand to increase the performance and on the second hand to find out if an algorithm is computational or memory bound by experiment.

Therefore several experiments will be undertaken. One experiment will involve the change of the core clock, second the change of the memory clock and lastly the combination of both clocks. All experiments were done with a free tool, *NVClock* [1]. NVClock is a small utility that allows to over-clock NVIDIA based GPUs and adjust the fan speed. This is important since the GPU will run out of specification and could overheat. The experiments will begin with changing the memory clock and

1 http://www.linuxhardware.org/nvclock/

evaluation of the results.

10.3.1 *Increasing the Memory Clock*

For the first experiment the memory clock will be changed by steps of 100 MHz. As a starting point the default 972 MHz of the memory clock will be used and the core clock will be fixed at the default of 648 MHz. Experiments have shown that the maximal achievable memory clock is 1101.600 MHz. All clocks were read out from hardware with *NVClock*. Additionally it must be noted that all frequency settings were not accepted one-on-one which means an increase per software of 100 MHz doesn't mean an increase of 100 MHz in hardware due to hardware restrictions. The Table A.2 shows the frequency set in software and the resulting real frequency in hardware.

The test was performed with three different image sizes: 256×256 pixel, 1024×1024 pixel and 2048×2048 pixel. The Figure **??** shows the run-time dependent on the memory clock.

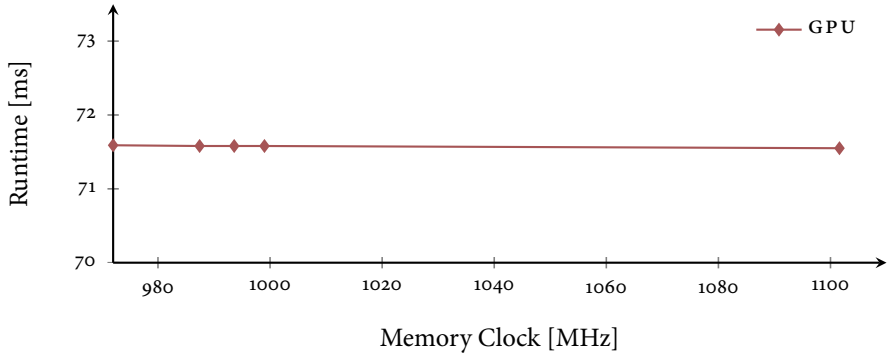


FIGURE 10.5 Image 256x256 increasing the memory clock

A straight line indicates that the algorithm is not memory bound. It does not make any difference how fast the memory one make the run-time will not decrease. Additionally it can be concluded that the algorithm is computational bound. This will be proven in the next section when the core clock is manipulated. The image sizes 1024×1024 and 2048×2048 pixel show the same behaviour. For the complete list of numbers see Table A.3.

10.3.2 *Increasing the Core Clock*

For the second experiment the core clock will be changed by steps of 100 MHz and as a dependency the shader clock as well. As a starting point the default 648 MHz of the core clock will be used and the memory clock will be fixed at the default of 972 MHz because the previous section has shown that the algorithm is not memory bound and the increase of the memory clock has no effect on the run-time. Experiments have shown that the maximal achievable core clock is 783 MHz. All clocks were read out from hardware with *NVClock*. The Table A.1 shows the frequency set in software and the resulting real frequency in hardware.

The test was performed with three different image sizes: 256×256 pixel, 1024×1024 pixel and 2048×2048 pixel. The Figure **??** shows the run-time dependent on the core clock. Additionally the normalised ratio between the differences of clock and ratio is depicted as well.

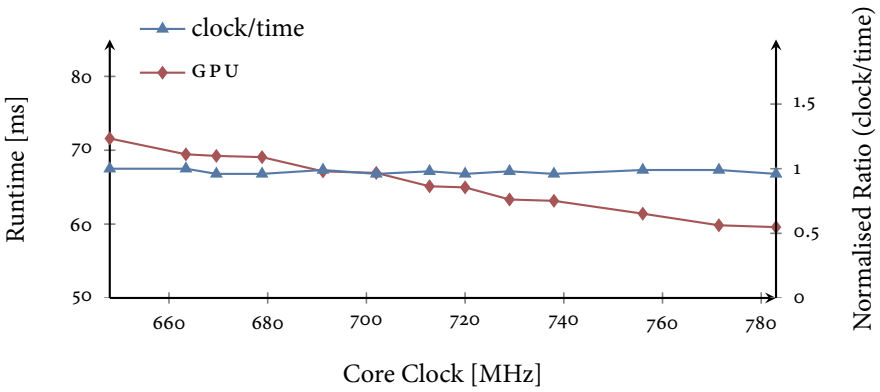


FIGURE 10.6 Image 256x256 varying the core clock with fixed mem clock

It's visible that the run-time is decreasing proportionally with the increase of the core-clock. This means that the algorithm is heavily computational bound. The run-time is determined in this implementation solely by the speed of the GPU. One way to get over this barrier is to (1) minimize the instructions executed by the GPU by restructuring the code or (2) adopting a another algorithm for finding feature point that fall into the search window, see **??** for enhancements that could be made to accelerate the search. Additionally, the image sizes 1024×1024 and

2048×2048 pixel show the same behavior. For completeness the next section shows the case when two GPUs are overclocked.

Multiple gpus overclocked

It's no surprise to see almost a identical diagram compared to the previous section. As the algorithm scales well with multiple GPU (see Section 10.2) and with the frequency of the core clock one can assume to see the same sequence of run-times just cut by half. The Figure **??** shows that, additionally attention should be paid to the shape of the curve as its the same as in Figure **??**.

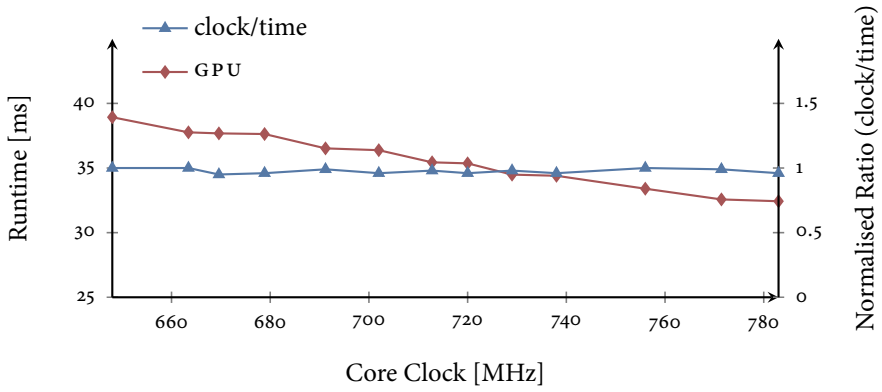


FIGURE 10.7 Image 256x256 varying the core on two GPUS

10.4 FINAL SPEEDUP

The previous sections showed the how the algorithm compares to the CPU over different work-sizes. Disregarding the ramp-up phase a single GPU achieved a speedup of about 160×. Computing with two GPUs the resulting speedup was of about 320×. This showed that a suitable algorithm can achieve reasonable speedups compared to traditional CPUs.

Additionally the scalability over the work-sizes, an additional GPUs and frequency were analyzed and depicted. The result is clear, the algorithm scales linearly over all three before mentioned parameters.

The speedup for an image of 2688×2688 pixel segmented by one overclocked GPU is

$$S_{single}(n) = \frac{T_{CPU}}{T_{GPU}} = \frac{1082570,14\,\text{ms}}{5465,28\,\text{ms}} = 198,08136 \tag{10.1}$$

and the final speedup by two overclocked GPUs is

$$S_{multi}(n) = \frac{T_{CPU}}{T_{GPU}} = \frac{1082570,14\,\text{ms}}{2736,14\,\text{ms}} = 395,65597 \tag{10.2}$$

That's an impressive number about 400× faster than the CPU version. Of course on has to take into account that the used algorithm is highly suitable for the GPU. Nevertheless this chapter showed to which extent algorithms can be accelerated on the GPU.

11 CONCLUSIONS & FURTHER WORK

The field of GPU computing is nowadays very broad. Starting from dense linear algebra, n-body, finite difference over graphics, image processing to computational modeling in science, engineering and finance the GPU is existent in all this fields. A lot of algorithms profited from using the GPU as an accelerator and many will profit too. There is a high demand on performance and computational power for scientific algorithms. Developers search for new architectures to solve problems in shorter time and the GPU is one promising candidate because of its incredible raw power.

New architectures meant new programming models and in the case of the GPU the first programming model was not build for general purpose computing and developers were accidently forced to use 3D APIS (OpenGL[1], DirectX[2]) to accomplish task like matrix multiplication or fast fourier transformation. These APIS implied some limitations and raised some myth for the new era of GPGPU when the switch from fixed pipelines to fully programmable pipelines happen.

Developers were concerned about the programming models will still have the limitations as the former had. The feasibility study showed that all the odds and ends are not more valid for the new architectures and programming models. This new sophisticated programming environments, like CUDA for the NVIDIA GPUS, offer a complete, easy and new interface to the hardware to exploit parallelism as much as possible that make GPGPU more feasible.

With the advent of these new programming environments more and more developers are using GPUS for general purpose computing. Many of them come from a world where everything is sequentiel and struggle with thinking in parallel. Developers often only needed to compile their code with another compiler to port the code to another architecture. For example porting code from x86 to PowerPC or x86 to Scalable Processor ARChitecture (SPARC) was easy enough if it was standard C or C++ without any special operating system calls. But the situation for parallel processors like the Cell, Niagara and here the GPU is completely different.

1 http://www.opengl.org
2 http://www.microsoft.com

One has to identify the critical parts and explicitly offload tasks to the GPU. It is obviuos that this is not the only step to do. Furthermore there are other facts which have to be considered, like load-balancing, deadlocks, concurrency, cache coherency and other. A workflow was presented throughout this thesis by the means of an algorithm how all this neccessary steps were untertaken.

The porting step is not the last when considering high performance code on a GPU. The next big thing is to optimize the code to some extent to run efficiently on the hardware. The main cause of unefficient code is bad data design. It is the biggest problem of an alogirthm when trying to attain maximum performance. If the algorithm is memory bound and not computational bound the limiting factor or bottleneck is the bandwidth to the memory. Good data design is essential for good, performant code. The data and how it is used must be known in order to effectively design and optimize a application for a GPU or other highly parallel systems like the Cell processor [27]. A simple computationally bound algorithm can be easily designed in a way that it becomes memory bound if data arrangement and access is neglected by the developer. A good data design is essential for any massively parallel system.

11.1 FURTHER WORK

For the mean shift algorithm there exist several acceleration teqhniques. These techniques could be applied to the algorithm implemented in this thesis to accelerate the algoirthm to some extent. There were several papers that describe the techniques in depth. All of them have on thing in common, they degrade the quality of the segmented image whereas the algorithm used throughout this thesis is the most accurate but also the slowest. But not all techniques imply a very high quality loss, often not seen by the human eye and can be used for tracking or cutout filters for image processing.

Further work could also extend the existing algorithm to segment not only pictures but also videos where tracking is a big field of interest. For image processing software different filters could be implemented like ‘‘paint by numbers’’ or ‘‘comic effect’’ and many more. Photoshop Creative Suite 4 (CS4) from Adobe e.g, is making heavy use of the GPU already and has a plugin architecture for developing custom filters.

Fermi

Fermi is a announced but yet not released new groundbreaking architecture from NVIDIA. Many of the limitations with the current hardware like non Error-Correcting Code (ECC) memory, single kernel, no C++ support, small main memory, register pressure and so on are being improved or completely eliminated. It features more shared memory, register, streaming multiprocessors and many more especially designe to meet the requirements of the HPC and general purpose computing communities. It would be worthwhile to run the algorithm on this new architecture to see how the promised scaling without recompiling is really doing. But for more performance it should be recompiled and adapted to all the new features available.

The problem with new hardware architectures is they have to be accepted by the developers to succeed. One way to achieve this is to have an easy programming model which adapts easily to common apis and existing systems. There are rarely branches of industry which want to, or are able to rewrite their code for bleeding edge architectures like the cell or GPUs if the outcome is only 2x faster code. In this case many just extend their cluster with another bunch of cheap systems. A factor of 10 is an undocumented limit when to invest in other, faster hardware.

OpenCL, CUDA and other programming model make a promising start in the field of massively parallel computers. More and more developers will or have to adapt to parallel architectures and parallel programming. Such programming models make the life easier and can help to understand parallelism.

BIBLIOGRAPHY

[1] IkkJin Ahn, Michael Lehr, and Paul Turner. Image processing on the gpu. Technical report, University of Pennsylvania, February 2005.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference Proceedings*, volume 32, pages 37–45, 1968.

[3] A. J. Bernstein. Program analysis for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, October 1966.

[4] Cynthia Bruyns and Bryan Feldman. Image processing on the gpu: a canonical example. Technical report, University of California Berkeley, 2005.

[5] Ian Buck, Paulius Micikevicius, Scott Morton, John Stone, Jim Phillips, and Patrick Legresley. Supercomputing 2008 tutorial intro. URL `http://www.gpgpu.org/sc2008/M02-01_Intro.pdf`.

[6] 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 Press.

[7] Chas. Data-parallel computing. *Queue*, 6(2):30–39, 2008. ISSN 1542-7730. doi: http://dx.doi.org/10.1145/1365490.1365499. URL `http://dx.doi.org/10.1145/1365490.1365499`.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof. doi: http://dx.doi.org/10.1016/j.jpdc.2008.05.014. URL `http://dx.doi.org/10.1016/j.jpdc.2008.05.014`.

[9] Yizong Cheng. Mean shift, mode seeking, and clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17

(8):790–799, August 2002. doi: 10.1109/34.400568. URL http://dx.doi.org/10.1109/34.400568.

[10] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002. doi: 10.1109/34.1000236. URL http://dx.doi.org/10.1109/34.1000236.

[11] Dorin Comaniciu and Peter Meer. Distribution free decomposition of multivariate data. In *SSPR '98/SPR '98: Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*, pages 602–610, London, UK, 1998. Springer-Verlag. ISBN 3-540-64858-5. URL http://portal.acm.org/citation.cfm?id=673222.

[12] Nvidia Corporation. Nvidia cuda c programming best practices guide. Technical report, 2009.

[13] Scott Draves and Erik Reckase. The fractal flame algorithm. Technical report, November 2008.

[14] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, November 2000. ISBN 0471056693. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471056693.

[15] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 257–266, New York, NY, USA, 2002. ACM Press. doi: http://dx.doi.org/10.1145/566570.566574. URL http://dx.doi.org/10.1145/566570.566574.

[16] Kayvon Fatahalian and Mike Houston. Gpus: a closer look. *Queue*, 6(2):18–28, 2008. ISSN 1542-7730. doi: http://dx.doi.org/10.1145/1365490.1365498. URL http://dx.doi.org/10.1145/1365490.1365498.

[17] Raanan Fattal, Dani Lischinski, and Michael Werman. Gradient domain high dynamic range compression. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics*

and interactive techniques, pages 249–256, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: http://dx.doi.org/10.1145/566570.566573. URL http://dx.doi.org/10.1145/566570.566573.

[18] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972. URL http://en.wikipedia.org/wiki/Flynn's_taxonomy.

[19] Andrew S. Glassner. *An introduction to Ray Tracing*. Academic Press, 1989.

[20] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31 (5):532–533, 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL http://dx.doi.org/10.1145/42411.42415.

[21] Mark Harris. Mapping computational concepts to gpus. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM. doi: http://dx.doi.org/10.1145/1198555.1198768. URL http://dx.doi.org/10.1145/1198555.1198768.

[22] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, 1994.

[23] Henrik W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, Ltd., July 2001. ISBN 1568811470. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/1568811470.

[24] Emmett Kilgariff and Randima Fernando. The geforce 6 series GPU architecture. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM. doi: http://dx.doi.org/10.1145/1198555.1198767. URL http://dx.doi.org/10.1145/1198555.1198767.

[25] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005. doi: http://dx.doi.org/10.1109/MM.2005.35. URL http://dx.doi.org/10.1109/MM.2005.35.

[26] Zvonko Krnjajic. Untersuchung und implementierung von algorithmen zur zerlegung großer zahlen. Technical report, University of applied sciences esslingen, June 2005.

[27] Zvonko Krnjajic. A raytracer implementation as an example for a cell broadband engine optimized application architecture. Master's thesis, University of Applied Sciences Esslingen, July 2006. URL #.

[28] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006. URL http://graphics.cs.ucdavis.edu/~lefohn/work/glift/.

[29] Jian-Ming Li, Xiao-Jing Wang, Rong-Sheng He, and Zhong-Xian Chi. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pages 855–862, 2007. doi: http://dx.doi.org/10.1109/NPC.2007.108. URL http://dx.doi.org/10.1109/NPC.2007.108.

[30] Fajie Li1 and Reinhard Klette. Adaptive mean shift-based clustering. Technical report. URL http://www.mi.auckland.ac.nz/tech-reports/MItech-TR-20.pdf.

[31] Pawel Maciol and Krzysztof Banas. Testing tesla architecture for scienti?c computing: the performance of matrix-vector product. volume 3, 2008.

[32] Rafal Mantiuk, Karol Myszkowski, and Hans P. Seidel. A perceptual framework for contrast processing of high dynamic range images. *ACM Trans. Appl. Percept.*, 3(3):286–308, 2006. ISSN 1544-3558. doi: http://dx.doi.org/10.1145/1166087.1166095. URL http://dx.doi.org/10.1145/1166087.1166095.

[33] Rafal Mantiuk, Grzegorz Krawczyk, Radoslaw Mantiuk, and Hans P. Seidel. High dynamic range imaging pipeline: perception-motivated representation of visual content. volume 6492. SPIE, 2007. URL http://scitation.

aip.org/getabs/servlet/GetabsServlet?prog=normal&id=PSISDG006492000001649212000001&idtype=cvips&gifs=yes.

[34] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004. ISBN 1568812299. URL http://portal.acm.org/citation.cfm?id=1024226.

[35] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin M. Yusof, Greg Roth, and Sharen Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11):648–662, September 2007. ISSN 0167-8191. doi: 10.1016/j.parco.2007.09.001. URL http://dx.doi.org/10.1016/j.parco.2007.09.001.

[36] Sally A. Mckee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, New York, NY, USA, 2004. ACM Press. ISBN 1581137419. doi: http://dx.doi.org/10.1145/977091.977115. URL http://dx.doi.org/10.1145/977091.977115.

[37] Rahul Narain. Image processing on gpus. Technical report, University of North Carolina Chapel Hill, NC 27599-3175, March 2007.

[38] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008. ISSN 1542-7730. doi: http://dx.doi.org/10.1145/1365490.1365500. URL http://dx.doi.org/10.1145/1365490.1365500.

[39] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007. ISSN 0167-7055. doi: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x. URL http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x.

[40] John Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM. doi: http://dx.doi.org/10.1145/1198555.1198766. URL http://dx.doi.org/10.1145/1198555.1198766.

[41] David Patterson, Krste Asanovic, Kurt Keutzer, and And. Computer architecture is back - the berkeley view on the parallel computing landscape. Technical report, January 2007.

[42] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. pages 184–592 Vol. 1, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1493930.

[43] Clifford A. Pickover. *Computers, Pattern, Chaos, and Beauty: Graphics from an Unseen World*. Dover Publications, Incorporated, 2001. ISBN 0486417093. URL http://portal.acm.org/citation.cfm?id=516853.

[44] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003. ISBN 1-58113-739-7.

[45] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, 2002. ISSN 0730-0301. doi: http://dx.doi.org/10.1145/566654.566575. URL http://dx.doi.org/10.1145/566654.566575.

[46] 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. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM. doi: http://dx.doi.org/10.1145/1399504.1360617. URL http://dx.doi.org/10.1145/1399504.1360617.

[47] Yuan Shi. Reevaluating amdahl's law and gustafson's law, October 1996. URL http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html.

[48] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[49] Alvy Ray Smith. A pixell is not a little square. *Microsoft Technical Memo 6*, 1995.

[50] Julien C. Sprott. *Strange Attractors: Creating Patterns in Chaos*. M & T Books, har/dsk edition. ISBN 1558512985. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/1558512985.

[51] Herb Sutter. The free lunch is over a fundamental turn toward concurrency in software. March 2005. URL http://www.gotw.ca/publications/concurrency-ddj.htm.

[52] Andrew Thall. Extended-precision floating-point numbers for GPU computation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: http://dx.doi.org/10.1145/1179622.1179682. URL http://dx.doi.org/10.1145/1179622.1179682.

[53] Marc Tremblay and Shailender Chaudry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc® processor. *International Solid-State Circuits Conference*, 2008.

[54] Turner Whitted. An improved illumination model for shaded display. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM. doi: http://dx.doi.org/10.1145/1198555.1198743. URL http://dx.doi.org/10.1145/1198555.1198743.

[55] Man-Leung Wong, Tien-Tsin Wong, and Ka-Ling Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293 Vol. 3, 2005. doi: http://dx.doi.org/10.1109/CEC.2005.1554979. URL http://dx.doi.org/10.1109/CEC.2005.1554979.

[56] Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. *Parallel Genetic Algorithms on Programmable Graphics Hardware*, volume 3612, pages 1051–1059. July 2005.

[57] Kai Zhang, James T. Kwok, and Ming Tang. Accelerated convergence using dynamic mean shift. In *ECCV (2)*, pages 257–268, 2006.

[58] K. Zhao, X. Chu, M. Z. Wang, and Y. Jiang. Speeding up homomorpic hashing using gpus. In *2009 IEEE International Conference on Communications*, pages 1–5. IEEE, June 2009. doi: 10.1109/ICC.2009.5199483. URL `http://dx.doi.org/10.1109/ICC.2009.5199483`.

GLOSSARY

CLOCK	Electronic pulses which are emitted periodically, usually by a crystal device, to synchronize the operation of circuits in a computer. Also known as clock signals. 71, 72
SHADER	A computational unit on the GPU. 71

ACRONYMS

1D	one-dimensional 16, 18, 54
1U	one rack unit 13
2D	two-dimensional 16, 54
3D	three-dimensional 6, 16, 54
ALU	Arithmetic Logic Unit 13
ANSI	American National Standards Institute 14
API	Application Programming Interface i, 2, 14, 17, 23, 28
ATI	ATI Technologies Inc. 11
CPU	Central Processing Unit i, 1, 5, 9, 11, 15–17, 21, 26, 43, 44, 53–56, 58, 59, 64, 69–71, 75, 82
CUDA	Common Unified Device Architecture i, 3, 6, 13–17, 25, 28, 53, 54, 57–59, 61, 62, 66, 88
DMA	Direct Memory Access 17, 55
EDISON	Edge Detection and Image SegmentatiON System 43–45, 58, 63
GB	Gigabyte 13
GDDR3	Graphics Double Data Rate v3 13
GFLOP	Giga FLoating point Operations Per Second 2, 13, 14, 26, 27
GMP	GNU Multiple Precision 8

GPGPU	General Purpose Computation on GPUS i, 1, 2, 5, 23, 28
GPU	Graphic Processing Unit i, 1–3, 5–9, 11–17, 19–21, 23, 25–28, 44, 46, 47, 49–51, 53–59, 62–64, 69–72, 74, 75, 79, 82, 85, 88
HDR	High Dynamic Range 8, 9
HPC	High Performance Computing 23
HSV	Hue Saturation Value 30
IBM	International Business Machines Corporation 5
KB	Kilobyte 13
LUV	L* u* v* 30, 38, 40, 42, 48, 55, 56
MHZ	Megahertz 72–74
MIMD	Multiple Instruction Multiple Data 11
MISD	Multiple Instruction Single Data 11
MISE	Mean Integrated Squared Error 33
MMX	Matrix Math Extensions 11
MPI	Message Passing Interface 1
MPMD	Multiple Programs Multiple Data 12
NVIDIA	NVIDIA Corporation 12
PCI-E	Peripheral Component Interconnect Express 13, 17
RGB	Red Green Blue 30, 40, 42, 48, 55, 56
RO	read-only 16
RW	read-write 16, 17
SDK	Software Development Kit 11, 27, 28
SIMD	Single Instruction Multiple Data 11–13, 23, 28, 55, 81
SISD	Single Instruction Single Data 11
SM	Streaming Multiprocessor 25
SPMD	Single Program Multiple Data 12, 50, 51, 53, 57

SSE	Streaming SIMD Extensions	11
USM	Unified Shader Model	12, 14
VMX	Vector Multimedia eXtension	11

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCE TABLES

A

Software [MHz]	Hardware [MHz]
650	648
660	663
670	670
680	679
690	691
700	702
710	713
720	720
730	729
740	738
750	756
760	756
770	771
780	783

TABLE A.1 Frequencies set in software and the resulting hardware frequencies

Software [MHz]	Hardware [MHz]
970	972
980	987
990	994
1000	999
1100	1102

TABLE A.2 Frequencies set in software and the resulting hardware frequencies

MEMCLK	256	1024	2048
972	71.59	954.31	3796.36
987.43	71.58	955.15	3796.79
993.6	71.58	955.07	3795.24
999	71.58	955.32	3798.35
1101.6	71.55	954.96	3795.84

TABLE A.3 Various memory clocks and the corresponding run times

HOST & KERNEL SOURCE CODE

B

```
#ifndef _MSFILTER_KERNEL_H_
#define _MSFILTER_KERNEL_H_

#include <stdio.h>
#include <cutil_inline.h>
#include "meanshiftfilter_common.h"

// declare texture reference for 2D float texture
texture<float4, 2, cudaReadModeElementType> tex;

__global__ void meanshiftfilter(
  float4* d_luv, float offset,
  float width, float height,
  float sigmaS, float sigmaR,
  float rsigmaS, float rsigmaR)
{

  // NOTE: iteration count is for speed up purposes only - it
  //       does not have any theoretical importance
  float iter = 0;
  float wsum;


  float ix = blockIdx.x * blockDim.x + threadIdx.x;
  float iy = blockIdx.y * blockDim.y + threadIdx.y;

  float4 luv = tex2D(tex, ix, iy + offset);

  // Initialize spatial/range vector (coordinates + color values)
  float yj_0 = ix;
  float yj_1 = iy + offset;
  float yj_2 = luv.x;
  float yj_3 = luv.y;
  float yj_4 = luv.z;

```

```
36    // Initialize mean shift vector
37    float ms_0 = 0.0f;
38    float ms_1 = 0.0f;
39    float ms_2 = 0.0f;
40    float ms_3 = 0.0f;
41    float ms_4 = 0.0f;
42
43    // Period-8 limit cycle detection
44    float lc_0 = 12345678.0f;
45    float lc_1 = 12345678.0f;
46    float lc_2 = 12345678.0f;
47    float lc_3 = 12345678.0f;
48    float lc_4 = 12345678.0f;
49    float lc_5 = 12345678.0f;
50    float lc_6 = 12345678.0f;
51    float lc_7 = 12345678.0f;
52
53
54    // Keep shifting window center until the magnitude squared of the
55    // mean shift vector calculated at the window center location is
56    // under a specified threshold (Epsilon)
57    float mag;  // magnitude squared
58
59    do {
60      // Shift window location
61      yj_0 += ms_0;
62      yj_1 += ms_1;
63      yj_2 += ms_2;
64      yj_3 += ms_3;
65      yj_4 += ms_4;
66
67
68      // Calculate the mean shift vector at the new
69      // window location using lattice
70
71      // Initialize mean shift vector
72      ms_0 = 0.0f;
73      ms_1 = 0.0f;
74      ms_2 = 0.0f;
75      ms_3 = 0.0f;
```

```
76      ms_4 = 0.0f;
77
78      // Initialize wsum
79      wsum = 0.0f;
80
81      // Perform lattice search summing
82      // all the points that lie within the search
83      // window defined using the kernel specified
84      // by uniformKernel
85
86
87      //Define bounds of lattice...
88      //the lattice is a 2dimensional subspace whose
89      //search window bandwidth is specified by sigmaS:
90
91
92      float lX = (int)yj_0 - sigmaS;
93      float lY = (int)yj_1 - sigmaS;
94      float uX = yj_0 + sigmaS;
95      float uY = yj_1 + sigmaS;
96
97      lX = fmaxf(0.0f, lX);
98      lY = fmaxf(0.0f, lY);
99      uX = fminf(uX, width - 1.0f);
100     uY = fminf(uY, height - 1.0f);
101
102
103
104     //Perform search using lattice
105     //Iterate once through a window of size sigmaS
106     for(float y = lY; y <= uY; y += 1.0f) {
107       for(float x = lX; x <= uX; x += 1.0f) {
108
109         //Determine if inside search window
110         //Calculate distance squared of sub-space s
111         float diff0 = 0.0f;
112
113         float dx_0 = x - yj_0;
114         float dy_0 = y - yj_1;
115
```

```
116         float dx = dx_0 * rsigmaS;
117         float dy = dy_0 * rsigmaS;
118
119         float diff0_0 = dx * dx;
120         float diff0_1 = dy * dy;
121
122         diff0 = diff0_0 + diff0_1;
123
124         if (diff0 >= 1.0f) continue;
125
126         luv = tex2D(tex, x, y);
127
128         float diff1 = 0.0f;
129
130         float dl_0 = luv.x - yj_2;
131         float du_0 = luv.y - yj_3;
132         float dv_0 = luv.z - yj_4;
133
134         float dl = dl_0 * rsigmaR;
135         float du = du_0 * rsigmaR;
136         float dv = dv_0 * rsigmaR;
137
138
139         float diff1_0 = dl * dl;
140         float diff1_1 = du * du;
141         float diff1_2 = dv * dv;
142         diff1 = diff1_0 + diff1_1 + diff1_2;
143
144         if((yj_2 > 80.0f)) {
145           diff1 += 3.0f * dl * dl;
146         }
147
148         if (diff1 >= 1.0f) continue;
149
150         // If its inside search window perform sum and count
151         // For a uniform kernel weight == 1 for all feature points
152         // considered point is within sphere => accumulate to mean
153         ms_0 += x;
154         ms_1 += y;
155         ms_2 += luv.x;
```

```
156           ms_3 += luv.y;
157           ms_4 += luv.z;
158           wsum += 1.0f; //weight
159
160         }
161
162       }
163       // When using uniform kernel wsum is always > 0 .. since weight == 1 and
164       // wsum += weight.
165       // determine the new center and the magnitude of the meanshift vector
166       // meanshiftVector = newCenter - center;
167       wsum = 1.0f/wsum;
168
169       float ws_0 = ms_0 * wsum;
170       float ws_1 = ms_1 * wsum;
171       float ws_2 = ms_2 * wsum;
172       float ws_3 = ms_3 * wsum;
173       float ws_4 = ms_4 * wsum;
174
175       ms_0 = ws_0 - yj_0;
176       ms_1 = ws_1 - yj_1;
177       ms_2 = ws_2 - yj_2;
178       ms_3 = ws_3 - yj_3;
179       ms_4 = ws_4 - yj_4;
180
181
182       // Calculate its magnitude squared
183       float mag_0 = ms_0 * ms_0;
184       float mag_1 = ms_1 * ms_1;
185       float mag_2 = ms_2 * ms_2;
186       float mag_3 = ms_3 * ms_3;
187       float mag_4 = ms_4 * ms_4;
188       mag = mag_0 + mag_1 + mag_2 + mag_3 + mag_4;
189
190
191       // Usually you don't do float == float but in this case
192       // it is completely safe as we have limit cycles where the
193       // values after some iterations are equal, the same
194       if (mag == lc_0 || mag == lc_1 ||
195           mag == lc_2 || mag == lc_3 ||
```

```
196         mag == lc_4 || mag == lc_5 ||
197         mag == lc_6 || mag == lc_7)
198     {
199       break;
200     }
201
202     lc_0 = lc_1;
203     lc_1 = lc_2;
204     lc_2 = lc_3;
205     lc_3 = lc_4;
206     lc_4 = lc_5;
207     lc_5 = lc_6;
208     lc_6 = lc_7;
209     lc_7 = mag;
210
211
212     // Increment iteration count
213     iter += 1;
214
215   } while((mag >= EPSILON) && (iter < LIMIT));
216
217
218   // Shift window location
219   yj_0 += ms_0;
220   yj_1 += ms_1;
221   yj_2 += ms_2;
222   yj_3 += ms_3;
223   yj_4 += ms_4;
224
225
226   luv = make_float4(yj_2, yj_3, yj_4, 0.0f);
227
228   // store result into global memory
229   int i = ix + iy * width;
230   d_luv[i] = luv;
231
232
233   return;
234 }
235
```

```
236
237
238 extern "C" void initTexture(int width, int height, void *d_src)
239 {
240   cudaArray* d_array;
241   int size = width * height * sizeof(float4);
242
243   cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4> ();
244
245   cutilSafeCall(cudaMallocArray(&d_array, &channelDesc, width, height ));
246   cutilSafeCall(cudaMemcpyToArray(d_array, 0, 0, d_src, size, cudaMemcpyDeviceToDevice));
247   tex.normalized = 0; // access without normalized texture coordinates
248         // [0, width -1] [0, height - 1]
249   // bind the array to the texture
250   cutilSafeCall(cudaBindTextureToArray(tex, d_array, channelDesc));
251 }
252
253
254 extern "C" void meanShiftFilter(dim3 grid, dim3 threads,
255     float4* d_luv, float offset,
256     float width, float height,
257     float sigmaS, float sigmaR,
258     float rsigmaS, float rsigmaR)
259 {
260   meanshiftfilter<<< grid, threads>>>(d_luv, offset, width, height, sigmaS, sigmaR, rsigmaS, rsigmaR);
261 }
262
263
264 #endif // #ifndef _MSFILTER_KERNEL_H_
```

LISTING B.1 Mean shift kernel