

Institute of Visualization and Interactive Systems  
University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Diplomarbeit Nr. 2924

**Algorithm Design and Algorithmic-Level  
Optimization of Video / Image Algorithm using an  
Abstract Common Interface for NVIDIA CUDA and  
Intel Larrabee Platforms**

Daniel Kauker

<b>Course of Study:</b>	Computer Science
<b>Examiner:</b>	Prof. Dr. Thomas Ertl
<b>Supervisor:</b>	Dipl.-Inf. Steffen Frey Dipl.-Inf. Harald Sanftmann
<b>Commenced:</b>	May 11, 2009
<b>Completed:</b>	November 10, 2009
<b>CR-Classification:</b>	I.3.6, G.1.2, D.2.11



## Abstract

*This diploma thesis has three topics: Make it possible to use the Fourier transform on CUDA devices using the NVIDIA CUFFT library, port a sub system of a motion estimator to the graphics card and finally the design and implementation of an abstraction layer for many core systems.*

*The problem of using the CUFFT library is its high memory consumption, especially for images of more than 8 Megapixel in size. Evaluating this shows, that it can bring current standard graphics cards to their limit. The solution to this problem is to split up the two-dimensional Fourier transform into two one-dimensional transforms. Its mathematical definition allows to transform larger images without any loss of performance.*

*The motion estimator sub system is the so called cleaning part. Its task is to smoothen the vectors which were created by the estimation when trying to find the direction a block of a frame moved to. It will be shown that today's CUDA hardware suffers from heavy performance losses when compared to a CPU implementation.*

*The third task was to design and implement an abstraction layer for the hardware of a many core system. Its particularly features are the hardware abstraction which allows to write code once and then compile and use it with every architecture available on the system. Additionally, it allows to split up its given tasks and distribute them over the computation devices available.*

---

## Zusammenfassung

*Diese Diplomarbeit beschäftigt sich mit drei großen Themenkomplexen: Die Fouriertransformation mittels der CUFFT Bibliothek von NVIDIA auf Grafikkarten mit wenig Speicher zu benutzen zu könne, einem Subsystem eines Bewegungsschätzers, der auf die Grafikkarte portiert werden soll und dem Design eines Abstraction Layers zur Abstrahierung von Hardware in einem Many-Core System.*

*Das Problem der CUFFT Bibliothek ist der hohe Speicherverbrauch, der besonders bei großen Bildern ab 8 Megapixeln zum Tragen kommt. Messungen zeigen, dass der benötigte Speicher handelsübliche Grafikkarten an ihr Limit bringt. Die Lösung besteht darin, die zweidimensionale Fouriertransformation in zwei eindimensionale Transformationen zu zerlegen. Durch diese mathematische Eigenschaft wird es möglich, auch viel größere Bilder transformieren zu können ohne dabei Geschwindigkeitseinbußen zu haben.*

*Bei dem Teilsystem des Bewegungsschätzers handelt es sich um den sogenannten Cleaning Part. Er sorgt dafür, die Vektoren, die die Verschiebung/Bewegung auf einem Bild darstellen sollen, zu glätten. Dadurch sollen mögliche Ausreißer, die durch Fehleinschätzungen entstanden sind, geglättet werden. Es wird gezeigt werden, dass dies auf der heutigen CUDA Hardware von NVIDIA nur unter großen Performanceeinbußen gegenüber einer CPU-Implementierung zu bewerkstelligen ist.*

*Die dritte Teilaufgabe war die Konzeption und Implementierung einer Abstraktionsschicht für die Hardware eines Many-Core Systems. Dabei wurde vor allem darauf geachtet, dass Code nur einmal geschrieben werden muss und dann mit jedem der unterstützten Systeme benutzt werden kann. Zudem kann das entwickelte System eine gegebene Aufgabe auf die zur Verfügung stehenden Berechnungsknoten verteilen.*

---

## Acknowledgements

First of all, I'd like to thank my family and every one who supported me over the last years during my studies. Thank you!

Secondly, I'd like to thank Prof. Dr. T. Ertl for making this diploma thesis possible. Together with Steffen Frey and Harald Sanftman as my supervisors he was a great help and even made it possible that a part of this theses could be presented on a conference.

Together with Prof. Ertl, Klaus Zimmermann, Matthias Wilde and Christian Unruh from Sony Deutschland made this thesis possible by supplying valuable information and ideas. It was a great time at Sony and I'd like to thank everyone there for his/her help and support.

And finally, thanks to every one how supported me during my diploma thesis, for inspiration, suggestions and as a place of tranquility.



# Contents

1	Introduction	13
2	Computation Platforms used for Evaluation	15
2.1	NVIDIA CUDA	15
2.2	Intel Larrabee	16
2.3	Others Platforms on the Market	17
2.4	Hardware used in this Thesis	18
3	Fourier Transformation on Graphics Cards	19
3.1	Mathematical Background of the Fourier Transformation	19
3.1.1	Discrete Fourier Transformation	19
3.1.2	Fast Fourier Transformation	21
3.1.3	Previous Work regarding Fourier Transform on GPUs	22
3.2	Notation Details	23
3.3	Motivation for Processing the Fourier Transform on a GPU	23
3.4	Solution for the Memory Problem	24
3.5	Description of the Experiments	26
3.5.1	Conditions	27
3.5.2	Measuring Methods Used	28
3.5.3	libfftw3	29
3.5.4	OpenCV	30
3.5.5	CUDA	31
3.5.6	Intel Performance Primitives	38
3.6	Results & Comparison	40
3.6.1	Correctness of FlexDFT	40
3.6.2	NVIDIA CUFFT	40
3.6.3	FlexDFT	41
3.6.4	libfftw	42
3.6.5	Interpretation and Comparison of the Results	42
3.6.6	Conclusions	46
4	Motion Estimator on Graphics Cards	49
4.1	Explanation and Problem of the Cleaning Algorithm	50
4.1.1	CUDA	53
4.1.2	Larrabee	57
4.2	Results	57
4.2.1	Evaluated Results	58

4.2.2	Ideas for Future Work . . . . .	59
4.2.3	Conclusions . . . . .	61
5	Cross Platform Computation Abstraction Layer	63
5.1	Introduction . . . . .	63
5.2	Motivation . . . . .	64
5.3	Existing Libraries and Frameworks . . . . .	65
5.3.1	Close to the Metal: OpenCL & DirectCompute . . . . .	65
5.3.2	Commercial Frameworks: Peakstream & RapidMind . . . . .	66
5.3.3	Differences to CROCAL . . . . .	67
5.4	Conception of CROCAL . . . . .	67
5.4.1	Global Functions . . . . .	68
5.4.2	Hardware Abstraction Layer . . . . .	68
5.4.3	Client Server System . . . . .	70
5.4.4	Server . . . . .	70
5.4.5	Client . . . . .	71
5.4.6	Job . . . . .	71
5.5	Examples for using CROCAL & Evaluation Results . . . . .	73
5.5.1	ASCII Pyramids . . . . .	73
5.5.2	Flexible Discrete Fourier Transformation . . . . .	75
5.6	Future Work . . . . .	78
5.7	Summary . . . . .	80
6	Summary	81
A	How to use libcrocal for Hardware Abstraction	83
B	How to use libcrocal for Many Node Computation	87
	Bibliography	97

# List of Figures

---

3.1	Example Fourier transformed Picture . . . . .	20
3.2	FlexDFT Diagram . . . . .	25
3.3	Usage of CUFFT Plan Buffers . . . . .	34
3.4	Time for creating a CUFFT plan . . . . .	36
3.5	NVIDIA CUFFT Performance . . . . .	40
3.6	FlexDFT Performance . . . . .	41
3.7	libfftw Performance . . . . .	42
3.8	Performance Comparison . . . . .	43
3.9	Stressed CPU Performance . . . . .	43
3.10	Reduced GPU Memory Performance . . . . .	44
3.11	Sizes of the FFT Plans . . . . .	45
3.12	Memory Usage Comparison . . . . .	45
3.13	Time needed for memory copies . . . . .	46
4.1	Linear Motion Interpolation . . . . .	49
4.2	Motion estimated for Interpolation . . . . .	49
4.3	Simple Single Threaded Cleaning Operation . . . . .	52
4.4	Parallelized Cleaning Operation . . . . .	52
4.5	Different Types of Blocks . . . . .	55
4.6	Vector Field split up into Puzzle Pieces. . . . .	56
4.7	Closer View on one Puzzle Piece. . . . .	56
4.8	Reordered Data . . . . .	57
4.9	Motionestimator Sony Reference . . . . .	58
4.10	Motionestimator Cleaning part Performance . . . . .	59
4.11	Scheme of How to Work on Multiple Frames . . . . .	60
5.1	CROCAL Design . . . . .	74
5.2	ASCII Pyramid . . . . .	75
5.3	Pyramid Task Graph Excerpt . . . . .	75
5.4	Pyramid Example Performance . . . . .	76
5.5	DFT Task Graph . . . . .	77
5.6	DFT Example Performance . . . . .	78
5.7	DFT Example Performance 2 . . . . .	78
5.8	DFT Partitioning . . . . .	78

## List of Tables

---

3.1	Target and Library Table . . . . .	27
3.2	Evaluated Image Sizes . . . . .	27
3.3	Bandwidth of CUDA Device . . . . .	29
3.4	Size of a Plan created by libfftw3 . . . . .	30
3.5	Size of a plan created by cufftPlan2d . . . . .	32
3.6	Size of a Plan created by cufftPlan1d . . . . .	36
3.7	Size of a Plan created by ippIDFTInitAlloc_R_32f . . . . .	39
3.8	NVIDIA CUFFT Performance . . . . .	41
3.9	FlexDFT Performance . . . . .	42
5.1	Pyramid Example Task Count . . . . .	76

## List of Listings

---

3.1	Measuring maximum memory consumption of an algorithm . . . . .	28
3.2	Measuring the used memory of the CUDA device . . . . .	29
4.1	Single Threaded Cleaning Operation . . . . .	51
A.1	Main Function . . . . .	84
A.2	demo.cpp: The Host Side of the Program . . . . .	85
A.3	demo.cu: The CUDA File . . . . .	85
B.1	Main Function . . . . .	88
B.2	DFTJob.cpp: Constructor, Destructor and Data Types . . . . .	89
B.3	DFTJob.cpp: DFT Real to Complex and Complex to Complex . . . . .	90
B.4	DFTJob.cpp: Transpose . . . . .	91
B.5	DFTJob.cpp: Partitioning Function . . . . .	92
B.6	DFTJob.cpp: Partitioning Function II . . . . .	93
B.7	DFTJob.cpp: Job Setup . . . . .	94

B.8 DFTJob.cpp: Job Setup II . . . . . 95

## List of Algorithms

---

3.1 Flexible DFT (here including the copies to and from the device) . . . . . 26



# 1 Introduction

In this diploma thesis the usability of general purpose graphic chips for standard image and video algorithms will be discussed. As today's graphics cards get more and more powerful in terms of GFlops (Million Floating point Operations per Second), this power can also be used for computations aside from 3D or video pipelines. The graphics card used in the thesis has around 933 GFlops [BBB<sup>+</sup>08] where as today's high end desktop main processors (CPU) have around 70 GFlops [Flops].

Modern examples for these kind of graphics cards are produced mostly by AMD and NVIDIA. Intel, on its side, is finishing the development of dedicated graphics cards. The hardware used for evaluation in this thesis is based on the GeForce architecture developed by NVIDIA. Intel's new Larrabee platform will be focused from a theoretical point only as there are no products available yet.

The problems discussed in this thesis are memory usage and recursive inter-pixel dependencies as well as the design and prototypical implementation of an abstraction layer for many core computation.

The first topic is discussed by the means of the Fourier transform for image processing. Previous investigations have shown that the Fourier transformation is a very memory consuming operation, especially when computed on NVIDIA CUDA graphics cards using NVIDIA's Fast Fourier Transformation library CUFFT.

The second topic of this thesis is to port a sub system of a motion estimator to the graphics hardware. It's challenge is a recursive dependency which is not straight forward to solve. The parallelization of the algorithm brought the CPU to its limit.

Working with two completely different architectures, the idea to create an unified interface for them comes up. This is the third part of the diploma thesis where an abstraction layer for many node systems will be designed. The first step there is to create a common interface which abstracts the different hardware types. And the next step is to create a framework which can be run on the targeted many node systems using the abstraction layer mentioned beforehand.

The thesis is basically structured into an introduction to the hardware used for evaluation the three main parts from the task formulation of this thesis and is closed by a short chapter summarizing the whole thesis.

**Chapter 2 – Computation Platforms used for Evaluation:** Introducing the graphics cards used for evaluation is the content of the second chapter. It will give an overview

of the NVIDIA CUDA and Intel Larrabee platform. Additionally, a short view on other computational platforms on the market will be given.

**Chapter 3 – Fourier Transformation on Graphics Cards:** This chapter describes the Fourier transformation on graphics cards, introduces a solution for a memory saving transformation and shows the measured results. A comparison between CPU and CUDA will show that the new mechanism can still compete for performance thus its using far less memory.

**Chapter 4 – Motion Estimator on Graphics Cards:** Multiple solutions for the motion estimator problem will be discussed in this chapter. It will be shown that there is no fast and universally valid method to solve the inter pixel dependency on todays graphics cards from NVIDIA.

**Chapter 5 – Cross Platform Computation Abstraction Layer:** Dealing with the third part of the thesis, this chapter introduces the abstraction layer for many core systems. A simple implementation is described and ideas are presented that will give an outlook of the further development.

**Chapter 6 – Summary:** The last chapter will summarize the preceding chapters and review the experience made.

In the appendix A and B, there are two examples of how to use the *libcrocal* which implements the abstraction layer developed in chapter 5. The first chapter describes the usage of the developed framework for simple hardware abstraction. For applying the Fourier transformation on an array of data and having the computation parallelized, the second appendix chapter shows how to do this exemplarily.

## 2 Computation Platforms used for Evaluation

This chapter will describe the two architectures that are used for evaluation in this thesis.

The two selected architectures are the NVIDIA GeForce platform [NVGeF] and the new Intel Larrabee platform [LRB, SCS<sup>+</sup>08]. Both platforms are designed as general purpose graphic processing units (GPGPU). As such, it is possible to make use of the parallel structures of the hardware design for not only graphic applications but all other computations, too.

The next sections, section 2.1 for CUDA and section 2.2 for Larrabee, will give a short introduction into the respective hardware architecture, its programming interfaces and how it is used in the later chapters of this thesis. After this, a short overview of other hardware used for multi and many node computation is given in section 2.3 and finally the specification of the development system in section 2.4.

### 2.1 NVIDIA CUDA

NVIDIA's GeForce series hardware architectures were developed over many development cycles, starting from the first programmable graphics cards having only Vertex Shaders (DirectX 7.0) in their fixed function pipeline. Later, there were Pixel Shader (DirectX 8.0) and Geometry Shader (DirectX 10.0) introduced as new features. All these shaders had their fixed position in the graphics pipeline but became more flexible. Today, there is only one shader architecture implemented on the hardware, the Unified Shader model.

The development of the graphics cards took rapid steps introducing new flexibilities for the programmer. Besides new possibilities in the graphic development and new effects, the graphics cards also became attractive for high performance computing. At first, the programmable shaders were used, programmed in the shader language like GLSL, HLSL or CG, to solve the problems until NVIDIA introduced its CUDA framework, the Computation Unified Device Architecture [NVCUD, NVMan]. It's main application is to simplify the development of non-graphic projects for NVIDIA graphics cards.

CUDA includes an API, compiler and debugger allowing to develop C/C++ applications which manage the data and the computation on the graphics card. The so called *kernels* used to compute the data on the device are compiled using the CUDA compiler and the resulting C code is then integrated into the developed application. The communication between host and device like kernel code uploading and argument passing is handled by the CUDA runtime system and the graphics driver.

The architecture of a CUDA device consists of a number of Streaming Multiprocessors (SM). The SM contains eight Scalar Processors (SP) which do the actual computational work. One SM has a specified amount of memory shared by the *blocks* running on it's SPs. A block is subdivided into *threads* which are mapped to the SPs of the device.

In short, this is a description of the notations regarding CUDA:

**Thread:** A thread describes the lowest execution layer in the system model. For x86 CPUs and CUDA devices, this is a simple (CUDA) thread.

**Warp:** In a Warps all threads execute the same instruction at a time. This represents the Single Instruction Multiple Data scheme of the CUDA device on thread and code level.

**Block:** One block is a CUDA block on a CUDA device containing a number of threads organized in an one, two or three dimensional array.

To use the hardware in an optimal way, 32 threads are combined into *warps*. CUDA is optimized to work with threads organized in half the warp size. This means, especially the memory cache is optimized for reading and writing 16 data elements, typically 4 Bytes each. Additionally, the execution code and time of the threads in a warp is the same. This means that for *if* constructs, the code will be executed by all threads even if only one thread fulfills the condition and no time can be saved by this.

When a kernel is invoked in the host application, the desired block and thread dimensions per block have to be set. The kernel is then divided into the blocks and distributed over the SMs of the device. A block's threads are the smallest logical units running on the device. The order of execution of blocks and threads are left to the scheduler and cannot be influenced by the programmer. This also implies that the synchronization of blocks is not supported by CUDA at this time.

These details will become important in chapter 4 where a part of a motion estimator is implemented in CUDA using kernel functions.

The next stage of the evolution was already announced on the NVIDIA GPU Technology Conference (GTC) in 2009. The Fermi architecture, the next generation after the GeForce technology, has even more parallel units rearranged in a new design and, according to NVIDIA, will give a significant performance improvement.

### 2.2 Intel Larrabee

The Larrabee platform by Intel is designed in a different fashion than the NVIDIA graphics cards. NVIDIA developed graphics cards which became more and more flexible over time, ending with the Unified Shader model nowadays. Intel, on the other hand, uses its x86-Pentium architecture as the shaders of their graphics card and added the display and video related functions. Larrabee and GeForce cards have a very different architecture. Resulting from this different architecture, the processors on the cards have different purposes.

As mentioned above, the Larrabee architecture is designed from x86-Pentium cores. The cores use an in-order code execution engine and are upgraded to run four simultaneous threads and have a vector processing unit capable of processing 16 elements, each 4 Bytes in size. The Larrabee design uses a number of these cores communicating via a ring bus with the memory.

Compared to a standard multi core CPU like an Intel Core 2 Quad, which has an out-of-order code execution engine, one Larrabee core can execute less single instructions per clock. The whole processor has a much higher vector throughput due to the 512 Bit VPU of Larrabee compared to the 128 Bit SSE instruction set of the Core 2 Quad.

To use the 512 Bit registers efficiently, Intel introduced a large number of new instructions allowing to make use of the gather and scatter implemented in Larrabee. While reading or writing a vector, its elements can be permuted or masked directly using the registers [LRBIns].

According to Intel, this high flexibility allows many operations to be done with less instructions and thus much faster. Combined with the parallel cores running four threads each, Intel aims to produce a competitive graphics card and computation co-processor.

Unfortunately, as there is no product from the Larrabee platform available, there cannot be made a statement about its performance. However, the Larrabee platform will be discussed based on the facts released to the public from Intel until now and it will have to proof its performance later on.

## 2.3 Others Platforms on the Market

Besides the above mentioned architectures, there are a lot of other architectures available. Ranging from the standard x86 multi core processors from Intel or AMD [Gee05] to the Cell Processor by IBM [PAB<sup>+</sup>05], those single systems can be combined to clusters.

The Cell processor consists of a PowerPC Processing Element (PPE) core with eight Synergistic Processing Elements (SPE). The PPE controls the execution and workflow of its SPEs which use the Single Instruction Multiple Data (SIMD) scheme for operating. Using the IBM Cell processor, Sony's Playstation 3 can be used for clustering and high performance applications [ps306].

NVIDIA's competitor AMD is also producing GPGPU graphics cards. AMD's devices implement the AMD Stream API [ATI] which is similar to CUDA. It also gives the programmer an easy access to the graphics card and allows him to use it as for computation different than graphic applications.

Talking about clusters, today's hype using Cloud Computing is a popular method for distributing heavy computational problems to a large number of nodes [AFG<sup>+</sup>09, NYTimes]. These nodes typically have access to the lower level hardware mentioned above and in the previous sections. The cloud concept combines the power of its elements providing the next level of abstraction.

### 2.4 Hardware used in this Thesis

For developing the benchmarks and the prototype of the abstraction layer, libcrocal, the following hardware was used:

- Intel Core 2 Quad 9550 @ 2.833 GHz
- Asus P5Q Pro Board
- 8 GBytes DDR2 main memory
- Windows 7 RC 1

All NVIDIA benchmarks were evaluated with the underlying graphics card and driver/API versions.

- NVIDIA GTX 280 with 1 GByte of Video RAM
- NVIDIA Driver for Windows 7 (64-bit) 185.85
- NVIDIA CUDA Toolkit and CUDA SDK Version 2.2

The Intel processor with the *Yorkfield* architecture was introduced in Januar 2008. The NVIDIA GTX 280 was released in June 2008.

## 3 Fourier Transformation on Graphics Cards

This chapter deals with the first topic of the thesis, the Fourier transformation and its implementation on graphics cards. The focus will be on the *NVIDIA CUFFT* library and how to use this library in a memory saving way.

The first section (3.1) will deal with the Fourier transformation itself and explain its mathematical background. Following the motivation (section 3.3) and the solution to the memory problem (section 3.4) the experiments are explained (section 3.5). Finally, the results are presented and discussed including a short summary of this chapter (section 3.6).

### 3.1 Mathematical Background of the Fourier Transformation

The Fourier transformation transforms a time domain function into the frequency domain [BK66, FCo6, SKMS93, Smied]. The data is then represented as the frequency. For humans this representation is hard to interpret as it does not show the visual information itself but the superposition of sine-waves.

$$F(\hat{x}) = \int_{-\infty}^{\infty} f(x) \cdot e^{-2\pi i x \hat{x}} dx \quad (3.1-1)$$

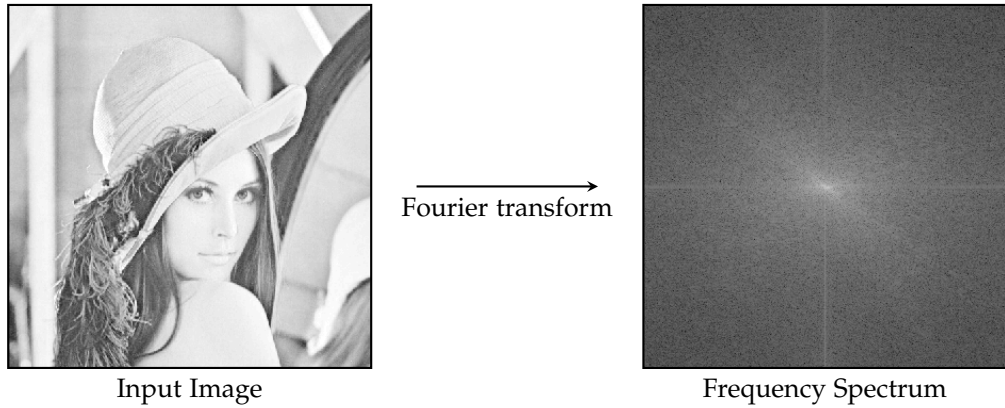
The Fourier transformation is applicable for higher dimensions, too. This makes it possible to work on two dimensional data like images.

#### 3.1.1 Discrete Fourier Transformation

The transformation can be applied on continuous as well as on discrete data which both can be real or complex. As this thesis focuses on an image as input data, the transformation used is the discrete Fourier transformation:

$$F_n = \sum_{x=0}^{N-1} f_x \cdot e^{-\frac{2\pi i x n}{N}} \quad (3.1-2)$$

$$F_{n,m} = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f_{x,y} \cdot e^{-2\pi i (\frac{xn}{N} + \frac{ym}{M})} \quad (3.1-3)$$



**Figure 3.1:** The famous “Lena” image and its Fourier coefficients.

Using this formula to transform the image from discrete real values into discrete complex values results in an array which has twice the size of the input data as the input is real using 4 Bytes floating point and the output has an additional complex part which is also a floating point value. Figure 3.1 shows an image [USC] and its Fourier coefficients. For the real discrete input there is the benefit that the resulting data is symmetrical because the complex value of the input is always zero and its complex conjugation (see equation 3.1-8). The multiplication of a real value with a complex one is simply the scaling of the complex value. This can also be seen in the image where the left half is the rotated right half.

The transformation can also be seen as two matrix multiplications [Smied]:

$$FT_{\text{Vector}} = \text{matRow} \times \text{Vector} \quad (3.1-4)$$

$$FT_{\text{Image}} = \text{matRow} \times \text{Image} \times \text{matCol} \quad (3.1-5)$$

With:

$$\text{matCol}_{k,j} = e^{\frac{-2\pi i k \cdot j}{N}}$$

$$\text{matRow}_{k,j} = e^{\frac{-2\pi i k \cdot j}{M}}$$

Vector has size  $N$  and  $\text{Vector}_{k,j} \in \mathbb{R}$ ,  
 Image has size  $N \cdot M$  and  $\text{Image}_{k,j} \in \mathbb{R}$ ,  
 matCol has size  $N \cdot N$  and  $\text{matCol}_{k,j} \in \mathbb{C}$ ,  
 matRow has size  $M \cdot M$  and  $\text{matRow}_{k,j} \in \mathbb{C}$

Here, both  $\text{matCol}$  and  $\text{matRow}$  are self-adjoint matrices:  $m_{k,j} = a_{k,j} + ib_{k,j} = a_{j,k} - ib_{j,k} = \overline{a_{j,k}}$ . Following this symmetry, the real input signal becomes a Hermite signal in the frequency domain:  $F(N - k) = \overline{F(k)}$ . Using this fact, there are only  $\frac{N}{2}$  independent Fourier coefficients

in the result. This means that half of the data is redundant in the result of the computation. The redundant data is not needed to be stored in memory for further operations:

Because of:

$$e^{2\pi i} = 1 \tag{3.1-6}$$

$$\begin{aligned} e^{i\phi} &= \cos \phi + i \sin \phi \\ &= \overline{\cos \phi - i \sin \phi} \\ &= \overline{e^{-i\phi}} \end{aligned} \tag{3.1-7}$$

the Hermite symmetry holds:

$$\begin{aligned} F(N - k) &= \sum_{x=0}^{N-1} f(x) \cdot e^{-\frac{2\pi i x(N-k)}{N}} \\ &= \sum_{x=0}^{N-1} f(x) \cdot e^{-\frac{2\pi i x N}{N}} \cdot e^{\frac{2\pi i x k}{N}} \\ &= \sum_{x=0}^{N-1} f(x) \cdot e^{-2\pi i x} \cdot e^{\frac{2\pi i x k}{N}} \\ &\stackrel{(3.1-6)}{=} \sum_{x=0}^{N-1} f(x) \cdot 1^{-x} \cdot e^{\frac{2\pi i x k}{N}} \\ &\stackrel{(3.1-7)}{=} \sum_{x=0}^{N-1} f(x) \cdot \overline{e^{-\frac{2\pi i x k}{N}}} \\ &= \overline{F(k)} \end{aligned} \tag{3.1-8}$$

#### 3.1.2 Fast Fourier Transformation

The standard discrete Fourier transformation requires the nested summation of all dimensions of the input data. For the 2d images this ends in a computation of  $O(N^2)$  per element. So, the complexity is  $O(N^4)$  operations as shown in formula 3.1-3 above. As the transformation kernel is separable, the formula can be rewritten:

$$\begin{aligned}
 F(n, m) &= \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) \cdot e^{-2\pi i(\frac{ym}{M} + \frac{xn}{N})} \\
 &= \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) \cdot (e^{-2\pi i(\frac{ym}{M})} + e^{-2\pi i(\frac{xn}{N})}) \\
 &= \sum_{x=0}^{N-1} \left( \sum_{y=0}^{M-1} f(x, y) \cdot (e^{-2\pi i(\frac{ym}{M})}) \right) \cdot e^{-2\pi i(\frac{xn}{N})} \\
 &= \sum_{x=0}^{N-1} e^{-2\pi i(\frac{xn}{N})} \cdot F_y(x, y, m)
 \end{aligned} \tag{3.1-9}$$

This reduces the complexity of the calculation from  $O(N^2 \cdot M^2)$  to  $O(N \cdot M \cdot (M + N))$ , simplifying the calculation from  $O(N^4)$  to  $O(N^3)$  for  $M \leq N$ . Using this schema the transformation is done in the y-dimension first and then in the x-dimension. Transforming one dimension can be done in a more efficient way. There are various algorithms to perform the 1d-transformation in less time then  $O(N^2)$ , e.g. the divide and conquer approach shown in formula 3.1-10 which recursively splits the data and has a complexity of  $O(N \cdot \log(N))$  for the input size of  $N$ . Combined with the separability, the Fourier transformation can be calculated much more efficiently.

$$\begin{aligned}
 F(n) &= \sum_{x=0}^{\frac{N}{2}-1} f(2x) \cdot e^{-\frac{2\pi i(2x)n}{N}} + \sum_{x=0}^{\frac{N}{2}-1} f(2x+1) \cdot e^{-\frac{2\pi i(2x+1)n}{N}} \\
 &= \sum_{x=0}^{\frac{N}{2}-1} f(2x) \cdot e^{-\frac{2\pi i(2x)n}{N}} + e^{-\frac{2\pi in}{N}} \cdot \sum_{x=0}^{\frac{N}{2}-1} f(2x+1) \cdot e^{-\frac{2\pi i(2x)n}{N}} \\
 &= F_{\text{even indices}}(n) + e^{-\frac{2\pi in}{N}} \cdot F_{\text{odd indices}}(n)
 \end{aligned} \tag{3.1-10}$$

For computing the Fourier coefficients using so called in-place-transforms, the size of the array needed has  $M \cdot (\frac{N}{2} + 1)$  complex elements. This array is large enough for holding the complete input image as this consists of real values only. Using this in-place-transform every row or column of the input has to be padded. For an image with the size of 24 Megapixel having a width of  $N = 6048$  and height of  $M = 4032$  pixels the required memory for the computation is 93.054 MB per channel using 4 Byte float values (so 8 Bytes per complex value in the resulting array).

#### 3.1.3 Previous Work regarding Fourier Transform on GPUs

Porting the Fourier transform to the GPU is a logical step when looking at the pure computational power of todays GPUs. Besides the NVIDIA CUFFT library discussed in this thesis, there are also other publications which handle this topic [GLD<sup>+</sup>].

Govindaraju et al. claim that their high performance implementation has a improvement of about 2 to 4 times over the NVIDIA CUFFT library. Towards memory consumption, they do not state an exact amount of what their algorithm needs. As they use out-of-place transforms in their algorithm, it can be assumed that they need at least the input and output buffers of the size  $M \cdot N$  and  $M \cdot (\frac{N}{2} + 1)$ , respectively.

Two older publications from Fialka et al. [FCo6] and from Moreland et al. [MA03] present methods of how to use the graphics cards from these days for Fourier transforms.

Fialka used the shader language HLSL under DirectX 9.0 to apply the Fourier transform and convolution to images. Moreland uses OpenGL and the Cg shaders, the predecessor of NVIDIA's CUDA, to program the vertex and fragment shaders which do the Fourier transform and convolution. He concentrates on fast one-dimensional Fourier transforms and claims a performance of 1.9 seconds for transforming and filtering an image of size  $1024 \cdot 1024$  doing the calculations for all 4 channels simultaneously.

### 3.2 Notation Details

In the equations in the following sections, the overline denotes the complex conjugation:  $c = a + ib = \overline{a - ib} = \bar{c}$

Additionally, the  $\sim$  denotes the complex data in the memory consumption descriptions.  $\tilde{N} = 2 \cdot N$  as the real and imaginary values are of the same format.

The listings describing an algorithm or code sample are based on the C syntax but are only meant as pseudo code for example purpose.

### 3.3 Motivation for Processing the Fourier Transform on a GPU

The motivation for this part of the thesis is the idea of using the graphics card as a mathematical coprocessor which computes the Fourier transformation. Today's sophisticated image manipulation tools allow the manipulation steps to be done with a standard pc by the photographer himself. Many algorithms in those tools require the Fourier transformation as a pre step to achieve a good performance. For example convolution with a filter kernel can be done in  $O(N^2 \cdot \log(N^2) + K^2 \cdot \log(K^2))$  for a quadratic image with the size of  $N \cdot N$  and a quadratic kernel of size  $K \cdot K$ . The standard convolution needs  $O(N^2 \cdot K^2)$  operations which will be a heavy problem for larger kernel sizes. Using the multiplication in the frequency domain of transformed input and convolution kernel the problem reduces to  $O(N^2 \cdot \log(N^2) + K^2 \cdot \log(K^2))$ .

Bringing this algorithm to highly parallel hardware like modern graphics cards, a large improvement in performance can be expected. For standard image sizes like two or six Megapixel this is no problem for the hardware itself. The problems arise for very big images produced by high end professional digital single lens reflex cameras which have up to 24

Megapixels (MP) these days. Here, an uncompressed image has the size of  $3 \cdot 24$  Megapixels = 72 Megabytes for all 3 channels, given one Byte per pixel. Depending on the data format this will range from 72 MBytes for 8 bit values up to 288 MBytes for 32 bit floating point values. Today's high end graphics cards are delivered with up to 2 GBytes of dedicated RAM but the standard graphics cards still have 256 MBytes or less, so there is no way of even storing the uncompressed image on the graphics card. In addition, the computation itself needs memory, to say nothing about the operating system and other applications which also require memory on the graphic device.

Making it possible to transform a larger image using the graphics card and evaluating the performance of this transformation is the topic of this chapter.

#### 3.4 Solution for the Memory Problem

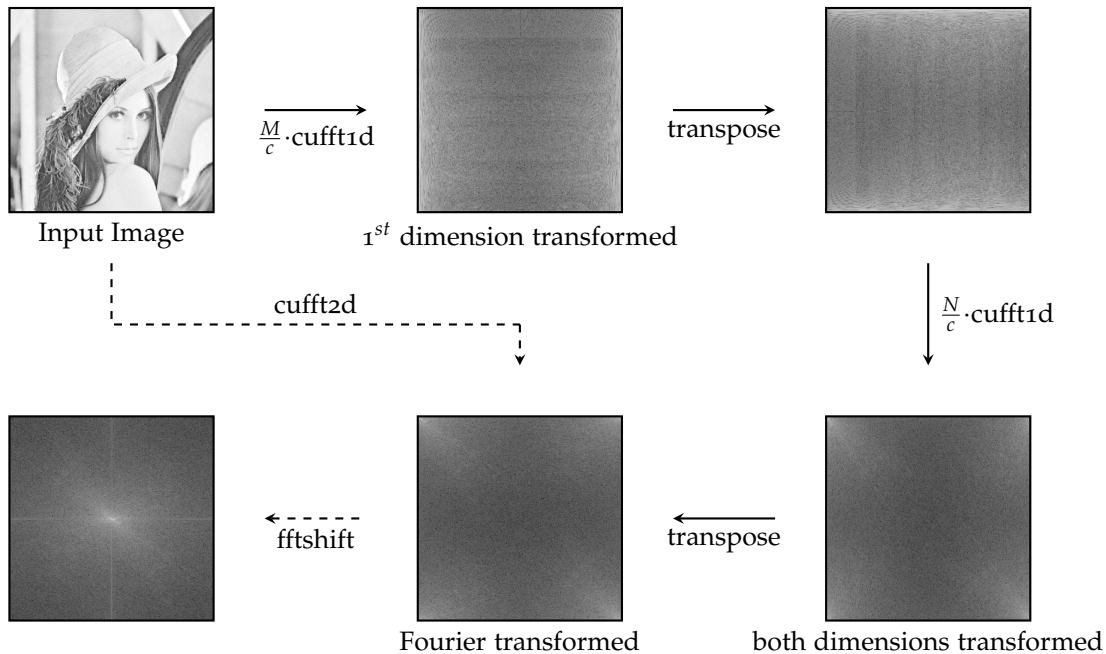
As shown in equation 3.1-9 in section 3.1.2 the separability of the Fourier transformation can be used to compute the coefficients more efficiently. The idea of splitting the computation by the dimensions separates it from one two dimensional operation into two one dimensional steps:  $O(M \cdot N \cdot \log(N) + N \cdot M \cdot \log(M)) = O(N \cdot M \cdot \log(N \cdot M))$ . Using this simple scheme, the memory consumption of both steps is lower than in the at-once transformation. This section will give a detailed explanation of the solution of the memory problem.

Using a CUDA device and the CUFFT library, the most memory is consumed by buffers which were created in the planning phase. In this planning state, the library arranges its calculations in so called *plans*. For an image of  $6048 \cdot 4032$  pixels this plan needs approximately 415 MBytes, which is far more than many standard consumer graphic devices can provide. In section 3.5.5 there is a detailed list of the plan sizes for CUFFT. On the host side, libraries like the Intel Performance Primitives or the *libfftw* also need additional buffers for their calculation. But the plans created by these libraries need far less memory than the CUFFT.

The idea of the solution to this is the following: Using the separability, the transformation of each dimension of the input data can be computed independently. Thus, a two dimensional transformation of  $M \cdot N$  is separated into  $M$  one dimensional transformations of  $N$  elements plus  $N$  transformations of  $M$  elements. This method uses far less memory for computation per transformation than the original two-dimensional CUFFT approach. Having a device with limited memory, the memory can be maxed out with as much input data as possible. The input data is then transformed and copied back to the host. This is done until the first dimension is transformed. For the second transformation, the array needs to be transposed piecewise, so that the second dimension lies in memory in consecutive order instead of interlaced order. This is necessary because the CUFFT library can only operate on data which is not interlaced. Other libraries might not need the transposition and thus can operate faster, as there are no additional memory copies and transpositions needed.

Transforming the second dimension in the same way as the first dimension then gives the Fourier coefficients of the input data. Because of the transposition in between the final data

also has to be transposed back. This last step can be avoided if the following algorithm which uses the Fourier coefficients is aware of this. Figure 3.2 shows a schematic diagram of this.



**Figure 3.2:** Diagram showing the execution of FlexDFT.  $c$  is the chunk size the  $M \cdot N$  image is split into. The last step is optional and shows the FFT with the lowest frequency coefficients in the center.

Depending on the memory available on the device, the data can either be copied completely or separated in chunks. When there is enough memory for the input and output data the first method should be preferred as it avoids the overhead of many memory operations from and to the device. In addition, the transposition of the data, if necessary in the intermediate and the last step, can be done on the device, when there is enough space for the buffers needed.

The method to transpose the data is taken from the NVIDIA SDK [NVCUD]. It provides a very fast transposition kernel which makes use of the shared memory to guarantee coalesce readings and writings so the array is transposed in the shortest time possible.

This approach is called *Flexible Discrete Fourier Transform* (FlexDFT). Depending on whether the input is copied chunk wise to the device, it is referred to as “FlexDFT, no copy” in the case that the input resides on the device completely and is **not** copied there in chunks. Both methods can be implemented as *in-place* or *out-of-place* variants. When talking about the *no copy* version, the memory operations needed for uploading and downloading the input and output to/from the device are not taken into account. It can be assumed that the image is processed before and after the transform on the device and therefore resides already there.

---

**Algorithmus 3.1** Flexible DFT (here including the copies to and from the device)

---

```

procedure DFT_FLEXIBLE(inputImage, freeDeviceMemory)
    (width, height) ← SIZE(inputImage)           // Pre-calculation work
    laneCount ← ESTIMATELANECOUNT(freeDeviceMemory)

    planWidth ← PLAN1D(width)                     // Transform the Rows
    repeat
        COPY(as much Rows as possible)
        outputLanes ← FFT1D(planWidth, inputLanes)
    until All Lanes were processed
    TRANSPOSE(outputLanes)

    planHeight ← PLAN1D(height)                  // Transform the Columns
    repeat
        COPY(as much Columns as possible)
        output ← FFT1D(planHeight, outputLanes)
    until All Lanes were processed
    TRANSPOSE(output)           // output now contains the Fourier transform of inputImage
end procedure

```

---

Algorithm 3.1 gives a pseudo-code implementation of the approach.

This algorithm is independent of the Fourier transformation algorithm and the device used. It can be applied using the libfftw or the libIpp, too. On the CPU side, this approach can be used if there is not enough main memory available. In this case, the data has to be read chunk wise from the hard disk or any other location and intermediate results have to be stored back there again.

Additionally, this computation scheme can be applied to higher dimensions as well. Separating three dimensional data into two dimensional chunks which can be solved following the idea developed here.

### 3.5 Description of the Experiments

This section will describe the procedure of the evaluation. The Flexible Fourier Transformation has been described in section 3.4. It has been implemented using NVIDIA CUDA and the CUFFT library. For performance evaluation, this approach has been compared with other existing libraries which are described here, too.

### 3.5.1 Conditions

For the experiments the hardware described in chapter 2 section 2.4, page 18 has been used.

The evaluated libraries implementing a Fast Fourier Transformation algorithm are:

Hardware / Target Platform	Evaluated Library
Intel Core 2 Quad (CPU Reference)	libfftw3 (3.2.2)
NVIDIA GTX 280 (1024 MB)	NVIDIA CUFFT library (delivered with CUDA 2.2)

**Table 3.1:** *Target and Library Table.*

For the experiments various image sizes have been used. The selection of these values are based on standard camera modes from professional single lens reflex cameras and mobile phones or common video formats. Table 3.2 shows the evaluated sizes:

Width	Height	Megapixel	Comment
6048	4032	24.38	Largest DSLR Camera Format
4096	4096	16.78	Power of 2 Example
4752	3168	15.05	Standard Digital Camera Format
3456	2304	7.96	Standard Digital Camera Format
2560	1920	4.91	Photo Mobile Phone Camera
2353	1568	3.69	Standard Digital Camera Format
1920	1080	2.07	Full High Definition Video Format & Desktop Resolution (16 : 9)
1280	1024	1.31	Typical Desktop Resolution (4 : 3)
720	480	0.35	Standard Definition Video Format
512	512	0.26	Power of 2 Example

**Table 3.2:** *Size of Images used for Evaluation.*

The largest picture is (logically) the most memory consuming one and will be used as an example in the following algorithms for memory consumption. The following section 3.5.2 will describe the methods used for evaluating the algorithms. The algorithms themselves are described in sections 3.5.3 to 3.5.6. In section 3.6 on page 40 the results will be discussed.

Note that the first dimension 2353 · 1568 is a very “Fourier-unfriendly”. This is the reason why the evaluation shows for this size a much worse performance than for little higher sizes. For every evaluated algorithm a clear peak in the curve is visible there.

---

**Listing 3.1** Measuring maximum memory consumption of an algorithm

---

```
void example_dft(..., int *memoryUsage, ...)  
{  
    // initialize memory variables  
    int iReferenceUsage = getMemoryUsage();  
    *memoryUsage = 0;  
    // allocate buffers and create plan  
    // assure maximum memory consumption is recorded  
    *memoryUsage = max(*memoryUsage, getMemoryUsage(iReferenceUsage));  
    dft(input, output);  
    // free memory no longer needed  
}
```

---

#### 3.5.2 Measuring Methods Used

In this section the methods which were used in the experiments to measure the memory consumption and time needed will be described by means of small pieces of (pseudo-) code.

In general, the memory usage can be determined by the Windows API on the host side and by the CUDA Driver API on a CUDA device. Usually, low level structures like arrays for the input image can be determined directly by evaluating the size argument of the memory allocation call. For determining the indirect memory usage like for the plans and theirs buffers the difference of the memory needed by the process before and after the creation of the plan is measured.

For the maximum memory consumption at a time the memory usage of an algorithm is determined directly after the function is called and then again directly before the call of the computation function(s) (see Listing 3.1). The difference between both values is the actual memory usage. The CUDA API has been used to measure the memory available on the device, see Listing 3.2. Using the *CUDA Context* it has come out that it needs memory on the device to. Creating a second CUDA Context and requesting the memory information reports around 50 MBytes less memory available.

For the speed of the algorithms, not only the pure computation time is interesting but the time for copying data from the host to the device and back must also be taken into consideration. Today's graphics cards do have a theoretical memory bandwidth of 100 GBytes/s and above, but the PCI Express slot is limited to 500 MBytes/s per lane giving a maximum memory transfer rate of 8 GBytes/s (16 lanes) for the graphics card per direction [PCISIG]. Measurements using the *bandwidthTest* in *Shmoo* mode from the NVIDIA CUDA SDK are shown in Table 3.3. The right column with a fully loaded system was created with the same tool and another program having four threads which constantly copy 128 MBytes and then sleep for 100 milliseconds. Unfortunately, as there are no physical prototypes of Larrabee available yet, there are no tests or information available. For this reason there can be made no statement about this at the time.

**Listing 3.2** Measuring the used memory of the CUDA device

---

```

int getMemoryUsageCUDA(int iReferenceUsage = 0)
{
    unsigned int iMemFree, iMemTotal;
    CUcontext pCudaContext;
    cuCtxCreate(&pCudaContext, CU_CTX_SCHED_AUTO, 0);
    // get Memory Information
    cuMemGetInfo(&iMemFree, &iMemTotal);
    cuCtxDestroy(pCudaContext);

    if(!iReferenceUsage)
        return iMemFree;
    else
        return iReferenceUsage - iMemFree;
}

```

---

Direction	Bandwidth (idle)	Bandwidth (full load)
Host to Device	1807.7	478.3
Device to Host	1919.1	528.2

**Table 3.3:** Bandwidth for 64 MBytes of data in MByte per Second.

To amortizing side effects of the running system, every test was running in a loop doing the same work at least 10 times. As smaller transforms run much faster than bigger ones, the number has been scaled accordingly. Whereas a 1 Megapixel image was iterated 256 times, a 8 Megapixel image was iterated at least  $\frac{256}{8} = 32$  times.

## 3.5.3 libfftw3

The *libfftw3* library [FJ98, FFTW] is an open source project implementing the Fourier transforms for various platforms. Its intention is to have a very fast implementation. This is reached by planning functions which try to plan the execution of the transform earlier and in time consuming operations for having an optimal execution time later.

For controlling this feature, there are several flags available so the programmer can decide how intensive the precalculation should be. Being able to save the “knowledge” from the planning makes it possible to once let the algorithm plan the transformation in a time consuming operation and then have the result available later.

In this thesis, the *libfftw3* is used as the reference CPU implementation. The planning flag used was `FFTW_PATIENT`. This mode is has a time consuming planning phase. It tries different methods for decomposing the transform and selects the fastest one for the actual transform later. Additionally, the library is capable of multi threaded calculations what is used for the later comparisons. So, the full power of the Intel Core Quad is used.

This library is very frugal and can limit its memory needed as intermediate result buffers to minimum as shown in 3.11 on page 45 in the results section (section 3.6). Its speed and the low memory consumption are the reasons why this library has been chosen for evaluation.

The plans created by `libfftw3` are shown in Table 3.4 below. They are very small compared to the plans needed by `CUFFT`. This could be explained by the knowledge provided to the `fftw_plan` function. This function gets the input and output pointers, so it can automatically distinguish between in-place and out-of-place transform. It “knows” about the data and its output: real to complex, complex to complex or complex to real. Additionally, the `libfftw` might have implemented more sophisticated algorithms which do not need that much memory.

Image Size		Resulting Plan Size
6048 · 4032 pixels	24.38 Megapixels	78.887 MBytes
4752 · 3168 pixels	15.05 Megapixels	53.969 MBytes
3456 · 2304 pixels	7.96 Megapixels	34.063 MBytes
2560 · 1920 pixels	4.91 Megapixels	25.168 MBytes
2353 · 1568 pixels	3.69 Megapixels	21.707 MBytes
1920 · 1080 pixels	2.07 Megapixels	16.840 MBytes
1280 · 1024 pixels	1.31 Megapixels	14.656 MBytes
720 · 480 pixels	0.35 Megapixels	12.137 MBytes
512 · 512 pixels	0.26 Megapixels	11.938 MBytes

**Table 3.4:** Size of a plan created by `libfftw3`, transforming two-dimensional real discrete input data to complex Fourier coefficients.

#### 3.5.4 OpenCV

*OpenCV* [BKo8] is an open source library from Intel which contains various algorithms for image processing and computer vision. The library focuses on real time processing and implements a Fourier transform algorithm, too.

For an optimal fast transformation, *OpenCV* pads the input image and then performs the discrete Fourier transformation on the padded array. A drawback is the high memory consumption as the input required by the algorithm is a complex image with the complex elements set to zero. The result is the full output of the transformation including the redundant coefficients:

$$size_{\text{opencv}} = 3 \cdot M \cdot N \text{ complex floats} \quad (3.5-1)$$

For an image of the size of 6048 · 4032 pixels the minimum required memory is 837.211 MB for one channel in float-format. This is an enormous overhead compared to the minimum

size needed for the computation of the Fourier coefficients. The minimum size is the input array in real format of  $N \cdot M$  and the output of  $M \cdot (\frac{N}{2} + 1)$  in complex values.

This high memory consumption disqualifies the OpenCV library for further investigation and evaluation. There is a *gpuCV* [gpuCV] version available which outsources many functions to the graphics card but this falls back to the CUFFT implementation for Fourier transforms. So the *gpuCV* would have the same problem as CUFFT brings it, the large amount of memory needed for the buffer for the intermediate results.

### 3.5.5 CUDA

The main focus of this part of the thesis lies on evaluating the idea presented in 3.4. The CUFFT library by NVIDIA delivers the algorithms for transforming one-dimensional lines. As it is possible to let the API execute a *batch job* which transforms many lines at once, one can save function calls and many small memory copies can be collected and executed at once.

#### CUFFT

The NVIDIA CUFFT Library [NVCUF] is an implementation for calculating the Fourier Transformation of 1d, 2d or 3d data on a CUDA enabled device. According to the documentation, the library uses various algorithms to calculate the transformation in an optimal way, similar to the *libfftw3* library (see section 3.5.3, page 29). For choosing the most suitable algorithm, the CUFFT library implements a planning API. When using this library, the memory for input and output has to be allocated on the device. Then, the library plans the Fourier transformation according to the size of the input data. Finally, the input data needs to be copied and then the transformation itself can be invoked. In this case, the memory on the device needed depends whether transformation is calculated in-place or out-of-place.

For an image of  $N \cdot M$  pixels in size the calculation looks as follows:

$$\begin{aligned}
 size_{\text{cufft, in-place}} &= \tilde{M} \cdot \left( \frac{N}{2} + 1 \right) \\
 &= 2 \cdot M \cdot \left( \frac{N}{2} + 1 \right) \\
 &= M \cdot (N + 2) \text{ Complex Floats} \\
 &= 8 \cdot M \cdot (N + 2) \text{ Bytes} && (3.5-2) \\
 \\
 size_{\text{cufft, out-of-place}} &= \tilde{M} \cdot \left( \frac{N}{2} + 1 \right) + M \cdot N \\
 &= 2 \cdot M \cdot \left( \frac{N}{2} + 1 \right) + M \cdot N \\
 &= M \cdot (N + 2) + M \cdot N \\
 &\approx 2 \cdot size_{\text{cufft, in-place}} \text{ Complex Floats} \\
 &\approx 2 \cdot size_{\text{cufft, in-place}} \text{ Bytes} && (3.5-3)
 \end{aligned}$$

For large images like a 24 Megapixel image with a size of  $6048 \cdot 4032$  pixels this results in 93.054 Megabytes 3.5-2 of device memory for the in-place transform and 186.108 Megabytes 3.5-3 for the out-of-place transformation for the pure image and the result per channel. Given the fact, that the operating system and the calculation, first of all the plan and its buffers, need memory, too, there might not be enough memory left on a standard graphics card for this computation.

The memory needed by the plan and its internal buffer varies, depending on the size of the transformation. Table 3.5 shows the memory consumption of the plan according to the size of the input.

Image Size		Resulting Plan Size
6048 · 4032 pixels	24.38 Megapixels	415.066 MBytes
4752 · 3168 pixels	15.05 Megapixels	229.750 MBytes
3456 · 2304 pixels	7.96 Megapixels	121.500 MBytes
2560 · 1920 pixels	4.91 Megapixels	75.000 MBytes
2353 · 1568 pixels	3.69 Megapixels	56.313 MBytes
1920 · 1080 pixels	2.07 Megapixels	31.688 MBytes
1280 · 1024 pixels	1.31 Megapixels	20.000 MBytes
720 · 480 pixels	0.35 Megapixels	5.313 MBytes
512 · 512 pixels	0.26 Megapixels	0.000 MBytes

**Table 3.5:** Size of a plan created by `cufftPlan2d`, transforming real discrete input data to complex Fourier coefficients

From this observation a formula can be derived which estimates the size of a plan of an image with the size of  $M \cdot N$ :

$$\begin{aligned}
 size_{\text{plan}} &\approx 4 \cdot 4 \cdot M \cdot N + \textit{overhead} \text{ Bytes} \\
 &= 4 \cdot M \cdot N + \textit{overhead} \text{ Floats} \\
 &= 2 \cdot M \cdot N + \textit{overhead} \text{ Complex Floats}
 \end{aligned}
 \tag{3.5-4}$$

As the computation is done in floats, which have a size of 4 Bytes each, it seems that 2 complex or 4 real additional buffers are needed internally. The usage of that much memory could be explained as follows:

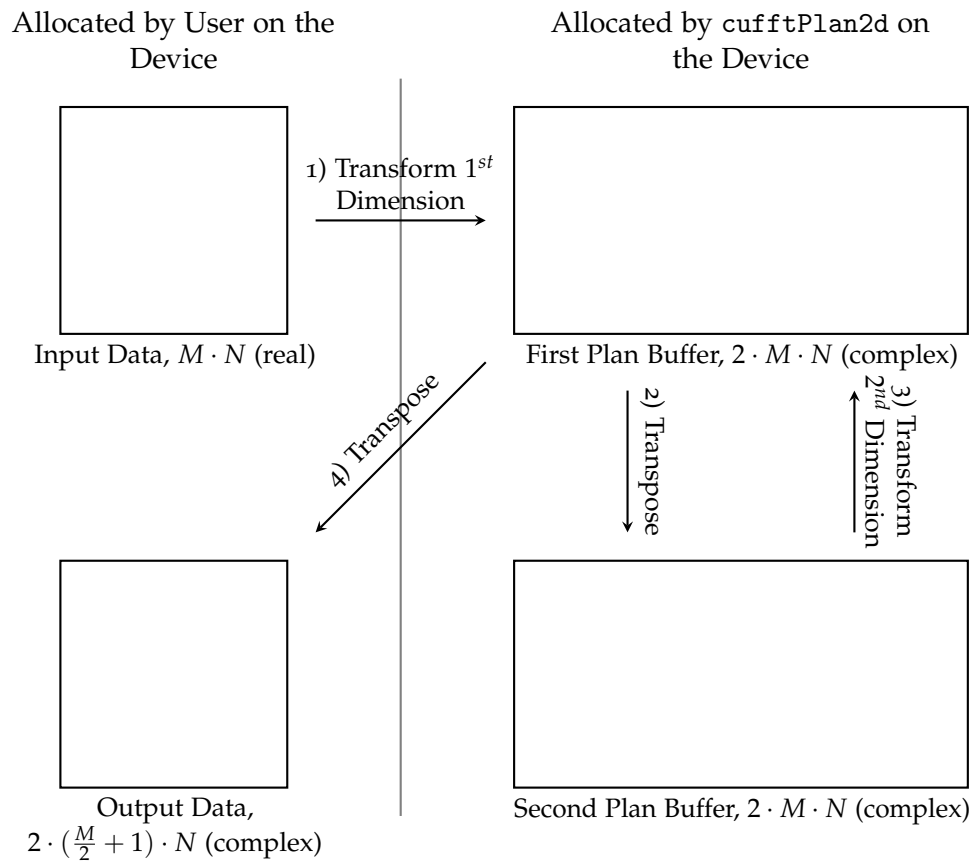
As the plan allocation is done before it is known whether the transformation is in-place or out-of-place, the algorithm cannot rely on the output array for storing intermediate results because it then could destroy the input. Additionally, the planning function does not have any information about the data, whether it is real and the result is complex, or a complex-to-complex transformation. So all the intermediate steps have to be done in two buffers which can hold the full complex to complex elements. In our case, there is half the allocated memory wasted as the transformation produces redundant output as shown in the introduction in section 3.1.1. The first buffer can hold the transformation of the first dimension whereas the second buffer is used for the transposed results. Then, the second dimension is transformed into the first buffer again and the result transposed into the output buffer, see Figure 3.3.

Reusing the buffers for another transformation is possible as they do not contain any information about the transformed data. This means the created plan can also be used for the next transformation if it has the right size. When the maximum chunk size is known, one API specific problem is left: Often, the last chunk does not have the same size as the others, e.g. due to a height which might be prime. This means, there has to be a second plan which only handles the last dimension and is only used once. This is a small drawback of using the FlexDFT with CUFFT.

So the huge memory consumption of the CUFFT library might be the result of the API design which does not allow the planning function to know more details about the intention of the user. Otherwise, it allows a more flexible usage than the planning functions of libfftw3 which require all information to be given at planning time and simply execute the plan later.

Using CUFFT, both versions of the algorithm have been implemented: Having both, the input and output, on the device so that no additional memory copies from the host to the device and back are necessary. The other version is copying chunks to the device only and transforms them there.

The latter version guarantees data to be transformed even when there is not enough space for the whole input and output. But if there is enough space, both versions can be run. Using the first method and having input and output on the device is the more memory wasting but therefore more efficient approach. The number of chunks which are transformed in one



**Figure 3.3:** How CUFFT's plans could work: Transforming the first dimension from the input into the first buffer (1), then transpose it into the second one (2) and transform the second buffer back into the first one (3). Finally, the content of the first buffer is transposed to the output (4).

function call is less than in the other version, but therefore there are far less memory copy calls, only two for copying the input to the device and the output back to the host.

The theoretical limit for the version having both input and output on the device having 1 GByte of memory would be a quadratic image of size  $N \times N$  with  $N \approx 11500$  pixels as demonstrated in equation 3.5-5.

$$\begin{aligned}
size_{input} + plan_{singleline} + size_{output} &\leq 1024 \text{ MBytes} \\
&\leq \frac{1024^3}{4} \text{ Floats} \\
N^2 + 4 * N + N^2 &\leq \frac{1024^3}{4} \text{ Floats} \\
N &\approx 11500 \text{ Floats}
\end{aligned} \tag{3.5-5}$$

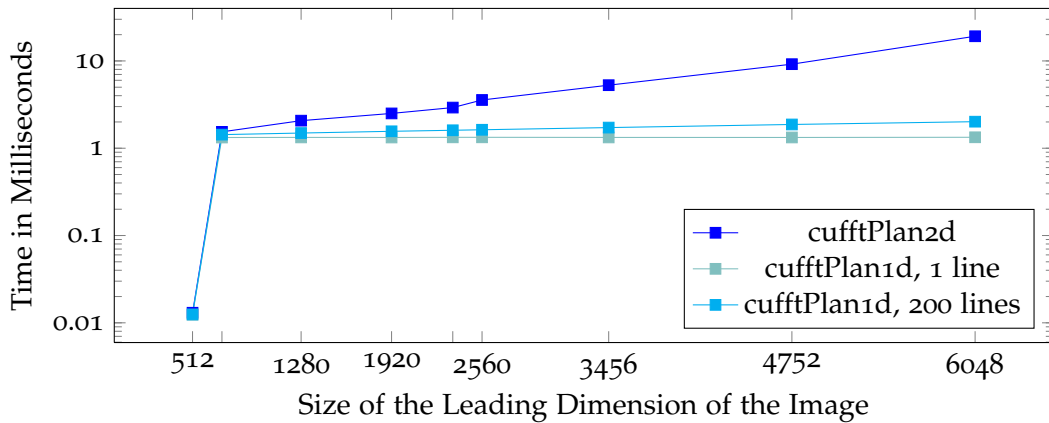
For the other approach having not enough memory for input and output, the largest dimension can theoretically be at around 45 Million elements for the same device, see equation 3.5-6.

$$\begin{aligned}
size_{input} + plan_{singleline} + size_{output} &\leq 1024 \text{ MBytes} \\
&\leq \frac{1024^3}{4} \text{ Floats} \\
N + 4 \cdot N + N &\leq \frac{1024^3}{4} \text{ Floats} \\
N &\approx 45 \text{ Million Floats}
\end{aligned} \tag{3.5-6}$$

Up to here, the method presented was working with single channel images only. For multi channel images, the options are to generate the plans first and keep them in memory until all channels are transformed or to destroy unneeded plans to make bigger chunks possible. Transforming the channels right after another, a trade-off between speed and memory consumption has to be made. All plans which are needed can be created **before** the computation and destroyed **afterwards**. Alternatively, a plan can be created on the fly but therefore more often as it is not kept in memory. As the CUFFT library implements various algorithms for computing the fast Fourier transformation, the time needed by the planning algorithm varies depending on the size of the image. In section where the results are presented, this dynamic planning version has the addition *reordered* as the sequence of transforming the multi channel picture is reordered. It therefore only works with chunk wise copies as the whole input of all channels is very large.

The graph in Figure 3.4 will give an overview of the time needed to create a CUFFT plan. Extrapolating the time needed for creating a plan for 200 one-dimensional transforms roughly equals the time needed for a `cufftPlan2d` call of the same size.

As the plans cannot be saved using the CUFFT API they have to be recreated what explains the worse behavior of *FlexDFT, reordered* vs. the simple *FlexDFT* approach. The evaluations will show that using plans only in a short time and therefore have bigger chunks makes the whole execution time a little bit longer. The amount is only small, but noticeable though, see Table 3.9 in the next section 3.6.



**Figure 3.4:** Time needed for the creation of a plan by CUFFT. For 512 and below, the size of the plan is zero and this might be the reason why the creation of the plan is so fast.

Measuring the memory consumption using the CUDA Driver API (see Listing 3.2 on page 29) before and after the initialization of the plan is the only way to make a statement about the memory used for this.

Table 3.6 shows the memory consumption of these plans according to the size of the data.

Leading Dimension	Resulting Plan Size
6048	18.500
4752	14.563
3456	10.563
2560	7.8125
2353	7.1875
1920	5.875
1280	4.671
720	2.25
512	0

**Table 3.6:** Size of a plan in MB created by `cufftPlan1d` with 200 batch jobs each.

Here again, the buffer size allocated by the planning function can be estimated like in the `cufftPlan2d` case. An input line of size  $N$  needs:

$$\begin{aligned}
 size_{\text{cufft,plan}} &\approx 4 \cdot 4 \cdot N + \text{overhead Bytes} \\
 &\approx 4 \cdot N + \text{overhead Floats}
 \end{aligned}
 \tag{3.5-7}$$

As the computation is done in floats, which have a size of 4 Bytes each, it seems that 2 complex or 4 real additional buffers are needed internally.

For the example picture size of  $6048 \cdot 4032$  pixels this results in a minimum of 110.25 Kilobytes device memory needed per transformation (see formula 3.5-8 on page 37). The problem would be copying every single line from the host to the device and back, so this is not efficient.

$$\begin{aligned} size_{\text{cufft,minimal}} &= input + size_{\text{cufft,plan}} + output \text{ Bytes} \\ &= 4 \cdot (Z + 4 \cdot Z + Z) \text{ Bytes} \\ &= (Z + 4 \cdot Z + Z) \text{ Floats} \end{aligned} \quad (3.5-8)$$

*Note:*  $Z$  is the  $\max\left\{\frac{N}{2} + 1, M\right\}$  as both dimensions need to be processed, but as only the one dimension is transformed at a time, so only the larger dimension matters here.

Using this approach, all memory available on the device can be used. For a given amount of free memory, the maximum number of transformations which are possible at a time is approximately  $\lfloor \frac{free_{\text{device}}}{size_{\text{cufft,minimal}}} \rfloor$ . There can be no statement made regarding the operations needed by this approach as there is nothing mentioned in the documentation. It can be assumed, that the algorithms implemented in CUFFT are fast because the Fourier transformation is one of the most needed operations for high performance computing.

The evaluation of the CUFFT library was done without measuring the memory copying operations from and to the device as it can be assumed that the other GPU operations might be applied before and after the transform, e.g. for filtering an image. Unless other stated, host-to-device and device-to-host memory operations are not taken into account for the timing evaluations.

## CUBLAS

NVIDIA's *CUBLAS* library [NVCUB] is a mathematics library implementing simple operations of linear algebra and vector/matrix operations. It is derived from the BLAS (Basic Linear Algebra Subprograms) library but specialized to run on CUDA enabled devices to exploit the highly parallel structures.

The standard Fourier Transformation formula (see formula 3.1-3 in section 3.1.1, page 19) for pictures can be interpreted as two matrix multiplications. For an image having the width of  $N$  pixels and the height of  $M$  pixels the calculation looks like:

$$FT = matRow \times Image \times matCol \quad (3.5-9)$$

With:

$$\begin{aligned} matCol_{k,j} &= e^{\frac{-2\pi i k \cdot j}{N}} \\ matRow_{k,j} &= e^{\frac{-2\pi i k \cdot j}{M}} \end{aligned}$$

Here, *matCol* has the size of  $N \cdot N$  and *matRow* has the size of  $M \cdot M$ .

The drawback of this approach is the fact that the two matrices have to be stored on the graphics card, too. So, the amount of memory needed to calculate the Fourier transformation of an image of size  $N \cdot M$  is:

$$size_{cublas} = M^2 + M \cdot N + N^2 \text{ Complex Floats} \quad (3.5-10)$$

For an image of 24 Megapixels having the size of  $6048 \cdot 4032$  pixels this results in 589.148 Megabytes of data per channel using 32 bit floating point values which is the smallest data type CUBLAS can operate on. As described in section 3.1.1, only half of the data would be needed because of the redundant output. But here, the full array is computed which produces a very large overhead. In addition, the two matrices have to be previously calculated which results in  $O(N^2 + M^2 + M^2 \cdot N + M \cdot N^2)$  (creating both matrices plus multiplying all three matrices) operations for the whole method. The precalculation can take place on the device itself, so it can be parallelized and there is no additional overhead for copying the matrices from the host to the device.

As the goal was to find a memory saving transformation and the CUBLAS approach needs very much device memory, it is not taken into the further evaluation.

#### 3.5.6 Intel Performance Primitives

The Intel Integrated Performance Primitives Library [Steo4, IPP] (*libipp*) is a commercial library which contains various algorithms for all kind of multimedia processing. It is designed for high performance on Intel processors using MMX, SSE and the multi core architectures for parallel computing. Using a planning function this algorithm is similar to the *libfftw3* (see section 3.5.3 on page 29). As it is a native library from Intel, it is also the library of choice for the Intel Larrabee device because it is very likely that Intel will support its own libraries on the new hardware, too. For the Larrabee device the same FlexDFT algorithm which was introduced in section 3.5.5 on page 26 can be used if there is not enough memory on the device for the picture and its transforms.

When Larrabee is available, a closer inspection of this library compared to the *libfftw3* on that device might be interesting. As it is not available these days, this is left to be evaluated in future work for now.

During the work of the thesis, the FlexDFT approach has been implemented on using a pre-release Larrabee SDK provided by Sony Deutschland GmbH and Intel GmbH. But as there are no simulators or even hardware boards available, no statement can be made regarding the speed of this implementation.

The *libipp* uses a special format called *RCPack2D* [IPPMAN] for storing the Fourier coefficients of an image. Exploiting the fact that a forward transformation from real to complex values has a redundancy of about 50% and the fact that only the compressed data of  $\frac{M}{2} \cdot N$  elements

needs to be stored in memory. Replacing  $\tilde{M}$  by  $2 \cdot M$  results in  $M \cdot N$  elements. This is an advantage of  $2 \cdot N$  elements over other approaches like the libfftw3 or NVIDIA's CUFFT library.

Using the libipp on the host the memory calculation is described in formula 3.5-11.

$$\begin{aligned} size_{ipp, host} &= input + output_{\text{complex,RCPack2D}} \\ &= M \cdot N + \frac{\tilde{M}}{2} \cdot N \end{aligned} \quad (3.5-11)$$

On the Larrabee device, the device needs the same amount of memory as the CPU version. On the host the same amount of memory is needed again because the host has to maintain the input and output data. Depending on the next algorithm which uses the Fourier coefficients, the output can be converted from RCPack2D to the normal or even the redundant version of the array. But this costs more memory and computational overhead which is not necessary.

$$\begin{aligned} size_{ipp, larrabee, host} &= input + output_{\text{complex,RCPack2D}} \\ &= M \cdot N + M \cdot N \end{aligned} \quad (3.5-12)$$

$$\begin{aligned} size_{ipp, larrabee, device} &= input + output_{\text{complex,RCPack2D}} \\ &= M \cdot N + M \cdot N \end{aligned} \quad (3.5-13)$$

Like the libfftw3 library (see section 3.5.3 on page 29) the libipp uses a planning function to determine the buffers needed and to optimize the calculation depending on the accuracy selected by the user. Table 3.7 shows the allocated memory for the evaluated pictures.

Image Size		Resulting Plan Size
6048 · 4032 pixels	24.38 Megapixels	81.406 MBytes
4752 · 3168 pixels	15.05 Megapixels	55.078 MBytes
3456 · 2304 pixels	7.96 Megapixels	34.789 MBytes
2560 · 1920 pixels	4.91 Megapixels	26.145 MBytes
2353 · 1568 pixels	3.69 Megapixels	22.426 MBytes
1920 · 1080 pixels	2.07 Megapixels	17.535 MBytes
1280 · 1024 pixels	1.31 Megapixels	14.914 MBytes
720 · 480 pixels	0.35 Megapixels	12.805 MBytes
512 · 512 pixels	0.26 Megapixels	12.129 MBytes

**Table 3.7:** Size of a plan created by `ippiDFTInitAlloc_R_32f` for a complete image, transforming real discrete input data to complex Fourier coefficients. The `hint` flag was set to `ippAlgHintAccurate` for accurate results.

## 3.6 Results & Comparison

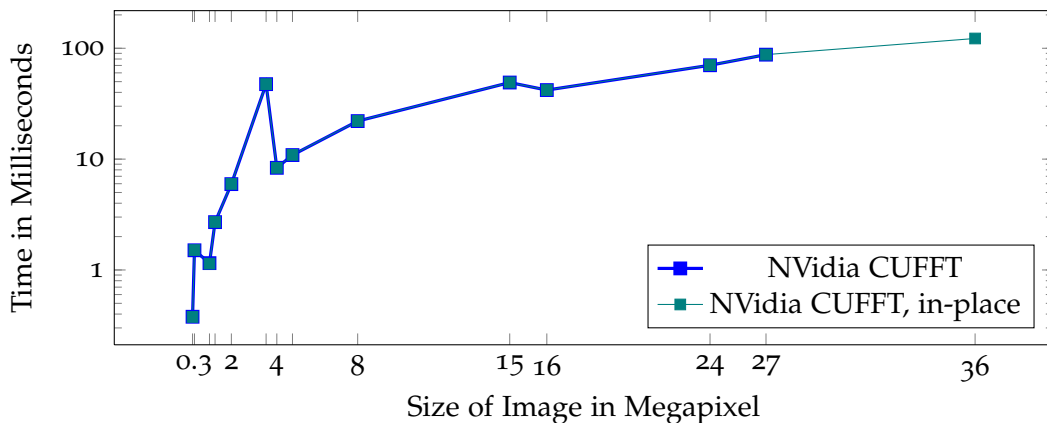
This section will present the results from the experiments described in section 3.5 beforehand.

### 3.6.1 Correctness of FlexDFT

Testing the correctness of the algorithms was done by correlating them with each other and calculating the mean squared error. The result was a correlation of 1 resulting in a mean squared error of 0. So the algorithms all calculate the same result and the same coefficients.

Due to rounding errors and the limits of single precision, very small differences might have been eliminated in the error and correlation calculations.

### 3.6.2 NVIDIA CUFFT

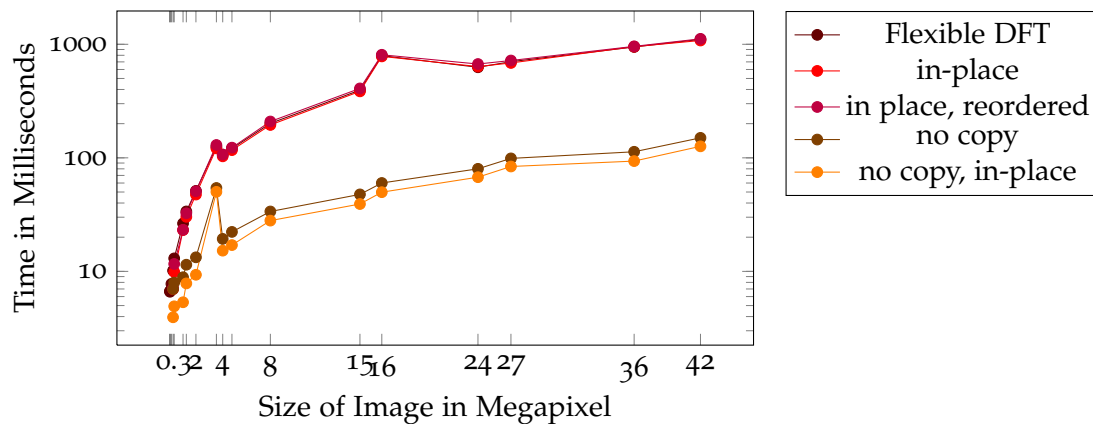


**Figure 3.5:** Performance evaluation of CUFFT library. Note that the both curves are very close. The normal version goes up to 36 Megapixels and the in-place version extends up to 42 Megapixels.

The graph in Figure 3.5 shows the standard NVIDIA CUFFT library's performance. It differentiates between the in-place and out-of-place transform but they are very close and there is hardly a noticeable difference. Table 3.8 shows some selected values. The "-" denotes the memory problem: even 1024 Megabytes of memory were not able to hold a 36 Megapixels or 42 Megapixels image, its Fourier transformed and the plans.

Megapixel	CUFFT out-of-place	CUFFT in-place
42.00	-	-
36.75	-	122.5241
27.00	87.5760	87.6557
24.39	70.2843	70.4869
16.78	41.9672	41.9787
7.96	22.0331	22.0764
1.05	1.1527	1.1398

**Table 3.8:** Performance Evaluation of CUFFT library. All values are Milliseconds.



**Figure 3.6:** Performance Evaluation of the developed FlexDFT approach

### 3.6.3 FlexDFT

Figure 3.6 and Table 3.9 display the performance of the FlexDFT approach developed in this thesis. The difference between the upper three and lower two curves can be explained by the copies from and to the device. Additionally, transposing the data is done on the CPU. As this is more time consuming due to copying the data from the device to the host in between, it is represented via the upper curves in the graph. The no-copies-approach saves a lot of time but consumes more memory as discussed above in section 3.4.

Here, the in-place and out-of-place are close but especially in the version having no additional copies from CPU to GPU the difference is noticeable. The reason for this might be that there are less memory allocations in the in-place version as the output does not need to be allocated or the reason lies deep in the CUFFT library. But there are no details available from NVIDIA regarding this topic so there cannot be made any further statement.

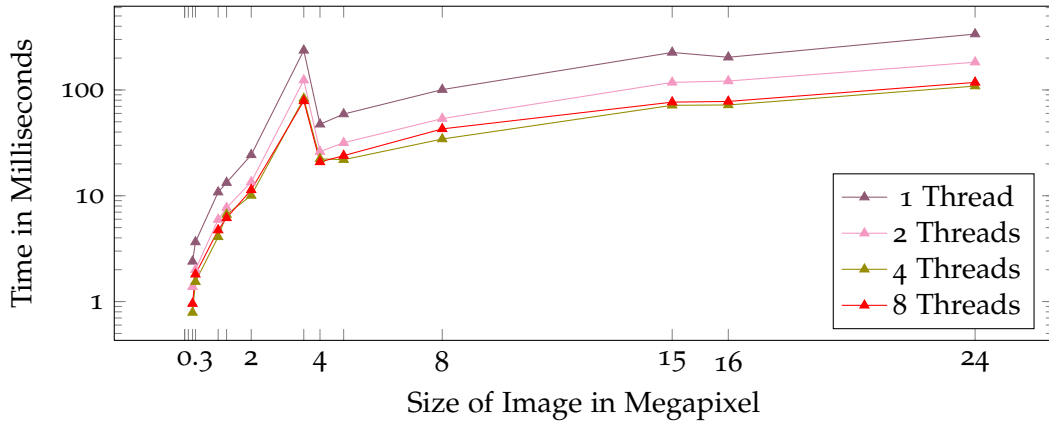
The in-place-transform version of this is chosen for further comparisons as it is the fastest and the memory consume is still passable.

### 3 Fourier Transformation on Graphics Cards

Megapixel	FlexDFT, with copy	FlexDFT reordered	FlexDFT, no copy
42.00	1080.0730	1117.0616	126.2715
36.75	953.0565	958.9585	93.6097
27.00	685.0000	719.7221	83.9141
24.39	636.7871	670.2104	67.5408
16.78	782.3658	808.5282	49.8302
7.96	194.8212	208.9004	28.0015
1.05	23.1000	23.2278	5.3464

**Table 3.9:** Performance Evaluation of the developed FlexDFT approach. The displayed values are taken from the in-place-transform versions. All values are Milliseconds.

#### 3.6.4 libfftw



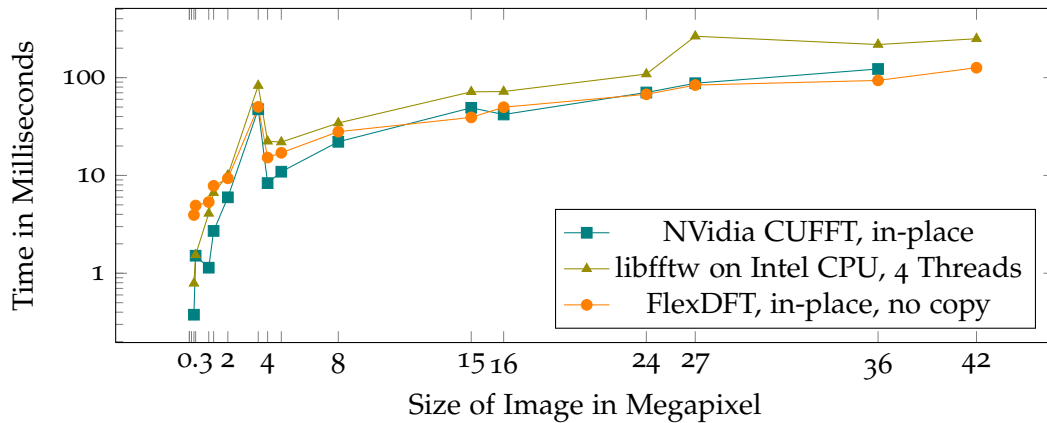
**Figure 3.7:** Performance Evaluation of libfftw using different Threads

Figure 3.7 shows the evaluated results for the libfftw3. As a quad core CPU was used the libfftw was set up using multiple threads, too. The fast versions using four or eight threads can be explained in with the power of the CPU on one hand and a good parallelizing scheme estimated by the planing function before on the other hand.

As using four threads provided the best results, it was chosen to be the CPU reference.

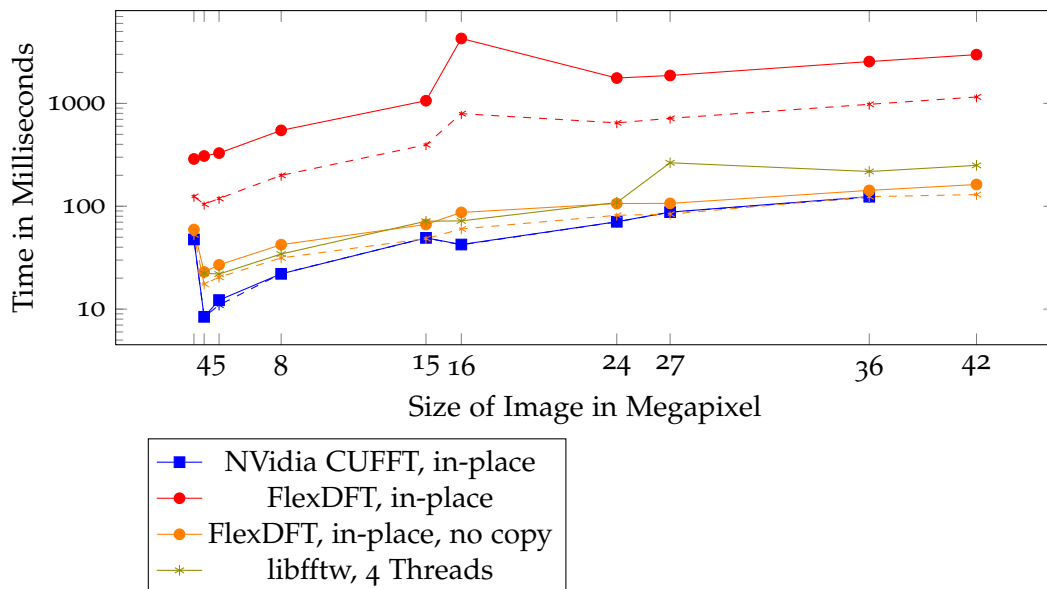
#### 3.6.5 Interpretation and Comparison of the Results

This section will compare the results shown above to have a direct view on all evaluated methods.



**Figure 3.8:** Performance Comparison of CUFFT, FlexDFT and libfftw.

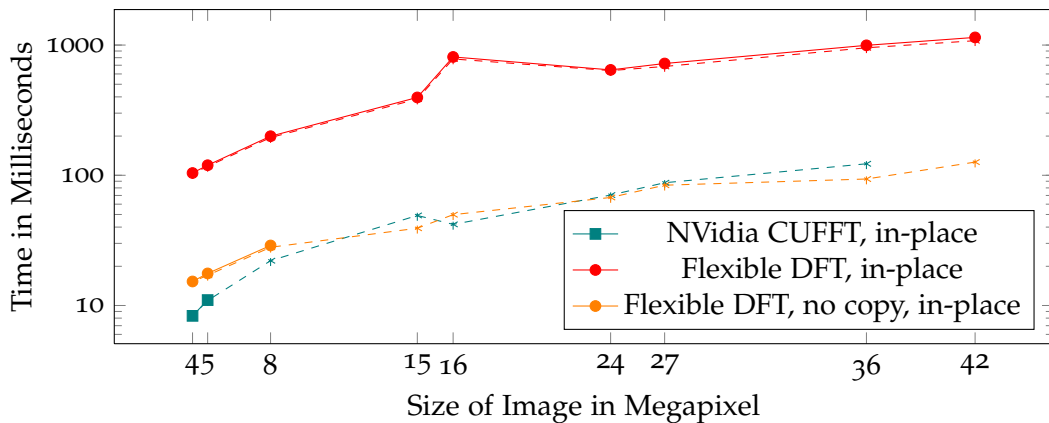
Beginning with the performance, Figure 3.8, the goal is to have a better performance than the CPU. Otherwise, the usage of the GPU would not make sense if it is too slow. A direct comparison of CUFFT and FlexDFT shows that they are very close for the bigger images starting at about 8 Megapixels. For the small images up to 8 Megapixels, CUFFT is faster. This advantage shrinks for images of larger size. From 36 Megapixels on, CUFFT is unable to compete with the approach of this thesis and libfftw3 any more.



**Figure 3.9:** Performance Comparison with a heavily loaded CPU. The dashed lines represent the timings measured with an idle CPU.

Having a CPU which is under heavy load could mess up the good performance of the previous graph. Figure 3.9 shows a comparison of an idle and loaded CPU. The dashed lines show the values in the idle situation. The straight lines show the performance of the CPU with all four cores loaded. Every core is copying memory blocks of 128 Megabytes forward and backward with a short sleep interval of 100 Milliseconds. The desired effect was to keep the CPU busy and also the memory bus is full of other data.

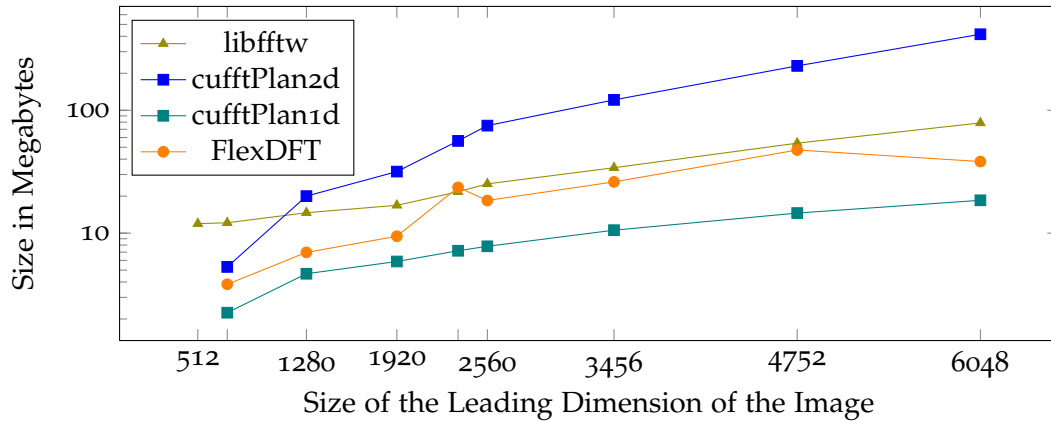
The black curve shows the CPU reference, the four-threaded libfftw3 with an idle CPU. The graph clearly shows that a fully loaded CPU influences the performance. But the CUFFT library and FlexDFT without copies is still faster than the libfftw3 on a CPU doing only this.



**Figure 3.10:** Performance Comparison with 700MB less available on the GPU, so around 160MB are left. The dashed lines represent the timings measured with the full amount of memory available.

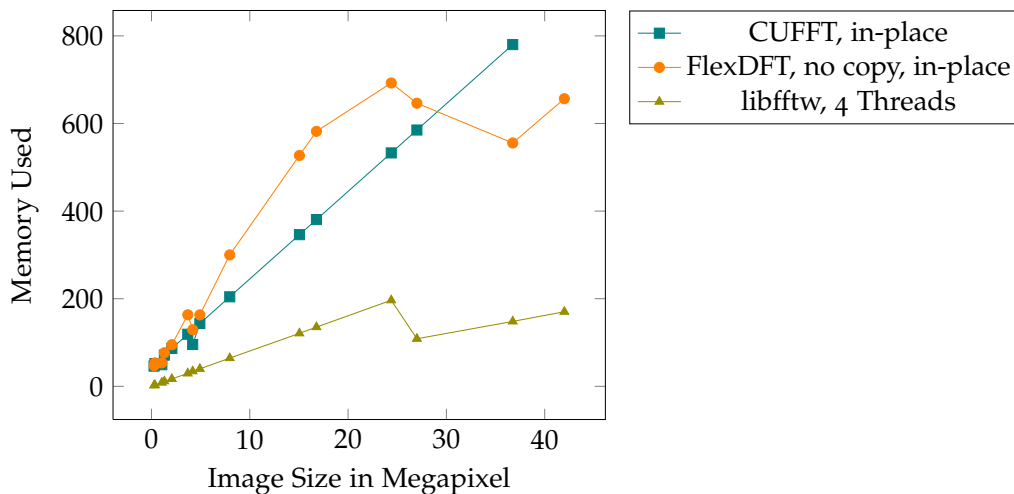
In Figure 3.10 the graph shows the performance of the same hardware but with the video memory of the NVIDIA GTX 280 reduced by 700 MB. As CUDA reports 860 MB left in idle state this leaves around 160 MB for the evaluation which is quite less. What can be seen on the graph is that the influence is only very small for the performance. Of course, CUFFT and the no-copy-approach will stop working when the plans or input+output, respectively, do not fit to the device any longer. But comparing the red line, the evaluated results, with the dashed red line, which was measured with the full memory available, there is hardly any difference.

Graph 3.11 gives an overview of the plans created by the algorithms. Whereas the libfftw3 keeps its memory consumption quite low compared to cufftPlan2d, the memory usage of the other both methods was set to a chunk size of 200. Changing the chunk size changes the memory usage of them linearly, too. FlexDFT relies on cufftPlan1d and this itself linearly sizes the buffer according to the number of batch jobs. Interestingly, the CUFFT versions do not report any memory usage for a dimension less or equal to 512 pixels in the leading dimension. This might be because there is no plan needed and one thread - as there are



**Figure 3.11:** Comparison of the Plan sizes. The Chunk size for FlexDFT and cufftPlan1d was set to 200.

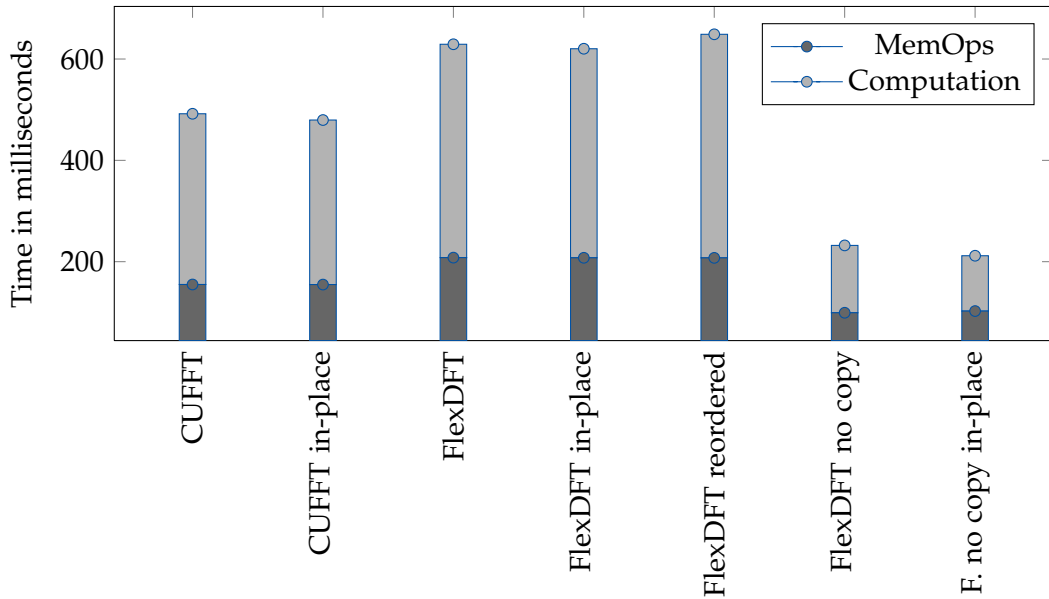
512 available on the device - computes the output of its element in the coefficient matrix. Additionally, the shared memory could also be the explanation. The 16 KB shared memory do not show up in the memory report but might be large enough for a transformation of 512 and less elements.



**Figure 3.12:** Comparison of the memory consumption of CUFFT, FlexDFT on the device and FFTW on host side.

Figure 3.12 shows a comparison of the memory usage of the CUFFT library and FlexDFT on the device as well as libfftw3 on host side. Here, libfftw3 is the clear winner as it has very small plans only. The FlexDFT needs more memory than CUFFT as it packs the graphics card's memory full with plans it needs later. These plans are allocated before the first line is

transformed and stay there for the whole computation time. The same reason applies for the bend for the larger images when the memory consumption is decreasing. There are more chunks with a smaller number of lanes transformed at a time as there is not enough memory for larger plans. Because the formula for calculating the plan size is only a rough estimation, there is space left on the device which is not fully filled. The bend on the `fftw`-curve might be a results from `libfftw`'s internal calculations and optimizations.



**Figure 3.13:** The whole time needed for the 24 Megapixel image split up into computation and memory operations.

The graph in Figure 3.13 shows the timings split up into the time needed for copying the data to the device and the time for the pure computation. As the CUFFT and the no-copy approaches of FlexDFT usually do not copy data from host to device and back, this has been added here. This is the reason why the numbers differ from the evaluated results in the other graphs. The timings measured in the *with copy* and *reordered* fit the ones measured for the standard algorithm. The variance of the other approach's values might result in memory copy latencies or hidden calculation latencies of the Fourier transform.

#### 3.6.6 Conclusions

In this chapter, the Fourier transformation was evaluated on NVIDIA CUDA devices, Intel Larrabee prototypes and on the x86 architecture CPU side. It has been shown that the most efficient way in transforming an input image into the frequency domain is using today's multi-core CPUs in combination with a good scheme of how to decompose the calculations.

As the standard approach of the CUFFT library uses very much memory for intermediate results, a less memory intensive method has been evaluated here. The Fourier transform can be separated for each dimension by its mathematical definition. This allows to transform both dimensions of an image by transforming all lines the first dimension independently and then the second dimension, also independently. Using this separation, the FlexDFT approach has been introduced. Internally, the FlexDFT used the CUFFT library's one dimensional transforming function for its calculation.

It was implemented in two different ways: For the first approach, the input and output reside completely on the graphics card. This is the faster of them both as it does not need any memory copy of parts of the calculation from or to the device. The second implementation transfers chunks of the input to the device, transforms them there and copies them back. The same is done with the second dimension until the whole image is transformed.

The evaluation was done using different, mostly larger image sizes. It has shown that it can compete with the CUFFT library and with the libfftw3 used as the CPU reference on the system when the first approach described beforehand is used. The second approach comes into play for image sizes the first method cannot handle any more. So, in theory, very large transforms can be made possible. The CUFFT library is still used for the one dimensional transforms but cannot compete for the two dimensional transforms any longer.

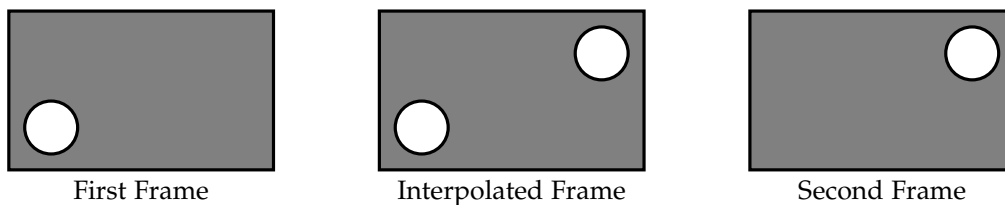
The results evaluated in this chapter have also been presented on the "GPU Technology Conference 2009" held by NVIDIA at the end of September 2009.



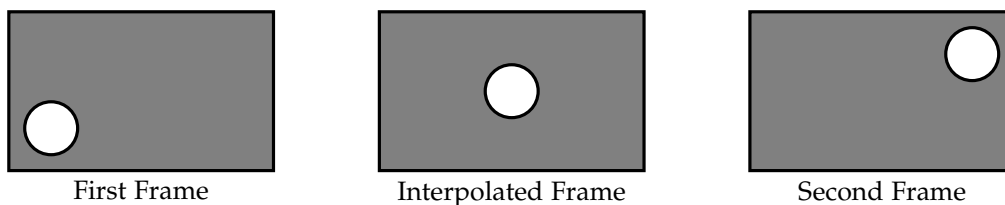
## 4 Motion Estimator on Graphics Cards

A motion estimator is designed to analyze a (short) sequence of frames and tries to estimate the global and local movement [Erd98]. Using this information, the algorithm can interpolate or extrapolate the movement. For interpolation, the algorithm can help to improve intermediate pictures, which are derived from preceding and succeeding images in a video stream. In the case of extrapolation, the motion estimation is used to predict the movement from a series of known images.

As today's TV and PC displays get bigger and the hardware gets faster in every generation, the standard frequency of 50 Hz is outdated. Today's TV sets can display 200 to 240 full images per second but the standard TV signal is still only 50 Hz. Also for standard video streams displayed on the PC, the motion estimator can be used to improve the quality of the pictures. Figure 4.1 and 4.2 show the results for interpolation. In the first Figure, the interpolated image is created by linear interpolation of both existing images. The second Figure simulates the motion estimator result where the created image has the complete object on its interpolated position.



**Figure 4.1:** Using linear interpolation results in strange interpolated pictures, the circle fading in and out.



**Figure 4.2:** Using a motion estimator moves the actual circle to an interpolated position.

The block matching motion estimator usually works in the following way:

Two pictures are separated into small blocks. The algorithm tries to find blocks from one image in the block set of the other image. This is done by generating a vector field for each block pointing to the position of the same or an equal block in the next picture. As the blocks cannot match perfectly in every case - depending on the compression used and other factors, the case of perfectly matching blocks is very rare - the vector field does not represent a clean motion but has some "holes" in it. These holes are *cleaned* for example by using an average filter. After this step, the vector field can be used to generate intermediate images which straddle from one picture to the next.

In this algorithm, it is a difficult task to find matching blocks as it is very time consuming to compare all blocks pairwise. Using larger blocks reduces the quality of the interpolated images because the probability raises that the tracked object in one block is not in another block at the same position or even crosses borders to other blocks. On the other hand, reducing the block size is very time consuming as there are more comparisons needed.

The method used to circumvent this problem is to use a so called "Update Star" which defines the positions where a block has to be searched for in the other block set. This reduces the complexity from  $O(N^2)$  for  $N$  blocks down to  $O(N)$  as every block is only compared to a fixed number of possibly matching candidates.

As the PC is one of the target platforms of the motion estimator, the idea comes up to run it on the graphics card. The flexibility of CUDA based graphics cards would allow this complex algorithm to be implemented on the device to use the highly parallel structures. As the frame buffer also resides on the graphics card, the generated image can directly be displayed without the overhead of copying it to the host and then back to the frame buffer.

In this thesis, the focus lies on only the cleaning part of the motion estimator and its parallelization.

One of the most common methods of erasing peaks or holes is the average filter. However, every algorithm needs to inspect the neighborhood of the element it is updating. The specific update mechanism varies in every implementation. A very simple neighborhood is the  $3 \times 3$  surrounding of the according block. In this thesis, the neighborhood is reduced to a  $2 \times 2$  blocks with the current block in the lower right corner.

In the following section 4.1, the ideas to parallelize the cleaning algorithm will be described on CUDA 4.1.1 and Larrabee 4.1.2 side. After that, the presentation of the results and a short summary will end this chapter in section 4.2.

### 4.1 Explanation and Problem of the Cleaning Algorithm

The basic idea of the cleaning algorithm is to iterate over every element in the vector field and update them like in a simple filter operation. One particular feature is that it updates the elements in place, so that the new value is written back to the current position in the vector field where it is read from. From this demand, the parallelization of the algorithm is a difficult task because the block dependencies have to be satisfied.

A very simple cleaning filter could look like the one described in equation 4.1-1, with  $i$  and  $o$  being one element in the vector field.  $i$  is the input of the current formula and  $o$  the result of the calculation. As shown, the output of the surrounding elements have to be calculated before the actual element itself can be determined.

From a mathematical point of view, this formula cannot be parallelized as it is not clear where to start the recursion. Choosing a direction where the dependency can be violated simplifies the problem.

$$\begin{aligned}
 o_{1,1} &= i_{1,1} \\
 o_{x,1} &= i_{x,1} \\
 o_{1,y} &= i_{1,y} \\
 o_{x,y} &= \frac{1}{4} \cdot ( o_{x-1,y-1} + o_{x,y-1} + \\
 &\quad o_{x-1,y} + i_{x,y} )
 \end{aligned}
 \tag{4.1-1}$$

Using this simplified filter, the algorithm starts as follows: the blocks left, in the upper left and above the block have to be updated first, before the vector at  $(x, y)$  itself can be updated.

Of course, the linear system from equation 4.1-1 can be solved mathematically and every  $o_{x,y}$  expressed without the recursion. The upper left corner is defined and so, every succeeding element can be calculated. The problem here is that the expression for an element gets bigger and bigger, the further it is away from the starting point as every element it depends on, has its own recursive dependencies and so on. For an element of position  $(x, y)$  that means a complexity of  $O(x \cdot y)$  for each pixel what is clearly quadratic. For the whole array of the size  $N \cdot N$ , this results in  $O(N^4)$  operations.

On the other hand, the computation can be done iteratively (see Listing 4.1), starting in the upper left corner for this case. Here, the complexity shrinks to  $O(N^2)$  for the given array as every element can access the already calculated values of its predecessor. The problem here is how to parallelize this operation.

For a simple single threaded (and thus not parallelized) loop this operation is no problem as shown in Listing 4.1 and Figure 4.3.

---

**Listing 4.1** Single Threaded Cleaning Operation

---

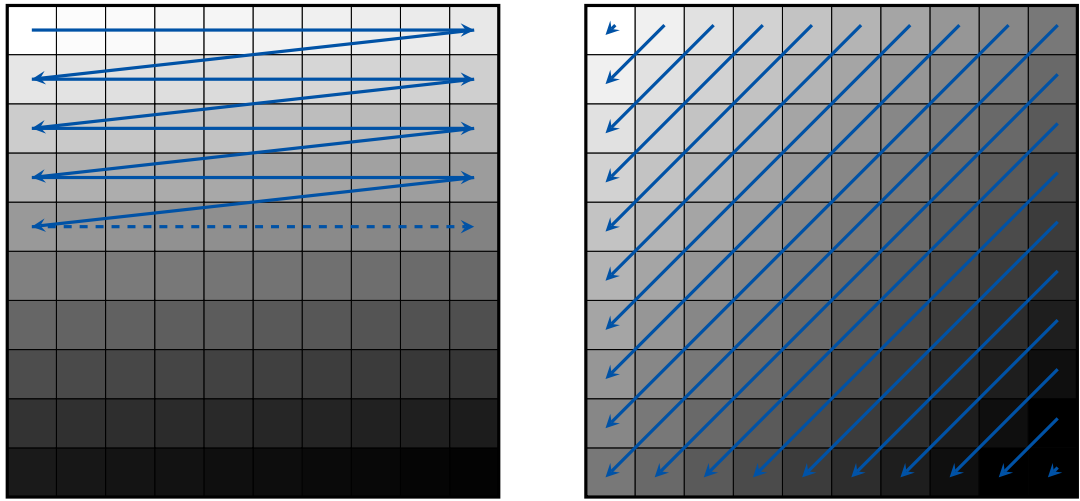
```

for(int y = 1; y < HEIGHT; ++y)
    for(int x = 1; x < WIDTH; ++x)
        v[x, y] = v[x-1, y-1] + v[x, y-1] + v[x-1, y] + v[x, y];

```

---

The dependencies are satisfied because the inner loop iterates first, so  $v_{x-1,y}$  is defined when  $v_{x,y}$  is used for updating  $v_{x+1,y}$ . The same claim holds for the  $v_{x-1,y-1}$  and  $v_{x,y-1}$ . The first row  $v_{x,0}$  and the first column  $v_{0,y}$  of the blocks are not updated by the algorithm and are implicitly assumed to be fixed. This is to have fixed values for first updates.



**Figure 4.3:** Iteratively update every element, following the arrows. **Figure 4.4:** Parallel cleaning operation with every element of the same color under an arrow computed in parallel.

To use the parallel hardware architectures of CUDA or Larrabee graphics cards, the algorithm has to be parallelized, too. In this case it is impossible to simply parallelize the inner or outer for-loops as the dependencies would be broken.

A closer inspection of the dependency shows that the pixels which are lying in a diagonal are independent from each other and thus can be updated in parallel. This is the key to parallelize the problem. In the following, one diagonal is called “lane”. A lane is always a  $45^\circ$  line starting in the first line or last column of the image, see Figure 4.4.

Separating an image into lanes gives three different types of lanes:

1. **Starting:** The starting lanes are the first lanes which have a length smaller than the height of the image. These lanes are in the upper left corner of the image and form a triangle. There are  $Height - 1$  lanes of this type.
2. **Middle:** The next lanes all have the same length equal to the height of the image. They form a parallelogram of  $Height \times (Width - Height - 1)$ .
3. **Ending:** Finally, the lower right is formed by all remaining lanes with a length smaller than the height of the image. Again, there are  $Height - 1$  lanes of this type.

All together, there are  $Width + Height - 1$  lanes in an image of the size  $Width \times Height$ .

In the following section the options to bring this to CUDA and Larrabee devices will be discussed.

### 4.1.1 CUDA

Porting this algorithm to the parallelized architecture of a CUDA device raises some problems which are described in the following:

**Limited number of threads:** Due to the design of the CUDA hardware, the number of threads in a block is limited. Currently, the maximum thread number is 512 and thus smaller than the Full HD resolution with a height of 1080 pixels.

**Synchronizing blocks not efficiently possible:** As there is no mechanism provided by the CUDA SDK to synchronize different blocks it has to be implemented by hand. A busy-wait construct would be possible where a thread is caught in a `while`-loop until a given variable has the correct value. This is possible but not efficient enough to guarantee real time operations.

These are problems which will come up when realizing the ideas to bring the algorithm to CUDA. In the next sections these ideas will be described and analyzed.

#### Single Loop

At first, the single threaded version of the algorithm could also be implemented on the CUDA device. This has been done for comparison reasons. As this version is not parallelized and thus does not benefit from the CUDA architecture at all. It is very slow as expected. Figure 4.3 shows the scheme of how it works.

#### One Block per Lane

To parallelize the problem, the block structure of CUDA is used. The idea is to create as many blocks as lanes are needed. The blocks are synchronized by a simple busy-wait loop which waits until a global variable, accessible by all threads, has their block id. When this condition is true, the threads in the block can work and every thread updates one pixel of the lane. Finally, when all threads have done their work and are synchronized via the `__syncthreads()` call, the last thread updates the global variable so the next block's threads can begin their work. Figure 4.4 shows this idea. The pixels are updated in the sequence of white to black. The pixels of the same color (under an arrow) are updated in parallel.

However, it came up that this method has two problems which cause each other:

1. **Block synchronization mechanism:** The blocking mechanism which ensures that only one block's threads are working is very inefficient. As it is not known what the internal scheduler is doing and how it is influencing the thread's and block's execution order much time can be wasted in the busy-wait loop. CUDA does not guarantee any order of execution and as such, much time can be wasted here.

2. **Inconsistent result:** When evaluating the result of this method there some inconsistencies in the “cleaned” array were found. There were false values at some places in the result which of course caused then wrong values in every depending element of the array. A closer inspection has shown that the values which have wrong results use old values in their calculation. So, maybe the caching mechanism does not update the values fast enough and the result of the calculation is wrong. This might be a result of the way CUDA’s scheduler works. As there are no guarantees about the execution order, the algorithm does not fit to the underlying hardware.

To solve these problems, a very fast block synchronization mechanism would be needed, ideally provided by the CUDA API itself to have a consistent and reliable behavior.

Using one block per lane also raises the problem of many unused threads in the upper left and lower right triangle’s lanes. There, every block has the same amount of threads. But not every thread is used as the number of elements to update is smaller than the number of threads. The threads with no real element to work on also occupy the processors on the device but do not produce any (valuable) output. They especially consume time in the busy-wait loop which is totally unnecessary as they are not needed at all in those lanes.

### One Kernel per Lane

The two problems and the overhead of unused threads can be encountered by using only one separate kernel call per lane. As all elements in a lane are completely independent from each other, their lane can either be split in multiple blocks or one thread can work on multiple elements. However, the problem with this method is the number of kernel calls made. As every lane has its own kernel working on it, the number of blocks and threads can be optimized. Having a large image, for example in Full HD resolution of  $1920 \times 1080$  pixel this results in 2999 kernel calls from the host to the graphics card. This is an enormous overhead in comparison to one kernel call with a fixed number of blocks and threads. The basic idea is shown in Figure 4.4.

### One Kernel per multiple Lanes

To reduce the number of kernel calls made, one thread, normally working on one lane, can work on multiple succeeding lanes. Unfortunately, this idea works for small lanes only which are as small as the warp size or even smaller. The conflicting points are the points on the edge of a warp which is executed. It cannot be guaranteed that all threads finish their work on one lane first and then continue on the next lane so that this dependency is satisfied. There is no mechanism provided by the CUDA API to control the execution order of the warps.

### Dynamically created Blocks

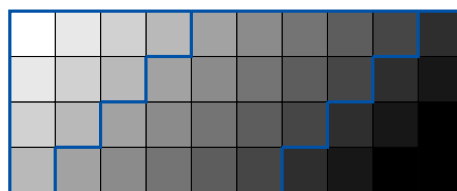
For a large set of data the 512 threads available on a CUDA device might not be enough for the whole lane. Using multiple blocks for this is the logical result from this. This approach calculates the number of blocks needed in the way that no additional thread is needed and doing unused work because its target is out of the field boundary. For 512 elements, one block is enough. For a larger number the minimum amount of blocks are used so that every thread in every block has work to do and its computation time is not wasted.

### Image split into Multiple Chunks & Puzzle approach

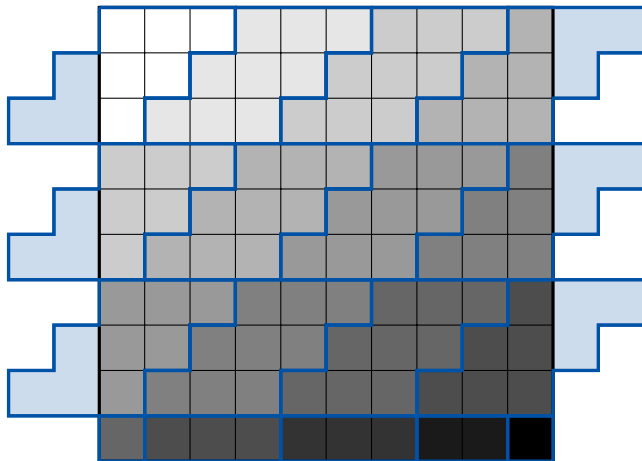
From the knowledge of the above ideas and the analyzation of their shortcomings, one last approach was evaluated. The image is partitioned in chunks of horizontal stripes which all have the same fixed height (except the last one which might be a little bit smaller to stay in the boundaries of the input image). These chunks can be treated as individual images and be worked on with the ideas from the above sections. As seen in section One Kernel per Lane more than one lane has to be updated by one kernel. As seen in section One Kernel per multiple Lanes above, the lanes for this have to be short. Otherwise, the result might be incorrect as the execution order is not guaranteed by CUDA.

Having this short lanes raises the kernel calls so that this solution cannot be the fastest of the presented variants. Speeding this variant up would be possible by the following idea:

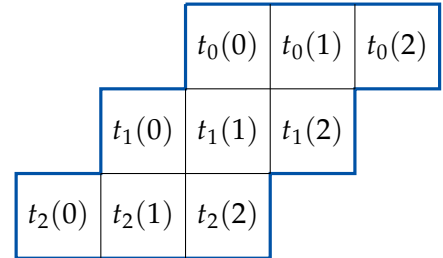
On every left and right side of one stripe, the block which is executed there forms a triangle. As described in the introduction of the algorithm above, these triangles are in fact parallelograms like the blocks in the middle but do not have values to work on in half of their elements. Having threads which can skip these unused parts would speed up these triangles and hopefully shorten the time needed for the algorithm. Figure 4.5 shows these three types. But as shown in section 4.1.1, it is only possible to work with very small stripes. Having stripes with the height of a warp means that every thread in the warp executes the same data path and thus there is no benefit. Because of some threads in the lane can skip the computational part, this would only bring an advantage in for a lane which needs more than one warp. But having these large lanes and stripes, the dependencies in the vector field cannot be guaranteed to be satisfied and thus the results might not be correct (see the sections above).



**Figure 4.5:** *The blue lines separate the three types of blocks for the given vector field.*



**Figure 4.6:** An image of  $10 \times 10$  pixels and its puzzle blocks. Puzzle blocks with the same color can be processed in parallel. Light blue areas show the wasted threads.



**Figure 4.7:** Execution order of one puzzle piece.  $t_x(y)$  denotes the  $x^{\text{th}}$  thread in its  $y^{\text{th}}$  iteration.

Figure 4.6 shows this so called “Puzzle” approach and Figure 4.7 next to it a closer view on one piece of the puzzle. Within one part of the puzzle, the dependencies are kept. Multiple pieces of the puzzle can be computed at the same time when they do not have a common border. In general, this depends on how the filter is designed. For the one used here (see equation 4.1-1) the parallel executed pieces have to be one pixel apart. This assures the dependencies over multiple puzzle pieces.

For small puzzle pieces of size  $16 \times 16$  the algorithm works correct but very slow. Using a larger size for the pieces, e.g.  $128 \times 128$  brings the problem discussed beforehand. The threads in the warps iterating over multiple elements finish their work after other threads have started which need the results.

The problem with this approach is the high number of kernels which have to be launched. Waiting for the completion of previous kernels makes the whole approach very slow.

#### Reorder the memory

A general idea to speed up the computation was to reorder the vector field elements in memory to exploit the cache structures of the CUDA device. These structures allow coalesce operations on the memory from the threads in a warp so that every thread can access its variable without delay and wasted cycles. The idea is that neighboring threads access aligned memory which is also neighbored [NVMan].

In this cleaning algorithm, neighboring threads which work on a lane do not access neighboring memory but have an offset of  $Width - 1$  as the next thread reads and writes the

left-handed element below. Reordering the vector field would bring the data in an aligned order according to the thread structure. Figure 4.8 demonstrates this.



**Figure 4.8:** Schematic drawing of the reordered data of the vector field. The letters mark the position of the block in both arrays and the arrows show the dependencies of block E.

The overhead of this idea is now taken away from the algorithm itself but put into the previous and later sections for “encoding” and “decoding” the vector field. This approach has been tested and does not bring any improvement of performance. The data used was just dummy data, so it was not actually reordered for simplicity. As it does not improve the time needed, there were no further investigations.

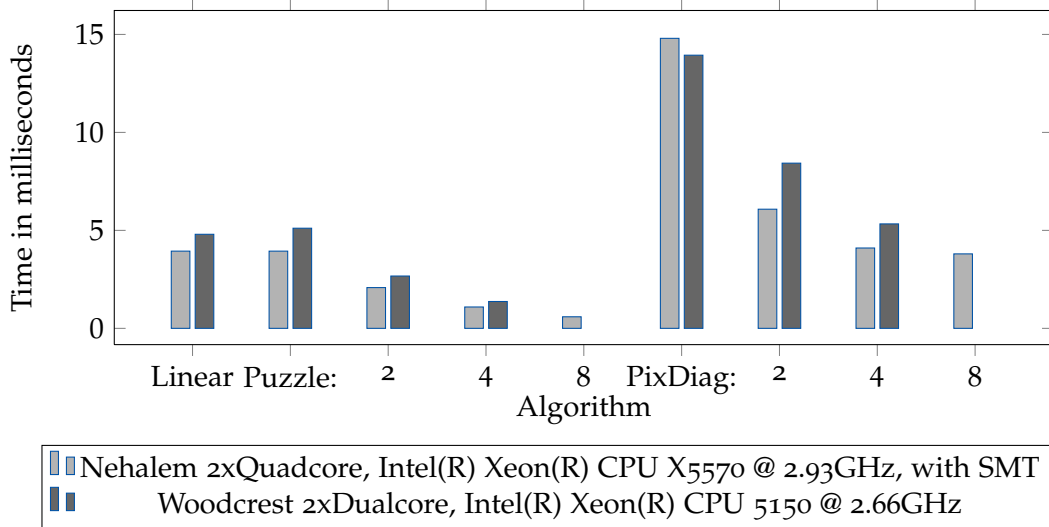
#### 4.1.2 Larrabee

For the Intel Larrabee, all the ideas developed above can be tested to find an optimal performance. Especially the ways of how many lanes a task will operate on or the size of a puzzle part need to be determined there.

The graph in figure 4.9 shows a reference implementation of the motion estimator cleaning part provided by the Sony Deutschland GmbH. As these numbers are measured on Intel processors, the Larrabee architecture should provide an equal or even better performance as it can parallelize even better. Depending on the scalability of the algorithm itself, Larrabee can at least be expected to have an equal performance. Making use of the 512 Bit vector units could also speed up the execution. The basic idea of the “One Block per Lane” scheme was implemented on Larrabee. The implementation has to be evaluated when there is a hardware board available.

## 4.2 Results

After describing the problem and the ideas for the algorithm in the last two sections, here, the evaluated results will be discussed. As stated before, not every idea works on CUDA graphics cards due to broken dependencies. Nevertheless, the values evaluated for the



**Figure 4.9:** Performance of the motion estimator cleaning part provided by Sony Deutschland GmbH for an array of  $1920 \cdot 1080$  in size. Puzzle and PixDiag are two algorithms equaling those above but are not disclosed in detail. The numbers right of "Puzzle" and "PixDiag" indicate the threads used for the according algorithm. The plain algorithm indicates only one thread like.

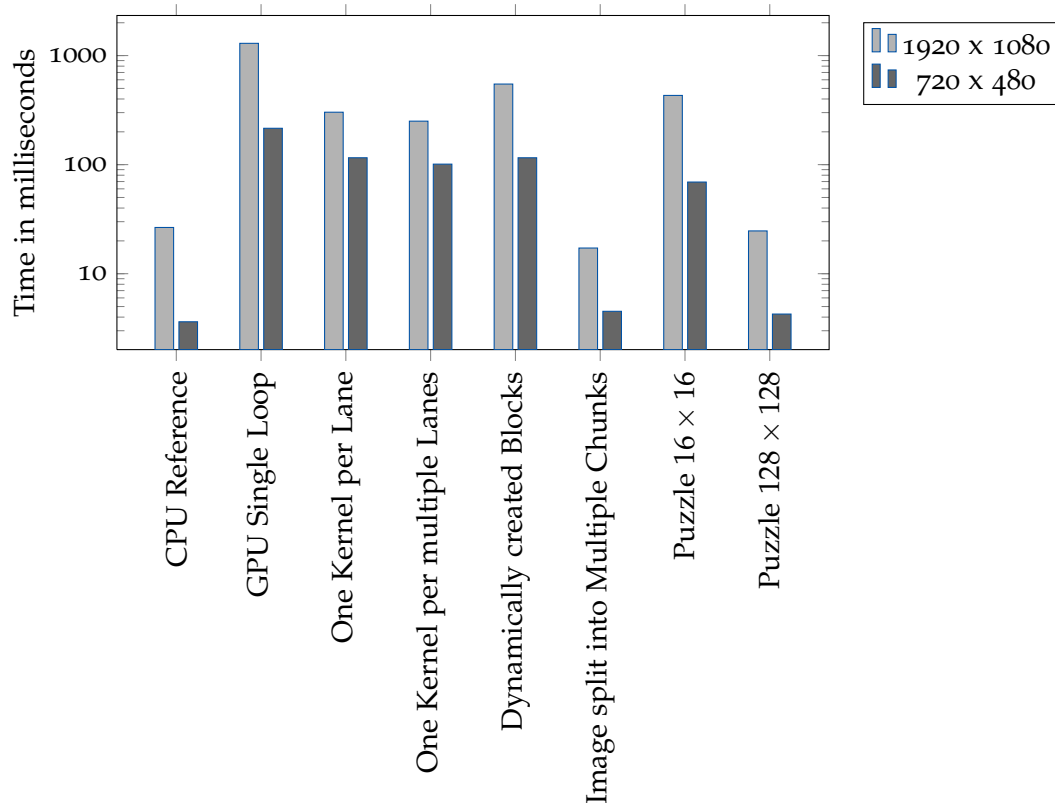
not-correct algorithms are also drawn in the graphs but have to be taken with a pinch of salt.

#### 4.2.1 Evaluated Results

Figure 4.10 shows the performance of the evaluated methods. For comparison, all methods above have been measured and are displayed here except for the *One Lane per Block* variant as it performs far to slow. Besides the first two bars all other bars are implemented on the GPU using CUDA.

Unfortunately, no algorithm reaches a good performance, for neither the Full-HD image or the smaller one. In addition, they suffer from the problems stated above.

The first two bars "CPU Reference" and "GPU Single Loop" show a nice comparison of the single threaded performance of CUDA on the GTX 280 and the Intel CPU. Both use a simple for loop to iterate over the vector field and the CPU is much faster. The GeForce card can only provide its full power when the task is (highly) parallelized. The problem here is the dependency of the data as mentioned above and thus the problem cannot be parallelized very easy.



**Figure 4.10:** Performance of the implementation of the cleaning algorithm. The rotated labels on the x-axis correspond to the subsection names in section 4.1.1.

The bad performance of the CUDA device comes from the high number of kernel launches to guarantee that the data dependencies are met. Even the “Puzzle” approach which tries to avoid unused computation fails here.

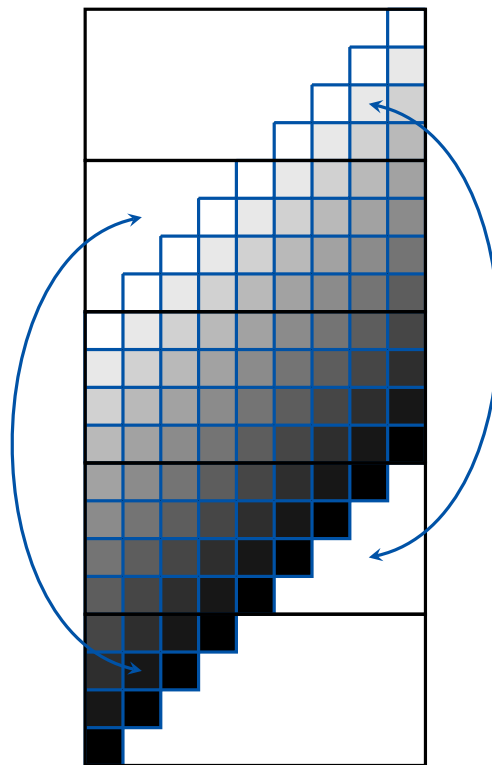
#### 4.2.2 Ideas for Future Work

Cleaning the blocks from runaways with the presented algorithm using CUDA hardware has the problems mentioned above. As it is not possible to synchronize blocks on the device (via CUDA kernel code) other than using *busy wait* or even more complex mechanisms which wastes a lot of computation bandwidth, the blocks have to be synchronized “outside”, which is on host side. This results in many kernel launches having only one block running.

To avoid too much overhead and unused threads doing nothing, one could place more than one data set among each other. There would still be a large number of kernels to start, but one kernel could complete more than one image. For  $1920 \times 1080$  pixels one lane has the size of 1920 elements and extends over three images/data sets. The same scheme would

be possible with data sets placed side by side. For handling the borders, a line filled with zeros could be added.

Additionally, the data the upper and lower triangles are working in could be filled useable data, so they are not wasted, too. Figure 4.11 shows this idea as an example. The black rectangles represent the memory allocated, the filled squares represent a vector field block. The field in the center is completely covered by the lanes and for the other frames, they can be partitioned as shown by the arrows. This would guarantee that all hardware threads are working and no computational power is wasted. Additionally, a number of frames can be worked on at once instead of one single frame only. But the drawback is, that the field is split and a mechanism for border handling has to be introduced, too (not shown in the figure).



**Figure 4.11:** *The black rectangles represent a vector field each. The lanes have all the same length and are represented by the same color. The arrows show where to put the part of an image which is not covered by working lanes.*

The algorithm designs mentioned above have been created keeping in mind not to break the dependencies of the blocks. Breaking the dependencies, e.g. by splitting the picture vertically in the middle, could improve the performance. The two parts can then be positioned among each other. Here, the same scheme as mentioned before could be applied which results in “shorter” kernels having less pixels to work on but reordering one half can reduce the number of threads having no work.

The broken dependencies might be insignificant to the algorithm as the next iteration on the following frame could be done in reversed order and then these particular points are at another location. According to internal studies supplied by the Sony Deutschland GmbH, the cleaning process still converges to a smooth set of vectors describing the overall motion of the latest frames.

Those ideas base on the assumption that CUDA will not allow to influence the scheduling or synchronization in any way. If a way comes up to do this, it will be worth a try.

What also can be tried is to transform the vector field into the frequency domain and work on the problem there. Of course, this causes the overhead of transforming the data into the frequency domain and back again. This gives  $O(2 * N^2 * \log(N^2))$  operations for a field of  $N \cdot N$  pixels, of course without the actual computation of the values. This would add at least  $O(N^2)$  operations as every element has to be taken and altered during the computation. In contrast, the proposed methods evaluated in this thesis need  $O(4 * N^2)$  operations. As seen in section 3.6 on page 40, transforming an image of size  $1920 \cdot 1080$  needs 5 ms (CUFFT), 9 ms (FlexDFT) or 10 ms (fftw), respectively - in one direction, so these values have to be doubled for the complete transform. According to the timing values provided by Sony Deutschland, the CPU version only needs around 0.5 ms for the whole cleaning of the data field (see Figure 4.9, page 40). This big performance difference is the reason why a possible Fourier approach was not investigated in this thesis.

Additionally, as *OpenCL* and *DirectCompute* are available now, their queueing of kernels and the scheduling might be more suitable for this type of problem.

It will be very interesting to see, how Larrabee performs in this section. As it has an equal architecture like the CPU from the Sony reference implementation above (Figure 4.9), it might give an significant improvement due to the much higher number of parallel nodes. Of course, this depends on the scalability of the algorithm and the tuning of the parameters.

### 4.2.3 Conclusions

This chapter was dedicated to a special part of a motion estimator, the cleaning part for eliminating holes and smoothing the vector field which identifies the motion of sub blocks from one frame to the next one.

The simple algorithm for doing the job was taken and tested on a CUDA enabled device to use its parallel structures and the idea was to get a speedup in comparison to the CPU version. It has come up that the algorithm and many ideas to speed it up do not work or are totally inefficient on the current CUDA devices. The approaches tried here are partly incompatible to CUDA and its hardware design. The algorithms that can work correctly on the CUDA device due to their parameterization perform not fast enough. Maybe, due to other architectures, new compilers or other improvements, the cleaning part will work better in the future.

What can be tried to speed up the cleaning for now is either to break the dependency in one frame and therefore use different directions of how to process this and the following

ones. Another possibility is to process multiple frames in one run to avoid overhead of idling threads. Alternatively, completely different approaches using OpenCL or DirectCompute can be tried, too.

It will be interesting to see how Larrabee performs on this using its own x86 architecture. The chances are good that it will leave CUDA far behind. But this will show when Larrabee is available and then compared to the up to date CUDA cards.

# 5 Cross Platform Computation Abstraction Layer

The following chapter contains the description of the third part of the diploma thesis, an abstraction layer for many core architectures. After introducing the specifics of the systems (see section 5.1) and the motivation (section 5.2) for developing the software, a selection of previous work and existing libraries in this field (section 5.3) is presented. Following a detailed description of the system (section 5.4) and two examples (section 5.5), the ideas for future work (section 5.6) and conclusion (section 5.7) close this chapter.

## 5.1 Introduction

Programming many core systems which might contain different architectures is non-trivial at all. Every system, let it be the standard CPU, NVIDIA GPU or others, has its own capabilities, API and maybe even different compilers. So a todays standard PC often has a multi core CPU and GPU. Using both processing units for the computation often gives a significant improvement of the performance. But on the other hand, the team play of them needs to be programmed by hand first.

The last task of this diploma thesis was to develop a prototype for easing the development of many node systems. The ambition is to present a system which handles and hides at most all the specific implementation details. At first, the design of *Cross Platform Computation Abstraction Layer* (CROCAL) will be discussed in theory. In the implementation developed in this diploma thesis, the *libcrocal*, the focus lies on a simple interface which abstracts the basic functions needed: A memory interface and calling the kernel functions that do the work. The other important feature of the system is the distribution of a given problem.

The abstracted hardware types provided by the current version are CPU and CUDA abstractions. An important point when developing the framework was to keep it flexible and extendable. It is designed to make it easy to include new hardware types. The problem specific code can be kept as is and only some details would be necessary to change when a new type is added and shall be used.

However, using the flexible and extendable abstraction layer has a drawback, too. Providing a very similar interface for all supported types reduces the detail and hardware specific parameters to tune the code for a specific product. Packaging all together reduces the functionality provided by CROCAL down to the least common denominator of all included

parts. Currently, what is provided by the library is a simple memory management for loading data from and to the device and an abstraction for calling simple kernels.

The implementation is written C++ and uses on its part the SDKs provided by NVIDIA for implementing the CUDA specific code. The host/CPU specific code is written using the standard C library. For CPU code, parallelizing the kernel code can be done by OpenMP if required. Using these simple mechanisms, CROCAL is supposed to compile & work with standard C/C++ compilers like the GNU Compiler Collection or the Microsoft C/C++ Compiler.

For clarification, a short explanation of some notations of the following sections are given below:

**Server:** The main thread coordinating the work. In the whole instance, there is only one CROCAL-server active.

**Client:** The computational thread which does the actual work.

**Scheduler:** A algorithm managing and ordering the execution of the given tasks. It is started by the server and running in its thread.

**Task:** The problem is divided into tasks by the programmer beforehand and then distributed to the clients by the scheduler.

**Node:** The underlying computational device of a client.

**Kernel:** The code which is executed and parallelized by CROCAL. It is the main part of a task.

**Thread:** The smallest executing unit on the hardware.

The current system is designed to run on one machine. The differentiation between the server and client is only given by threads running under the same process.

### 5.2 Motivation

Having different kind of hardware architectures built into a computer system, the complexity of the system, the code and the maintaining overhead rapidly grows. Usually, every hardware vendor develops its own software framework for his products which can directly access the features of it, like the NVIDIA CUDA or ATI Stream framework.

Developing software for a heterogenous system would then require to use all of the frameworks. Either, the programmer chooses to write the problem specific code for every system or develops its own abstraction layer. This is where CROCAL comes into play and offers a simple system and interface to handle multiple different APIs. Having the basic hardware layer abstracted, the next step is to combine the multiple nodes and make them transparent to the programmer.

In addition to this, the extensibility of the framework is what can keep the work for updating the code to a new platform at a minimum. Ideally, the interface of libcrocal does not change, so the problem specific code can be kept. Adding a new architecture to the system would only require a few minor changes, e.g. setting the right flags.

Altogether, the motivation for this part of the diploma thesis is to simplify the development for many node architectures by providing an C/C++ interface.

### 5.3 Existing Libraries and Frameworks

Using not only one processor but many computers in a cluster to solve a problem is not new [Par]. From simple multi processor numerical calculations to the famous SETI@home project [ACK<sup>+</sup>02] researchers make use of a many node architecture. Those systems target multiple computers connected over a network. But nowadays, the multi core main processor and the general purpose co-processor form a small “cluster”, too. Making it easy for the programmer to access the power of the whole machine is the goal of frameworks and libraries developed by various companies. The current developments in this sector are mentioned in the next sections. The presented systems target heterogeneous platforms which are capable of compiling code for different architectures like x86, NVIDIA or AMD GPUs and so on.

#### 5.3.1 Close to the Metal: OpenCL & DirectCompute

Two of the APIs available for many node computing are OpenCL [Gro08], developed by Khronos Group, and Microsoft’s DirectCompute [MSDC]. Both of these are connected to their graphics relatives OpenGL and DirectX, respectively.

##### OpenCL

The Khronos Group standardized OpenCL for using this API as an open standard for simplifying the programming of parallel computing. The “Open Computation Language” itself is a subset of the C standard to write kernels. These kernels are compiled at runtime for the underlying hardware. OpenCL is organized in a host-device structure. The host compiles the kernels, puts it into queues and then those queues are distributed to the device/devices. Here, all the platform specific code is handled by OpenCL so the programmer can concentrate on writing his code.

A closer look on the OpenCL API reveals a close relationship to CUDA. It uses similar functions but in a more general way as there are more platforms supported. Limitations like non-recursive functions or the lack of function pointers for very dynamical programs are the same like in CUDA. At the end of this thesis, the first certificated drivers became public.

### DirectCompute

Microsoft's DirectCompute is the pendant to OpenCL as DirectX is to OpenGL. It is designed to preliminarily integrate seamlessly into the rendering process of a Direct3D application. The idea behind DirectCompute is to work with so-called *Compute Shader* which allow a more general computation instead of the standard pixel or vertex shader. It is a further development of Microsoft's HLSL. It is developed to make use of DirectX hardware in modern PCs and use their computational power, especially in Windows 7.

DirectCompute runs on every graphics hardware supporting DirectX10, but some specific features are available from DirectX11 on. This shows how generalized and flexible the previous DirectX10 hardware is.

Interestingly, Microsoft mentions the Fast Fourier transform as an example application of DirectCompute. In a presentation, they claim to reach 100 GFlops in the transform and say newest hardware needs 3 ms for the transform of a  $1024 \times 1024$  image (For comparison: *CUFFT, in-place*: 1.1 ms *FlexDFT, no-copy, in-place*: 5.3 ms *libfftw3*: 4.0 ms, see 3.6.5 on page 42).

### 5.3.2 Commercial Frameworks: Peakstream & RapidMind

The RapidMind [Mono8] framework and the Peakstream [PeakSt] library are commercial products which target the parallelization of heavy data computation. They are both frameworks which provide an API for a high level parallelism of a given task. A closer view is given below.

#### Peakstream

Peakstream is a commercial product derived from the Brook library by the Stanford University [BFH<sup>+</sup>04, Brook]. It was one of the first projects combining the CPU and GPUs. The concept of Peakstream is a further development of Single Instruction Single Data (SISD) and Single Instruction Multiple Data (SIMD) scheme. Using SISD, every instruction is applied to one data object. Using vector techniques, this advances to the SIMD concept where one instruction is applied to multiple data values in parallel.

The new concept of Peakstream uses kernels implementing multiple operations which operate on streams of data.

For developing with Peakstream, it provides a virtual machine which takes the function of the scheduler distributing the data. A just in time compiler runs on the virtual machine and handles the code generation for the targeted hardware.

### RapidMind

The RapidMind framework by RapidMind, recently bought by Intel, provides an C++ framework intended to simplify the process of parallelizing computations.

It provides its own classes for data types and arrays as well as an structure for the program which is later distributed.

Instead of compiling the programs, the programmer defines a program object. This concept is called Single Program Multiple Data (SPMD). The program object works on RapidMind's data types and uses its abstractions. When the source code is compiled, there is no specialized code for a given target platform created. This is not done before the code is executed for the first time. At runtime, the system can decide how to distribute the code and then the right back ends are chosen and the codes get compiled for it.

This encapsulates the machine specific details into the library and the programmer using RapidMind can concentrate on the algorithm he implements. Therefore, it gives no real control over the generated code for the targeted system it is working on.

### 5.3.3 Differences to CROCAL

The framework developed and implemented in this work is a simple prototype only. It cannot compete with the designs and implementations mentioned above.

In its current version, CROCAL targets a single computer featuring multiple nodes for computation. The code it is used with is completely generated at compile time. It is not able to rearrange its code like RapidMind's approach and is not using virtual machines for code distribution. However, although it lacks these type of features, it is designed to be extensible in any direction.

Its current strength lies in the extensibility and the flexibility of how the given tasks are dynamically divided into sub tasks at runtime so they suit the available hardware.

The next section will give a closer view on the conception of CROCAL and the implementation in libcrocal.

## 5.4 Conception of CROCAL

The following sections describe the ideas behind CROCAL. At first, some global and useful functions are explained. In the next section, the abstraction for different computation nodes is described. The focus lies on memory allocation and a transparent way of calling the actual procedure which does the computation on the nodes.

The later sections describe the next level of hierarchy. After abstracting the simple hardware API provided by the SDK, driver or operating system, the next step is to have a system which

can operate on multiple cores. To provide an abstraction for the many core architecture, a Client-Server-System was developed.

### 5.4.1 Global Functions

The implementation of the CROCAL concept provides some global functions for operating system abstraction like starting and stopping a timer or putting a thread to sleep for a specified amount of milliseconds.

But more important are the functions which automatically detect the systems hardware and spawn the maximum number of threads useful for this system: `crocalSpawnCPU` and `crocalSpawnCUDA` can both spawn CPU and CUDA threads, respectively. Their return values are a list with the new `CrocalClient` created by them. For implementing new hardware and extending the current functionality, a function pointer type is defined. Every new probe function which fits that type can be added to the list of probing functions for the automatic client spawning.

### 5.4.2 Hardware Abstraction Layer

For abstracting different computation nodes like a CPU or NVIDIA CUDA devices, the developed abstraction layer provides a C++ class called `CrocalBaseNode`. This class provides a number of pure virtual functions which define an interface. The code behind the interface is hardware / computation node specific and is intended to be implemented in a child class.

Besides the standard C++ constructor/destructor and other implementation details, the interface defines the following methods:

**malloc:** This function allocates memory of the given size on the node and returns the pointer. The pointer does not necessarily point to a location in memory but can also contain a structure with further information regarding the allocated memory.

**free:** Allocated memory on the node is freed by this procedure.

**transferToNode, transferFromNode:** These functions are used to transfer memory from the host to the node and vice versa. A flag can be set whether the destination memory is already allocated.

**transferOnNode:** Copying data on the node from one location to another is managed by this method.

Now that the device is initialized and the data is transferred to allocated memory, the function call of the working method is the last step for abstraction. The function to be called has a standardized parameter interface:

```
unsigned int crocalThreadId, unsigned int crocalNumThreads, void *p, size_t
        sizeofP
```

The arguments introduced are explained in the following:

**crocalThreadId:** The id of the current thread. For multithreaded code using OpenMP this would refer to `omp_get_thread_num()` and on CUDA the equivalent constant is `threadIdx.x`.

**crocalNumThreads:** The total number of threads working. A single thread has  $\frac{1}{\text{crocalNumThreads}}$  of the total work to do. Here, the OpenMP function would be `omp_get_num_threads()` and the CUDA constant is `blockDim.x`.

**p:** `p` hides the user defined arguments which need to be passed to the function. This can basically be everything the compiler accepts as a `void *` and is totally unknown to the abstraction layer.

**sizeofP:** This parameter holds the size of the structure hidden by `p` above which for example is useful to send the size of an array.

For abstracting the different identifiers for the current thread and all threads running (including the blocks on CUDA), the two variables `crocalThreadId` and `crocalNumThreads` are used.

CROCAL provides some preprocessor macros to ensure the correct interface of the function and unify the call of this function. Basically, the programmer writes his kernel code once using the provided interfaces and macros. Depending on the pre compiler defines set up at compile time, the generated code contains either only the selected node settings and kernels, e.g. the CUDA part or all functions. The macros ensure that the names of the functions are unique so that a CPU kernel can be compiled together with a CUDA kernel in one file. Having all options available, it is left to the user which kernel to call in the code or it can be decided at compile time. However, the kernel code only has to be maintained once but is compiled as often as needed. This reduces the overhead for the programmer.

Currently, there are two “nodes” implemented:

**CPU:** The CPU macros define a simple for loop which iterates over all “threads”. This will execute the code in serial order. As the macro evaluates to a for loop in its first line, it is possible to parallelize the loop with OpenMP to introduce real parallelism.

**CUDA:** The NVIDIA CUDA SDK brings a special compiler for CUDA kernels. This means, the kernel itself and the calling function must be compiled by the `nvcc`. The CROCAL threads map to the CUDA thread and block scheme. The macros automatically spawn as many blocks as necessary to run all requested threads.

For both systems, a `CrocalBaseNode` derivate exists which provides the memory interface for this type of device. In the appendix, chapter A shows a simple demonstration of the explained functions and macros. Note, that `CrocalBaseNode` is only the generic interface. Its instance defines the behavior of the function.

### 5.4.3 Client Server System

The next level of abstraction is to have a system which consists of multiple computation units. That units do not necessarily have to be of the same architecture. So, the previously introduced hardware abstraction layer can be used and on top of that a new layer is introduced for managing the nodes available.

This version of CROCAL focuses on CPU and CUDA nodes, so only these both modes are supported yet. In addition, the physical hardware has to be one computer, so any computer with at least one CPU and/or one CUDA enabled graphics card can be used with this version. In section 5.6 a concept for extending this limit will be introduced.

The idea behind the client server system is to have a server which controls everything in the system. It uses a scheduler to distribute tasks to its clients and controls the data pool with the data objects the clients operate on.

As the current system is a single computer solution the server and its clients are implemented as threads using the *boost* library [Kar05, boost]. *boost* implements platform independent mechanisms like the thread concept including mutexes and other extensions for C++.

On page 74 Figure 5.1 shows a schematic diagram of the following description.

### 5.4.4 Server

The server in CROCAL is a simple class which at first “collects” all its clients and initializes the job the clients are working on. It starts the scheduler and the whole complex work is done by it.

In addition, the server handles the data objects in the *data pool* (see below). Every memory block which is shared by the clients should be acquired via the server to ensure a consistent, thus simple, memory management.

#### Scheduler

In the current version of CROCAL a very simple scheduler is implemented running in the thread-context of the server itself. It has a *waiting list* of unfinished tasks, picks one task from that list and tries to find a matching client which is idle. When a client is found, the task is assigned and removed from the waiting list. If there is no fitting client, it is idle, too, and waits for the next iteration to match an idle client with a task.

When a client has finished its task, the scheduler adds the tasks which depend on the finished task to the waiting list. When the last task of a job is done, the scheduler is finished and returns to the server.

## Data Pool

The data pool contains all allocated data objects. It ensures thread safety for requesting and releasing memory. As the clients usually need an array or similar to work on, they request it from the servers data pool. Working on one single machine in one process makes it possible to only use pointers for sharing data. By this, the data is directly altered by the clients without the need of copying data from the data pool to the clients. As the server manages the whole process, it also needs control the data objects and need the memory where they do exist. The data is typically allocated when setting up the task graph and can be released explicitly by the clients *work* function when it is no longer needed.

### 5.4.5 Client

The working units of the system are the so called *clients*. As mentioned before, the clients are implemented as threads. Ideally, the number of clients is oriented on the specification of the underlying hardware. CROCAL is able to detect the current hardware and “spawn” the maximum useful number of clients. For a quad core system with a CUDA enabled graphics card the system spawns one server and three clients, each being a thread. This number is oriented on the number of physical processors: One hardware thread is left to the server, one hardware thread controls the CUDA device and the other two hardware threads are left to two separate CPU clients. This is the default behavior of CROCAL but the user is free to change this as the clients are created by a single function with a fixed public interface.

In contrast to a *node*, introduced in section 5.4.2 above, a client is the logical unit that works on a piece of code. The node is a more hardware specific interface.

The client contains a flag, which computational unit it is assigned to (e.g. CPU or CUDA device). This is how the scheduler decides if a client is suitable for a given task.

### 5.4.6 Job

The actual problem which is given to the system to compute or solve is the so called *job*. A job usually can be split into several sub tasks. These tasks are explained below in their own section. For having a working Job, there has to be at least on task. From a wide view, the tasks are intended to form a directed graph. The scheduler gets a defined set of entry points (can be one or more) and a defined finishing task. Once the finishing task is completed, the whole job is done.

The setup of the task is user/problem specific. CROCAL provides a virtual base class but the whole construction of the task graph and the implementation of the kernels is left to the programmer. This guarantees a wide range of possible implementations, optimizations and flexibility.

### Kernel

A kernel is the code which is executed on the clients and doing all the computational work. It has a fixed interface for the users arguments and a return code similar to the `CROCAL_KERNEL` from section 5.4.2 above.

Picking up the comparison of the client and the node from above, the client executes the kernel. The node is one of the kernel parameters defining the way the client is operating, e.g. where to copy the data to. The node could be passed as an argument to the kernel function and then could be used in there. As it makes no sense to mix up a CUDA kernel with a CPU-only client, the creation of the node is done beforehand. But to have a simple kernel code, the code is written once and the decision which client running underneath is made by the framework.

### Task

The task is a logical unit within the problem definition. Usually, when the job is split into several tasks, they do have a specific order, e.g. setup the data, process the data using different methods and finally produce the result. To keep the order, the tasks are implemented as nodes in a cycle-free graph having preceding and/or succeeding tasks. A task can be processed when all its prerequisites are completed. When a task is finished, its own successors are added to the list of the scheduler when all of their prerequisites are fulfilled.

As the list of following tasks can have many elements which are depending on the currently finished task, the task pool gets filled and hopefully all the clients can work in parallel and max out the system.

In addition to the simple `CrocalTask` which consists of one function and its parameters only, there are two special kinds:

**CrocalTaskDynamic:** This type of task gets an additional function as an input argument which is able to split the current task into several small ones. Imagine the task is to transform an image from spatial into frequency domain. As seen in chapter 3 this can be a memory consuming job, maybe even to big for the CUDA device it is supposed to run on.

The `CrocalTaskDynamic` uses its *partition function* to spawn several new tasks that all work on a partition of the original problem. How the problem is partitioned is problem dependent and again left to the programmer.

All spawned child tasks are automatically linked into the task graph as a successor of their parents graph. The normal succeeding task of the `CrocalTaskDynamic` becomes the succeder of the new children's tasks. Of course, all of the children's tasks have to be finished before the next task can start to meet all dependencies.

**CrocalTaskSwitch:** Having a heterogeneous system, it cannot be known in advance which capabilities a client will have when the task is created. `CrocalTaskSwitch` accepts a number of different tasks and decides at runtime, when the scheduler tries to pair a client with a task, which of the tasks is executed. In the best case, the kernel function is written once and the only difference between the tasks given to `CrocalTaskSwitch` is the node object. The kernel then can decide via a simple switch which `CROCAL_KERNEL` call to make.

Using these three types of tasks, a powerful arrangement can be set up to solve a given problem.

For identifying a task in the code of its working function, every task has an internal identifier. Additionally, the kernel parameters are set individually, so the programmer can add its own identification scheme there, too.

Figure 5.1 shows the flow of CROCAL. The server receives the registration of the clients, initializes the job to be done and then the scheduler takes over. It works its way through the task graph by matching idle clients and unfinished tasks. When the job is finished, everything is cleaned up and the computation is finished. In the appendix section B a code example for the user defined functions are shown.

## 5.5 Examples for using CROCAL & Evaluation Results

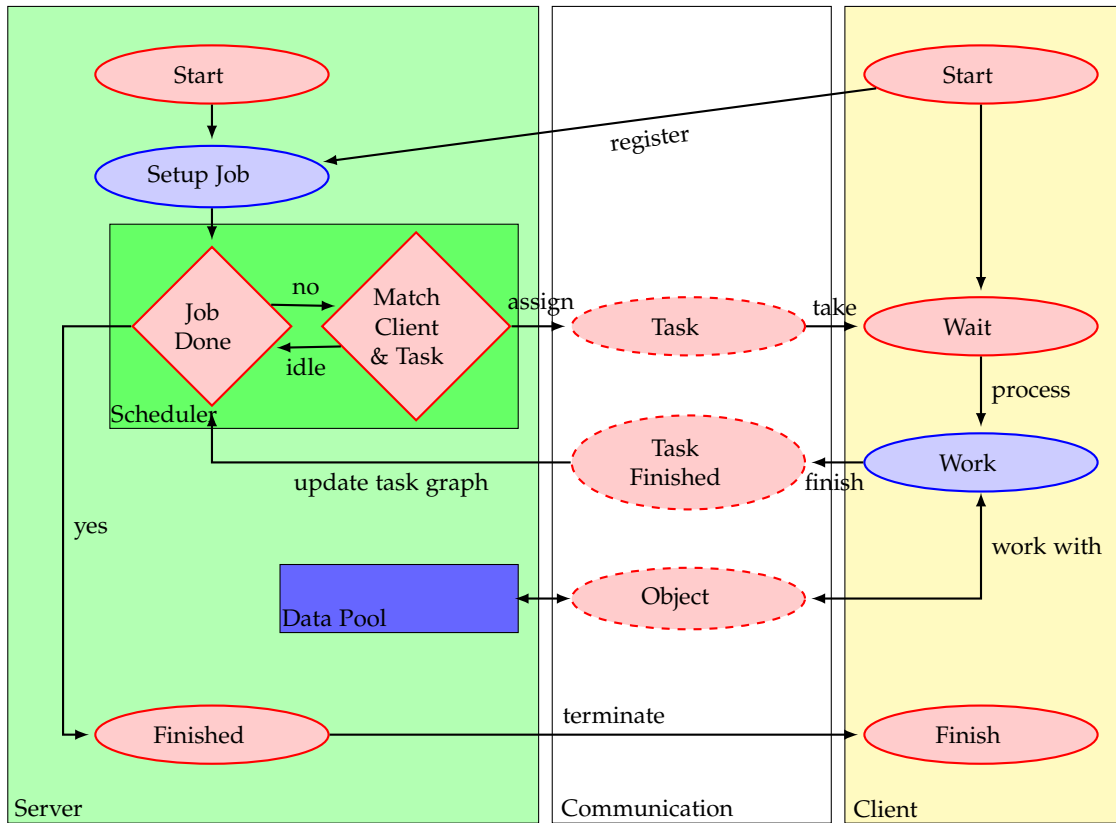
This section will show two short examples for using the `libcroc`. The first example will simply print ASCII pyramids to the screen/console with every character being printed by its own task. The second example will compute the Fourier transformation like the scheme introduced in chapter 3. Both examples will make use of the parallel and hardware independent schemes supported by CROCAL.

### 5.5.1 ASCII Pyramids

The pyramid example uses simple CPU only `CrocalTasks` connected to each other to print the drawing to the screen. The example is intended to show a simple problem and how to parallelize it with CROCAL. For an example of such a pyramid, see Figure 5.2. The setup function adds one task for every character printed to the screen. Every equal consecutive character (the `□` or `_`) can be printed in parallel. A group of these parallel task is "caught" by the next unique character which on his side spawns the next group of parallel tasks.

Figure 5.3 below shows a section of the task graph. The last sequence of the previous line printing `\ + \n` spawns the parallel tasks of the next line which print the `□`. They on their hand spawn the `/` and so on.

The schematic flow chart in Figure 5.4 shows an evaluation of `libcroc` versus the pure C implementation of the pyramid printout. The displayed values are the average of 10 iterations



**Figure 5.1:** Design Diagram of CROCAL. The blue circles are the job dependent parts implemented by the user. The colored rectangles and their layout symbolize the design of the components.

to eliminate side effects and system stalls caused by other processes. The curve describing the result of the C-only implementation is the fastest as there is no task management included at all. The other curves show a noticeable difference. This is because of the different sleep interval which was set at compile time. Idling, e.g. when a client does not have a task assigned or the scheduler has no idling client or available task in its pools, the system calls the operating systems sleep routine to grant other processes/threads access to the CPU.

The polynomial behavior of the graph comes from the number of tasks in a pyramid. A pyramid of height  $N$  has  $N^2 + \frac{N^2 + N}{2}$  tasks. Table 5.1 shows the numbers of the tasks for better understanding.

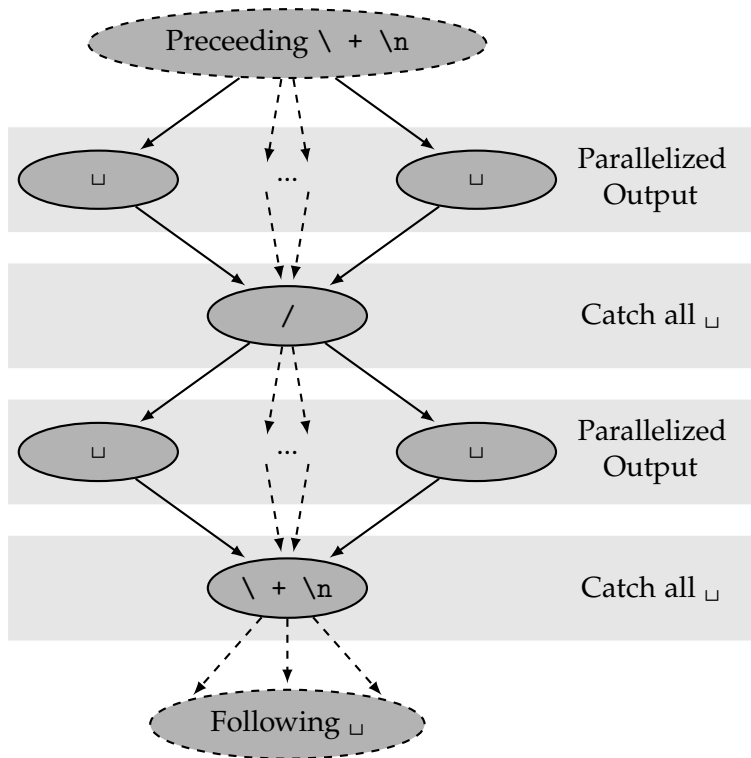
A longer sleeping time causes longer runtimes of the whole execution. Having short sleep times results in an overhead and wasted CPU power as the client still does not have a task or the scheduler is still unable to assign one. On the other hand, clients might be idle to long, when the scheduler assigned a task to them but they are still asleep. In the last case, the

system has a lower need of CPU power but a longer runtime. The tradeoff between time and intensity of the computation is to be made here.

```

uuuuu/\
uuuu/uu\
uuu/uuuu\
uu/uuuuuu\
u/uuuuuuuu\
/-----\
    
```

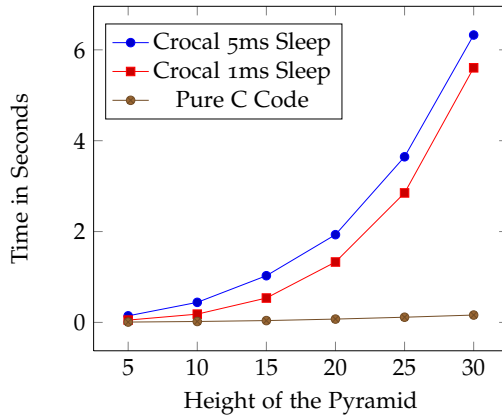
**Figure 5.2:** 6 Level ASCII Pyramid



**Figure 5.3:** Task Graph Excerpt displaying one line of the pyramid.

### 5.5.2 Flexible Discrete Fourier Transformation

The second example is taken from chapter 3 and implements the idea of FlexDFT. The four steps to be done are the transformation of the two dimensions and two transpositions between and after them. Additionally, a comparing step was added at the end to verify



Height	Total Number of Tasks
5	30
10	155
15	345
20	610
25	950
30	1365

**Figure 5.4:** Performance Evaluation, *libcrocal* vs. pure C **Table 5.1:** Pyramid Height and Task Count

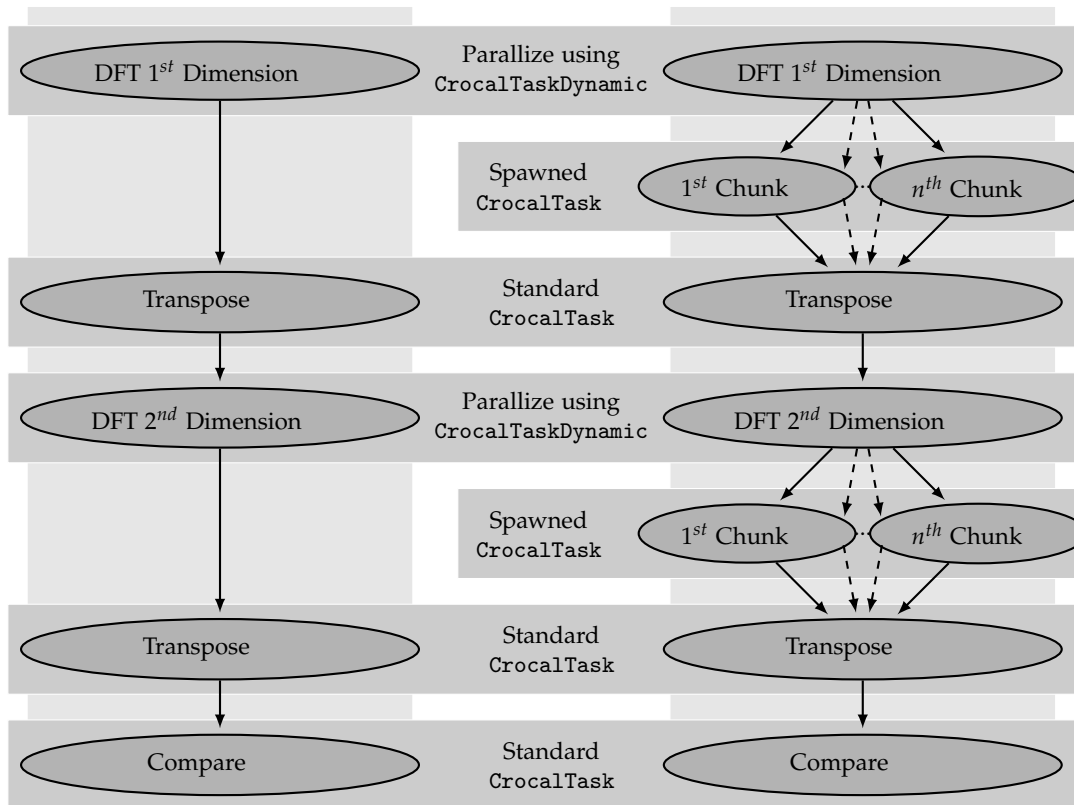
the computation. Putting these five steps in a row creates the task graph of this example. This is shown on the left side of Figure 5.5. The transforms are implemented using the `CrocalTaskDynamic` to dynamically create children tasks and split up the transformation for parallel computing. Additionally, one task transforming a large array might be too much for the CUDA device built in.

The right side of Figure 5.5 shows the task graph after the child tasks are spawned. The *transpose* tasks now “catch” those tasks as all of their output is needed for the next step. The partition one task is working on depends on the smallest amount of memory available on the participating nodes and the partitions are equally distributed to the tasks.

The graphs displayed in Figure 5.6 and 5.7 show the comparison of different types and numbers of threads and also different amounts of memory used. The setup of a bar is displayed in the legend of the graphs. The task was to transform a square of floating point values of the given size into the frequency domain and at the end compare the output with a reference. The last comparison task is one single task and thus adds a constant amount to the bars of each size and does not influence the relativity of the values.

What can be learned from graph 5.6 is that parallelization only becomes useful from a bigger amount of work on. For sizes of up to  $1024 \times 1024$  the 3 measured combinations are nearly equal, one thread is even faster as there is far less overhead for the scheduler to do and no memory transfer from host to GPU. For larger sizes, the parallelization comes into play and significantly improves the performance compared to the single threaded version having only one working client. The difference between 1 GByte and 128 MBytes is also visible. The smaller amount of memory requires more threads to be spawned and to complete them takes more time.

Figure 5.7 has the focus on the memory available for the computation and thus the task created by `CrocalTaskDynamic`. Here again, at a size length of 1024 elements the overhead of the scheduler can be seen when it has more than one client to handle. For lower sizes



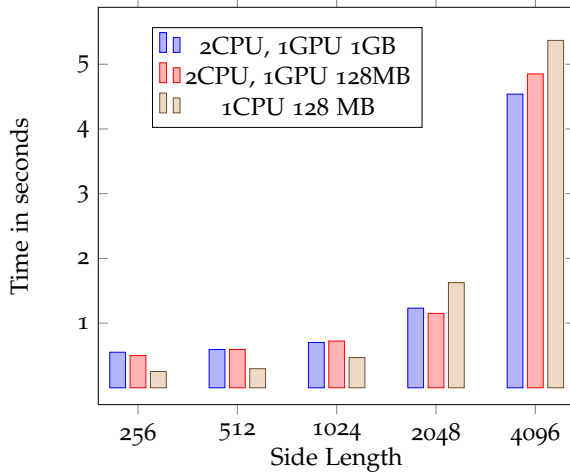
**Figure 5.5:** Task graph of the DFT example. The left part shows the setup defined by the user and the right side shows the expanded task graph.

all values are nearly equal, again. So the difference seen in Figure 5.6 for  $256^2$  and  $512^2$  elements might be a result of the transfers to and from the GPU when CUFFT was used.

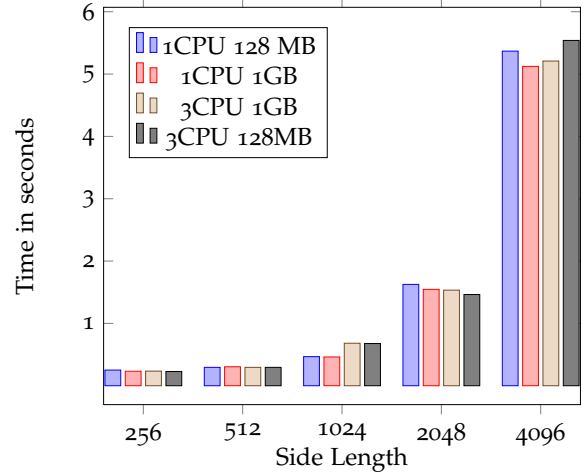
From size lengths of 2048 and higher two effects can be seen: First, the single threaded version begins to work slower as the parallelization in the 3 CPU version comes into play. From 4096 on the second effect, that the versions having more memory and thus less tasks are ahead of the others.

The partition function splits the transformations up into independent parts. For 128 MBytes the 4096 square is partitioned into chunks of 460 lines. Figure 5.8 shows the output of the example with the unordered sequence in each transformation.

In the appendix B there are some extracts from the actual code of how this example is realized using libcrocal. The description there gives a more detailed view on how to create tasks, connect them in the task graph and set up CROCAL.



**Figure 5.6:** Performance Evaluation, Comparison of parallelized Computations



**Figure 5.7:** Performance Evaluation, Comparison of different Thread Count and available Memory.

```

r2c - offset-lane: 0 lane-count: 460
r2c - offset-lane: 920 lane-count: 460
r2c - offset-lane: 460 lane-count: 460
r2c - offset-lane: 2300 lane-count: 460
r2c - offset-lane: 1380 lane-count: 460
r2c - offset-lane: 1840 lane-count: 460
r2c - offset-lane: 3220 lane-count: 460
r2c - offset-lane: 2760 lane-count: 460
r2c - offset-lane: 3680 lane-count: 416
c2c - offset-lane: 0 lane-count: 460
c2c - offset-lane: 920 lane-count: 460
c2c - offset-lane: 460 lane-count: 460
c2c - offset-lane: 1380 lane-count: 460
c2c - offset-lane: 1840 lane-count: 209
    
```

**Figure 5.8:** Output of the transformation functions of the example. *r2c* is Real To Complex, *c2c* Complex to Complex Transformation.

## 5.6 Future Work

As mentioned before, the abstraction framework developed in this diploma thesis has some limitations. As it is only implemented in as a prototype for the given design, it could be

extended in many ways. The structural design was kept modular so that single parts can be updated without the need of changing other parts of the system.

**Scheduler:** The current scheduler implements a simple queue for its tasks. A more sophisticated scheduling system can help to improve the performance [Pino8]. Adding more information to the tasks, like e.g. estimated runtime or memory access on the data objects, can help to create a more intelligent scheduler which finds better matchings of clients and tasks. It could defer tasks if there is a client expected to be free soon which has a much better performance on the task than any free client at this time might have.

**Multi Computer Systems:** Working on one computer, CROCAL implements all basic features for having the server and its clients run on the same machine. The next step is to expand this limit by adding a inter computer connection system like TCP or similar to the system. The server would still run on a single thread but listen on ports for connections. The scheduler sends and receives the tasks via TCP instead of simple function calls and a client still “orders” the required data objects.

**Communicating Clients:** In this context, the system could benefit from clients communicating with each other. Especially memory transfers could be simplified for consecutive tasks. The first tasks result can be sent to another tasks client directly instead of detouring to the server and back.

**Memory Management:** A more problem oriented memory management for the nodes can help to prevent the storage of doubled data. Knowing that the input is no longer needed by any other client makes it possible to overwrite it and thus save memory.

The implemented memory manager can allocate and deallocate memory on the server side and provide this to clients. This is sufficient for single computers which use CROCAL as all the memory resides there. For more complex solutions a more sophisticated memory management can help to keep the data overhead low. Especially for network transfers, having a good memory manager which can keep track of the data objects, the clients which need them etc, the overhead of sending data could be reduced.

**Other Architectures:** Using the well defined class hierarchy of CROCAL, it can easily be extended for the use with other techniques like ATI Stream [ATI], Intel Larrabee or other emerging systems on the market.

Following these ideas, CROCAL can become a powerful framework for larger scale computations. Of course, this will rise the complexity of the project. But in contrast, the system gets more flexible and the overall performance can increase compared to the simple approach in this version.

For integrating Intel’s Larrabee architecture into the libcrocal, the following needs to be done:

A class derived from `CrocalBaseNode` has to implement all necessary functions, see section 5.4.2 for details. Additionally, the macros used for defining the kernel prototype and calling the kernel have to be adjusted to the calling conventions of Larrabee’s API. Having done

this, Larrabee can be used in the code. See the code sample in Listing B.3 in the appendix B to get an idea.

### 5.7 Summary

Designing and implementing a prototype of a many node computation abstraction layer was described in this chapter. The focus of designing it was to create an abstraction layer for many node systems having different architectures built into one PC system. The extensibility of the presented system was another focus so it can keep up with further developments on the market, for example adding a sub system for Larrabee when it is available.

The basic layer was intended to use for simplifying the programming of the hardware. The functions are reduced to the common functions of all participating architectures. So for example, there is no recursion available for the systems when CUDA is involved.

On top of this system built very near to the actual hardware comes a task distribution system. A task graph with the dependencies of the graphs are given to the system which then distributes them over the available hardware. Here, the interfaces for the components are designed for later extensibility, too. The system could be upgraded for usage of multiple computers.

Using two examples, the system was demonstrated with the current implementation. Especially the FlexDFT approach developed in chapter 3 is a good example as it can be separated in sub tasks. The tasks on their hand can be parallelized to make use of the multiple nodes available on a system starting with a multi core processor.

Of course, the problem to solve has to have an appropriate size for such a project to justify the overhead. But even small projects can benefit from CROCAL in the meaning that the complexity for programming and maintaining the system is lower.

So using this approach can combine the parallel power of the GPU with the CPU's flexible design. Having these both components working together, and maybe more components in a further step, and an efficient task setup will improve the usability and performance as their combination will show the advantage of all of their power.

## 6 Summary

This diploma thesis had three large topics which all have been accurately discussed in the corresponding chapters above.

The memory problem from using NVIDIA's CUFFT library on a CUDA device was solved by splitting up the data which have to be transformed into the frequency domain. The developed FlexDFT approach can reach a similar performance compared to the original CUFFT. Using this method, the memory does not play a role for transforming data using the Fourier transform anymore. Instead, for large transforms, the time needed for the transformation raises as a large number of memory copies from the host to the device will be necessary. This is the typical speed vs. memory tradeoff which can be seen all over the computer science. Additionally, the FlexDFT and the results having been evaluated were presented on the "GPU Technology Conference 2009" in San Jose, USA.

In contrary to the first topic, the motion estimator cleaning part has shown to not be able to be implemented efficiently on a CUDA device. Having a strict automated scheduler on the CUDA device makes it impossible to reach the performance promised by NVIDIA in means of computational power. The inflexible programming model of CUDA is unable to suit the data dependencies which were necessary here. The standard CPU approach is much faster and thus CUDA is no competitor on this sector.

The last topic was the design and implementation of an abstraction layer for many node computation devices. The base layer abstracts the programming of the actual hardware, in the current version CPU and CUDA. It is possible to write the code once and let it run on both hardware architectures but with the restriction that only the common features can be used. The computation abstraction layer was designed using a client - server structure with the server being the head of operations. The clients do the actual work, e.g. the parallelized fourier transform of an image using the FlexDFT approach developed in this thesis.

All those topics have been discussed regarding Intel's Larrabee architecture as well. As there is no further information or hardware available yet, only theoretical views of how the FlexDFT, the motion estimator cleaning part or the sub system for libcrocal could be implemented were given. As Larrabee has a similar structure and architectures compared to todays CPUs, it can be expected that those topics will work very well.

The task of the thesis was to investigate the heavy memory usage for Fourier transforms and regarding this, the developed approach is successful, in both low memory consumption and performance. However, the recursive pixel dependency problem does not perform good on CUDA. Especially for complex dependencies like the motion estimator problem has, the

current CUDA architecture is not flexible enough. On the other hand is the speed of today's graphics cards enormous, when their parallelism is used effectively.

The last topic combines the flexibility of the CPU and the power of the GPU and shows that both parts can complement one another.

The contributions of this work are the following:

At first, an approach for having a memory saving Fourier transform available on CUDA devices has been developed. Although the underlying algorithm has a large memory consumption, the new method can handle the same input sizes on less memory.

The second achievement is the design of a hardware abstraction layer including a task distribution system for many node computing. Using the prototypical implementation reduces the overhead for using the available hardware in the system and adapting the problem to it.

## A How to use *libcroc*al for Hardware Abstraction

The following listings will demonstrate the usage of *libcroc*al for abstracting specific hardware code by means of the important excerpts from the source code files. As mentioned in the corresponding chapter 5, the code which can be written may use the common features of all participating hardware architectures. In this example, CUDA and x86 are possible. When the example is compiled with the precompiler flag `CROCAL_USE_CUDA` defined, the CUDA binary is generated. Without the define, the CPU version is compiled. The code shown here does not include the full capability of CROCAL as its purpose is to demonstrate the hardware abstraction level only. For an example showing the full feature set of CROCAL including the task splitting and parallelization, see appendix B later on.

Listing A.1 shows the main function called by the operating system. It sets up a node, initializes it and then calls the `start_demo` function. This function is implemented in the second Listing A.2 which also holds the actual kernel, here called `demo`.

The CPU and CUDA specifics are hidden in the macros beginning with `CROCAL_`. As the CUDA compiler requires its CUDA specifics to be in a file with the `.cu` ending, the third Listing A.3 shows this for the sake of completeness.

When compiled with `CROCAL_USE_CUDA` defined, the compiler will generate a function prefixed `__crocal_cuda_` and otherwise a function with prefix `__crocal_x86_`. Using different files which do and do not define the `CROCAL_USE_CUDA` macro allows to generate files which include both, the CPU and CUDA version. The function which is then called depends on the definition of `CROCAL_KERNEL_CALL`. Of course, the functions can be called individually, if defined.

### Listing A.1 Main Function

---

```
// main function
int main(int argc, char* argv[])
{
    /*! hardware abstraction */
    #if defined CROCAL_USE_CUDA
        CrocalNodeCUDA *C = new CrocalNodeCUDA();
    #else
        CrocalNodeCPU *C = new CrocalNodeCPU();
    #endif // defined CROCAL_USE_CUDA

    // start the demo
    if(C->init() == CROCAL_OK)
        start_demo(C);

    // cleanup
    C->destroy();
    delete C;

    return 0;
}
```

---

---

### Listing A.2 demo.cpp: The Host Side of the Program

---

```
// demo definition. simply set the input array to a thread-dependend value
CROCAL_KERNEL(demo,
{
    ((unsigned int*) p)[crocalThreadId] = crocalThreadId*1000 + crocalNumThreads;
})

// host-side function doing the setup of data and cleanup afterwards
crocalStatus start_demo(CrocalNodeBase *C)
{
    unsigned int uiArraySize = 512;
    unsigned int uiNumThreads = uiArraySize;
    unsigned int *aiDemoArrayHost = NULL,
    *aiDemoArrayNode = NULL;

    // allocate the data and copy the (internally zeroed) array to the device
    Crocal::malloc((void**)&aiDemoArrayHost, sizeof(unsigned int) * uiArraySize);
    C->transferToNode((void**)&aiDemoArrayNode, aiDemoArrayHost, sizeof(unsigned int) *
        uiArraySize);

    // call the kernel "demo"
    CROCAL_KERNEL_CALL(uiNumThreads, demo, aiDemoArrayNode, uiArraySize);

    // copy the data back
    C->transferFromNode((void**)&aiDemoArrayHost, aiDemoArrayNode, sizeof(unsigned int) *
        uiArraySize, CROCAL_ALLOCATE_ALREADYALLOCATED);

    // print the output
    for(int i = 0; i < uiArraySize; ++i)
        printf("%d\n", aiDemoArrayHost[i]);

    // cleanup
    C->free(aiDemoArrayNode);
    ::free(aiDemoArrayHost);

    return CROCAL_OK;
}
```

---

### Listing A.3 demo.cu: The CUDA File

---

```
#if defined CROCAL_USE_CUDA
#include "demo.cpp"
#endif // defined CROCAL_USE_CUDA
```

---



## B How to use libcrocal for Many Node Computation

This chapter will give a short overview of how to use *libcrocal* and its interfaces by the means of the Fourier transformation example. The idea and structure of the example is discussed in section 5.5.2.

Listing B.1 shows the part of the main function like instantiating the class objects, initializing the clients, server and cleaning every thing up when the server returns. The clients are created automatically by `crocalSpawnCPU` and `crocalSpawnCUDA` which are both part of the *libcrocal* project. The resulted `CrocalClient` list is then given to the server by initializing the clients.

## B How to use libcrocal for Many Node Computation

---

### Listing B.1 Main Function

---

```
#!/ test two: dft */
constant int size = 4096;

CrocalServer *server = new CrocalServer();
CrocalDemoJobDFT *job = new CrocalDemoJobDFT(size, size);

// automatically spawn clients
std::list<CrocalClient*> listClients;
std::list<crocalDeviceInformationProbeFunction> listProbeFunctions;
// spawn the clients by these probe functions
listProbeFunctions.push_back(&crocalSpawnCPU);
listProbeFunctions.push_back(&crocalSpawnCUDA);
Crocal::spawnClients(listProbeFunctions, &listClients);

// initialize the clients
for(std::list<CrocalClient*>::iterator itClient = listClients.begin(); itClient !=
    listClients.end(); ++itClient)
    (*itClient)->init(server);

// do the actual work
server->init(job);

// clean up
delete job;
delete server;

for(std::list<CrocalClient*>::iterator itClient = listClients.begin(); itClient !=
    listClients.end(); ++itClient)
    delete (*itClient);

listClients.clear();
```

---

---

Listing B.2 describes the constructor, destructor and the `clientWorkInfo` for the *DemoDFT* job. The structure is allocated on heap memory and given as anonymous void pointers to the work function which casts them to `clientWorkInfo` to access the members.

---

**Listing B.2** DFTJob.cpp: Constructor, Destructor and Data Types

---

```
// constructor of the class
CrocalDemoJobDFT::CrocalDemoJobDFT(unsigned int uiWidth, unsigned int uiHeight) : CrocalJob()
{
    this->uiWidth = uiWidth;
    this->uiHeight = uiHeight;
}

// destructor of the class
CrocalDemoJobDFT::~CrocalDemoJobDFT(void)
{
}

// Info for the clients. Cast the void pointer to clientWorkInfo* and they are accessible.
struct clientWorkInfo
{
    // input and output data objects
    CrocalDataObject *pDataInput,
                    *pDataOutput;
    // information about the dimensions
    unsigned int uiWidth,
                uiHeight,
                uiOffset,
                uiLaneCount;
    // node information
    CrocalNodeBase *pNode;
};
```

---

## B How to use libcrocal for Many Node Computation

---

Listing B.3 shows the `dft_r2c` function which is the main function doing the first transform. After casting the void pointer  $p$  the input and output data is requested from the objects and then the switch does its work, depending on the current node. `dft_c2c`, the transform from complex to complex values, looks exactly the same. Only the parts inside the case clauses have to be adjusted, if necessary.

---

### Listing B.3 DFTJob.cpp: DFT Real to Complex and Complex to Complex

---

```
// first transform from real to complex values
crocalStatus dft_r2c(void *p, size_t sizeOfP)
{
    if(!p || !sizeOfP)
        return CROCAL_INVALIDARGUMENT;

    // cast p to access the info
    clientWorkInfo *psWorkInfo = (clientWorkInfo*)p;

    float *afDataInput = NULL,
          *afDataOutput = NULL,
          *afDemoArrayNode = NULL;

    // get the content of the arrays
    psWorkInfo->pDataInput->getContent((void**)&afDataInput);
    psWorkInfo->pDataOutput->getContent((void**)&afDataOutput);

    // get the node info of the hardware owned by this client
    crocalNodeInformation nodeInfo;
    psWorkInfo->pNode->getNodeInformation(&nodeInfo);

    // do the actual fft
    switch(nodeInfo.computationNodeType)
    {
    case CROCAL_NODE_NVIDIA:
        // work with e.g. CUFFT
        break;
    case CROCAL_NODE_CPU:
        // work with e.g. libfftw3
        break;
    default:
        return CROCAL_INSUFFICIENT_CAPABILITIES;
    }

    return CROCAL_OK;
}

// second transform similar to dft_r2c
crocalStatus dft_c2c(void *p, size_t sizeOfP)
{
    // ...
    return CROCAL_OK;
}
```

---

---

Listing B.4 implements the transpose function which is the third of all working functions and simply transposes the input array into the output array.

---

**Listing B.4** DFTJob.cpp: Transpose

---

```
// transpose the data
crocalStatus transpose(void *p, size_t sizeOfP)
{
    if(!p || !sizeOfP)
        return CROCAL_INVALIDARGUMENT;

    // cast p to access the info
    clientWorkInfo *psWorkInfo = (clientWorkInfo*)p;

    float *afDataInput = NULL,
          *afDataOutput = NULL;

    // get the content of the arrays
    psWorkInfo->pDataInput->getContent((void**)&afDataInput);
    psWorkInfo->pDataOutput->getContent((void**)&afDataOutput);

    // iterate over every item to transpose
    for(int y = 0; y < psWorkInfo->uiHeight; ++y)
        for(int x = 0; x < psWorkInfo->uiWidth; ++x)
            {
                afDataOutput[2*psWorkInfo->uiHeight*x + 2*y + 0] =
                    afDataInput[2*psWorkInfo->uiWidth*y + 2*x + 0];
                afDataOutput[2*psWorkInfo->uiHeight*x + 2*y + 1] =
                    afDataInput[2*psWorkInfo->uiWidth*y + 2*x + 1];
            }

    // release the input as we do not need it any longer
    psWorkInfo->pDataInput->release();

    return CROCAL_OK;
}
```

---

## B How to use libcrocal for Many Node Computation

---

Listing B.5 and B.6 show the partitioning function for splitting up the big transforms into several small tasks. After getting the smallest amount of memory available on the clients in Listing B.5, Listing B.6 shows the creating of the new `crocalWorkParameters` holding the `clientWorkInfo`, its size and the node information.

---

### Listing B.5 DFTJob.cpp: Partitioning Function

---

```
// partition the dft
crocalStatus part_dft(crocalWorkParameters *psOriginalParameters,
    std::list<crocalNodeInformation> listNodeInfos, std::list<crocalWorkParameters>
    *pListWorkParameters)
{
    if(!psOriginalParameters || !listNodeInfos.size() || !pListWorkParameters)
        return CROCAL_INVALIDARGUMENT;

    // cast p to access the info
    clientWorkInfo *psWorkInfo = (clientWorkInfo*)psOriginalParameters->p;

    // look for a NVIDIA gpgpu and get its available memory as the maximum
    unsigned long long ulAvailableGPUMem = 1 << 30;
    unsigned int uiMaxWidthHeight = max(psWorkInfo->uiHeight, psWorkInfo->uiWidth);

    for(std::list<crocalNodeInformation>::iterator itInfo = listNodeInfos.begin(); itInfo !=
        listNodeInfos.end(); ++itInfo)
    {
        if((*itInfo).computationNodeType == CROCAL_NODE_NVIDIA)
            ulAvailableGPUMem = min(ulAvailableGPUMem, (*itInfo).ulMemoryAvailable);
    }

    ulAvailableGPUMem = (unsigned long long)(9*ulAvailableGPUMem / 10);
    unsigned int iTTaskCounter = 0;
```

---

---

## Listing B.6 DFTJob.cpp: Partitioning Function II

---

```
// create new working info with reduced memory
psWorkInfo->uiLaneCount = min(psWorkInfo->uiLaneCount, ulAvailableGPUMem / (64 *
    uiMaxWidthHeight));
for(unsigned int i = 0; i < psWorkInfo->uiHeight; i += psWorkInfo->uiLaneCount)
{
    clientWorkInfo *psWorkInfoSubTask = (clientWorkInfo*)malloc(sizeof(clientWorkInfo));
    *psWorkInfoSubTask = *psWorkInfo;

    psWorkInfoSubTask->uiLaneCount = ulAvailableGPUMem / (64 * uiMaxWidthHeight);
    psWorkInfoSubTask->uiOffset = i;

    if(i > psWorkInfo->uiHeight - psWorkInfo->uiLaneCount)
        psWorkInfoSubTask->uiLaneCount = psWorkInfo->uiHeight - i;

    psWorkInfoSubTask->uiLaneCount = min(psWorkInfo->uiLaneCount,
        psWorkInfoSubTask->uiLaneCount);

    crocalNodeInformation sNodeInfo = psOriginalParameters->sNodeInfoRequired;
#ifdef CROCAL_USE_CUDA
    if(iTaskCounter % listNodeInfos.size() == 0)
    {
        sNodeInfo.computationNodeType = CROCAL_NODE_NVIDIA;
        sNodeInfo.ulMemoryAvailable = ulAvailableGPUMem;

        psWorkInfoSubTask->pNode = new CrocalNodeCUDA();
    }
    else
#endif // defined CROCAL_USE_CUDA
        psWorkInfoSubTask->pNode = new CrocalNodeCPU();

    crocalWorkParameters sWorkParameters = { psWorkInfoSubTask,
        psOriginalParameters->sizeofP, sNodeInfo };
    pListWorkParameters->push_back(sWorkParameters);

    ++iTaskCounter;
}

return CROCAL_OK;
}
```

---

## B How to use libcrocal for Many Node Computation

---

Listing B.7 and B.8 are the examples setup function where the task graph is created. At the beginning, the memory objects are allocated and the input array would be filled with data. Then the `clientWorkInfo` are created and the `CrocalTask` is created. At the end, the tasks are put into a simple bidirectional linked list and the first and last task are made known to the system.

---

### Listing B.7 DFTJob.cpp: Job Setup

---

```
// setup up the job
crocalStatus CrocalDemoJobDFT::setup(CrocalServerClientInterface *pClientInterface)
{
    if(!pClientInterface)
        return CROCAL_INVALIDARGUMENT;

    // set up the data
    CrocalDataObject *pDataImage = NULL,
        *pDataImageDFT1D = NULL,
        *pDataImageDFT1DT = NULL,
        *pDataImageDFT2D = NULL,
        *pDataImageDFT2DT = NULL;

    unsigned int uiDataSize = 2*sizeof(float)*(this->uiWidth/2+1)*this->uiHeight;

    // request memory for the data
    pClientInterface->requestMemory(uiDataSize, &pDataImage);
    pClientInterface->requestMemory(uiDataSize, &pDataImageDFT1D);
    pClientInterface->requestMemory(uiDataSize, &pDataImageDFT1DT);
    pClientInterface->requestMemory(uiDataSize, &pDataImageDFT2D);
    pClientInterface->requestMemory(uiDataSize, &pDataImageDFT2DT);

    float *afData;
    pDataImage->getContent((void**)&afData);
    // setup the data in pDataImage
```

---

---

## Listing B.8 DFTJob.cpp: Job Setup II

---

```
// create the working infos for the clients
clientWorkInfo *workInfoD1 = (clientWorkInfo*)malloc(sizeof(clientWorkInfo));
workInfoD1->pDataInput = pDataImage;
workInfoD1->pDataOutput = pDataImageDFT1D;
workInfoD1->uiOffset = 0;
workInfoD1->uiHeight = this->uiHeight;
workInfoD1->uiWidth = this->uiWidth;
workInfoD1->uiLaneCount = this->uiHeight;
workInfoD1->pMutexfftwPlan = this->pMutexfftwPlan;
// note that this is a CrocalTaskDynamic
CrocalTaskDynamic *dft1d = new CrocalTaskDynamic(pClientInterface->requestUniqueId(),
    this, &dft_r2c, workInfoD1, sizeof(clientWorkInfo), &part_dft);

// set up the additional tasks:
// create workInfoTranspose
CrocalTask *dft1dtranspose = new CrocalTask(pClientInterface->requestUniqueId(), this,
    &transpose, workInfoTranspose, sizeof(clientWorkInfo));
// create workInfoD2
// note that this is a CrocalTaskDynamic, too
CrocalTaskDynamic *dft2d = new CrocalTaskDynamic(pClientInterface->requestUniqueId(),
    this, &dft_c2c, workInfoD2, sizeof(clientWorkInfo), &part_dft);
// create workInfoTranspose2
CrocalTask *dft2dtranspose = new CrocalTask(pClientInterface->requestUniqueId(), this,
    &transpose, workInfoTranspose2, sizeof(clientWorkInfo));

// build the task graph
this->listInitialTasks.push_back(dft1d);

dft1d->addDependentTask(dft1dtranspose);
dft1dtranspose->addPreconditionedTask(dft1d);

dft1dtranspose->addDependentTask(dft2d);
dft2d->addPreconditionedTask(dft1dtranspose);

dft2d->addDependentTask(dft2dtranspose);
dft2dtranspose->addPreconditionedTask(dft2d);

this->pFinishTask = dft2dtranspose;

return CROCAL_OK;
}

// clean up
crocalStatus CrocalDemoJobDFT::cleanup()
{
    // ...
    return CROCAL_OK;
}
```

---



# Bibliography

- [ACK<sup>+</sup>02] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002. doi:<http://doi.acm.org/10.1145/581571.581573>. (Cited on page 65)
- [AFG<sup>+</sup>09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. S. M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009. (Cited on page 17)
- [ATI] URL <http://www.amd.com/stream>. (Cited on pages 17 and 79)
- [BBB<sup>+</sup>08] K. Barros, R. Babich, R. Brower, M. A. Clarck, C. Rebbi. Blasting through lattice calculations using CUDA. *The XXVI International Symposium on Lattice Field Theory*, 2008. (Cited on page 13)
- [BFH<sup>+</sup>04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pp. 777–786. ACM, New York, NY, USA, 2004. doi:<http://doi.acm.org/10.1145/1186562.1015800>. (Cited on page 66)
- [BK66] R. Bracewell, P. B. Kahn. The Fourier Transform and Its Applications. *American Journal of Physics*, 34(8):712–712, 1966. doi:10.1119/1.1973431. URL <http://link.aip.org/link/?AJP/34/712/3>. (Cited on page 19)
- [BKo8] D. G. R. Bradski, A. Kaehler. *Learning opencv, 1st edition*. O'Reilly Media, Inc., 2008. (Cited on page 30)
- [boost] URL <http://www.boost.org/>. (Cited on page 70)
- [Brook] URL <http://graphics.stanford.edu/projects/brookgpu/>. (Cited on page 66)
- [Erd98] O. Erdler. *Untersuchung zur Implementierung von Algorithmen der Bewegungsschätzung auf dem Signalprozessor TriMedia*. Master's thesis, Universität Dortmund, Diplomarbeit D11-98. (Cited on page 49)
- [FCo6] O. Fialka, M. Cadik. FFT and Convolution Performance in Image Filtering on GPU. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pp. 609–614. 2006. doi:10.1109/IV.2006.53. (Cited on pages 19 and 23)
- [FFTW] URL <http://fftw.org/>. (Cited on page 29)

## Bibliography

---

- [FJ98] M. Frigo, S. Johnson. FFTW: an adaptive software architecture for the FFT. *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference*, 3:1381–1384, 1998. (Cited on page 29)
- [Flops] URL [http://techgauge.com/article/intel\\_core\\_i7\\_performance\\_preview/9](http://techgauge.com/article/intel_core_i7_performance_preview/9). (Cited on page 13)
- [Gee05] D. Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005. doi:<http://dx.doi.org/10.1109/MC.2005.160>. (Cited on page 17)
- [GLD<sup>+</sup>] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. Microsoft. (Cited on page 22)
- [gpuCV] URL [ref:http://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/](http://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/). (Cited on page 31)
- [Groo8] K. Group. OpenCL (Open Computing Language). <http://www.khronos.org/openc1/>, 2008. URL <http://www.khronos.org/openc1/>. (Cited on page 65)
- [IPP] URL <http://software.intel.com/en-us/intel-ipp/>. (Cited on page 38)
- [IPPMAN] *Intel® Integrated Performance Primitives 6.1 – Documentation*, volume 2. Intel, 2009. (Cited on page 38)
- [Karo05] B. Karlsson. *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005. (Cited on page 70)
- [LRB] URL <http://software.intel.com/en-us/articles/larrabee/>. (Cited on page 15)
- [LRBIns] URL <http://software.intel.com/sites/billboard/archive/larrabee-new-instructions.php>. (Cited on page 17)
- [MA03] K. Moreland, E. Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003. (Cited on page 23)
- [Mono8] M. Monteyne. RapidMind Multi-Core Development Platform. Technical report, RapidMind Inc, 2008. (Cited on page 66)
- [MSDC] Microsoft. Microsoft WINHEC Session GRA-T517. [http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/GRA-T517\\_WHo8.pptx](http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/GRA-T517_WHo8.pptx). (Cited on page 65)
- [NVCUB] NVIDIA. NVIDIA CUDA CUBLAS Library. (Cited on page 37)
- [NVCUD] URL [http://www.nvidia.com/object/cuda\\_sdks.html](http://www.nvidia.com/object/cuda_sdks.html). (Cited on pages 15 and 25)
- [NVCUF] NVIDIA. NVIDIA CUDA CUFFT Library. (Cited on page 31)

- [NVGeF] URL [http://www.nvidia.com/object/geforce\\_family.html](http://www.nvidia.com/object/geforce_family.html). (Cited on page 15)
- [NVMan] NVIDIA. NVIDIA CUDA Programming Guide Version 2.2.1. (Cited on pages 15 and 56)
- [NYTimes] URL <http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/>. (Cited on page 17)
- [PAB<sup>+</sup>05] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa. The design and implementation of a first-generation CELL processor. pp. 184–592 Vol. 1. 2005. URL [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=1493930](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1493930). (Cited on page 17)
- [Par] URL [http://www.computerworld.com/s/article/65878/The\\_Power\\_of\\_Parallelism](http://www.computerworld.com/s/article/65878/The_Power_of_Parallelism). (Cited on page 65)
- [PCISIG] URL <http://www.pcisig.com/specifications/pciexpress/base2/>. (Cited on page 28)
- [PeakSt] URL <http://arstechnica.com/hardware/news/2006/09/7763.ars>. (Cited on page 66)
- [Pino8] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 2008. (Cited on page 79)
- [ps306] 2006. URL <http://www.consolewatcher.com/2006/08/building-supercomputer-using-playstation-3/>. (Cited on page 17)
- [SCS<sup>+</sup>08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp. 1–15. ACM, New York, NY, USA, 2008. doi:<http://doi.acm.org/10.1145/1399504.1360617>. (Cited on page 15)
- [SKMS93] J. K. Sanjit K. Mitra, K. Sanjit. *Handbook for Digital Signal Processing*. John Wiley & Sons, 1993. (Cited on page 19)
- [Smied] J. O. Smith. *Mathematics of the Discrete Fourier Transform (DFT)*. <http://ccrma.stanford.edu/jos/mdft/>, November, 2009(date accessed). Online book. (Cited on pages 19 and 20)
- [Steo4] E. Stewart. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004. (Cited on page 38)
- [USC] URL <http://sipi.usc.edu/database/database.cgi?volume=misc&image=12>. (Cited on page 20)

All links were last followed on November 5<sup>th</sup>, 2009.



## **Declaration**

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

---

(Daniel Kauker)