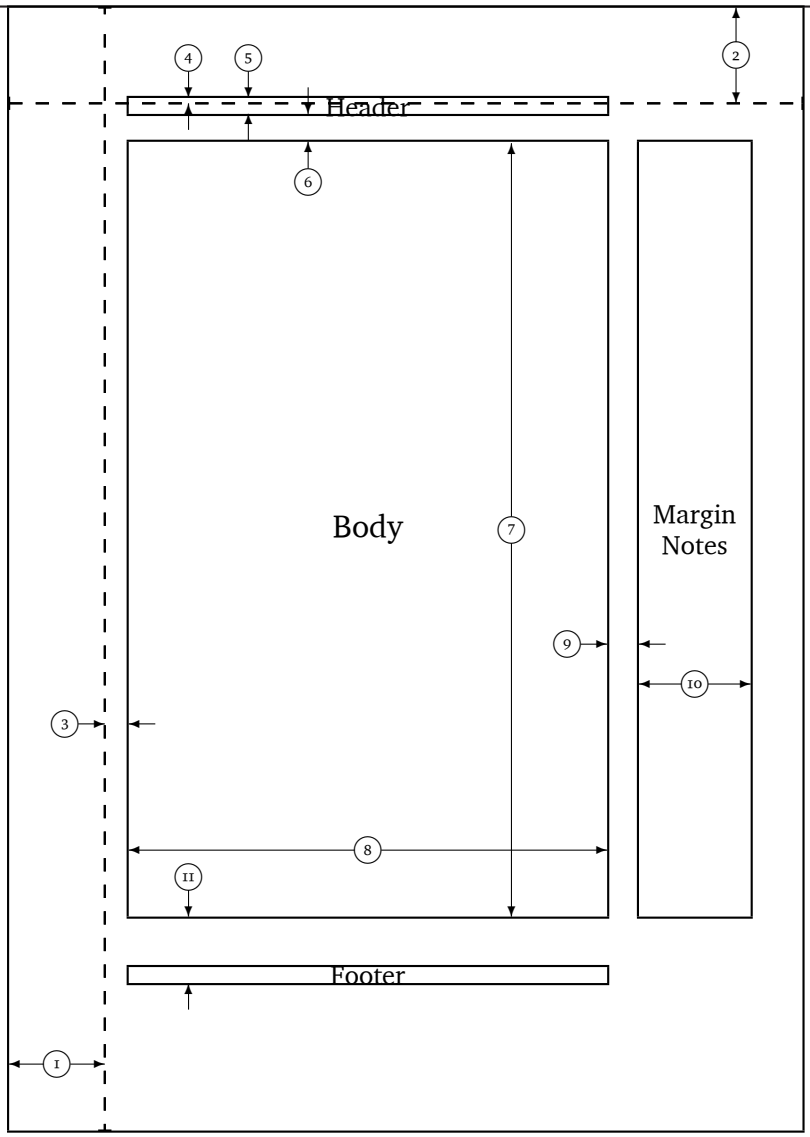
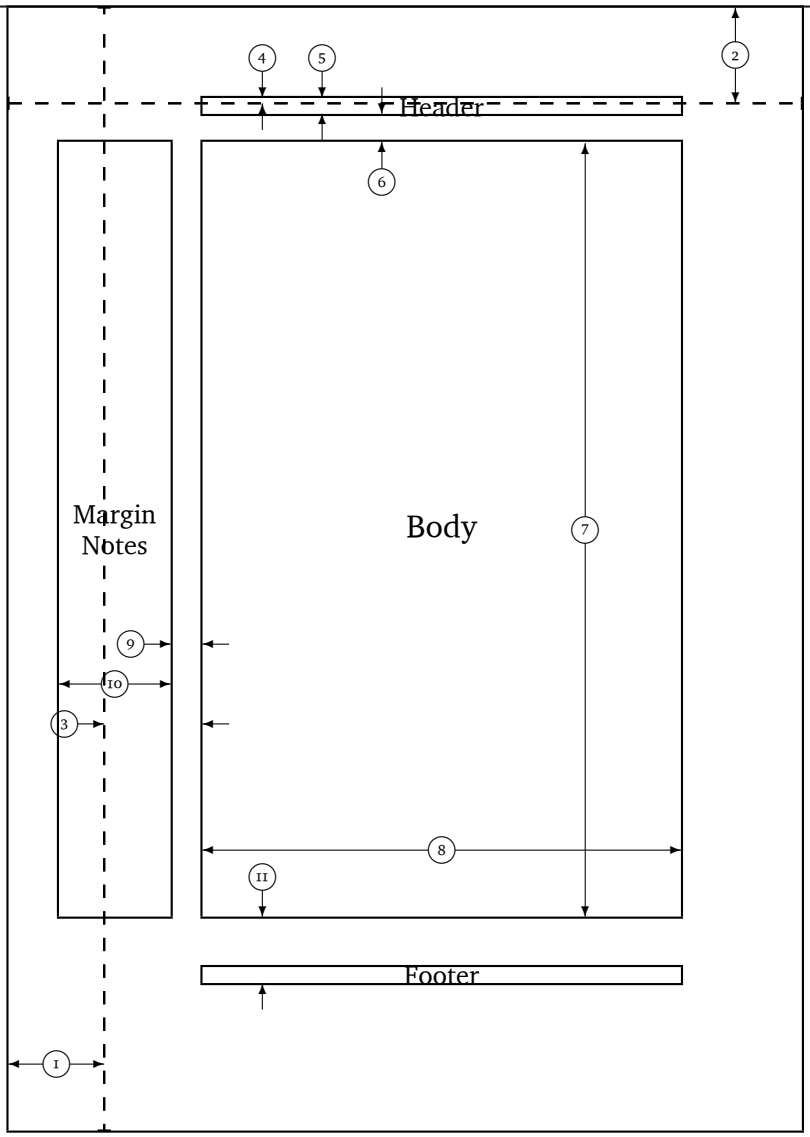
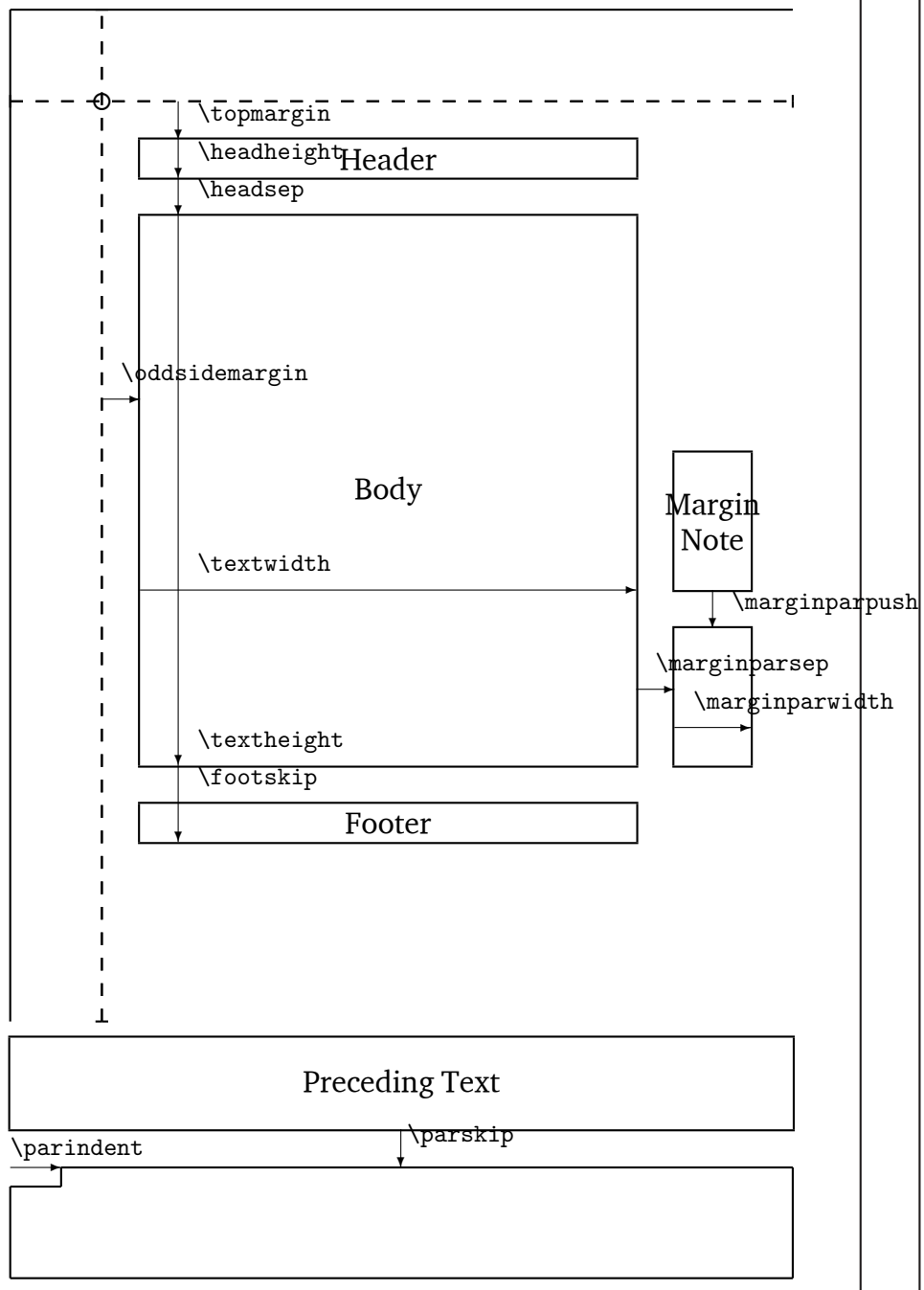



| | | | |
|---|---|---|---|
| 1 | one inch + \hoffset | 2 | one inch + \voffset |
| 3 | \oddsidemargin = 18pt | 4 | \topmargin = -4pt |
| 5 | \headheight = 12pt | 6 | \headsep = 21pt |
| 7 | \textheight = 583pt | 8 | \textwidth = 360pt |
| 9 | \marginparsep = 24pt | 10 | \marginparwidth = 84pt |
| 11 | \footskip = 50pt | | \marginparpush = 6pt (not shown) |
| | \hoffset = 0pt | | \voffset = 0pt |
| | \paperwidth = 597pt | | \paperheight = 845pt |




| | | | |
|---|---|---|---|
| 1 | one inch + \hoffset | 2 | one inch + \voffset |
| 3 | \evensidemargin = 74pt | 4 | \topmargin = -4pt |
| 5 | \headheight = 12pt | 6 | \headsep = 21pt |
| 7 | \textheight = 583pt | 8 | \textwidth = 360pt |
| 9 | \marginparsep = 24pt | 10 | \marginparwidth = 84pt |
| 11 | \footskip = 50pt | | \marginparpush = 6pt (not shown) |
| | \hoffset = 0pt | | \voffset = 0pt |
| | \paperwidth = 597pt | | \paperheight = 845pt |

The circle is at 1 inch from the top and left of the page. Dashed lines represent (`\hoffset + 1 inch`) and (`\voffset + 1 inch`) from the top and left of the page.




parindent 11.74988pt
parskip 0.0pt plus 1.0pt

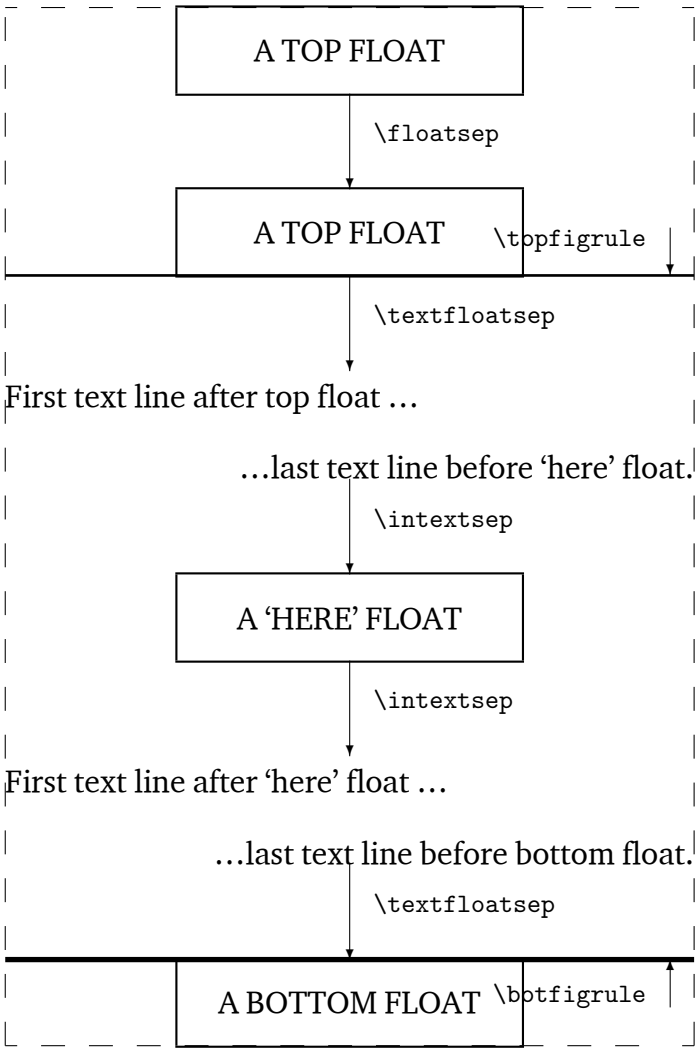
…end of last line of preceding text.

||*beforeskip*|| + `\parskip` (of text font) + `\baselineskip` (of heading font)

*indent* 3.5 Heading Title

*afterskip* + `\parskip` (of heading font) + `\baselineskip` (of text font)

This is the start of the after-heading text, which continues on …
second line of text following the heading …




152.93977pt

School of Engineering & Design
Electronic & Computer Engineering

MSc Distributed Computing Systems Engineering

Brunel University

# High Performance Computing on Graphic Processing Units

*Mean Shift Image Segmentation*

Zvonko Krnjaji

Supervisor: Dr. Abbes Amira

## 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, 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.

*supervisor, family, sponsor, others ...*
*We have seen that computer programming is an art,*
*because it applies accumulated knowledge to the world,*
*because it requires skill and ingenuity, and especially*
*because it produces objects of beauty.*

— ? [? ]

## Acknowledgments

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri, Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, and the whole LaTeX-community for support, ideas and some great software.

CONTENT OVERVIEW

# 1 INTRODUCTION

In recent years GPUs have moved from fixed pipeline processors to a fully programmable processor. This evolvement has attracted many developers to do general purpose computing on GPUs. GPUs have devoted there silicon (transistors) for computing engines rather than for control engines like caches, branch prediction, coherency protocols and more. This incredible computing power made algorithms with an high arithmetic density run by an order of magnitude faster than on CPUs. Speedups of 100× 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.

This thesis will cover all the topics to understand the architecture, programming model, software eco system, drawback and pitfalls when doing GPGPU. The GPU is a highly parallel processor with thousands of threads and a peak performance of 600 Giga FLoating point Operations Per Seconds (GFLOPs) (G92 core[1]).

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. CPU manufacturer 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 stigma is more cores means higher performance. Moore's law says that for every 2 years the amount of cores on a chip will double. 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.

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

1 http://www.nvidia.com/page/geforce_8800.html

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.

# 2 LITERATURE REVIEW

The first thing to answer is why should someone do general purpose computing on GPUs (GPGPU) anyway. For the most peopleCPUsare 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 [? ] and in [? ] CPU manufacturers are facing problems which they cannnot overcome just by increasing the frequency. The often cited *Walls* are first, *The Memory Wall*[? ], *The Frequency Wall* and *The Power Wall*. The only way seen by CPU 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* [? ]. IBM, Toshiba and Sony developed the *Cell Broadband Architecture* a 9 core chip [? ]. SUN 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 [? ].

Compared toCPUsGPUs went years ago to multicore and multithreading.GPUsare maybe the kind of processor whereCPUsare 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* [? ]. User will only benefit from this growth if software can make use of all the cores. 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 livelock, 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 [? ]. 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 for general purpose computing (Brook, Sh, RapidMind, ...).

The focus of this work will be CUDA[1]. CUDAis the only environment which is not based on a graphics library and officialy released by NVIDIA for their GPUs. CUDAis a minimal extension to C and C++ programming languages. The technique employed by CUDAis single process, multiple data (SPMD). Tasks are split up and run simultaneously on multiple threads (cores) with different input [? ].

## 2.1 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 theGPUsto make a decision if an algorithm can be mapped on GPUs. A pretty good overview over the NVIDIA GPU gives the *NVIDIA CUDAProgramming Guide* [? ]. A little bit outdatet but still of interest is *The GeForce 6 Series GPU Architecture*[? ] 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 intoGPUsthe article [? ] is highly recommended.

### 2.1.1 *Computations which Map Well to GPUs*

It is important to understand thatGPUsare 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 [? ]. 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*. GPU memory systems are architected to

1 www.nvidia.com

deliver high bandwidth, rather than low-latency, data access. As such computations that benefit most of theGPUshave a high arithmetic intensity [? ]. The next sections will represent some algorithms which could fit to GPUs.

### 2.1.2 *Ray Tracing*

Ray Tracing [? ] is an embarrassingly parallel alorithm which could fit well to GPUs. The author has a extensible knowledge of Ray Tracing on massively parallel computers [? ]. Thats why Ray Tracing was first considered for porting to the GPU. Unfortuneately there were several implementations already done for the GPU. Nevertheless equipped with all the knowledge about Ray Tracing and how to split up the work, arrange the data on a parallel machine to run efficiently the Ray Tracing algorithm [? ] will be used for initial benchmarks and the feasibility study Chapter 5.

### 2.1.3 *Photon Mapping*

Ray Tracing 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* [? ] can create all the effects that Ray Tracing cannot. There are implementations of photon mapping on GPUs [? ] 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 the first candidate for porting to the GPU is *Photon Mapping*.

### 2.1.4 *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 [? ]. 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 [? ].

The idea was to implement some of the factoring algorithms to the GPU. The only thing needed is the multiple precision library. In [? ] the Gnu Multiple Precision (GMP) library was used to implement the algorithms. So the multiple precision arithmetic is another candidate for porting to the GPU.

### 2.1.5 *High Dynamic Range*

Image processing is always a candidate for embarassingly parallel algorithms. A pretty new algorithm to enhance images is *tone mapping* [? ]. Tone mapping is the compression of dynamics in High Dynamic Range (HDR) pictures. There are several algorithms for tone mapping: Mantiuk [? ], Reinhard [? ], Durand [? ], Fattal [? ] and many more. All of this algorithms are present in the pfs-tools library written by Krawczyk [? ]. As this document was written Krawczyk was already implementing the pfs-tools on GPU but not publishing it. Furthermore there is an complete editor written for hdr image processing which is running on the GPU. So HDR was discarded but there are several other algorithms in image processing which are considered as candidates. Some algorithms in no particular order: segmentation, tracking, filtering, ... and so on. This is put as another candidate to the list of the possible algorithms for porting.

### 2.1.6 *Genetic Algorithms*

Parallel genetic algorithms are usually implemented on parallel machines but fine-grained parallel genetic algorithms can be mapped to GPUs [? ]. In [? ] its shown what kind of genetic algorithm map well toGPUsand how the work and communication is handled. It is *pretty* easy to implement simple genetic algorithms on the GPUs but with increasing complexity one has to consider many more things: load balancing, communication pattern, dynamic memory allocation, resolving of recursion ... . Another paper [? ] compares genetic algorithms implemented onCPUsandGPUsand 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 imple-

mentation of the core algorithm. So this topic excluded from the candidate list.

### 2.1.7 *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 [? ]. 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 [? ] 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 [? ] 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.

### 2.1.8 *Image Processing*

Many image processing algorithms are embarrassingly parallel as one can calculate filters on idividual pixels without considering the neigbouring pixels.

## 2.2 OLD STUFF

Amdahl's Law: The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the

faster mode can be used. http://www.cag.csail.mit.edu/ps3/lectures/6.189-lecture5-parallelism.pdf

*Typical Software Development Flow*

- Algorithm complexity study
- Data layout/locality and Data flow analysis
- Experimental partitioning and mapping of the algorithm and program structure to the architecture
- Develop CPU Control, CPU scalar/multicore code
- Develop CPU Control, partitioned GPUscalar code (Communication, synchronization, latency handling)
- Transform GPUscalar code to GPUthreaded code, multi GPUcode
- Re-balance the computation / data movement
- Other optimization considerations (load balancing, bottlenecks...)

### 2.2.1 *An Overview of Stream Computation*

streaming programming model... sdk's ..

### 2.2.2 *General Programming of Streaming Processors*

SDK's, CTM, Anbieter, verschieden Programiersprachen

### 2.2.3 *Stream Computing in Field Use*

Einsetzbarkeit, Leistung und Effizienz von GPUApplikationen im nicht-grafischen Applikationsbreich Kontext, Referenzen (wo wird eingesetzt im nicht-grafik Bereich)

### 2.2.4 *NVIDIA CUDA*

Tesla, Stil, Aufwand, Debugbarkeit, Portieraufwand für General Purpose Anwendungen, Vergleich zu SPUFS

### 2.2.5 *The Classic GPU Pipeline*

### 2.2.6 *The GeForce 8 Series Architecture*

## 2.3 Algorithmic point of view to GPUs

Welche Algorithmen eignen sich fuer GPUs

*Computational Concepts*

*Efficient Data Structures*

*Program Optimization*

*…or your supervisor might use the margins for some comments of her own while reading.*

# EXAMPLES

3

## 3.1 Proposed Solution

## 3.2 Experimental Results

Zugriffsarten, Profiling, Latencies, Bandbreiten, Berechnungen Is the DMA engine determnisitic?

## 3.3 ... on the GPU

*Analysis*

10.2 Reducing Cost of Fitness with Caches

*Design*

## 3.4 Methodology and Discussion

Methods applied, Results achieved if not why, Benchmarks, it would be good if results are discussed in first placed and then discrepancies here discussed.

## 3.5 Conclusion, Questions, Perspective

## 3.6 Summary

bal bla blabub

# 4 PARALLEL PROCESSING WITH GPUS

## 4.1 Parallel Architectures

*Single Program Multiple Data (SPMD)*

## 4.2 The Tesla Architecture

. The Tesla architecture announced 1999 and developed by NVIDIA is the first GPU highly specialized for raster operations and more important for general purpose computing. Formerly GPUs.ad 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 [? ]. Furthermore manufacturers introduced the Unified Shader Model (USM) that unifies the processing units allowing for better utilization 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 [? ].

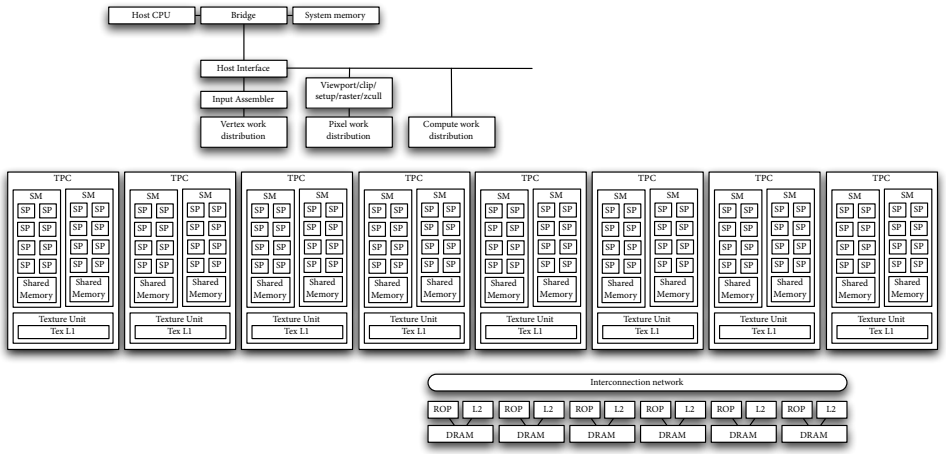



Figure 4.1: Tesla Architecture

The figure Figure 4.1.shows the Tesla architectures. As mentioned above the new GPU.architectures are a radical departure from traditional GPU.design (USM). The Tesla 8 Series has 16 multiprocessors. Each multiprocessor is composed of 8 streaming processors, 128 processors in total. Each streaming multiprocessor has 16 KB of shared memory and a L1 cache attached and has access to a texture unit. A streaming processor consists of a scalar ALU and performs floating point operations. 32 streaming processor build up a SIMD unit (warp) in that one instruction is executed. The 8800 GTX has 768 MB of graphics memory, 620 GFlops of peak performance and 86 GB/s peak memory bandwidth. Many massively data-parallel algorithms can be run sufficiently on this specialized architecture [? ]. Programming for this architecture is done with CUDA.Common Unified Device Architecture) the C language extension which will be covered in the next section.

## 4.3 Common Unified Device Architecture (CUDA)

. In 2007, NVIDIA introduced an extended ANSI C programming model and software environment the Compute Unified Device Architecture (CUDA). The reason why CUDA was is 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 goals when CUDA.as developed were that it scales to 100s of cores, 1000s 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.nd GPUs.citepciteulike:3839013.

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 [? ].that are exposed to the developer. CUDA.ses 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 aggresively parallel. The problem has to be split into indepedent coarse sub-problems and at finer level into fine-grained sub-problems that can be solved cooperatively [? ].

1 Processor count is doubling every 18 - 24 months

CUDA has been accepted be many developers which can be seen by the huge amount of already developed software and contributions to the CUDA.one: *www.nvidia.com/cuda*. A brief look shows the CUDA.omputing sweet spots[? ].

- 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.n GPUs. For a more extensive list and speedups compared to high end CPUS see *www.nvidia.com/cuda*.and *www.gpGPU.org*.

## 4.4 CUDA Programming Model

. The CUDA.rogramming model exposes the graphics processor as a highly multithreaded coprocessor. The graphics processor is viewed as a compute device that is a coprocessor 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 efficience where CPUs.nly 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[2].and efficiently sharing data through the low latency shared memory. Two threads from two different blocks can cooperate with atomic functions through the global memory. The identification

2 For hazard free shared memory access

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 have 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 1D or 2D layout. Thread ids can be accessed either with 1D, 2D or 3D coordinates similar to multidimensional arrays. It simplifies memory addresing when processing multidimensional data. For example image processing, matrix multiplication or solving partial differential equations on volumes and so on. 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 (host) can (rw) global, constant and texture memories.

Global, constant and texture memory have long latency accesses. They reside off chip where registers local[3].and shared memory reside 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 textures are initialized.

## 4.5 A Simple Example

. This simple example will show the structure of an CUDA.rogram. 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.rogram 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. Listing ??.shows the initalization.

3 Not true for older GPUs.hips where local data is spilled out to global memory

```
int main( int argc, char** argv) {
    CUT_DEVICE_INIT(argc, argv);

    uint32_t num_threads = 32;
    uint32_t mem_size = sizeof(float) * num_threads;

                    ...
```

Since the GPU.is attached to the PCIe 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 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 ??.

```
    // allocate host memory
    float* h_idata = (float*) malloc(mem_size);
    // allocate device memory
    float* d_idata; cudaMalloc((void**) &d_idata,
        mem_size);
    // allocate device memory for result
    float* d_odata; cudaMalloc((void**) &d_odata,
        mem_size);
    // allocate mem for the result on host side
    float* h_odata = (float*) malloc( mem_size);
    // copy host memory to device
    cudaMemcpy(d_idata, h_idata, mem_size,
        cudaMemcpyHostToDevice);
```

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 ??.shows the call of the kernel with its input and output data.

```
    // setup execution parameters
    dim3  grid( 1, 1, 1);
```

```
            dim3  threads(num_threads, 1, 1);

            // execute the kernel
            kernel<<< grid, threads, mem_size >>>(d_idata,
                d_odata);
```

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 ??.shows the last steps before exiting the program.

```
            // check if kernel execution generated and
                error
            CUT_CHECK_ERROR("Kernel execution failed");

            // copy result from device to host
            cudaMemcpy(h_odata, d_odata, sizeof( float) *
                num_threads,
                        cudaMemcpyDeviceToHost);

            // cleanup memory free(x), free(y), free(z) ...
            CUT_EXIT(argc, argv);
        }
```

The previous listings showed the host code and how to launch a kernel on the device. The listing ??.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 [? ].

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 ?? the *__shared__*.qualifier is used. The device will use the shared memory to preload the data for faster access.

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

```
#define SDATA(index) CUT_BANK_CHECKER(sdata, index)
// Simple test kernel for device functionality
__global__ void kernel( float* g_idata, float*
    g_odata)
{
  // shared memory
  // the size is determined by the host application
  extern  __shared__  float sdata[];

  // access thread id
  const unsigned int tid = threadIdx.x;
  // access number of threads in this block
  const unsigned int num_threads = blockDim.x;

  // read in input data from global memory
  // use the bank checker macro to check for bank
      conflicts during host
  // emulation
  SDATA(tid) = g_idata[tid];
  __syncthreads();

  // perform some computations
  SDATA(tid) = (float) num_threads * SDATA( tid);
  __syncthreads();

  // write data to global memory
  g_odata[tid] = SDATA(tid);
}
```

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.

## 4.6 Porting Strategies for GPUs

.

# FEASIBILITY STUDY

5

There are several myths about GPGPU which will be examined and solved with a feasibility study. 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 Single Instruction Multiple Data (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 a 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.

## 5.1 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].

## 5.2 Ray Casting

Ray casting was first introduced by ? [? ] 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 [? ]. 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 [? ].

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

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 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

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

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. ? [? ] estimates that anywhere from 75 percent to over 95 percent 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 30.720.000 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.

## 5.3 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 is depend 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 REF. to CUDA Application Stuff 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 we 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 SM. These calculations can easily be done with the *CUDA occupancy calculator* supplied with the SDK.

After fiddling around with compiler switches especially *-maxrregcount=x* the application used only 14 register which meant we could have a block-

*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*

size 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.

The Table 5.1 shows the run configuration used for the sample runs.

| blockSize.x | blockSize.y | gridSize.x | gridSize.y | Regs. per Thread |
|---|---|---|---|---|
| 8 | 8 | 96 | 96 | 14 |

TABLE 5.1: Run 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 Table 5.2 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 5.2: Comparison between CPU and GPU.

As it can be seen in Table 5.2 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!

The reasons for this uneffective usage of processing power can be seen in Table 5.3a and Table Table 5.4. 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 actually reflected by the value 22578032.

*All performance values are gathered with the NVIDIA Performance Profiler availabe as well with the Software Development Kit (SDK)*

| Load | Store |
|---|---|
| 63,294,179 | 328,180,560 |

(A) Local memory

| Sum | Divergent |
|---|---|
| 22,578,032 | 26,834 |

(B) Branches

TABLE 5.3: 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 Table 5.4 for the values.

| Load Incoherent | Load Coherent | Store Incoherent | Store Coherent |
|---|---|---|---|
| 1,115,657,863 | 67,961 | 2,506,752 | 0 |

TABLE 5.4: 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 G90 chip with peak 620 GFLOPs

## 5.4 Demystifing the Myths

There were several myths about GPGPU see Chapter 5 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.

# MEAN SHIFT

6

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 ? [? ].

## 6.1 Density Estimation

In probability and statistics it is known that for different tasks there exist more or less suitable features. In ? [? ] 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.

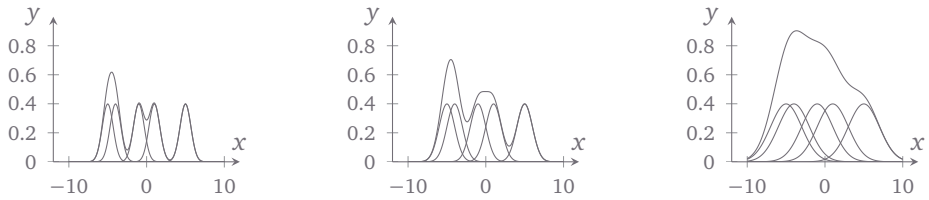
The *mean shift* method is based on the gradient method. For the gradient estimation a function is upon estimations of discrete observations in the feature space. For this kernel density estimators are used also known as *Parzen Window* method.

## 6.2 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 6.1. 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 obser-

vation, 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 6.1: 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

$$\hat{f}(x) = \frac{1}{N} \sum_{n=1}^{N} K_{\mathbf{H}} (x - x_n) \tag{6.1}$$

where

$$K_{\mathbf{H}}(x) = \frac{1}{\sqrt{|\mathbf{H}|}} K \left( \frac{x - x_n}{h} \right) \tag{6.2}$$

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 6.1 can be written as

$$\hat{f}(x) = \frac{1}{Nh^D} \sum_{n=1}^{N} K \left( \frac{x - x_i}{h} \right). \tag{6.3}$$

The kernel density estimator is valid for several kernels and successive considerations will deal with several kernel the Equation 6.3 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 ? [? ]. 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

$$K(x) = c_{k,d} k(\|x\|^2) \tag{6.4}$$

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 6.3 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 6.3 is inserted into Equation 6.2, the Equation 6.3 is transformed to

$$\hat{f}_{h,K}(x) = \frac{c_{k,D}}{Nh^d} \sum_{n=1}^{N} k\left( \left\| \frac{x - x_n}{h} \right\|^2 \right) \tag{6.5}$$

## 6.3 Kernel and their Properties

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

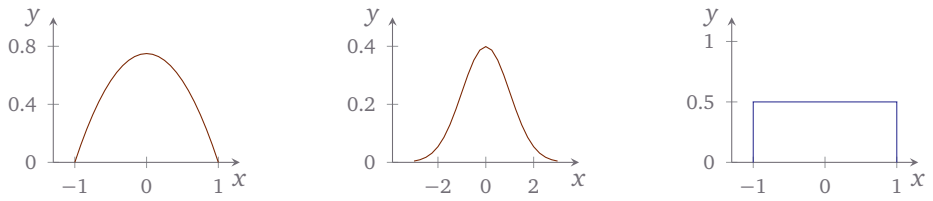
From the Epanechnikov profile

$$k_E x = \begin{cases} 1 - x & 0 \leq x \leq 1 \\ 0 & x > 1 \end{cases}, x \in \mathbb{R} \tag{6.6}$$

follows a radial symmetric kernel

$$K_E(x) = \begin{cases} \frac{1}{2} c_D^{-1} (D + 2)(1 - \|x\|^2) & \|x\| \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R}^D \tag{6.7}$$

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) [? ]. The Figure 6.2a shows the epanechnikov kernel. The dervative of the kernel is a uniform profile.




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

FIGURE 6.2: Kernel density estimators

From the normal profile

$$k_N(x) = exp\left(-\frac{1}{2}x\right) \text{ where } x \geq 0, x \in \mathbb{R} \tag{6.8}$$

follows the normal kernel

$$K_N(x) = (2\pi)^{-D/2} exp\left(-\frac{1}{2}\|x\|^2\right), x \in \mathbb{R}. \tag{6.9}$$

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

$$k_U(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R} \tag{6.10}$$

follows the uniform kernel

$$K_U(x) = \begin{cases} 1 & \|x\| \leq 1 \\ 0 & otherwise \end{cases}, x \in \mathbb{R}^D \tag{6.11}$$

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

$$G(x) = c_{g,D} g(\|x\|^2) \tag{6.12}$$

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 [? ]. The mean shift vector of a kernel points to the same direction as the gradient of the shadow kernel (See ??).

## 6.4 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 ? [? ].

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 ? [? ]).

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

$$m_{h,G}(x) = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \tag{6.13}$$

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 6.2 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.

### 6.4.1 *Density Gradient Estimation*

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

$$\hat{\nabla} f_{h,K}(x) \equiv \nabla \hat{f}_{h,K}(x) = \frac{2_{C_{k,D}}}{Nh^{D+2}} \sum_{n=1}^{N} (x - x_n) k'\left(\left\|\frac{x-x_n}{h}\right\|^2\right) \tag{6.14}$$

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

$$\begin{aligned} \frac{\delta}{\delta x} k\left(\left\|\frac{x-x_n}{h}\right\|^2\right) &= \left(\left\|\frac{x-x_n}{h}\right\|\right)' k'\left(\left\|\frac{x-x_n}{h}\right\|^2\right) \\ &= 2\left(x - x_n\right)\left(\frac{1}{h}\right) k'\left(\left\|\frac{x-x_n}{h}\right\|^2\right) \\ &= \frac{2}{h^2}(x - x_n) k'\left(\left\|\frac{x-x_n}{h}\right\|^2\right). \end{aligned} \tag{6.15}$$

Using $g(x) = -k'(x)$ and with Equation 6.12 a new kernel $G(x)$ with profile $g(x)$ can be defined. Transforming Equation 6.14 with the new profile $g(x)$ the gradient of the density estimator becomes

$$\hat{\nabla} f_{h,K}(x) = \frac{2c_{k,D}}{Nh^{D+2}} \sum_{i=1}^{N} (x_n - x)\, g\left(\left\| \frac{x - x_n}{h} \right\|^2\right) \tag{6.16a}$$

$$= \frac{2c_{k,D}}{Nh^{D+2}} \left[\sum_{n=1}^{N} g\left(\left\| \frac{x - x_n}{h} \right\|^2\right)\right] \left[\frac{\sum_{i=1}^{N} x_i g\left(\left\| \frac{x-x_n}{h} \right\|^2\right)}{\sum_{n=1}^{N} g\left(\left\| \frac{x-x_n}{h} \right\|^2\right)} - x\right]. \tag{6.16b}$$

The first term of Equation 6.16b conforms with the density estimator $\hat{f}_{h,G}(x)$ for kernel $G$ (compare with Equation 6.5 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 6.13. 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 6.16b follows

$$\hat{\nabla} f_{h,K}(x) = \frac{2c_{k,D}}{h^2 c_{g,D}} \hat{f}_{h,G}(x) m_{h,G}(x) \tag{6.17}$$

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

$$m_{h,G}(x) = \frac{1}{2} h^2 c \frac{\hat{\nabla} f_{h,K}(x)}{\hat{f}_h, G(x)}, \text{ whereas } c = \frac{c_{g,D}}{c_{k,D}}. \tag{6.18}$$

The denominator of Equation 6.18 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 ? [? ].

Let be

$$mi_{h,K}(x) = \frac{\sum_{i=1}^{n} x_i k\left(\left\| \frac{x-x_i}{h} \right\|^2\right)}{\sum_{i=1}^{n} k\left(\left\| \frac{x-x_i}{h} \right\|^2\right)} - x \tag{6.19}$$

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 $G$

$$m_{h,G}(x) = mi_{h,G}(x) - x = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \tag{6.20}$$

lies in the gradient density estimator direction with kernel $K$

$$\hat{f}_{h,K}(x) = \frac{c_{k,D}}{Nh^d} \sum_{n=1}^{N} k\left(\left\|\frac{x - x_n}{h}\right\|^2\right) \tag{6.21}$$

In the following sections gradients will not more be considered as it was shown in Equation 6.18 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.

### 6.4.2 *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 6.5
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 6.1: 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 6.5 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. ? [? ]).

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:

$$y_{j+1} = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} \quad j = 1, 2, \ldots \tag{6.22}$$

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 ? in [? ].

## 6.5 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 ? [? ] 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

$$K_{h_r,h_f}(x) = \frac{C}{h_r^2 h_f^p} k\left(\left\|\frac{x^r}{h_r}\right\|^2\right) k\left(\left\|\frac{x^f}{h_f}\right\|^2\right) \tag{6.23}$$

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.

### 6.5.1 *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 6.3: 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 6.2.

Given $x_n = (x_n^r, x_n^f), n = 1, \ldots, N$ are the $D$-dimensional pixels of the original image and $z_n, n = 1, \ldots, N$ are the $D$-dimensional filtered pixels.
For each pixel $n = 1, \ldots, N$

1. Initialize $k = 1$ and $y_k = x_n$
2. Calculate $y_{k+1}$ according to Equation 6.21, repeat until convergence
3. Set $z_n = (x_n^r, y_{conv}^f)$

ALGORITHM 6.2: 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 6.3 was smoothed with the algorithm of Algorithm 6.2. A uniform kernel with $h = (6.5, 7)$ was used.

### 6.5.2 *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.

The Algorithm was applied on the image Figure 6.4a. A uniform ker-

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

FIGURE 6.4: Mean shift segmentation with parameters $(h_r, h_f, M) = (6.5, 7, 20)$ applied to a color image

nel 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 6.4 shows the result of a segmentation. After the image segmentation follows a edge detection by examining the cluster regions Figure 6.4c.

In filtering as well as in segmentation algorithm a fixed sized window size is used. In ? [? ] 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 segmenta-

tion part of the item 3 has no significant impact on the run time.

$x_n = (x_n^r, x_n^f), n = 1, \ldots, N$ are the $D$-dimensional pixels of the original image and $z_n, n = 1, \ldots, N$ are the $D$-dimensional filtered pixels.
For each pixel $n = 1, \ldots, N$

1. Convert feature vector $x_n^f$ to Luv color space
2. Initialize $k = 1$ and $y_k = x_n$
3. Calculate $y_{k+1}$ according to Equation 6.21, repeat until convergence
4. Set $z_n = (x_n^r, y_{conv}^f)$

ALGORITHM 6.3: Mean shift segmentation

# 7 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 simultaenously to 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 size 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 ? of [? ] offers functionality to filter, segment and detect edges in images.

The Figure 7.1 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 6.3a. 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 obviuos, as stated in Chapter 6 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

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

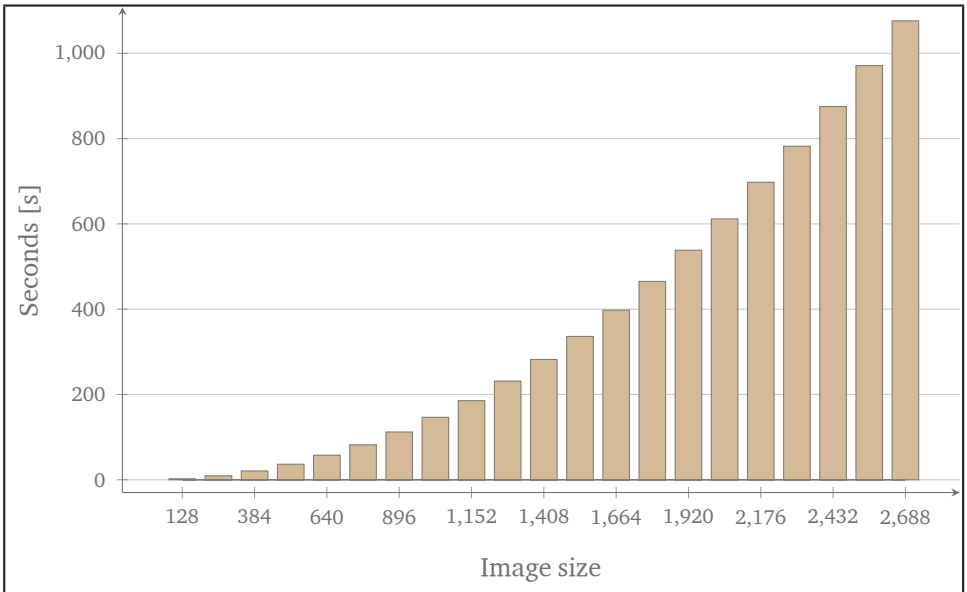



FIGURE 7.1: CPU run time depending on the image size

clue to which 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.

## 7.1 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 7.1 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

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 7.1: EDISON run time analysis

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 and the remaining columns are irrelevant for now.

Now having a look at the table there is one function which is executing 98.2% of the run time, the function *MeanShift::uniformLSearch(double*, double*)*. In this function the algorithm tries to find feature points that fall into the search window with radius $h$ (see Section 6.4). 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

$$S(n) = \frac{T_1}{T_n} \tag{7.1}$$

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

## 7.2 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

$$S_{max}(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{7.2}$$

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 ? [? ].

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 5.2 for the filtering step:

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

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 ? 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:

$$S_{max}(n) = \frac{s' + p' * n}{s' + p'} = n + (1 - n) * s' \tag{7.3}$$

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

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

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 [? ].

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.

## 7.3 Data and Task Parallelism

### 7.3.1 *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 ? 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 [? ]:

$$I_j \cap O_i = \emptyset \tag{7.4a}$$
$$I_i \cap O_j = \emptyset \tag{7.4b}$$
$$O_i \cap O_j = \emptyset \tag{7.4c}$$

Equipped with the knowledge how to identify independent parallel tasks, the mean shift segmentation algorithm item 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 and the output data of each step is written to the same location for each pixel. 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 item 3. The longest path of dependent calculations is the critical path. For the mean shift algorithm there exist no shorter path.

Having a look again at Table 5.2 where the most computationally intensive parts are identified, one could think of having different tasks for the filtering step. But again here the same situation each calculation is inherently dependent on each other.

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.

### 7.3.2 *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 of the 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.

### *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 7.3.1 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 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. In Section 7.3.2 it was said that each processing unit gets a chunk of the image for processing. The mean shift method takes

CUDA Emulator Output

As many people have noticed the same code executed in Emulator mode gives different floating point results from the kernels run in Debug or Release mode. Although I know what causes this I have never bothered to investigate the actual differences as most of the stuff I write runs entirely on the GPU. Recently I have had to compare results on the CPU<->GPU and wrote some code to change the FPU settings. Firstly a quick explanation: By default the CPU (FPU) is set to use 80 bit floating point internally. This means that when you load in an integer (fild) or a single / double float (fld) it gets converted to a 80 bit number inside the FPU stack. All operations are performed internally at 80 bits and when storing the result it converts back to the correct floating point width (single / double) (fst / fstp). This method of operation is desirable as it reduces the effect of rounding / truncating on the intermediate results. Of course while very useful for computing on the CPU this is not how the CUDA devices operate.

In CUDA all operations on a float occur at 32 bits (64 bits for a double) which means your intermediate operations will sometimes lose precision. In CUDA Emulator mode your code is actually run on the CPU and it uses the FPU's default precision and rounding settings. This causes the difference in output.

For my testing I modified the Matrix Mul sample in the CUDA SDK to include code to change the CPU settings before running the Gold Kernel. (Code link follows below)

I turned down the CPU internal precision to 32 bits in order to match the 32bit floats the CUDA kernel uses. For emulator mode I made sure the CPU was turned down to the same precision before running the Kernel. As expected the Gold and CUDA kernels outputs match perfectly.

Next I ran in Debug mode (the kernel will now execute on the GPU). As both the Gold kernel and Cuda kernel are now at 32 bits I expected the results to be the same. Rather interestingly it turned out that they are slightly different. I then tried changing the CPU rounding settings hoping to get the results to match up.

After trying all the rounding settings I discovered that the default set-

ting (round to nearest or even) gave the closest results to the Gold kernel BUT they are still slightly out. I suspect this is down to differences in the internal workings of the FPU units on the GPU.

So in summary: If you are trying to compare kernel results between Emulator and Release mode you will never get exactly the same results but the differences can be mitigated somewhat by changing the CPU/FPU's internal precision settings.

# 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 onyl valid for CUDA, they can be applied to many parallel machines which are build after the shared memory model.

## 9.1 Offload Compute Intensive Parts

The amount of performance benefit an application will realize by running on CUDA depends entirely on the extent to which it can be parallelized. As mentioned previously, code that cannot be sufficiently parallelized should run on the host, unless doing so would result in excessive transfers between host and device. Amdahl's law specifies the maximum speed-up that can be expected by parallelizing portions of a serial program. Essentially, it states that the maximum speed-up (S) of a program is

$$S = \frac{1}{(1-p) + \frac{P}{N}} \tag{9.1}$$

where P is the fraction of the total serial execution time taken by the portion of code that can be parallelized and N is the number of processors over which the parallel portion of the code runs. The larger N is (that is, the greater the number of processors), the smaller the P/N fraction. It can be simpler to view N as a very large number, which essentially transforms the equation into $S = 1/1 - P$. Now, if 34 of a program is parallelized, the maximum speed-up over serial code is $1/(1 - 3/4) = 4$ . For most purposes, the key point is that the greater P is, the greater the speed-up. An additional caveat is implicit in this equation, which is that if P is a small number

(so not substantially parallel), increasing N does little to improve performance. To get the largest lift, best practices suggest spending most effort on increasing P; that is, by maximizing the amount of code that can be parallelized.

The first naive implementation of the mean shift filter resulted in a speedup of

$$S = 11310\ ms/1942\ ms = 5,82389$$

times faster than the CPU version. The reference time was generated from EDISON. It reports timings for filtering segmentation.

### Optimize Access to Global Memory

One important fact for coalescing is the sequential accesses to global memory should be grouped together. A simple optimization is to use the *float4* data type available in CUDA. After rewriting the algorithm the new speedup was

$$S = 11310\ ms/1464\ ms = 7,7254$$

basin of attraction, random access to global memory, not knowing where each pixel is moving to. Switching to *textures* (glossary texturess) yield to a huge speedup of

$$S = 11310\ ms/260\ ms = 43,5$$

### Avoid Expensive Divisions

If possible precalculate divisions. For example if it looks like this

```
dl = (luv.x − yj_2) / sigmaR;
du = (luv.y − yj_3) / sigmaR;
dv = (luv.z − yj_4) / sigmaR;
```

Listing 9.1: Divison

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

```
dl = (luv.x − yj_2) * rsigmaR;
du = (luv.y − yj_3) * rsigmaR;
```

```
dv = (luv.z − yj_4) * rsigmaR;
```

Listing 9.2: Precalculated Divison

This optimization yielded a speedup of

$$S = 11310\ ms/203\ ms = 55,71428$$

## 9.2 Run Configurations

It is important to check the best run configuration. Each run configruation has its benefit. Some can exploit the bandwidth to the global memory some can benefit from the cache of the texture. After testing all configurations the best fit was a $8x32 = 256threads$ configuration.

$$S = 11310\ ms/181\ ms = 62,48618$$

## 9.3 Use the Float Data Type where Ever Possible

One should use the $float$ data type where ever possible. GPUs are highly optimized for floating point calculations. After converting all integer calculations to floating point operations another significant speedup was achieved.

$$S = 11310\ ms/175\ ms = 64,62857$$

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

$$S = 11310\ ms/125\ ms = 90,48$$

## 9.5 Shared Memory

The optimization guides state that one should use shared memory to avoid redundant transfers from global memory. But in this case the accesses are not known in advance and would lead to heavy reduce in run time as the access to shared memory from the threads would lead to many bank conflicts.

## 9.6 Know the Algorithm

experiments to know how many iterations are used per pixel.... varying the $limit$ Setting the $lim = 10$ and executing the CPU and GPU version we get

$$S = 11310\ ms/125\ ms = 254,10344$$

Setting the $lim = 50$ and executing the CPU and GPU version we get

$$S = 11310\ ms/125\ ms = 126,5$$

(A)Period-0 Limitcycle (B)Period-4 Limitcycle (C)Period-8 Limitcycle

Figure 9.1: Effect of limitcycle detection on iteration count

Looking at the iteration count iter.txt one can see that pixel 10992 has an iteration count of 100. $i = 10992 mag = 2,5 iter = 100$. Examining the sequence of the magnitude one can easily see that after some iterations the magnitude is fixed to 2.5. This behaviour is known as a fixed point limit cycle.

FIGURE 9.2: Visualization of Iteration Count

*Limit Cycle, Iterating Dynamical Systems*

bla bla limit cycle attractors...

examining $i = 15762$ shows that the iteration is a period-2 limit cycle.

Implementing a simple period-4 limit cycle detection yields to a speedup of

$$S = 11310\ ms/99\ ms = 114,24242$$

Implementing a simple period-8 limit cycle detection yields to a speedup of

$$S = 11310\ ms/87\ ms = 130$$

## 9.7 UNROLLING LOOPS AND MULTIPLICATIONS

After unrolling the multiplications

$$S = 11310\ ms/83\ ms = 136,26506$$

## 9.8 Extra luv to rgb Kernel

After a lot of optimization the most computational task became really small. Now small function which where a tiny fraction of the complete run time became significant. After implementing a $luvtorgb$ kernel the speedup is

$$S = 11310\ ms/75\ ms = 150,8$$

explains how the PPM image is read in and written back to. cutLoadPPM4ub() function reads the PPM file which is in RGB (0-255) format and pads a zero to the fourth byte, thereby making the data to be four bytes in total. This padding is very helpful because memory coalescence and better performance is achieved by accessing 1/2/4 consecutive memory locations at a time.

# MANAGMENT OF THE PROJECT

11

# LIST OF FIGURES



# LIST OF TABLES



# LIST OF LISTINGS

## Colophon

This thesis was typeset with LaTeX $2_\varepsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by ? 's genius as presented in *The Elements of Typographic Style* [? ]. It is available for LaTeX via CTAN as "`classicthesis`".

NOTE: The custom size of the textblock was calculated using the directions given by Mr. Bringhurst (pages 26–29 and 175/176). 10 pt Palatino needs 133.21 pt for the string "abcdefghijklmnopqrstuvwxyz". This yields a good line length between 24–26 pc (288–312 pt). Using a "*double square textblock*" with a 1:2 ratio this results in a textblock of 312:624 pt (which includes the headline in this design). A good alternative would be the "*golden section textblock*" with a ratio of 1:1.62, here 312:505.44 pt. For comparison, `DIV9` of the `typearea` package results in a line length of 389 pt (32.4 pc), which is by far too long. However, this information will only be of interest for hardcore pseudo-typographers like me.

To make your own calculations, use the following commands and look up the corresponding lengths in the book:

```
\settowidth{\abcd}{abcdefghijklmnopqrstuvwxyz}
\the\abcd\ % prints the value of the length
```

Please see the file `classicthesis.sty` for some precalculated values for Palatino and Minion.