VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EXPLOITATION OF GPU IN GRAPHICS AND IMAGE PROCESSING ALGORITHMS

DISERTAČNÍ PRÁCE

DOCTORAL THESIS

AUTOR PRÁCE ING. RADOVAN JOŠTH

AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ GPU PRO ALGORITMY GRAFIKY A ZPRACOVÁNÍ OBRAZU

EXPLOITATION OF GPU IN GRAPHICS AND IMAGE PROCESSING ALGORITHMS

DISERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE AUTHOR	ING. RADOVAN JOŠTH
VEDOUCÍ PRÁCE SUPERVISOR	DOC. ADAM HEROUT, PH.D.

BRNO 2014

Abstract

This thesis introduces several selected algorithms, which were primarily developed for CPUs, but based on high demand for improvements; we have decided to utilize it on behalf of GPGPU. This modification was at the same time goal of our research. The research itself was performed on CUDA enabled devices.

The thesis is divided in accordance with three algorithm's groups that have been researched: a real-time object detection, spectral image analysis and real-time line detection. The research on real-time object detection was performed by using LRD and LRP features. Research on spectral image analysis was performed by using PCA and NTF algorithms and for the needs of real-time line detection, we have modified accumulation scheme for the Hough transform in two different ways.

Prior to explaining particular algorithms and performed research, GPU architecture together with GPGPU overview are provided in second chapter, right after an introduction. Chapter dedicated to research achievements focus on methodology used for the different algorithm modifications and authors' assess to the research, as well as several products that have been developed during the research.

The final part of the thesis concludes our research and provides more information about the research impact.

Keywords

GPU, CPU, GPGPU, CUDA, LRP, LRD, PCA, NTF, Line Detection, Parallel Coordinates, Hough Transform, Spectral Image, Object Detection

Bibliographic Citation

Jošth, R.: Exploitation of GPU in graphics and image processing algorithms, Ph.D. thesis, Faculty of Information Technology, Brno University of Technology, Brno, CZ (2014)

Abstrakt

Táto práca popisuje niekoľko vybraných algoritmov, ktoré boli primárne vyvinuté pre CPU procesory, avšak vzhľadom k vysokému dopytu po ich vylepšeniach sme sa rozhodli ich využiť v prospech GPGPU (procesorov grafického adaptéra). Modifikácia týchto algoritmov bola zároveň cieľom nášho výskumu, ktorý bol prevedený pomocou CUDA rozhrania.

Práca je členená podľa troch skupín algoritmov, ktorým sme sa venovali: detekcia objektov v reálnom čase, spektrálna analýza obrazu a detekcia čiar v reálnom čase. Pre výskum detekcie objektov v reálnom čase sme zvolili použitie LRD a LRP funkcií. Výskum spektrálnej analýzy obrazu bol prevedný pomocou PCA a NTF algoritmov. Pre potreby skúmania detekcie čiar v reálnom čase sme používali dva rôzne spôsoby modifikovanej akumulačnej schémy Houghovej transformácie.

Pred samotnou časťou práce venujúcej sa konkrétnym algoritmom a predmetu skúmania, je v úvodných kapitolách, hneď po kapitole ozrejmujúcej dôvody skúmania vybranej problematiky, stručný prehľad architektúry GPU a GPGPU. Záverečné kapitoly sú zamerané na konkretizovanie vlastného prínosu autora, jeho zameranie, dosiahnuté výsledky a zvolený prístup k ich dosiahnutiu. Súčasťou výsledkov je niekoľko vyvinutých produktov.

V závere nechýba stručné zhodnotenie celého výskumu, jeho vplyv či využitie a dopad na budúce štúdie a výskum.

Kľúčové slová

GPU, CPU, GPGPU, CUDA, LRP, LRD, PCA, NTF, detekcia objektov, spektrálna analýza obrazu, detekcia čiar, Houghova transformácia, paralelné koordináty;

Bibliografická citácia

Jošth, R.: Využití GPU pro algoritmy grafiky a zpracování obrazu, Dizertačná práca, Fakulta informačních technologií, Vysoké učení technické v Brně, Brno, CZ (2014)

Declaration

I hereby declare that this thesis is my own work that has been created under the provision of Adam Herout. It is based on eight publications ([31, 32, 73, 30, 41, 42, 28, 4]) that have been written in cooperation with the following list of co-authors: Pavel Zemčík, Lukáš Polok, Michal Hradiš, Roman Juránek, Jiří Havel, Marku Hauta-Kasari, Jukka Antikainen, Markéta Dubská, Vitězslav Beran and Martin Žádník. Where the sources of information have been used, they have been duly acknowledged.

Radovan Jošth
20th August, 2014

Acknowledgements

I would not have been able to write this doctoral thesis without help and support of several people, and I would like to express my sincere gratitude to all of them.

First of all, I would like to thank my supervisor Adam Herout, for his valuable guidance, advices and recommendations. This thesis would not have been possible without my colleagues from university, all fruitful discussions related to topic, their initiative and help during research and publicizing. I would therefore express my gratitude to Markéta Dubská, Vítězslav Beran, Jukka Antikainen, Markku Hauta-Kasari, Jiří Havel, Michal Hradiš, Roman Juránek, Lukaš Polok and Martin Žádník.

I would like to thank Faculty of Information Technology, Brno University of Technology as it have provided me with a tremendous graduate education and financial, academic and technical support. I thank Pavel Zemčík, who was my initial supervisor.

Last, but not least, I thank all my good friends, who were anxious for this thesis, my supportive family – my mum, grandmother, sister, aunt and cousins; and my girlfriend for her support, patience, and for keeping a sense of humour when I had lost mine.

Contents

Chapter 1

Introduction

Back in 2007, when we started the research for the needs of this thesis, there was a limited number of implementations which could enable an effective and energy-efficient use of Graphics Processing Units (GPUs). There were many publications describing the new, fast implementations of algorithms on Central Processing Unit (CPU), but there was a gap and very high demand for improvements of current algorithms that were primarily designed for CPUs that time (even though there were also some solutions for specialized processors such as FPGA, DSP, etc.). Based on very high computational potential of GPU, we have decided to utilize it on behalf of general-purpose computing on graphics processing units (known as GPGPU, or GPGP or GP2U) - focusing on computer vision and image processing algorithms. These pixel-based applications are very well suited to GPGPU technology.

We have selected a set of existing and successfully implemented algorithms with good performance results and optimized them for GPGPU. For computer vision, we have decided for object detection using the Local Rank Differences (LRD) and Local Rank Pattern (LRP) functions, and for image processing improvement we had chosen Non-Negative Tensor Factorization (NTF) and Principle Component Analysis (PCA) algorithms. Research on line detection was performed using high-resolution Hough transformation and parallel coordinates.

As all commonly available PCs include GPU, our goal was to off-load CPU (that is optimized for a small number of threads) and move the part (or even the whole blocks) of program to GPGPU; and therefore enable better usage of computer resources, and enhance the effectiveness of whole PC in the manner of costs and data processing speed.

Historically, the initial purpose of GPU was to serve as graphic accelerator, which supports only specific fixed-function pipelines. Later on, after almost 10 years of development in 90's, they became increasingly programmable. NVIDIAs' GPU was for the first time introduced in 1999, but further development was carried on, moving forward to high-performance computing methodology - GPGPU, which uses GPU to crunch the data.

At the time of the research there were an NVIDIA GPUs available for our needs. These GPUs had a CUDA parallel computing platform interface and shaders. That time, the shaders were on a very high level from the GPU programming availability point of view; however they were tailored for graphics rendering and their usage for GPGPU needs would cause significant difficulties in optimization when comparing with CUDA parallel computing platform. NVIDIA invented CUDA in 2006, and that was the world's first solution for general-computing on GPUs. In parallel with CUDA development, there was an OpenCL developed. These are the two different interfaces for programming the GPUs. While OpenCL is an open standard that can be used to program CPUs, GPUs or other devices; CUDA is specific to NVIDIA GPUs.

For the needs of this thesis, CUDA was used as primary computing platform, as at the time the research, OpenCL was in a stage of development - it was not stable enough, and was unable to use the full potential of GPUs. Moreover, several months later, NVIDIA introduced Parallel Nsight, a development platform for heterogeneous computing, what enabled us to debug, and fully optimize the performance of GPU. This tool was used to identify and analyze bottlenecks, and to observe the behaviour of the system.

GPGPU makes a significant impact affecting wide range of application domains, such as weather forecasting, fluid-flow, or molecular dynamics. Algorithms that we were focusing on, can find an application on the field of computer vision, physics, astronomy, medicine and many others.

Thesis Structure

After introduction, the second part of this thesis discusses the background and architecture of GPU and GPGPU. Beside products available at the time of our research, the past, current and further GPUs are listed.

Next three chapters discuss different types of algorithms, we have been focusing on. Namely - LRD and LRP features are explained in Chapter 3., research on PCA and NTF algorithms is discussed within Chapter 4, and Hough transform with parallel coordinates are described in Chapter 5. These three chapters provide an insight, and basic information about the particular algorithms, with an outline of related researches, performance analysis and results for each research.

Chapter 6 points out the research gains of the author, his contribution and results.

Whole thesis is concluded within Chapter 7., which except overall conclusion includes also the citation analysis performed in order to assess research impact.

Within the thesis, the list of published papers is enclosed as Appendix A. Appendix B conclude the list of products developed during our research.

1.1 Research Motivation

As already stated, our whole research can be divided into three main parts:

- Speed-up of real-time object detection algorithms using CUDA;
- Optimizations of spectral image analysis algorithms;
- Modifications of real-time line detection algorithm (Hough transform).

Each of these topics had a different reason for research initiative and those are further explained in a consequent section.

1.1.1 Real-Time Object Detection Algorithms - Problem Formulation

Object detection, having a wide range of applications, was in 2001 subject of research for Viola and Jones [92], who introduced very successful face detector which was combining boosting, Haar low-level feature calculated on integral image, and a focus-of-attention cascade of classifiers. The detector provided a precision of detection high enough for practical applications. Success of Viola and Jones encouraged further research in similar approaches and resulted in a great number of modifications to this original detector.

It was a popular trend to use statistical classifiers (such as AdaBoost and its modifications) for object detection. Statistical classifiers, as very powerful and common approach to object detection, classified individual locations of the input image and made a binary decision whether the location contains the object or not. The result was a set of candidate locations, which was further proceeded, typically by a non-maxima suppression algorithm. Face detector of Viola and Jones was a combination of techniques that all together well minimized the average decision time. The classifier extracted relevant information from the image with Haar-like features, which were computed very fast and in constant time using an intermediate image representation called the integral image. Viola and Jones used AdaBoost, a general boosting algorithm, for feature selection by keeping weak hypotheses very simple and each based only on single Haar-like feature. However, either this or any other consequent proposed approaches ([86]) were still not fast enough for real-time applications.

As in this period of time, the GPGPU was introduced, we have decided to the accelerate object detection in images and video sequences using graphic processors (GPUs). Our task included algorithmic modifications and adjustments, constructing variants of efficient implementations, and evaluation by comparing with efficient implementations on CPUs. CUDA offered a maintainable and portable way of programming general-purpose code for the GPUs (in the scope of NVIDIA), and GPUs controlled by CUDA offered a high computational power for tasks that were highly parallel. The aim of our research (Chapter 3) was also to evaluate the suitability of the CUDA platform for object detection by classifiers, and to design efficient version of object detection algorithm.

1.1.2 Spectral Image Analysis Algorithms - Problem Formulation

The topic that led to initiation of the research on hardware-accelerated Principal Component Analysis (PCA) algorithm has been revealed from the start-up project called Optical Sensor Technology in Medical Applications, introduced by University of Eastern Finland. PCA was primarily targeted on real-time spectral image analysis. It could have been used on very large data sets, where its utilization has previously been unthinkable. The computational speed of PCA, especially the speed of creation of the co-variance matrix, was however critical and any improvement was appreciated.

PCA is often used for data of high dimensionalities. Generally, in the case of spectral imaging, the dimensionality of the input data was not high (commonly 6–81 channels) but the number of samples (i.e. number of pixels in image or video) was large - millions to billions. Existing solutions (e.g. [39, 38, 2, 67]) did not exactly suit this purpose, and so this unique situation must have been covered by a particular solution.

Within this research (Chapter 4), also motivated by the need of using PCA on spectral images in the context of real-time medical imaging, we have optimized two implementations of algorithm, one utilizing the SSE instruction set of contemporary CPUs, and the other running on graphics processors, using CUDA environment.

Spectral imaging is except medicine used in many different scientific and industrial fields, such as wood analysis, mineral detection or textile industries. Non-Negative Tensor Factorization (NTF) can be used for image compression [3], optimal filter generation [29], and feature extraction [43], or in fields of global climate analysis, neuroscience, psychometrics, etc. [75], [6], [11], [57], [84], [48]. The problem that led us to perform the research on this algorithm was that dimensionalities of these problems are often so high, that NTF computation takes hours, therefore the acceleration of this process was desirable.

Having the possibility to use GPU for GPGPU, we were not forced to use shading languages and rendering libraries, but were able to use CUDA. Our task was to speed-up the NTF computation, enhance the efficiency, and compare it with other available solutions.

1.1.3 Real-Time Line Detection - Problem Formulation

The Hough transform is a well-known algorithm for detecting shapes and objects in raster images. Originally, Hough [34] defined the transformation for detecting lines. Later it was extended for more complex shapes, such as circles, ellipses, etc., and even generalized for arbitrary patterns [5]. However, as standard Hough transform was rather slow to be usable in real-time, different accelerated and approximated algorithms existed. Previously, several research groups invested an effort to deal with computational complexity of Hough transform based on the θ-ϱ parametrization, which uses a very straightforward transformation from the image space to one bounded space of parameters, and because its uniform distribution of discretization error across the Hough space. There have been different methods developed ([71], [82], [98], [51] or [8]) focusing on spacial data structures, non-uniform resolution of the accumulation array, or special rules for picking points from the input image, but there was still a need for real-time implementation of the Hough transform.

Our first research (Section 5.1.) on modification of accumulation scheme for the Hough transform was using θ-ϱ parametrization. The algorithm used a modified strategy for accumulating the votes in the array of accumulators in the Hough space. The strategy was designed to meet the nature of GPUs available at the time of research.

The second part of this research (Section 5.2.) used new parametrization of lines – PClines. Both algorithms were suitable for computer systems with a small but fast read-write memory, such as GPUs available at the time of the research. Our second algorithm required no floating-point computations or goniometric functions, what made it suitable for special, or low-power processors and special-purpose chips. Our task was to evaluate proposed algorithm solutions both on synthetic binary images, and on complex high resolution real-world photos.

Chapter 2

GPU Architecture and GPGPU

The aim of this chapter is to provide an overview of GPU and GPGPU evolution and theory, together with different GPU products available both at the time of our research (years 2007-2011), and those that were not addressed in our research, as they became available later on.

Currently, there are basically two dominant GPU producers. The first one is NVIDIA, which invented their first GPU (GeForce 256) in 1999, and unveiled CUDA architecture in 2006. NVIDIA GPUs are now powering millions of desktops, notebooks, workstations and supercomputers around the world, accelerating computationally-intensive tasks for all types of potential customers – professionals, scientists, researchers or random consumers.

The whole scale of products is now available; such as Tesla for technical and scientific computing, Quadro for professional visualization, NVS products for financial industries, or their primary product line called GeForce, which is for a years in a competition with AMD's Radeon product, and will be the subject of our further discussion. Radeon brand was originally launched by ATI Technologies, and acquired by Advanced Micro Devices (AMD) in 2006. AMD is therefore the second dominant producer.

At the time of the research there were NVIDIA GPUs available for our needs. The reason why AMD products were not considered for our research is that AMD, supported only by OpenCL language, what was that time less mature and less stable than CUDA architecture - the base of our research.

2.1 GPU Architecture

Historically were GPUs designed as non-programmable 3D-graphics accelerators, supporting only specific fixed-function pipelines. The evolution of this kind of GPUs started from large expensive systems in early 1980s to small workstations and then PC accelerators in the mid to late 1990s, when the hardware became increasingly programmable. [91, 90] They were being developed by multiple companies, and during this period the performance increased from 50 million pixels to 1 billion pixels per second and from 10 000 vertices up to 10 million vertices per second. Within years 1994 and 2001, the progress on chips development moved from simplest pixel-drawing functions to implementing the full 3D pipeline including transforms, lighting, rasterization, texturing, depth testing and display, when the surface of an object was drawn as a collection of triangles.

Fixed-function pipeline GPUs were represented by products as S3 ViRGE (1995) or 3DFx VooDoo products line (starting in 1996), followed by first NVIDIA products: “pre”-GeForce NV3 (known also as RIVA 128 or N3 only, 1997), GeForce 256 (or NV10, 1999), later NV11 up to NV16, crowned by GeForce3 (NV20, in 2001), which for the first time allowed limited amount of programmability in the vertex pipeline. All of these are further explained within next sections of this chapter.

As the chip programmability of GeForce3 was very limited, later GeForce products became more flexible and faster, adding separate programmable engines for vertex and geometry shadings. This evolution culminated in the GeForce 7800 that had three kinds of programmable engines for different stages of the 3D pipeline together with several stages of configurable and fixed-function logic. With GeForce 7800, the era of programmable pipeline had begun.

At this point, also GPGPU started its evolution, as to perform non-graphics processing on graphics-optimized architectures. This was typically performed by running carefully crafted shader code against data presented as vertex or texture information, and retrieving the results from a later stage in the pipeline.

GeForce 7800 and its three engines management led to unpredictable bottlenecks, so in 2006 NVIDIA introduced GeForce 8800 (G80 series of Tesla product line) design that featured “unified shader architecture” with 128 processing elements distributed among eight shader cores, where each of them could have been assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance. With GeForce 8800, CUDA development environment was introduced (see 2.1 or 2.2 for more information about CUDA).

With G80 series, NVIDIA introduced their Tesla product line, beginning with PCI Express add-in boards, and drivers optimized for GPU computing beside of 3D rendering. From now on, GPU could become to be treated like a many-core processor. Tesla product line was followed by introduction of Fermi and Kepler product lines, as well as latest Maxwell and expected future Pascal products. All of these are further, and more in detail, discussed within next sections of this chapter.

Application Programming Interfaces (APIs)

In parallel with GPU development, an Application Programming Interfaces (APIs) were being developed. Generally, the API is standardized layer of software that allows an application (e.g. game) to send commands to graphics processing unit to draw objects on a display.

The leader in 3D graphics for workstation was American manufacturer Silicon Graphics, Inc. (SGI). Their IrisGL API became industry standard, overshadowing the open standards-based PHIGS, due to its usage simplicity and immediate mode rendering support. As the market started to accommodate more and more competitors, SGI decided to turn the IrisGL into an open standard – OpenGL.

OpenGL is now known as a rendering API, providing hardware accelerated (GPU) rendering functions. Unlike other popular APIs (like DirectX), OpenGL is platform agnostic, in the meaning that you can write an OpenGL application on one platform and at the same time OpenGL program can be compiled and run on another platform.

In 1995, Microsoft released the main competitor of OpenGL – Direct3D interface, and consequently in 1997 an SGI initiated the Fahrenheit project, which was a joint effort with the goal of unifying the OpenGL and Direct3D. [89] In 1998 Hewlett-Packard joined the project. Even though it initially showed some promise of bringing order to the world of interactive 3D computer graphics APIs, due to financial constrains at SGI, strategic reasons at Microsoft and general lack of industry support, it was abandoned in 1999. [90]

Several years later, in 2006, the OpenGL Architecture Review Board voted to transfer the control of OpenGL API standard to the Khronos Group, but still keeping the ARB acronym to prefix the name of OpenGL core extensions. [70]

In the same year, when NVIDIA introduced GeForce 8800, also CUDA was introduced. CUDA (stands for Compute Unified Device Architecture) is considered to be industry's first C-based development environment for GPUs, which delivers an easier and more effective programming model than earlier GPGPU architectures.

2.1.1 OpenGL and Shader Evolution

The first version of OpenGL had a fixed-function pipeline (Fig. 2.1), what means that all the functions performed by OpenGL were fixed and could not be modified except through the manipulation of various rendering states. Programmers therefore didn't have a control over the rendering pipeline. [90] The scheme of fixed-function pipeline OpenGL is shown at Fig. 2.1, where blue stages are still being used within current versions of OpenGL and orange ones represent stages of the fixed-function pipeline, that have been replaced by different stages in the programmable shader pipeline.

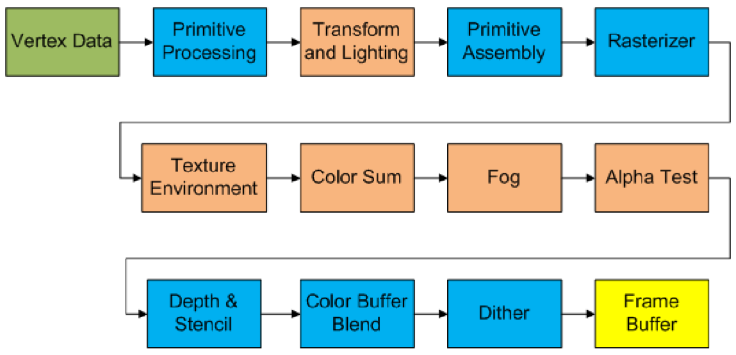


Figure 2.1: OpenGL Fixed-Function Pipeline.

OpenGL version 2.0 released in 2004 provided the ability to programmatically define the vertex transformation and lightening (T&L), and introduced fragment operations. Vertex shaders offer programmers with more flexibility regarding how the vertices are transformed, and it is even possible to perform the lighting computations in the fragment shader to archive per-pixel lighting. The primary responsibility of vertex shader is to transform the vertex position into "clip space". This is often done by multiplying the vertex position by the model-view-projection matrix (known also as MVP matrix). The output of vertex shader can go directly to rasterizer (OpenGL version 2.0) or to geometry shader (if present; from OpenGL version 3.0). [77]

In Fig. 2.2, orange stages from Fig. 2.1 are replaced by vertex and fragment program.

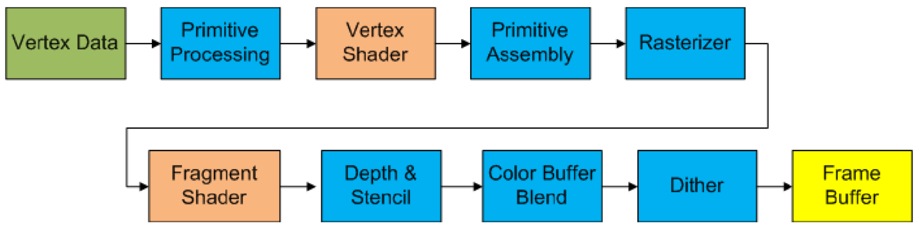


Figure 2.2: OpenGL 2.0 Programmable Shader Pipeline with Vertex Shader.

Another type of shader, available in OpenGL 2.0 is fragment shader, known also as pixel shader that compute colour and other attributes for each fragment. It replaces all of the complicated texture blending, colour sum, and fog operations from Fig. 2.1. Fragment shader can be used to compute the per-pixel lighting as well as blend together multiple textures to determine the final fragment colour.

OpenGL version 3.2 introduced in 2009 came with additional stage of the programmable shader pipeline called geometry shader (Fig. 2.3). This shader comes after the vertex shader in the programmable shader pipeline and therefore the output of vertex shader becomes an input to the geometry shader. Geometry shader can generate new graphics primitives, such as points, lines, and triangles. They are typically used for point sprite generation, geometry tessellation, shadow volume extrusion, or single pass rendering to a cube map.

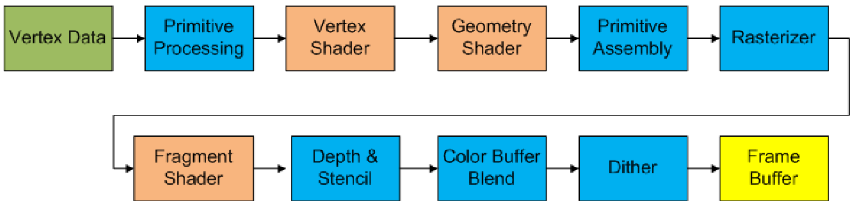


Figure 2.3: OpenGL 3.2 Programmable Shader Pipeline with Geometry Shader Included.

Tessellation stages that come after vertex but before geometry shaders were introduced in OpenGL 4.0 (2010) (Fig. 2.4). Tessellation Control Shaders and Tessellation Evaluation Shaders together allow for simpler meshes to be subdivided into finer meshes at run-time according to a mathematical function.

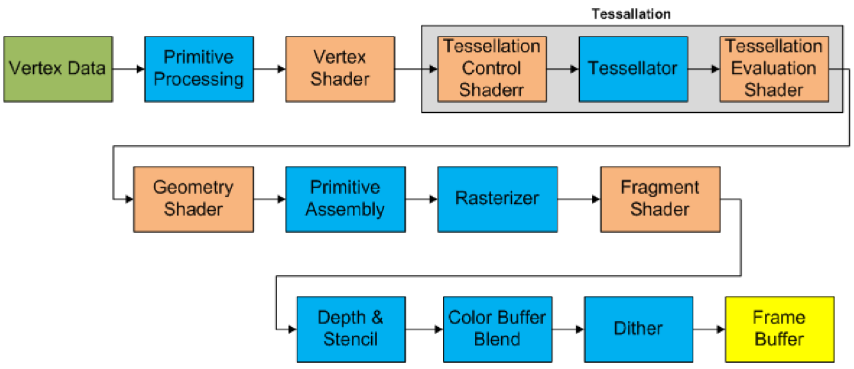


Figure 2.4: OpenGL 4.0 Programmable Shader Pipeline with Tessellation.

2.2 GPGPU

First naive graphic cards had almost no ability to change their graphic pipeline computations. NVIDIA Geforce 3/4 brought the first chance of shader programming for programmers. However, shaders in early stages couldn't compute very complex algorithms, due to graphic hardware limitation and were still used primarily for vertex and pixel processing. Lately introduced Geforce 8 provided almost unlimited programmability, not only thanks to better shading language, but also for support of CUDA. The CUDA uses the same hardware shaders, but the interface for accessing hardware is more straightforward for GPGPU programmers. Shading language therefore became to be used for other purposes than graphic computations.

As already stated, at the time of the research there were an NVIDIA GPUs available for our needs. The reason why AMD products were not considered for our research is that AMD, supported only by OpenCL language, was that time less mature and stable than CUDA architecture, so the performance with OpenCL implementation may not match the expected performance specified in particular GPU specification and figures provided further in the next chapters of the thesis. Initially, we have been working on an OpenCL implementation as well, as it was expected to bring more portability, ease of programming and software maintenance. However, we have found out that OpenCL was always one step behind, and a full utilization of hardware (NVIDIA) seemed to be impossible. It was just too unstable and in comparison with CUDA, it was missing the functionality.

Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between the data elements. The GPUs usage of data parallelism can be described as follows [77]:

- It uses Single Instruction Multiple Data (SIMD) or Thread (SIMT) model, while CPUs maps multiple tasks to multiple threads;
- It runs thousands of lightweight threads on hundreds of cores, while CPUs runs tens of relatively heavyweight threads on tens of cores;
- The threads are managed and scheduled by hardware, while on CPU each thread is managed and scheduled explicitly (Fig. 2.5);
- The programming is done for batches of threads, while on CPU each thread has to be programmed individually;

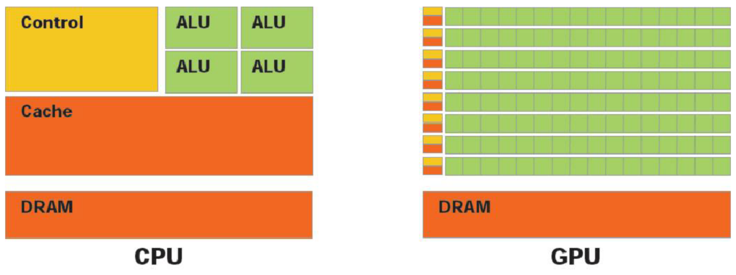


Figure 2.5: CPU and GPU Architecture Overview. [64]

There is also a different hardware architecture needed when performing tasks with GPU data parallelism. GPGPU capable graphic card contains several multiprocessors that contain a fast and small memory shared between the cores and a register set, and large DRAM which can be accessed directly or by using a cache. As already discussed, GPU architecture was originally designed for real-time rendering purposes: processing 3D vertices, rasterization of primitives and processing of pixels. All these tasks are performed in parallel and are done by a simple code with limited requirements for advanced programming structures (recursion, branching, loops, etc.). The possibilities and limitations of the GPUs are defined by SIMT concept and the different levels of memory.

Getting back to shaders evolution, as we have already mentioned in 2.1.1, at a particular evolution level of GPUs, shaders became to be able to execute very sophisticated and complicated computations. Suddenly it was possible to use shaders also for GPGPU needs. But to use them this way, we have had to choose special approach to algorithm decomposition. If we could divide the whole problem/algorithm in pixel/vertex manner, than we could use shaders for GPGPU. But there was still a high probability that we will have implementation problems, due to shader limitations. For example, it could be impossible to communicate between pixels in one pass, problems with storing temporary data, etc. Each program must be kind of “drawn” - even if you draw nothing. And this was the main disadvantage - the fact that you cannot focus on the problem itself, but wrap the problem into drawing of primitives.

CUDA/OpenCL was more generalized approach to overcome some of the limitations of shading languages, providing benefits such as:

- CUDA/OpenCL access to spatial information is much more flexible, than in shading language;
- CUDA/OpenCL provides thread synchronization and atomic functions;
- CUDA/OpenCL enables to define your own compute space (Fig. 2.6), while shading language will hard-wire the vertex/fragment compute space to your shader.

2.2.1 CUDA

With GPGPU, the programmers are not forced to use shading languages and rendering libraries to use the GPUs, but CUDA [64] – the first and currently the most mature C-like programming language. In other words, CUDA is a scalable parallel programming model and software environment for parallel computing. It offers a maintainable and portable way of programming general-purpose code for the GPUs. CUDA is often linked with the three abstractions that are simply exposed to the programmer as a minimal set of language extensions: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism.

This scalable programming model (Fig. 2.6) allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions:

from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs.

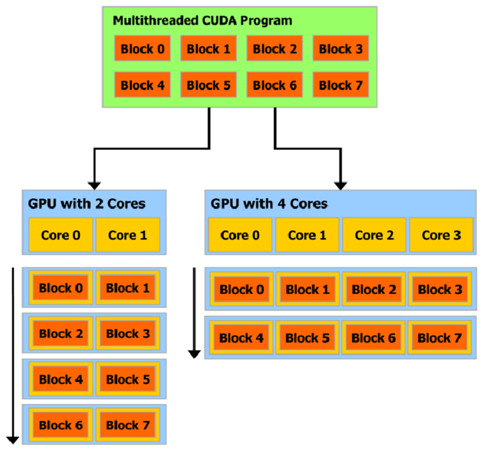


Figure 2.6: Automatic Scalability [64].

Next section introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. The following description was due to its complexity kept in an original wording of CUDA C Programming Guide [64], in order to provide the reader with accurate and compact set of essential information.

Kernels

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new `<<<...>>>` execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable. [64]

Here, each of the N threads that execute `VecAdd()` performs one pair-wise addition. Two vectors A and B of size N are added and stored into vector C. [64]

Thread Hierarchy

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a

Algorithm 2.1 Illustration of Kernel invocation.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y),the thread ID of a thread of index (x, y) is $(x + yD_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x+yD_x+zD_xD_y)$. [64]

As an example, the following code adds two matrices A and B of size $N \times N$ and stores the result into matrix C:

Algorithm 2.2 Illustration of Kernel invocation.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  ...
  // Kernel invocation with one block of N * N * 1 threads
  int numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Fig. 2.7. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

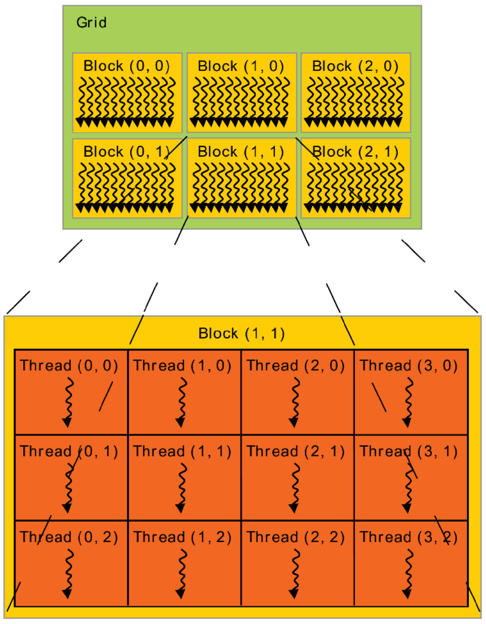


Figure 2.7: Grid of Thread Blocks.

The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type *int* or *dim3*. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in *blockIdx* variable. The dimension of the thread block is accessible within the kernel through the built-in *blockDim* variable.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

A thread block size of 16 $\times$ 16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that

Algorithm 2.3 Illustration of Kernel invocation.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  ...
  // Kernel invocation
  dim3 threadsPerBlock(16, 16);
  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  ...
}
```

dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Fig. 2.6, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight. [64]

Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Fig. 2.8. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.[64]

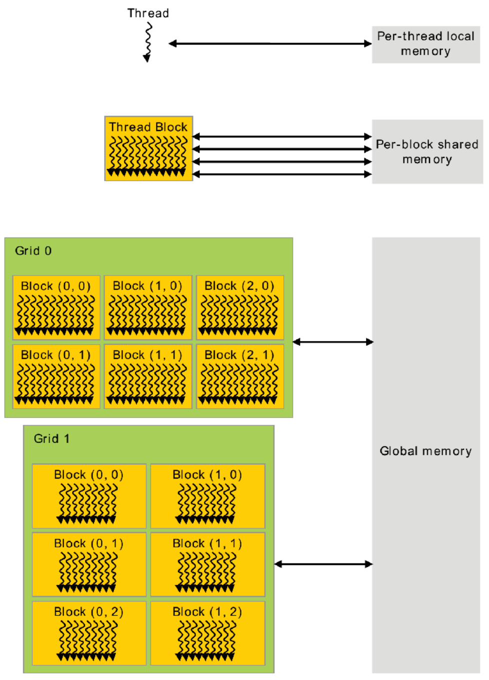


Figure 2.8: Memory Hierarchy.

Heterogeneous Programming

As illustrated by Fig. 2.9, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.[64]

Compute Capability

The compute capability of a device [64] is defined by a major revision number and a minor revision number. Devices with the same major revision number are of the same

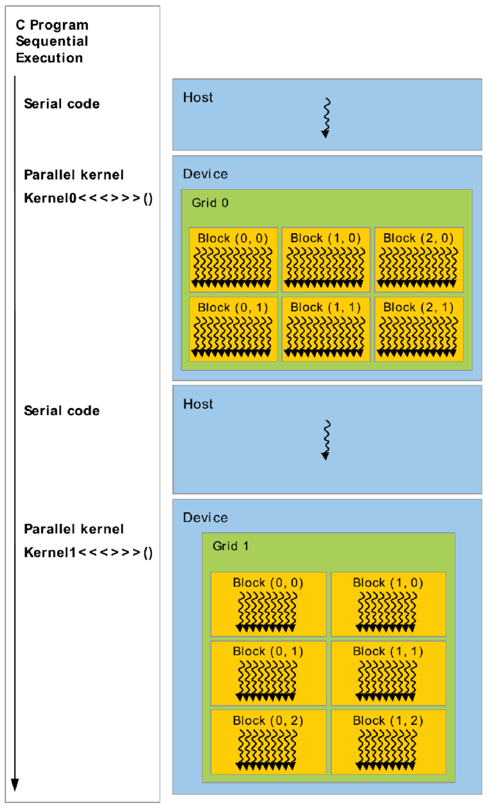


Figure 2.9: Heterogeneous Programming.

core architecture. The major revision number is 5 for devices based on the Maxwell architecture, 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture.

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

CUDA-enabled GPUs lists of all CUDA-enabled devices along with their compute capability. Compute capabilities gives the technical specifications of each compute capability. [64]

2.2.2 OpenCL

OpenCL language [47] was introduced due to lack of compatibility between hardware producers. It was merging various architecture interfaces into one unified OpenCL interface and could be used on different hardware. The interface is almost the same as CUDA for C interface and reimplementing already existing algorithms from CUDA to OpenCL is not difficult at all. CUDA and OpenCL differentiate in several ways:

- CUDA has better marketing, as it is directly supported by its GPU vendor;

- CUDA has developer-support in one package, well-written manuals, examples, tutorials, etc.;
- CUDA has more built-in functions and features, as it is re-released always together with a new product (up-to-date with NVIDIA products);
- CUDA was more stable in the time of our research;

The only disadvantage of CUDA was that it only works on GPUs of NVIDIA, while OpenCL is a completely open standard and has a support of more types of processor architectures; however it is supplied by many vendors, not provided as one packet or centrally orchestrated.

2.2.3 GPGPU Debugging

There are several debugging tools available for GPGPU programming. [24] *NVIDIA Visual Profiler* was introduced for a fist time in 2008 and provides a strategic metrics to find potential performance problems. It provides the performance analysis for CUDA apps Linux, Windows or Mac, and delivers developers vital feedback for optimizing CUDA C/C++ applications. It supports all CUDA capable NVIDIA GPUs and is available as part of the CUDA Toolkit. *CUDA-GDB* command line debugger seamlessly debug both the CPU and GPU code, setting the breakpoints on any source line or symbol name, executing only one wrap per single step. It is capable of handling thousands of threads running simultaneously on each GPU in the system. *CUDA-MEMCHECK* memory analyser accurately identifies the source and cause of memory access errors in GPU code and allows their quick locating. It also reports runtime execution errors, identifying situations that could otherwise result in an “unspecified launch failure” error when an application is running. One more tool, mostly during our research is *NVIDIA Nsight*, an ultimate development platform for heterogeneous computing, explained more in detail within next section.

NVIDIA Nsight

NVIDIA Nsigh enables full optimization of the CPU and GPU performance. This feature-rich tool provides generally better understanding of the code by identifying and analysing the bottlenecks and observing the behaviour of all system activities. An environment integrated into Microsoft Visual Studio extends the debugging and performance analysis capabilities of Visual Studio to support GPU computing, and is useful for game development, high-performance computing, supercomputing or workstation and content creation software. Nsight can be divided three functional parts:

1. GPU Debugger that helps to debug applications that uses CUDA. It enables to set the breakpoints in CUDA source code, inspect the memory, view the values of local variables, perform memory checks, or other common debugging tasks.

2. Graphics Debugger debug frame by each draw call (vertex shaders, pixel shaders, view pipeline states). It creates performance makers and profiles the execution of graphics code.

3. System Analysis and Profiling Tools provide an understanding how workloads are distributed across an application and the whole system in general. It enables programmer to see API calls (including CUDA, OpenCL, DirectX, and OpenGL), memory copies, kernel executions, draw calls, and CPU/GPU activity events along a visual time-line. Some key features are source code correlation, deep kernel analysis to detect factors limiting maximum performance, or unlimited experiments on live kernels.

2.3 GPU Product Lines Overview

This subsection will provide you with the selection of some major GPU releases aligned chronologically, beginning with first generations of fixed-function pipeline GPUs, crossing through programmable pipeline NVIDIA Tesla, Fermi, Kepler and Maxwell product lines. Section is concluded by some proposed future GPU products.

During our research, we have been mostly using three particular graphics cards:

- NVIDIA GeForce 9800GTX (Tesla product line);
- NVIDIA GeForce GTX 280 (Tesla product line);
- NVIDIA GeForce GTX 480 (Fermi product line).

The major differences between the product lines are described below.

2.3.1 Fixed-Function Pipeline Products

First Generation of GPUs

The two significant products since the beginnings of 3D graphics that are worth to be mentioned within this section are [53]:

S3 ViRGE (1995) graphics chipset is known as first 2D/3D accelerator that has been intended for mainstream consumers. The acronym ViRGE stands for Virtual Reality Graphics Engine, and S3 powerhouse equipped their first 3D the chipset with 4MB of on-board memory (core and memory clock-speeds of up to 66MHz). ViRGE's pixel throughput gained somewhat faster than the best software-optimized 3D-rendering available that time, when performing basic 3D-renering with only texture mapping without other advanced features. In addition, it offered better (16bpp) colour fidelity. However, with additional operations to the polygon load, such as perspective-correction, Z-depth fogging or bilinear filtering, the rendering throughput dropped to the speed of software-based rendering on an entry-level CPU. This feature was unacceptable for the most of the gamers, and after introduction of competing product (by 3dfx), the S3 as a company was unable to adapt the rapidly evolving market.

3dfx Voodoo (1996) product line introduced by 3dfx Interactive was the company's initial flagship. It heralded a new era of high-performance and high-quality 3D graphics for gaming and became a standard for many 3D games. The typical Voodoo Graphics PCI expansion card consisted of a DAC, a frame buffer processor and a texture mapping unit, along with 4 MB of EDO DRAM. RAM and graphics processors operated at 50 MHz. While other video-cards fused on both 2D and 3D functionality onto a single board, the Voodoo1 concentrated solely on 3D and lacked any 2D capabilities. The consumers therefore still needed a 2D graphics card for day to day computing, which would be connected to the Voodoo1 via a VGA pass-through cable.

NVIDIA's First Generation of GPUs

Several years after NVIDIA has been founded, they came with their first 3D GPU [77, 53]:

NVIDIA "pre"-GeForce (NV3) (1997) product, also known as RIVA 128 or N3, was introduced to target the performance segment of the volume PC graphics market. It was designed with OpenGL 1.0 and Microsoft's DirectX 5 API in mind. NVIDIA packed 3.5 million transistors on its first performance part, along with a single pixel pipeline. It was also a 2D/3D combo card, whereas 3dfx's Voodoo line still required a separate 2D card. This proposition was relatively costly, what was not a welcome feature in gamer's community. However, image quality was poor compared to the Voodoo line, at least early on, and some games at the time were embracing 3dfx's proprietary Glide API.

Nowadays, NVIDIA desktop cards carry the GeForce nomenclature. This naming scheme first began in 1999 as a result of contest called "Name That Chip".

NVIDIA GeForce (NV10, NV11-NV16) (1999) firstly introduced NV10, also known GeForce 256. This architecture offered significant speed gains over its predecessor, almost twice as fast in some cases, and allowed NVIDIA to snatch the performance crown from 3dfx in dramatic fashion. The GeForce 256's quad-pixel rendering pipeline could pump 480 M/texels, which was about 100-166M more than other video-cards on the market that time. It was also equipped with hardware T&L (Transformation and Lighting) and a multi-texturing (giving bump maps, light maps, cube environment mapping for creating real-time reflections and others). Later NV11-NV16 got second texture map unit (TMU) to each of its 2-4 pixel pipelines what helped to boost the performance.

NVIDIA GeForce3 (NV20) (2001) was introduced with Shader Model 1.0. It was the first time when a limited amount of programmability in the vertex pipeline was allowed, together with volume texturing and multi-sampling for anti-aliasing.

Consequently, several generations of graphics cards from NVIDIA, ATI and other manufacturers were developed. Their enhancements were mainly based on tweaking the speed of memory and other computing units, bus expanding, etc. However, any of these enhancements did not enter the market with significant step forward.

2.3.2 NVIDIA Programmable Pipeline Products

Year 2002 brought the first GPU with programmable pipeline. This section presents the selection of some of the major NVIDIA GPU releases having this ability [77, 53]:

NVIDIA GeForce FX (NV30) (2002) is also known as GeForce5. The NVIDIAs' FX series was the fifth generation of the GeForce line and the first generation of fully-programmable graphics cards. They were the company's first video-cards to support Shader Model 2.0 with support of Cg, HLSL and GLSL shading language. Shader model 2.0 allowed more flexibility in complex shader/fragment programs and much higher arithmetic precision.

NVIDIA GeForce6 (NV40) (2004) series provided innovative feature set of computing, including full support of Shader Model 3.0 for unparalleled gaming effects. This series implemented also high dynamic range imaging and introduced scalable link interface (SLI) and PureVideo capability. The main benefit of this series, from programmable pipeline point of view, was dynamic flow control in vertex and pixel shaders (branching, looping, predication, etc.), increased efficiency, longer shader lengths and vertex texture fetch.

NVIDIA GeForce7 (G70/NV47) (2005) series was the last generation of GPUs that could support the AGP bus. It was just the refined version of 6th GeForce generation, but providing the major improvement of being a widened pipeline and increase in clock speed. The GeForce 7950GT was used in origins of our research, where object-detection algorithm, which used LRD and Haar features, was implemented by using shaders.[73]

TESLA Product Line

Tesla, as NVIDIA's first micro-architecture (Fig. 2.10) was the first product line which implemented unified shaders (Fig. 2.11)[77]. Prior to this point, pixel shaders and vertex shaders existed as separate units. Tesla started with 8th generation of GeForce and covers also GeForce 9 Series, GeForce 100 Series, GeForce 200 Series and GeForce 300 Series, and then it was replaced by Fermi product line. Tesla is at the same time NVIDIA's third generation of micro-architecture designed as a GPGPU.

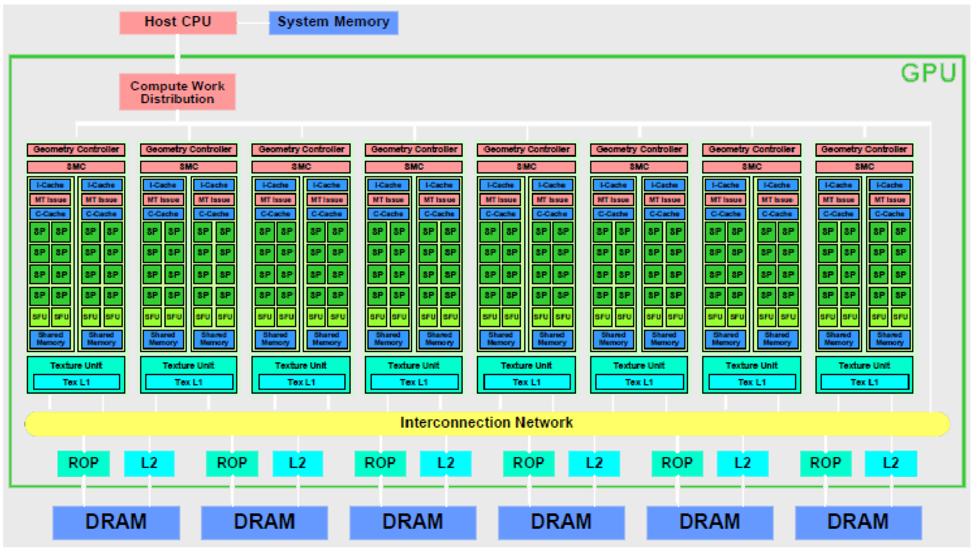


Figure 2.10: Tesla Architecture [88].

With unified shaders, unlike the vector processing approach taken with older shader units, each stream processor (SP) is scalar and thus can operate only on one component at a time. This makes them less complex to build while still being quite flexible and universal. Each streaming multiprocessor (SM) consists of eight scalar SPs. Two special function units (SFUs) for transcendentals such as exponential function, logarithm, and trigonometric functions, an MTIU (multi-threaded instruction unit), and on-chip shared

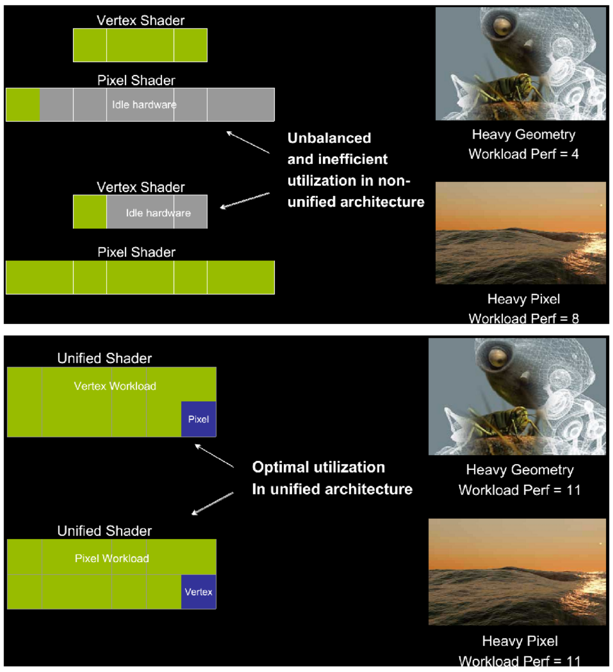

Vertex Shader
Pixel Shader
Idle hardware
Unbalanced and inefficient utilization in non-unified architecture
Heavy Geometry Workload Perf = 4
Vertex Shader
Idle hardware
Pixel Shader
Heavy Pixel Workload Perf = 8
Unified Shader
Vertex Workload
Pixel
Optimal utilization In unified architecture
Heavy Geometry Workload Perf = 11
Unified Shader
Pixel Workload
Vertex
Heavy Pixel Workload Perf = 11

Figure 2.11: Unified Shaders [77].

memory. The SM creates, manages, and executes hundreds of concurrent threads in hardware with zero scheduling overhead. It can create as many as eight CUDA thread blocks concurrently, limited by thread and memory resources. To manage hundreds of threads running several different programs, the Tesla SM employs single-instruction, multiple-thread (SIMT) architecture. For more details, refer to [64]. Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management. [88]

During our research, we have been using two Tesla GPUs: GeForce 9800GTX and GeForce GTX280.

NVIDIA GeForce 9800GTX (2008) when compared with previous versions (8800GTX), benefited in two SLI connectors, higher clock speed, and support for NVIDIA Hybrid Power, a technology allowing the discrete GPU to shut off during non-resource intensive applications, and use integrated GPU instead. This feature made this product relatively expensive. The memory interface width was 256-bit. It supported Shader Model 4.0 and Compute Capability 1.1. GeForce 9800GTX has 128 CUDA cores (SPs) which are divided into 16 multi-processors (SMs). CUDA capability 1.1 enabling 8 192 registers and 16 KB of shared memory per one SM. See CUDA documentation [60] for more details on Compute Capability. Three months after releasing this version, NVIDIA introduced GeForce 9800GTX+ with even faster core and shader clocks. This design is since March 2009 manufactured as GeForce GTS

250. [53]

NVIDIA GeForce GTX280 (2008) was almost identical with GeForce 9800GTX but providing 240 CUDA cores (SPs) divided into 30 multi-processors (SMs). The most interesting feature was the support of Compute Capability 1.3 (while already Compute Capability 1.2 supported Atomic function in shared memory). See CUDA documentation [60] for more details on Compute Capability and differences between any two versions. [53]

FERMI Product Line

Fermi micro-architecture introduced (Fig. 2.13) in the beginning of 2010 was represented by GeForce 400 Series and GeForce 500 Series. A complex architecture managed by a multi-level programming model allowing programmers to focus on an algorithm design to improve the productivity. Fermi was based on collection of four Graphics Processing Clusters (GPCs), each of which contained a raster engine and four SM units. Fermi supports concurrent kernel (Fig. 2.12) execution, where different kernels of the same application context can execute on the GPU at the same time thus fully utilizing GPU capacity.[61, 20, 83, 96]

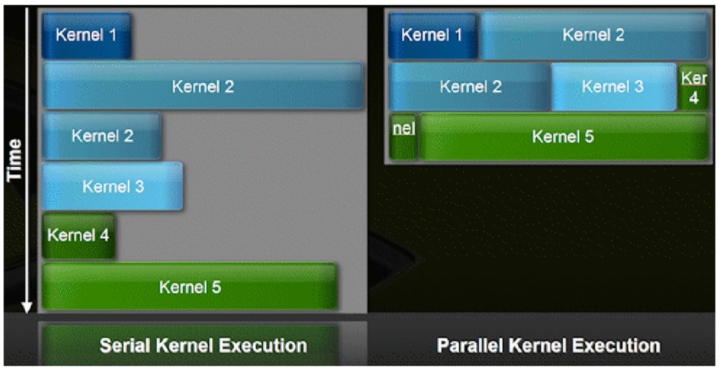


Figure 2.12: Fermi Concurrent Kernel.

At this time, NVIDIA has also introduced Nexus (further renamed to Nsight, 2.2.3), which is claimed to be the world's first integrated heterogeneous computing application development environment within Microsoft Visual Studio.

Double precision throughput has increased by a factor of eight compared to the previous generation. NVIDIA has also added support for ECC memory, which was a critical requirement for data-centres and supercomputers looking to deploy GPUs on a large scale.

During our research, we have been using one Fermi GPU – a GeForce GTX 480.

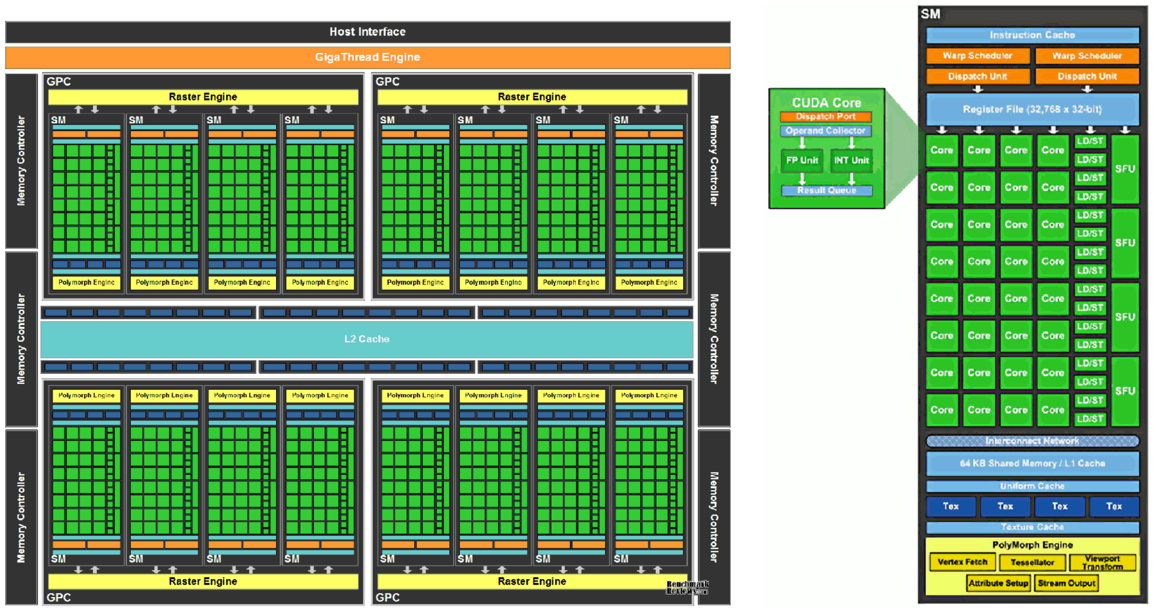


Figure 2.13: Fermi Architecture GeForce GT300 (GF100).

NVIDIA GeForce GTX 480 (2010) provided 480 CUDA cores (SPs) divided into 15 multi-processors (SMs) and included GDDR5 memory with memory width of 384-bit, what is less than the Tesla GTX 280, but the overall maximal bandwidth is up to 177 GB/s. It supported Compute Capability 2.0. See CUDA documentation [60] for more details.

KEPLER Product Line The same year as Fermi, NVIDIA introduced Kepler micro-architecture (Fig. 2.14) which brought some very important architectural changes. Kepler is represented by GeForce 600 Series and GeForce 700 Series. Taking into account that Kepler was still organized into CUDA cores, SMs, and GPCs, and the way how the warps were executed, from a high-level view, Kepler was identical to Fermi. However, the key new features of Kepler compared to previous Fermi were new SMX processor architecture and enhanced memory subsystem (offering additional caching capabilities, more bandwidth at each level of the hierarchy, and a fully redesigned and substantially faster DRAM I/O implementation). [63]

Kepler replaced SM with SMX consisting of 192 CUDA cores (SPs), 32 Special Function Units (SFU), and 32 Load/Store units (LD/ST). It was designed from ground up to maximize computational performance with superior power efficiency. SMX was 3 times more energy efficient than previous Fermi multiprocessor. Each SMX featured four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's quad warp scheduler selected four warps, and two

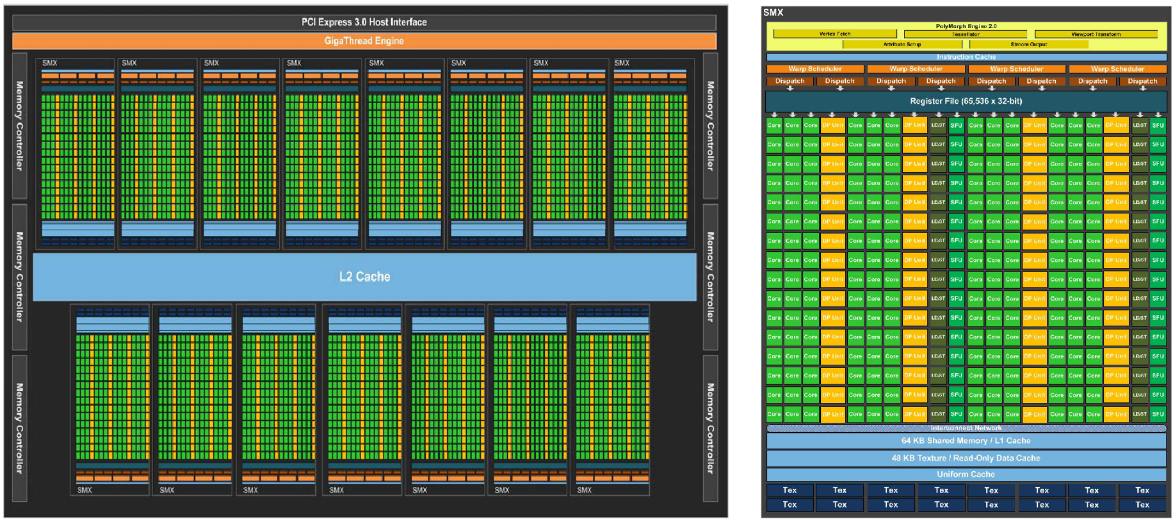

PCI Express 3.0 Host Interface
GigaThread Engine
SMX
Memory Controller
L2 Cache
Register File (65,536 x 32-bit)
Instruction Cache
Warp Scheduler
Dispatch
64 KB Shared Memory / L1 Cache
48 KB Texture / Read-Only Data Cache
Uniform Cache
Tex

Figure 2.14: Kepler SMX Architecture.

independent instructions per warp could be dispatched each cycle.

Another ability introduced with Kepler was Dynamic Parallelism (Fig. 2.15, 2.16) for kernels to be able to dispatch other kernels. By giving kernels the ability to dispatch their own child kernels, GK110 could both save time by not having to go back to the CPU, and in the process free up the CPU to work on other tasks. Dynamic Parallelism is all about scheduling work on GPU based on the data without the need for the CUP to coordinate work. The accelerator can generate work for itself; it can launch its own kernels unlike in the past where CPU was solely responsible for launching all kernels allowing more of a program to directly run on the GPU without communication with the CPU. [40, 62]

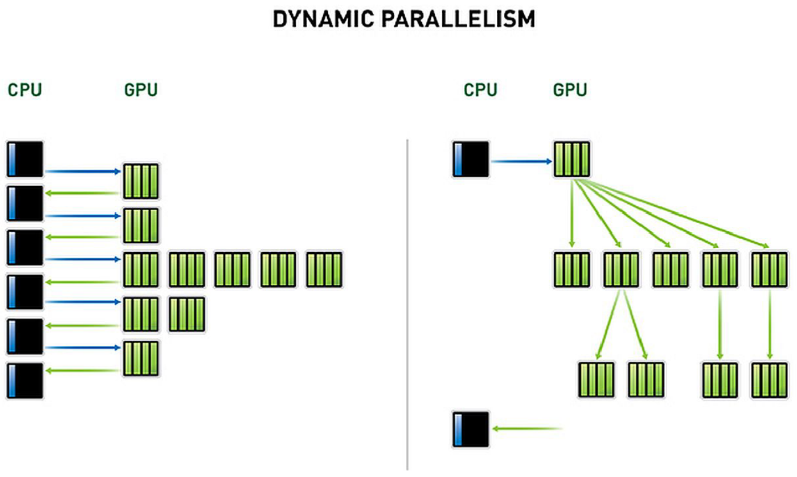


Figure 2.15: Kepler Dynamic Parallelism.

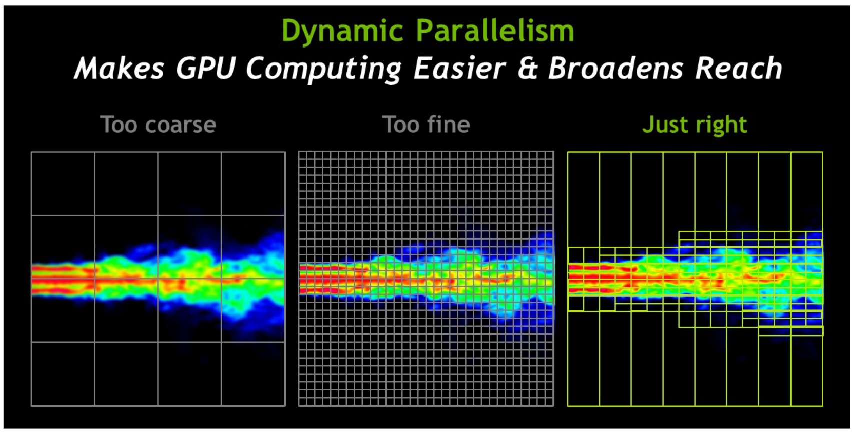


Figure 2.16: Dynamic Parallelism Decomposition.

MAXWELL Product Line Maxwell was introduced in 2013 and is represented by GeForce 800 Series and GeForce 900 Series. The number of CUDA Cores per SM has been reduced to a power of two, however with Maxwell's improved execution efficiency, performance per SM is usually within 10% of Kepler performance, and the improved area efficiency of the SM means CUDA cores per GPU will be substantially higher versus comparable Fermi or Kepler products. The Maxwell SM retains the same number of instruction issue slots per clock and reduces arithmetic latencies compared to the Kepler design.

As with SMX, each SMM has four warp schedulers, but unlike SMX, all core SMM functional units are assigned to a particular scheduler, with no shared units. Number of active thread blocks per multiprocessor has been doubled over SMX to 32, which should result in an automatic occupancy improvement for kernels that use small thread blocks of 64 or fewer threads. A significant improvement in SMM is that it provides 64KB of dedicated shared memory per SM and per-thread-block limit remains 48 KB. [65]

Future

In 2013 (the same year when Maxwell architecture was introduced) NVIDIA announced that their next GPU architecture will be Volta; with on-package DRAM, utilizing Through Silicon Vias (TSVs) to die stack memory and place it on the same package as the GPU. However, in the first quarter of this year, Volta was pushed back and architecture named Pascal (Fig. 2.17) was announced for year 2016. Pascal is supposed to be the first GPU to use stacked, 3D chip packing, and should incorporate a new PCI Express-based interconnect technology called NVLink. With Pascal, NVIDIA will achieve 2.5 times the capacity and four times the energy efficiency of Maxwell while boosting memory bandwidth for multi-GPU scaling even further.

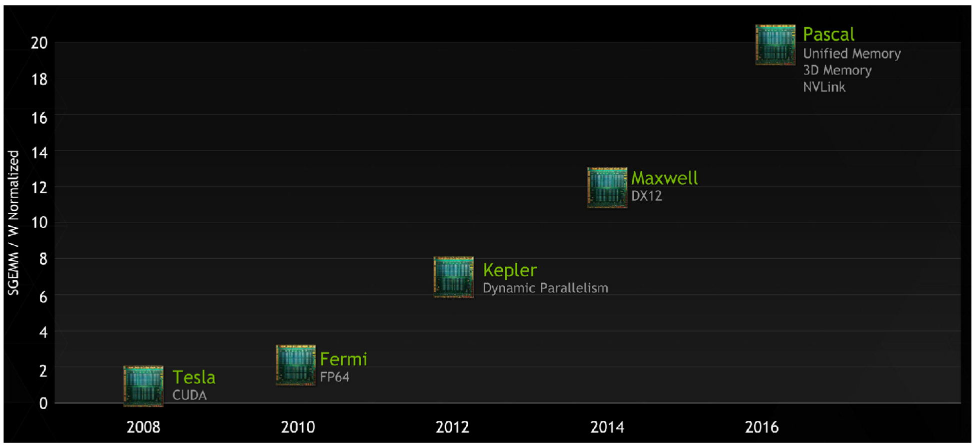


Figure 2.17: Nvidia Roadmap.

Another NVIDIA-announced future event is Echelon (Fig. 2.18) - NVIDIA's Extreme-Scale Computing Project. NVIDIA Research team Echelon project aims to address energy-efficiency and memory-bandwidth challenges and provide features that facilitate programming of scalable parallel systems. Echelon is a general-purpose fine-grained parallel-computing system that performs well on a range of applications, including traditional and emerging computational graphics as well as data-intensive and high-performance computing. At a 10 nm process technology in 2017, the Echelon project's initial performance target is a peak double-precision throughput of 16 Tflops, a memory bandwidth of 1.6 TB per second, and a power budget of less than 150 W. The goal is to integrate CPUs and GPUs on the same die with unified memory architecture. Such a system eliminates some of accelerator architectures' historical challenges, including requiring the programmer to manage multiple memory spaces, suffering from bandwidth limitations from an interface such as PCI Express for transfers between CPUs and GPUs, and the system-level energy overheads for both chip crossings and replicated chip infrastructure. Echelon aims to achieve an average energy efficiency of 20 pJ per sustained floating-point instruction, including all memory accesses associated with program execution. [45, 19]

Echelon processor promises global address space, flexible memory hierarchy, efficient bulk parallelism and heterogeneous cores.

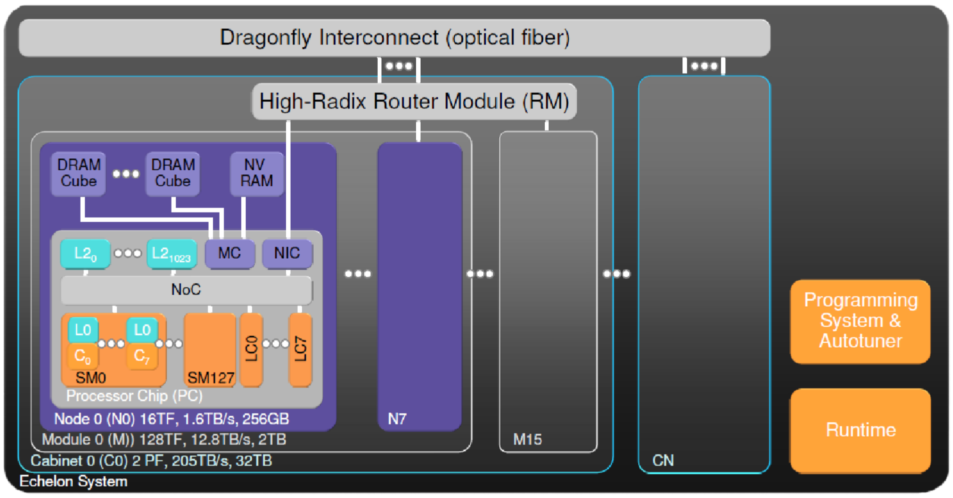

Dragonfly Interconnect (optical fiber)
High-Radix Router Module (RM)
DRAM Cube
DRAM Cube
NV RAM
L2$_0$
L2$_{1023}$
MC
NIC
NoC
L0
C$_0$
L0
C$_7$
SM0
SM127
LC0
LC7
Processor Chip (PC)
Node 0 (N0) 16TF, 1.6TB/s, 256GB
N7
Module 0 (M)) 128TF, 12.8TB/s, 2TB
M15
Cabinet 0 (C0) 2 PF, 205TB/s, 32TB
CN
Echelon System
Programming System & Autotuner
Runtime

Figure 2.18: Echelon System Sketch.

Chapter 3

Real-Time Object Detection Using CUDA

This chapter presents object detection in still images and video sequences. Image classification and detection tasks can be used as a base for various image processing and computer vision applications. While it is a very costly task from the computational resources point of view, very high demand exists for efficient object detection methods and implementations.

One of the frequently used techniques of fast object detection is usage of classifiers to scan the image and attempt classification of every potential object position or even every potential position in the image being searched. Classifiers can be implemented as statistical classifiers based on supervised machine learning and can take as their input low-level features (sometimes called weak classifiers) extracted from the window being classified. In principle, such features can be immediately the image pixels, but by using more complex feature extractors, the classifiers can achieve better performance - both in the detection rate and speed.

The research on real-time object detection presented within this chapter was performed in cooperation with the following list of co-authors: Adam Herout, Pavel Zemčík, Lukáš Polok, Michal Hradiš, Roman Juránek and Jiří Havel.

3.1 Background of Object Detection by Boosting

In 2001 Viola and Jones [92] presented the first real-time frontal face detector which provided a precision of detection high enough for practical applications. This performance was achieved by combining ideas which together very well minimize the average computation time. The individual parts are the Haar-like features used to efficiently extract discriminative information from images; the AdaBoost learner which combines simple hypotheses into a powerful decision rule; and the attention-cascade structure of the detector which greatly reduces the average decision time. Additionally, bootstrapping was used when training the detector to achieve very low false positive rates needed when detecting objects in images. The significant success of the Viola and Jones face detector consequently encouraged further research in similar approaches and resulted in a great number of modifications to this original detector.

The performance of the detection classifiers largely depends on the type of features they use. The ideal features should be computationally inexpensive, and to some degree, invariant to geometry and illumination changes, and should provide high discriminative power – all at the same time. High discriminative power is needed to achieve high precision of detection and it also implies more compact and faster classifiers as lower number of features is needed to be computed for the classifier to make a decision. In general, the ideal type of features can differ for different types of objects [87]. However, simple image filters have been proven to generalize well across various types of objects [79]. These filters decorrelate the neighbouring pixel values; utilize knowledge about frequency properties of images; and they also provide low tolerance to geometric transformations. Most of the filters which are used for object detection do not respond to the zero-frequency component, and they can be also normalized to compensate lighting changes.

When using simple filters, it is possible to transform the data in such a way that all the information in the original data is represented with the same number of coefficients (wavelet transformation). However, it is more efficient to consider all the possible filters and choose only the most discriminative for the classifier. This way, the most relevant information is extracted in the least amount of time and the classifier can be simpler. For example, Viola Jones ([92]) used a highly over-complete set of Haar-like features totalling 180,000 for samples 24×24.

Algorithm 3.1 The original version of AdaBoost [18] with notation modified according to [78].

```
Require: $S = \langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$, $x_i \in \mathbb{X}$, $y_i \in Y = \{-1, +1\}$
Ensure: $D_1(i) = 1/m$
1: for $t = 1, \ldots, T$ do
2:     Train weak learner using distribution $D_t$
3:     Choose $\alpha_t = \frac{1}{2} \ln \left( \frac{1-\varepsilon_t}{\varepsilon_t} \right)$
4:     Update: $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$ where $Z_t$is a normalization factor
5: end for
6: Output the final hypotesis: $H(x) = sign \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right)$
```

Viola and Jones used AdaBoost [18] algorithm to both select informative features and create the classifier. AdaBoost (Alg. 3.1) is one of the boosting algorithms. It combines simple (weak) classifiers into a very accurate prediction rule (strong classifier). If each of the weak classifiers is based on only a single feature, the boosting algorithm then effectively performs feature selection. The weak classifiers are selected in a greedy fashion and combined to minimize an exponential loss function. AdaBoost creates large-margin classifiers in the weak classifier space. The AdaBoost algorithm has certain properties which makes it especially useful for real-time detection. The strong classifier is a linear combination of the weak classifiers which makes it very efficient to compute. Also, the algorithm rapidly converges to a good solution on training data which minimizes the size of the strong classifier. Finally, the AdaBoost algorithm has been proven to reach an arbitrarily low classification error rate on the training data as long as the weak classifiers provide at least some useful information. This can be generalized in that the AdaBoost algorithm is guaranteed to reach a specific error at any operating point. In the Viola & Jones detector, this fact is exploited when creating classifiers for the cascade stages, where the reaching of a specified error at a specific operating point is used as the stopping criterion. This way, the complexity of the classifier is kept low while maintaining the required error rate.

The ensemble classifier created by AdaBoost can be itself a powerful and efficient classifier capable of detecting objects in images. However, such a classifier would have to still be composed of hundreds of weak hypotheses. Such a large classifier would certainly not provide real-time performance in most of the desired scenarios. To reduce the computational complexity of the detector, Viola and Jones exploited the fact that the vast majority of samples classified when scanning images for desired objects belong to background. They created an object-specific focus-of-attention mechanism which they called cascade and which is essentially a degenerated decision tree (Fig. 3.1), where each of the nodes is a strong classifiers created by AdaBoost. The individual stages of the cascade either reject the processed sample as background or they send the sample to the next classifier. As the decision task becomes harder for the later stages, the classifiers become longer. The cascade is the first mechanism which allows creation of such focus-of-attention mechanisms at least partially automatically.

The detection cascade can be created according to the desired false positive rate and false negative rate of each stage. In such a case, AdaBoost increases the size of the strong classifier until the required rates are reached. However, in [92], the authors set the lengths of the individual stages manually. Moreover, the cascade is in many aspects suboptimal. First, all information between the consecutive stages is lost, even though the previous stage already provides a very good solution to the problem of the next stage. Second, the operating points of the classifiers and their lengths are set ad-hoc and not optimally.

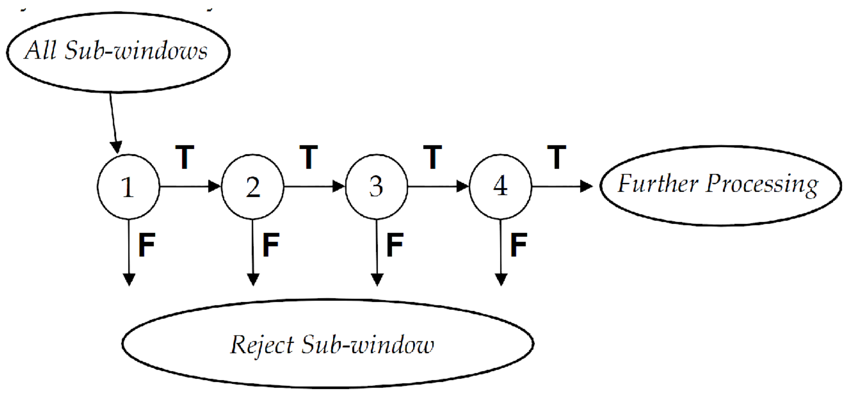


Figure 3.1: The detection cascade. The cascade is composed of a series of increasingly more complex classifiers which either reject the classified sub-window as background or pass it to the subsequent stage. The object is detected only if the corresponding sub-window successfully passes through all of the stages. [92]

These two problems were addressed many times ([7][85] [97]), most notably, [86] presented WaldBoost algorithm which solves these two problems in a natural way. The WaldBoost algorithm is a combination of real AdaBoost [78] and Wald's [94] sequential probability ratio test. In WaldBoost, rejection thresholds are set after each iteration of the AdaBoost algorithm. The thresholds are set as Wald proposes in the sequential probability ratio test, which he proves is the fastest possible classification strategy for a given target error rate. Also, as the resulting classifier is monolithic, no information is lost.

3.1.1 Image Features Based on Haar Wavelets

The Haar features were introduced by Papageorgiou et al. [72], who used them as an input for support vector machine to create a very accurate classifier. Viola and Jones [92] used the Haar features for rapid object detection in a framework with an AdaBoost classifier and thresholding weak hypotheses. The features, in their basic form, are based on the difference of adjacent rectangular regions of the input image. They respond strongly on edges and line segments of the image.

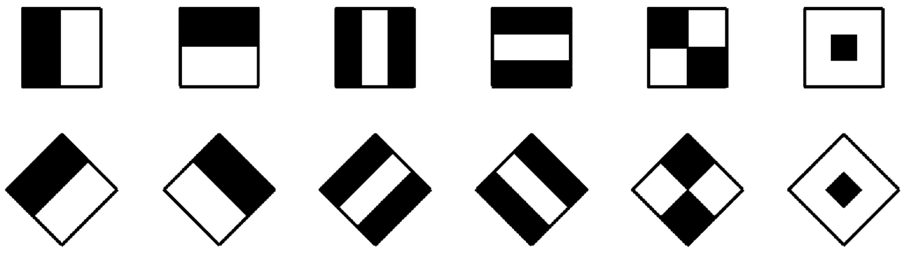


Figure 3.2: Shapes of Haar features. Standard shapes on top and extended set on the bottom.

The shapes of the wavelets typically used in pattern recognition are displayed in Fig. 3.2. The Haar features are very popular for their extremely low computational cost when evaluated on integral image and for providing good amount of information at the same time. The extended Haar feature set was introduced by Lienhart and Maydt [52]. The difference from the commonly used features is that new features are rotated by 45 degrees.

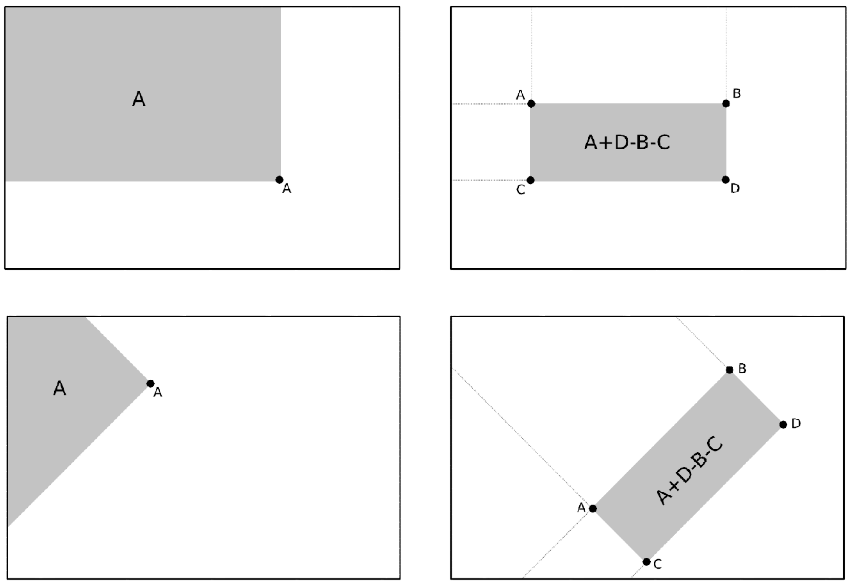


Figure 3.3: Integral images. Standard integral image (top) and integral image required to evaluate 45 degree rotated Haar features (bottom).

Efficient evaluation of Haar features is achieved by using integral image (Fig. 3.3). The integral image stores in each pixel the sum of all pixels above and to the left of it. As a consequence, the sum of pixels of an arbitrary axis-aligned rectangular region in the image can be obtained by referencing only the corner pixels. For the extended set, a different type of integral image is required (Fig. 3.3 bottom). An important advantage of the features is that the response can be obtained in constant time regardless of the size of the feature in the image. A preprocessing stage is required to create the integral images, though. Similar to other convolution-based features, the Haar features need to be normalized to achieve (at least partial) invariance to lighting conditions, which can significantly increase computational demands. The typical choice of the normalization value is the standard deviation of local intensity for which another integral image is required.

Cg Implementation as a Reference

Cg implementation was created as a reference for LRD implementation introduced further in this chapter. LRD implementations have been compared to Cg implementation, and the results are presented within Chapter 6.

Only the simplest (two-fold) Haar wavelet features were used in this testing implementation (though also three-fold features are used in the object detectors, whose evaluation is slightly slower). The Haar wavelets require normalization by the energy in the classified window – both to evaluate the energy and to evaluate the features themselves, integral images are used, which is the fastest method available to our knowledge. The calculation of the integral images constitutes the preparatory phase evaluated in the comparison. Please note that (to our knowledge) there is no effective way of calculating the integral image in the shading language, so the preparatory phase is implemented in the CPU. The shader evaluating the classifiers is illustrated in Alg. 3.2.

3.1.2 Local Binary Patterns

The Local Binary Patterns (LBP) are widely used in texture processing. They were introduced by Ojala et al. [68] and some improvements have been proposed since then. LBPs in their basic form capture information about local textural structures by thresholding samples from a local neighbourhood by its central value and forming the pattern code (Fig. 3.4). The code is calculated as a weighted sum of the threshold samples. The weights correspond to powers of 2, so each sample sets a single bit in the pattern value.

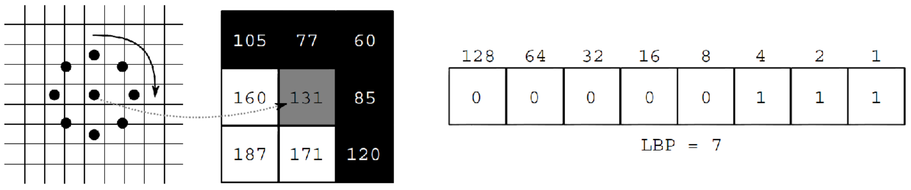


Figure 3.4: Example Of LBP Evaluation: sampling of the neighbourhood (left), thresholding sampled values by the central value (middle) and forming of the LBP code (right).

Typically, the circular neighbourhood with 8 samples is used (8 bit pattern), but other variants are also possible (Fig. 3.5). LBP are most frequently used in combination with local histograms to describe a local image area and segment the image.

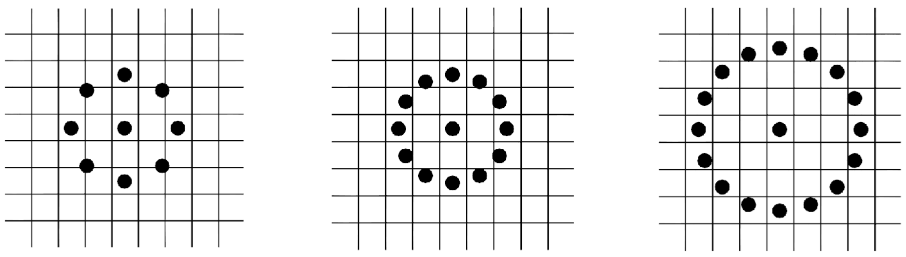


Figure 3.5: Different Sizes Of Local Binary Patterns.

Algorithm 3.2 Evaluation of the Haar-like features in the GPU (Cg).

```
float GetHaar(float2 p0, float2 p1, float2 p2,
              float2 p3, float2 p4, float2 p5,
              uniform samplerRECT IntegTexId)
{
return - texRECT(IntegTexId, p0).a
       + texRECT(IntegTexId, p1).a * 2.0f
       - texRECT(IntegTexId, p2).a
       + texRECT(IntegTexId, p3).a
       - texRECT(IntegTexId, p4).a * 2.0f
       + texRECT(IntegTexId, p5).a;
}

float Horizontal(float2 p0, float2 d, float WIntensity,
                 uniform samplerRECT IntegTexId,
                 uniform samplerRECT AlphaTexId,
                 float HaarId)
{
  float2 dx1 = float2(d.x,0.0f);
  float2 dx2 = float2(d.x+d.x, 0.0f);
  float2 p3 = p0 + float2( 0.0f, d.y);

  float haar = GetHaar(p0, p0+dx1, p0+dx2, p3, p3+dx1, p3+dx2, IntegTexId);
  haar /= d.x*d.y * WIntensity; // Normalization
  haar = clamp((haar+1.0f)*0.5f * 120.0f, 0.0f, 120.0f); // quantization

  return texRECT(AlphaTexId, float2(HaarId, haar)).a;
}

sOutPS FragmentProgram(sVS2PS IN,
                       uniform samplerRECT IntegTexId,
                       uniform samplerRECT IntegSqTexId,
                       uniform samplerRECT AlphaTexId)
{
  sOutPS OUT;
  float window_energy =
             +texRECT(IntegSqTexId, IN.texcoord0).a
             -texRECT(IntegSqTexId, IN.texcoord0 + float2(WND_W, 0.0f)).a
             -texRECT(IntegSqTexId, IN.texcoord0 + float2( 0.0f, WND_H)).a
             +texRECT(IntegSqTexId, IN.texcoord0 + float2(WND_W, WND_H)).a;

  float haarid = 0;
  float sum = 0;

  sum += Horizontal(IN.texcoord0+float2( 0.0f, 0.0f), float2( 8.0f, 8.0f),
                    window_energy, IntegTexId, AlphaTexId, haarid);
  haarid++;
  sum += Vertical  (IN.texcoord0+float2( 3.0f, 3.0f), float2( 2.0f, 8.0f),
                    window_energy, IntegTexId, AlphaTexId, haarid);
  haarid++;

  sum += // ...
  OUT.color.r += sum/haarid;
  OUT.color.a = 1.0f;

  return OUT;
}
```

The LBP is not rotationally invariant, it is dependent on which sample is considered first when forming the code. Rotational invariance can be achieved by normalization of the pattern by shifting the bits – the lowest value is selected as the LBP result. The LBPs exhibit very good performance when used as features in object detection. [100]

3.1.3 Local Rank Functions

The experience with known features, such as Haar features and LBP, suggests that in many cases the classification benefits from the intensity information. On the other hand, the intensity information is subject to changes due to brightness and contrast adjustments of the images while invariance to these changes is very often wanted. This fact causes the applications using features directly based on intensity, such as Haar features, to normalize the image window being classified (e.g. through equalization of its histogram to have a constant energy and zero mean value or through other comparable techniques). However, regardless of the normalization method, the normalization can be very costly from the computational point of view especially comparing it to the cost of, for example, the computation of Haar features evaluation itself. The novel LRF is based on the idea that the intensity information in the image can be well represented by the order of the values (intensities) of the pixels or small pixel regions (e.g. summed 2×2 pixel rectangular areas). This idea is backed by the fact that calculation of the values of features based on the order of pixels is equivalent to (or based on the exact evaluation method at least very close to) normalizing the image through histogram equalization [1] and then evaluation of the feature value based on the pixel or small regions intensities.

The LRF based on the order of pixel values rather than the values of pixels themselves – have several principal advantages over the functions based on the values themselves:

- Invariance to illumination changes – the LRF are invariant to most of the functions used to brightness and contrast adjustments/normalization in the images. More specifically, LRF are invariant to nearly all monotonic gray-scale transformations;
- Strict locality – LRF of objects (parts of objects) do not change locally when the object's image is being captured under changing conditions (similar to for example SIFT);
- Reasonable computational complexity – computation and memory accesses can be optimized thanks to regular geometric structure. No explicit normalization is needed, which is specifically important in some classification schemes, such as WaldBoost ([86]);

Formal Definition of LRF

Let us consider a scalar image $f : \mathbb{Z}^2 \rightarrow \mathbb{R}$. On such image, a *sampling function* can be defined ($\mathbf{x}$, $\mathbf{u} \in \mathbb{Z}^2$, $g : \mathbb{Z}^2 \rightarrow \mathbb{R}$):

$$S_x^g(\mathbf{u}) = (f * g)(\mathbf{x} + \mathbf{u}) \tag{3.1}$$

This sampling function is parametrized by convolution kernel g which is applied before the actual sampling, and by vector $\mathbf{x}$ which is the origin of the sampling. Next, let us introduce a vector of relative coordinates ($n \in N$):

$$\mathbf{U} = [\mathbf{u}_1 \mathbf{u}_2 \dots \mathbf{u}_n],\ \mathbf{u}_i \in \mathbb{Z}^2 \tag{3.2}$$

This vector of two-dimensional coordinates can define an arbitrarily shaped neighbourhood and it will be used together with the sampling function to obtain a vector of values describing the neighbourhood of this shape on position x in the image:

$$\mathbf{M} = [S_{\mathbf{x}}^g(\mathbf{u}_1)\, S_{\mathbf{x}}^g(\mathbf{u}_2)\ \dots\ S_{\mathbf{x}}^g(\mathbf{u}_n)] \tag{3.3}$$

This n-tuple of values will be referred to as the *mask* in the following text. The term mask is reasonable as the vector was created by “masking” global information from the image and leaving only specific local information. Note that in general, the sampling function does not have to be uniform over the mask:

$$\mathbf{M} = [S_{\mathbf{x}}^{g_1}(\mathbf{u}_1)\, S_{\mathbf{x}}^{g_2}(\mathbf{u}_2)\ \dots\ S_{\mathbf{x}}^{g_n}(\mathbf{u}_n)] \tag{3.4}$$

but the implementations described in this text all use the uniform sampling function.

For each element k in the mask, its rank can be defined as:

$$R_k = \sum_{i=1}^{n} \begin{cases} 1 & \text{if } M_i < M_k \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

i.e., the rank is the order of the given member of the mask in the sorted progression of all the mask members. This way an n-tuple of ranks R is obtained. Note that the ranks are independent on the local energy in the image.

On the n-tuple of ranks R, a variety of functions which extract discriminative information can be defined. These LRF have the form:

$$LRF : \mathbb{Z}^n \rightarrow \mathbb{Z} \tag{3.6}$$

One of the possible variants of LRF is the Local Rank Pattern (LRP) image feature

[35], which selects two specific ranks and encodes their values. The LRP from their nature produce a large set of possible results, which can in the context of recognition/detection cause problems when only small training datasets are available and when the memory available on the target computational platform is limited. One way to deal with this issue - and to shrink the output set of the image features - is the Local Rank Differences (LRD). The LRD computes the difference of two ranks which is very similar to the Haar-like features with added (Fig. 3.2) with added local image contrast normalization.

These are just brief definitions of LRP and LRD. However, both LRD and LRP were the subject of our interest and research and are further explained in the next sections of this chapter (3.2,3.3).

3.2 Local Rank Patterns

Local Rank Patterns (LRP) are low-level image features introduced in [35] and described in detail in [32]. They were designed to constitute an alternative of the commonly used Haar wavelets, which would be suitable for hardware implementations (in FPGA and ASIC chips). Though designed for implementation by circuitry, they perform very well also when implemented on processors and graphics chips.

The LRPs are based on the idea that the intensity information in the image can be well represented by the order of the values (intensities) of the pixels or small pixel regions (e.g. summed 2×2 pixel rectangular areas).

Our research addressed in [32] and [30] presents the LRP low-level image feature extractor and its efficient implementations on several hardware architectures.

Formal Definitions of LRP

Local Rank Patterns [35] are defined as:

$$LRP(a,b) = nR_a + R_b,\ a,b \in 1,\dots,n \tag{3.7}$$

Note that n is the number of samples taken in the neighbourhood and therefore the result of LRP is unique for each combination of values of the two ranks R_a and R_b. This fact suggests an alternative definition of the LRP when we allow the results of LRP to be pairs of values instead of a single value:

$$LRP(a,b) = [R_a\, R_b] \tag{3.8}$$

The LRP have some interesting properties which make them promising for image pattern recognition. Mainly, LRP are invariant to monotonous gray-scale changes such as changes of illumination intensity. This invariance results from using ranks instead of absolute values to compute the value of the feature. In fact, using the ranks has the same effect as locally equalizing the histogram of the convolved image $f * g$.

Further, LRP are strictly local – their results are not influenced by image values outside the neighbourhood defined by $\mathbf{U}$. This is a clear advantage over wavelet features (e.g. Haar-like features) which, in the way they are commonly used, need global information to normalize their results. This locality makes the LRP highly independent, for example, on changes of background and on changes of intensity of directional light.

The meaning of the values produced by the LRP can be understood in two ways. First and most naturally, the results give information about the image at the locations of the two ranks $x + u_a$ and $x + u_b$ and information about their mutual relation. On the other hand, the results also carry information about the rest of the neighbourhood, especially if the neighbourhood is small. In such cases the results of LRP carry good information about the local pattern in the image.

In the previous text, the LRP have been defined for two-dimensional images. However, the notation allows very a simple generalization for higher-dimensional images by changing the dimensionality of $\mathbf{x}$, $\mathbf{u}$ and of the relative coordinates in $\mathbf{U}$ to Z^3 for 3D or Z^{k} for general dimensionality. Furthermore, it is possible to use more than two ranks to compute the results of the LRP. For example:

$$LRP(a, b, c) = R_a \cdot n^2 + R_b \cdot n + R_c \tag{3.9}$$

3.2.1 Implementations

As the core of this thesis was to find the possibility of using CUDA in object detection, the main implementation of LRP was CUDA implementation. Consequently was this CUDA implementation compared to SSE implementation on multi-threaded CPU.

CUDA Approach

The efficient implementation solves problems of two main domains: the classifier operating on one fixed-size window and parallel execution of this classifier on different locations of the input image. Making the object detector with these two issues separate simplifies the design. However, some extra speed-up could possibly be gained from exchanging information between different classifier instances. The implementation presented in this section keeps the classifier instances as “black boxes” and does not share information between them. Experiments with sharing the information lay outside the scope of this research.

The problem of object detection by statistical classifiers (from the CUDA implementation point of view) can be divided into following steps:

1. loading and representing the classifier data;
2. image pre-processing;
3. object detection and
4. retrieving results.

Loading and Representing the Classifier Data The constant data containing the classifier (image features' parameters, prediction values of the weak hypotheses summed by the algorithm, WaldBoost thresholds) could be accommodated in texture memory or constant memory of the CUDA architecture. This data is accessed on evaluation of each feature at each position, so the demand for access speed is critical. Although the access would be slightly simpler and faster if the data was stored in the texturing memory of the CUDA environment, the experiments showed that the overall detection times are better when the classifier data is stored in the constant memory. This is mainly because the image is stored in the texturing memory and is heavily accessed, so offloading the access to the classifier data to the constant memory relieves a bottleneck of the system. The constant memory (as well as the texturing memory) is cached and the referencing to the classifier data exhibits a large locality of reference – all the threads are typically processing the same weak classifier.

Input Image Pre-Processing The classifier is trained on a training dataset of fixed-scale examples. To be able to detect the object in different scales, the image must be scanned in multiple resolutions. The common approach benefits from the ability of the Haar wavelets calculated using the integral image to be evaluated in arbitrary scales in constant time. The LRP features could be evaluated in a similar manner as well, but experiments showed that especially on the graphics card, it is notably more efficient to construct a multi-resolution pyramid from the input image and scan it by the detector. See Fig. 3.6 for an illustration of how the pyramid is built. Note that some pixels of the pyramidal image, which is the actual input of the detection algorithm itself, are left unused. More compact layouts of the images of different resolutions could possibly be found and the amount of the unused pixels could be slightly reduced. However, thanks to the nature of the WaldBoost algorithm, only a very small number of weak classifiers

(˜2) are evaluated on the unused locations, which are filled with a constant colour. The time spent on evaluation of these areas is a tiny fraction of the whole processing time and sparing a fraction of this amount would not be worth the relatively complicated and error-prone layout algorithm.

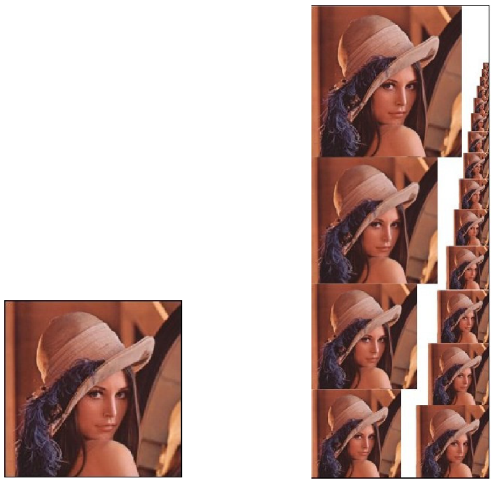

(a) Original image (b) Multi-res pyramid

Figure 3.6: Multi-Resolution Pyramid Constructed from the Input Image.

OpenGL rasterization is used for creation of the multi-resolution pyramid: a pixel buffer object of sufficient resolution is created and the input image is rendered for each scale. After the rendering is done, this pixel buffer is converted into a CUDA texture (Alg. 3.3).

Algorithm 3.3 Pixel Buffer conversion to CUDA texture.

```
cudaGLRegisterBufferObject( PixelBufferObject );
cudaGLMapBufferObject( CudaData, PixelBufferObject );
cudaMallocArray( CudaArray, ...  );
cudaMemcpyToArrayAsync( CudaArray, CudaData);

cudaBindTextureToArray( CudaTexture, CudaArray);
```

Object Detection – Overall Algorithm Design Programs that are run on the graphics hardware using CUDA are executed as kernels, each kernel has a number of blocks and block is further organized into threads. The code of the threads consumes hardware resources: registers and shared memory; this limits the number of threads that can be efficiently executed in a block (both the maximal and minimal number of threads).

One thread computes one or more locations of the scanning window in the image. One thread could as well perform a task of smaller granularity - e.g. one or more weak classifiers, but that would imply too much inter-thread communication. The image pixels (or window locations, more precisely) are therefore divided into groups which are calculated by the threads. The final solution divides the image into rectangular tiles which are solved by different thread blocks (see Fig. 3.7). We have been experiencing with various layouts of the position-thread assignments, but this design is both simple and achieves no less performance than any other design experimented with.

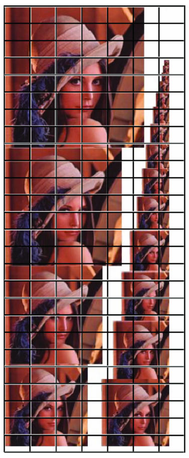


Figure 3.7: Blocks of Threads.

Experiments showed that the suitable number of threads per block is around 128. Executing blocks for only 128 pixels of the image would not be efficient, so we choose that one thread calculates more than one pixel - a whole line of pixels in the rectangular tile. A nice consequence of this layout is an easy control of the resources used by one block: the number of threads is determined by the height of the tile, the width controls the whole number of processed window positions by the block. The tile can extend over the whole width of the image or just a part of it. Because of thread rearrangement described below in 3.2.1, the total number of pixels processed by one thread block is limited proportionally to the size of the shared memory (fast memory in one multiprocessor, which is shared between the threads of one block), and so the image is divided vertically into several columns of tiles.

When the kernel is started, the image data are referenced by texturing units from the multi-resolution pyramid and the parameters of the classifier are read from the constant memory. When object is recognized at window position, the coordinates are written to the global memory. To avoid collisions of concurrently running threads and blocks, atomic increment (`atomicInc()`) of one shared word in the global memory is used for synchronization. This operation is rather costly, but the positive detections are so rare that this means of output can be afforded. As a consequence, the results of the whole process are at the end available in one spot of the global memory, which can be easily fetched to the host computer. The whole architecture is depicted in Fig. 3.8.

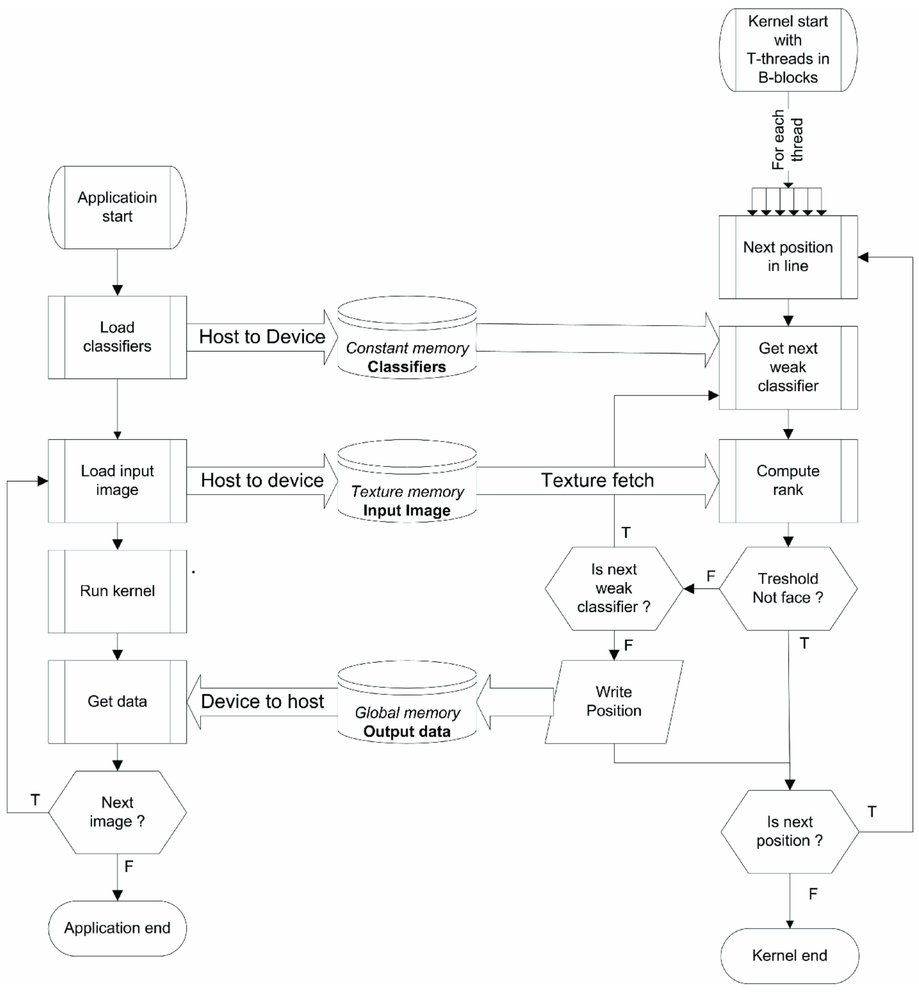


Figure 3.8: CUDA Object Detection Architecture. On the left side of the figure is the host process, on the right is the device kernel.

Thread Rearrangement The CUDA architecture imposes some requirements on the threads to run efficiently. Because of the SIMD (single instruction, multiple data) nature of CUDA, at one time the threads must perform identical operations. In case of branching, the threads are split into groups according to the variant of code they execute, and the groups of identical execution paths are run separately. Not all threads in the block are handled in this manner, but the threads are organized into *warps* - groups of threads of fixed count (32 in current hardware implementations). Organization of the threads into the warps is done at kernel start and the threads remain in a warp till their end.

The scanning classifiers indeed execute identical code - they load image data from identical positions (differing only by an additive offset), they evaluate identical weak classifiers, compare the intermediate sum to identical thresholds etc. However, due to the (desired) focus-of-attention capability of WaldBoost, some threads terminate with negative decision earlier than others (Fig. 3.9), but the warp continues to evaluate until the very last thread terminates. This leads to relatively low utilization of the hardware resources.

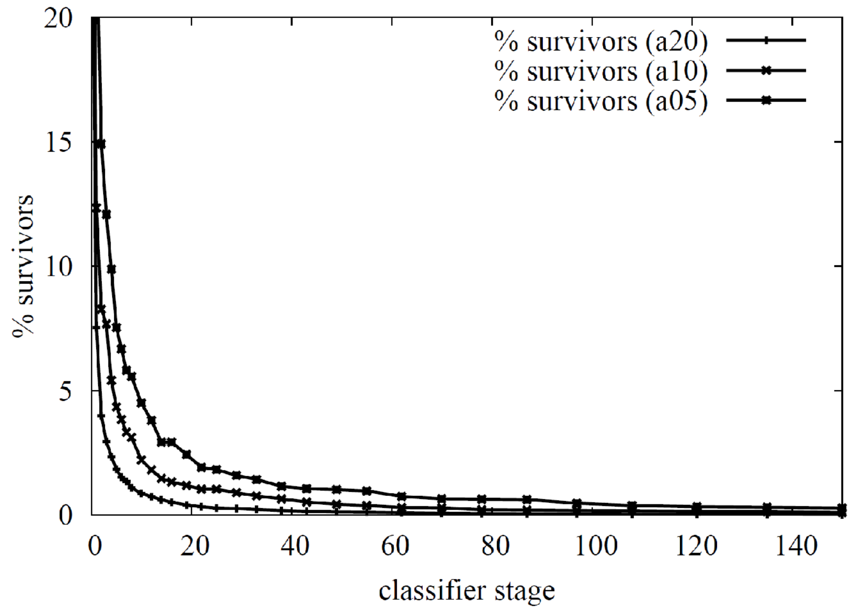


Figure 3.9: Fraction of locations in the image still evaluated after a number of evaluated image features. After the first evaluated feature, 60 – 70% (depending on the used classifier) locations are eliminated. The classifier is trained with different target false negative rate ($a20$, $a10$, $a05$).

For illustration, Fig. 3.10 contains the situation in a block after 10 weak classifiers evaluated – white pixels indicate that the classifier evaluation was terminated, blue pixels indicate positions still evaluated. Note that the threads are arranged into warps of 32 threads and all threads within one warp must evaluate the same code path or wait for the others. In this case it means that the majority of threads is waiting for several threads exploring a fraction of the image; note that this happens in each column again. However, the situation is not tragic thanks to locality of reference, i.e. that the threads evaluate locations close to each other and the responses of the classifiers are therefore highly correlated.

To address this issue, we propose thread rearrangement: at some stage of the classifier, all locations in the image that have not been classified as negative are written into a memory block shared between the threads, and another phase of the classification is started, that processes only these locations. This rearrangement can be performed several times during the whole classification process ($\sim 500 - 1{,}000$ stages). See Fig. 3.11 for an illustration of two rearrangements.

The intermediate positive (more accurately not-yet-negative) samples are stored into the shared memory of the multiprocessor similarly as the final detections are written to the global memory, as described above. The shared memory is very fast (as fast as the

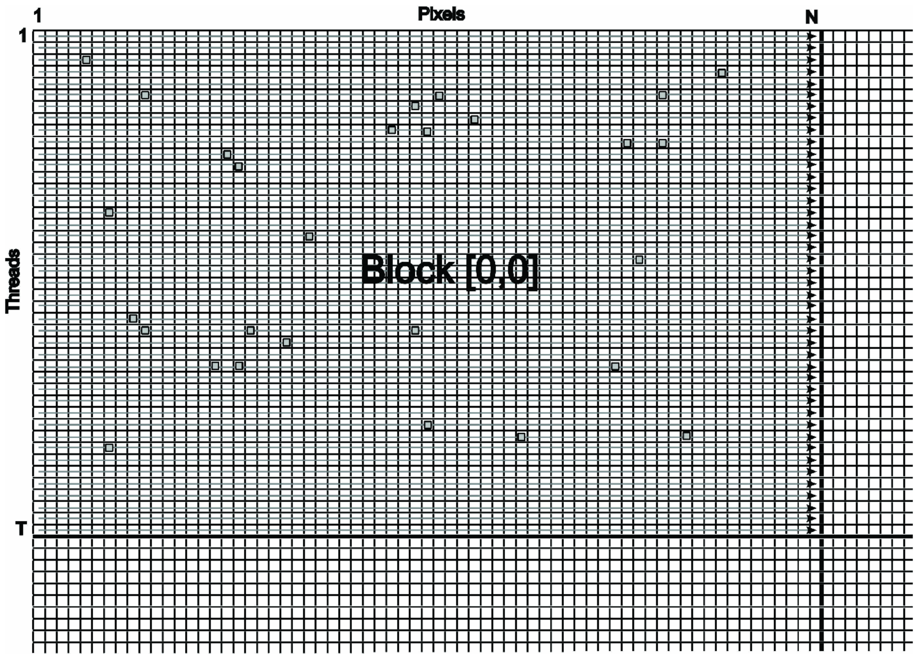


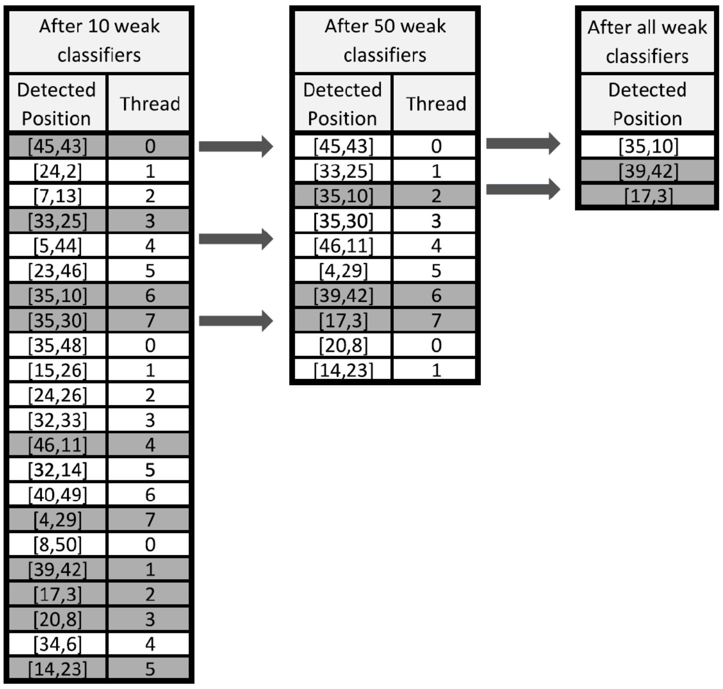
Figure 3.10: Remaining Candidates for Positive Response after 10 Weak Classifiers.

registers) and even the instruction of atomic increment in the shared memory is not as costly as in the case of global memory. The scope of accessibility of the shared memory is only within one block of threads, which is only appropriate, because the rearrangement happens within one block.

The exact count and locations of the rearrangement steps needs to be determined experimentally. Analytical expressions can be sought for, that would determine these from some characteristics of the algorithm and the platform. Such expressions, however, would depend on many variables: cost of one weak classifier, cost of the rearrangement, speed of the classifier in different phases of the classification process, locality of information in the processed image and many others and still would be only crude approximations. Further in this chapter are described experiments carried out to determine an optimal locations rearrangement and the discussion on the measurement results. Generally, the major influence of the rearrangements is during the beginning of the classifier, because the most of the locations are dropped out very early (see Fig. 3.9) and only a small fraction of computational load remains to the further stages.

Considerations of Alternative Algorithm Designs The purpose of this section is to mention several elements of the algorithmic design that were considered for the object-detection architecture but were found to be inferior to the solution described above.

Many efficient image processing CUDA implementations use the shared memory for



After 10 weak classifiers	
Detected Position	Thread
[45,43]	0
[24,2]	1
[7,13]	2
[33,25]	3
[5,44]	4
[23,46]	5
[35,10]	6
[35,30]	7
[35,48]	0
[15,26]	1
[24,26]	2
[32,33]	3
[46,11]	4
[32,14]	5
[40,49]	6
[4,29]	7
[8,50]	0
[39,42]	1
[17,3]	2
[20,8]	3
[34,6]	4
[14,23]	5

After 50 weak classifiers	
Detected Position	Thread
[45,43]	0
[33,25]	1
[35,10]	2
[35,30]	3
[46,11]	4
[4,29]	5
[39,42]	6
[17,3]	7
[20,8]	0
[14,23]	1

After all weak classifiers
Detected Position
[35,10]
[39,42]
[17,3]

Figure 3.11: Thread Rearrangement after 10 and after 50 Weak Classifiers.

storing the processed image. The shared memory is very fast and is dozens of kilobytes large – tiles of the processed image can be loaded into it and processed by thread blocks. We have tried variants of this arrangement and experiments show that using the texture memory is more efficient. The texturing units perform bilinear interpolation between neighbouring pixels, which can be used for evaluation of LRP. Most importantly, when using the texturing memory, the execution is as fast as when using shared memory (apparently because the bottleneck is in the calculation, not memory access), and the shared memory remains spared for other helpful purposes, as is the thread rearrangement above.

As discussed in the previous section, one of the factors limiting the performance is that the evaluation of different locations in the image is terminated after varying number of stages of the classifier and due to the SIMD nature of CUDA some threads are idly waiting. We have tried several arrangements, where the threads are assigned the work dynamically, so that when the evaluation at one location terminates, the thread "asks for" another location in the image and processes it. The idea is that the work unit would not be one location in the image, but one weak classifier. The control required by this arrangement and especially the need to synchronize the threads seems to be too complex and these attempts were much slower than the finally achieved solution with the thread rearrangement (although some threads are still idle).

We have made several experiments (see 3.2.1) with the placement and representation of the classifier data (constant for all images and all locations in the images). A texture could be used for storing it, shared memory or constant memory. Both texture memory and constant memory were cached; shared memory was very fast by itself. Placement of the classifier data into the shared memory required pre-loading it upon start of each block from another location and so it was the least efficient solution. The rest two options (texture memory or constant memory) seemed to be performing equally well, so storing the classifier in the constant memory was preferred to offload the texturing units which were used for accessing the pyramidal image.

SSE Approach

The performance of the CUDA implementation was evaluated in comparison to an efficient SSE implementation of the same classification principle. For details on the implementation please refer to [33]; these paragraphs will summarize briefly its main characteristics.

This implementation addresses two crucial issues: memory accesses performed by the algorithm (minimizing the number of memory accesses and ensuring their speed by aligning the operands) and the algorithmic evaluation of the local ranks and their differences. It uses the SSE2 instruction set which has extensive support of instructions working with sixteen 8bit values in a single 128bit register.

To simplify feature evaluation as much as possible, the convolutions of the input image with the sampling function were pre-computed and stored in the memory in such a manner that all the results of the LRP grid could be fetched into the CPU registers through two 64bit loads. Compared to a naive LRP implementation, the described implementation benefits from parallel processing when calculating the ranks. The disadvantage was the limited number of convolution kernels because for each grid size a separate pre-calculated image was required. In our experiments we used four feature sizes 1×1px, 1×2px, 2×1px and 2×2px and therefore four interleaved convolution images needed to be pre-computed.

The evaluation part (Fig. 3.12) first expands selected values (A and B) to full 128bit length. The value of A (resp. B) is then compared to all other values loaded from the sampling function. Comparison result is masked and the result is summed – the number of positive comparisons corresponds to the rank of A (resp. B). Results for A and B are then combined to produce the LRP value.

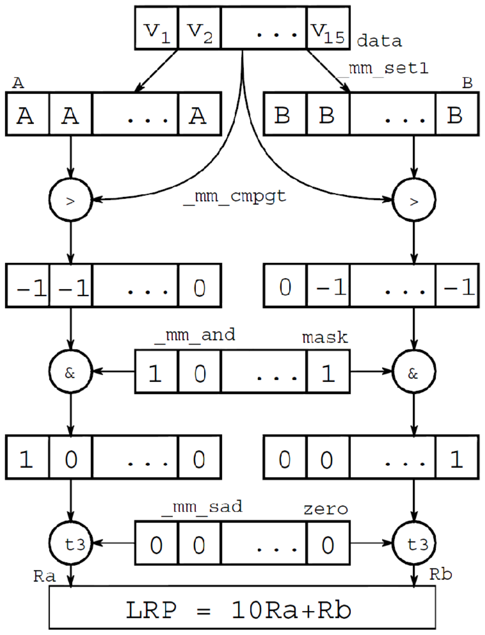


Figure 3.12: Evaluation of LRP using SSE instruction set of Intel CPU. The input is a vector data of 16 values, mask and indexes A, B of values from which LRP is calculated.

3.3 Local Rank Differences

An algorithms exhibits real-time performance in detecting complex patterns, such as human faces [92], while achieving precision of detection which is sufficient for practical applications. Work of Sochman and Matas [87] even suggests that any existing detector can be efficiently emulated by a sequential classifier which is optimal in terms of computational complexity for desired detection precision. In their approach, human effort is invested into designing a set of suitable features which are then automatically combined by the WaldBoost [93] algorithm into an ensemble. This approach may significantly reduce the development time of detectors and it may even lead to more computationally efficient detectors - Šochman and Matas report successfully emulating the Kadir-Brady saliency detector [44], while achieving 70× faster detection times over the original implementation.

In practical applications, the speed of the object detector or other image classifier is crucial. Real-time performance is required in many applications such as surveillance, even when processing several input streams. Use of specialized hardware in image processing and computer vision is nothing new (e.g.[81], [55]). The advances in development of graphics processors, at the time of our research, were attracting many researchers and engineers to the idea of using GPU's not for their primary purpose - rendering 3D graphics scenes. Different approaches to so-called GPGPU [69] existed and also the field of image processing and computer vision have had seen several successful uses of these techniques (e.g.[81], [55]).

Statistical classifiers were built by using low level weak classifiers or image features and the properties of the classifier largely depended on the quality and performance of the low level features. In face detectors and similar classifiers, Haar-like wavelets [52], [93], [87], [92] are frequently used, since they provide good amount of discriminative information and they provide excellent performance. Other features are used in different contexts, such as the Local Binary Patterns [68]. Recently, designed especially for being implemented directly in programmable or hard-wired hardware, Local Rank Differences [35] have been presented. These features are described in more detail in section 3 of this paper. The main strengths of this image feature are inherent gray-scale transformation invariance, the ability to capture local patterns and the ability to reflect quantitative changes in lightness of image areas.

Prior to this GPGPU [69] in CUDA [60] implementation and related research, we have implemented the LRD features in the GPU as shaders [73]. The Cg implementation was fairly efficient, the main disadvantage was the need of complicated control of the rendering pipeline from the CPU (by issuing commands to render quads, lines or other primitives in a complex pattern that covered the searched area of the image). This disadvantage was minimized by the properties of the GPGPU philosophy. The CUDA implementation presented in [31], compared to the Cg one [73] benefits also from some memory arrangement improvements, from improved training process and other minor advances.

The following part of this section briefly presents the Local Rank Differences (see [35] for more detail) image feature.

3.3.1 Formal Definition of LRD

The LRP from their nature produce a large set of possible results, which can in the context of recognition/detection cause problems when only small training datasets are available and when the memory available on the target computational platform is limited. One way to deal with this issue – and to shrink the output set of the image features – are the Local Rank Differences (Polok et al., 2008), which can be defined as:

$$LRD(a, b) = R_a - R_b \tag{3.10}$$

The LRD computes the difference of two ranks which is very similar to the Haar-like features (Fig. 3.2) with added local image contrast normalization.

The definition of the LRP (and LRD) which was given in the previous text is very general. It allows arbitrary sizes and shapes of the neighbourhoods and arbitrary convolution kernels. However, we can define a set of LRP which is suitable for creating classifiers for detecting objects in images – which is both informative and efficient to compute. This particular version is used in the reported experiments.

Fig. 3.13 shows the simplified flow for evaluating a single LRD classifier. It begins with the detection window (e.g. 31×31 pixels) being classified where rectangular mask M_{xy}^{mn33} is positioned (considering e.g. 3×3 masks). Each field of the mask spans across several pixels which need to be convolved (see Eq. 3.11 below). Next, the ranks are evaluated and finally the rank difference is used as index into the alpha table, selecting the weak classifier's result.

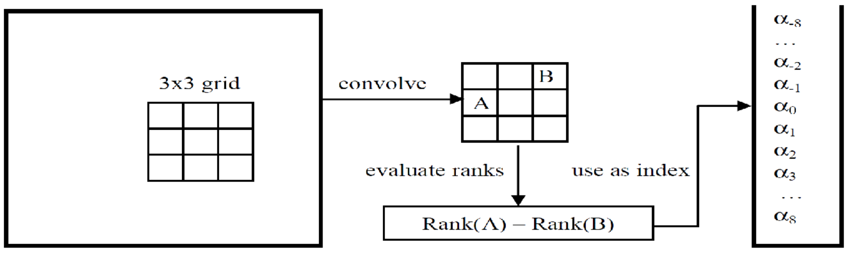


Figure 3.13: Use of Local Rank Differences in the Classifier.

Input Image Pre-Processing

For increasing the performance of the LRD evaluation, the function S_{xy}^{mn} defined on the input image can be pre-calculated. As stated above, low number of combinations of$m \times n$ is sufficient for learning an object classier - experiments show that 1×1, 2×2, 2×4 and 4×2 combinations are enough. The input image I can be convolved with:

$$h_{2\times 2} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}, \qquad h_{4\times 2} = \begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \end{bmatrix}, \qquad h_{w\times h} = \begin{bmatrix} \frac{1}{wh} & \cdots & \frac{1}{wh} \\ \vdots & \ddots & \vdots \\ \frac{1}{wh} & \cdots & \frac{1}{wh} \end{bmatrix} \tag{3.11}$$

and the resulting images at given location (x, y) can contain the values of the sampling function. Such pre-processing of the input images can be done efficiently and the LRD evaluation then only consists of 9 look-ups to the memory (for the case of 3×3 LRD mask) into appropriate pre-processed image and then evaluation of ranks for two members of the mask. The evaluation then can be done in parallel on platforms supporting vector operations; both GPU and FPGA are strong in such kind of parallelism.

3.3.2 LRD Compared to Haar Wavelets

Comparing LRD with Haar wavelets is only natural as both of these types of features were first intended to be used in detection classifiers. There are two fundamental aspects in respect to the detection classier which must be addressed: the computational complexity of evaluating the features and the amount of discriminative information the features provide.

Haar wavelets can be computed very rapidly on general purpose CPUs by using the integral image representation [92] which can be created in a single pass through the original image. The simple Haar wavelets of any size can be computed using only six accesses into the integral image, six additions and two bit-shifts. When scanning the image in multiple scales, this gives the possibility to scale the classier instead of down-sampling the image. The Haar wavelets are usually normalized by the size of the feature and the standard deviation of pixel values in the classified sub-window. Computation of the standard deviation requires additional integral image of squared pixel values and uses square root.

While the Haar wavelets can be computed relatively efficiently on general purpose CPUs, it may not be the same on other platforms. On FPGAs, the six random accesses into memory would significantly limit the performance (only single feature evaluated per every six clock cycles) and the high bit-precision needed for representing the integral images would make the design highly demanding. On the other hand, the nine values needed to compute LRD with grid size 3×3 can be obtained on FPGAs with only single memory accesses [35] (when preprocessed as shown in 3.3.1) and on GPUs with three or six accesses [73].

Some detection classifiers evaluate on average very low number of features (even less than 2). In such cases, computing the normalizing standard deviation poses significant computational overhead. Further, the square root which is needed cannot be easily computed on FPGAs. The LRD inherently provide normalized results, whose normalization is in fact equivalent to local histogram equalization.

The detection performance of classifiers with the LRD has been evaluated on the frontal face detection task and it has been compared to the performance of classifiers with the standard Haar features. The results suggest that the two types of features provide similar classification precision. One of the two classifiers compared in (Fig. 3.14) uses the same Haar wavelets as in [92] and the other uses the LRD with block sizes of the sampling function (Eq. 3.3) restricted to 1×1, 1×2, 2×1 and 2×2. The classifiers were trained using 5000 hand annotated faces normalized to 24×24 pixels and the non-face samples were randomly sampled from a pool of 250 million sub-windows from more than 3000 non-face images. The results were measured on a set of 89 group photos which contain 1618 faces and total 142 million scanned positions (scale factor 1.2, displacement 2/24). Although the set of LRD features is very limited in this experiment, the detection performance it provides is similar to the full set of Haar wavelets. This is probably due to the localized normalization of the results of the LRD which provides information about local image patterns that goes beyond simple difference of intensity of image patches.

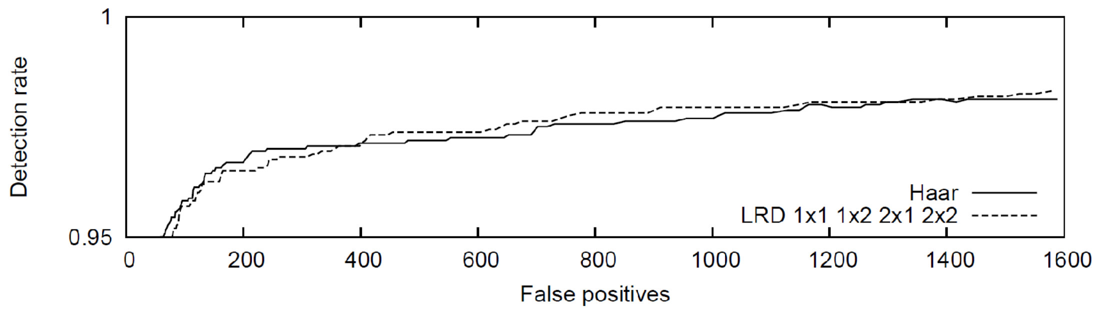


Figure 3.14: ROC of two WaldBoost classifiers on a frontal face detection task. Length of the classifiers is 500 and they differ only in type of features which they use (Haar features, LRD).

3.3.3 Implementations

Since previous section introduced several LRP implementations, there are also several approaches to LRD. The goal of the particular LRD implementations was the comparison of their effectiveness. Apart from those mentioned below, two more straight-forward (without any optimization) implementations has been developed:

- "Simple" LRD implementation on CPU, and
- "Simple" Haar implementation on CPU.

These "Simple" implementations are due to their simplicity not addressed within this thesis.

CUDA Approach

CUDA approach is already described in 3.2.1. The only difference is that in this section, the LRD evaluation is being used.

The implementation of the LRD using CUDA corresponds with the theoretical description of the LRD in a straightforward way. It appears that a wise choice is relying on the combinations 1×1, 1×2, 2×1 and 2×2 of the LRD sampling function. Such sampling limits the descriptive power of the features slightly, but allows nice performance improvements. Thanks to the built-in texture sampling with bilinear interpolation on the usable graphics cards, sums of 2 neighbouring pixels in vertical or horizontal direction or sum of four neighbouring pixels consume the same amount of time as sampling just one source pixel. The scanned image can be used in such way without any pre-processing

Algorithm 3.4 The central part of the CUDA implementation code. The LRD() function loops over all the weak classifiers in the boosted cascade (stored in a 1D texture), gets the rank difference (by calling GetRankDi(...) and uses the difference as an index to the table of alpha values obtained by training the classier.

```
__device__ int GetRankDiff(
    unsigned int posX, unsigned int posY,
    unsigned int BlockSizeId, unsigned int BlockABId)
{
  unsigned int mempos = threadIdx.x*9; // address to the temp mem
  float uiBlockWidth = _uiBlockSizeId >> 3; // mask size
  float uiBlockHeight = _uiBlockSizeId & 7; // mask size

  // current pixel [px,py]
  float px = posX + AbsX1 + float(BlockSizeId >> 3)/2.0f;
  float py = posY + AbsY1 + float(BlockSizeId & 7)/2.0f;

  // get sums of each matrix block (1x1, 1x2, 2x1, 2x2)
  s_fBlockSum[mempos+0] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+1] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+2] = tex2D(tImage1, px, py).x;

  px -= 2.0f*uiBlockWidth; py+=uiBlockHeight; // shift to next line

  s_fBlockSum[mempos+3] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+4] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+5] = tex2D(tImage1, px, py).x;

  px -= 2.0f*uiBlockWidth; py+=uiBlockHeight; // shift to next line

  s_fBlockSum[mempos+6] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+7] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+8] = tex2D(tImage1, px, py).x;

  // compute the rank difference between blockA and blockB
  int iRank = 0;
  unsigned int uiBlockA = _blockABId >> 4;
  unsigned int uiBlockB = _blockABId & 15;
  for (unsigned int bi = 0; bi < 9; bi++)
  {
    if (s_fBlockSum[mempos+bi] < s_fBlockSum[mempos+uiBlockA]) iRank--;
    if (s_fBlockSum[mempos+bi] < s_fBlockSum[mempos+uiBlockB]) iRank++;
  }

  return iRank;
}

__device__ unsigned char LRD()
{
  float ret = 0.0f;
  // loop over weak class
  for (unsigned cid=0; cid < WCCount-1; cid++)
  {
    uint4 w0 = tex1D(tWeakParam, cid); // get WeakClassifier parameters
    // Compute WeakClassifier rank and convert it to predictor value
    ret += tex2D(PredValues, GetRankDiff(w0.x, w0.y, w0.z, w0.w)+8, cid);
  }

  return (unsigned char)ret;
}
```

stage. The following Alg. 3.4 contains the central part of the CUDA code implementing the LRD evaluation.

Compared to the previously published Cg implementation of the LRD [73], the CUDA offers some advantages. The biggest problem of the shader version was the need for rather complicated drawing of geometric primitives on the "screen" to control the object detection process. The whole of the input image needs to be covered by the primitives, but for efficiency reasons, simple drawing of one rectangle of the same size as the input image was not possible. In the GPGPU version, all the coding and control is simpler and more straightforward. As shown further in this chapter, the price to pay for such feasibility of programming is the performance, or rather performance distribution depending on the input size and on the count of the weak classifiers.

MMX Approach

The performance of the GPU implementation was compared to an implementation on standard Intel CPU using MMX instructions. To simplify feature evaluation as much as possible, the convolutions of image are pre-computed and stored in the memory in such manner that all the results of the LRD grid can be fetched into the CPU registers through two 64-bit loads. This positively affects the evaluation that is performed in MMX CPU instructions (introduced by Intel).

A pseudo-code of the MMX implementation is shown in Alg. 3.5 and the block diagram of the evaluation is shown on Fig. 3.15. The LRD are parametrized by the feature's position (x, y) and the block size (w, h) which determine the convolution image to use. First the data from the subsequent rows of the convolved images are loaded into registers (*row1*, *row2*). The values of the rank pixels are loaded from the data (*pixelA*, *pixelB*) and expanded to the MMX registers. The registers with the data are then compared to the expanded values of pixelA and pixelB and the result of the comparison is masked (since we are interested in 3×3 grid only and 4×4 pixels were loaded). The comparison's results are summed – the resulting registers, therefore, contain the rank sum of differences of a pixel and vale A and B. Finally, the 8-bit values in the resulting registers are summed together which corresponds to the LRD response.

The code, compared to CPU without MMX, is more optimal since the values are compared in one step. The slowest step of evaluation is the expansion of 8 bit value to the 64 bit MMX register. Since the instruction set lacks a single instruction to do this, the expansion must be done by a sequence of shift-left and or instructions. A similar problem is the final sum of rank differences - eight 8 bit values in a register must be summed together. Again, there is no support in instruction set.

Algorithm 3.5 Pseudo-code of the MMX implementation of the LRD.

```
row1 = convolution_{w,h}(x, y)
row2 = convolution_{w,h}(x.  y+1)
pixelA = (A < 8) ?  row1[A] : row2[A-8];
pixelB = (B < 8) ?  row1[B] : row2[B-8];
mm0 = expand(pixelA)
mm1 = expand(pixelB)
mm2 = load(row1)
mm3 = load(row2)
mm4 = cmp(mm2, mm0)
mm5 = cmp(mm2, mm1)
mm6 = cmp(mm3, mm0)
mm7 = cmp(mm3, mm1)
mask(mm4, valid0)
mask(mm5, valid1)
mask(mm6, valid0)
mask(mm7, valid1)
mm4 = add(mm4, mm6)
mm5 = add(mm5, mm7)
mm0 = sum_pi8(mm4)
mm0 += sum_pi8(mm5)

return mm0
```

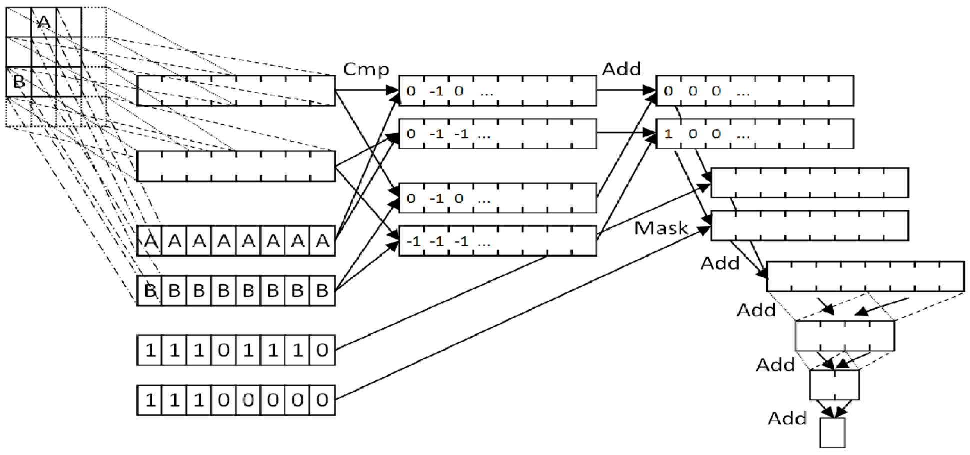


Figure 3.15: Block Diagram of the MMX Implementation of the LRD.

GPU(Cg) Approach

As shown in 3.3.1, the sampling function for a given sampling block size used by the LRD can be pre-processed by convolving the original input image by a simple convolution matrix. On GPU, built-in texture sub-sampling can be used to achieve this pre-processing efficiently. This is done using very simple fragment shaders and the whole convolution calculation usually takes less than 10% of frame time and was not further optimized.

The step that uses the pre-calculated images is the evaluation of the LRD weak classifiers. Early analysis of the algorithm revealed that its bottleneck would be texture sampling. Therefore, the main goal was to minimize the number of texture samples per pixel and to improve texture sampling coherency in order to achieve the best performance. A trick was used to do this – interleaving the convolution image into different layers of a

3D texture. The dimensions of the texture are:

$$w_t = \frac{w_i}{m} \qquad h_t = \frac{h_i}{n} \qquad d_t = mn \tag{3.12}$$

Where w_i, h_i are the input image's dimensions, m, n are the sampling block's dimensions and w_t, h_t, d_t is the texture size. The texture organization is illustrated in Fig. 3.16. Such way of storing image data ensures the texture samples needed to evaluate single LRD classifiers are tightly connected to each other.

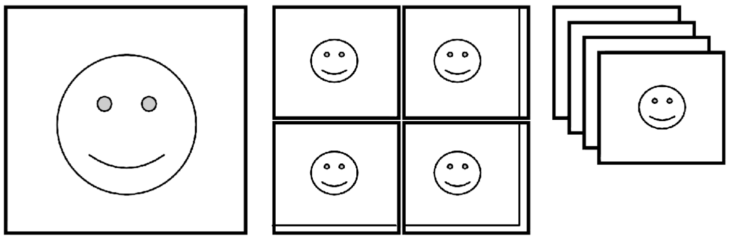


Figure 3.16: (from left to right) Original image, interleaved convolution images (for 2x2 kernel) and interleaved images stored as a 3D texture.

To read the 3×3 LRD mask in a naive way, nine texture samples are needed; however, most of today's hardware is not capable of loading nine samples without stalling the pipeline. To avoid this limitation, the (8-bit grayscale) pixels of the convolution texture are packed by four into RGBA vectors stored in the texture memory. Then it takes three or six texture samples, depending on the modulo 4 position, to read all the nine pixels of the mask (in contrast to the nine reads without the use of 3D texture).

Pixel unpacking is done in the fragment shader and it needs to choose one of four different branches. It could be solved by a simple if statement, but the (expensive) branching instruction can be avoided by rasterizing the image in vertical stripes, one pixel wide and four pixels apart, using a different shader for each modulo 4 position.

Having read the 3×3 grid, the next step is to evaluate the local ranks. The SIMD nature of the GPU can be exploited by keeping the pixels in three 3D vectors. First, the pixels on positions a and b are picked. Unfortunately, no index parameter can be used in a shader so the pixels are selected using dot product (which is fairly efficient on GPU). The ranks are calculated using the Alg. 3.6.

The AdaBoost/WaldBoost Object Detection Runtime Framework in GPU One fragment shader evaluates several LRD's and accumulates them in an accumulated (see above). After accumulating all the weak classifiers in the learned AdaBoost classifier, a decision is made based on a threshold. The overall AdaBoost classifier structure implemented using the shader is in Fig. 3.17.

Algorithm 3.6 Calculation of the local rank difference; *row0*, *row1* and *row2* are vec3 and contain the input pixels, A and B are pixel values on positions a and b. The *lessThan* function compares its arguments by component and the result is vec3, containing zeros or ones based on comparison. The dot product sums up the Local Rank Difference. This snippet of code evaluates in approximately 14 GPU instructions. Finally, *alpha* is chosen from table (texture).

```
vec3 accum = lessThan(vec3(A), row0);
accum += lessThan(vec3(A), row1);
accum += lessThan(vec3(A), row2);
accum -= lessThan(vec3(B), row0);
accum -= lessThan(vec3(B), row1);
accum -= lessThan(vec3(B), row2);

float rank_difference = dot(vec3(1,1,1), accum);
```

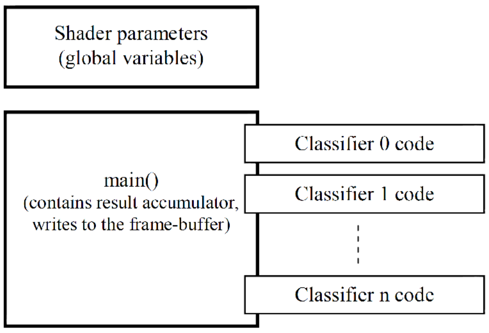


Figure 3.17: AdaBoost/WaldBoost object detection GPU runtime shaders with several classifiers.

The WaldBoost [93] pipeline is fairly similar to the one of AdaBoost (described above), it only needs facilities to terminate the calculation on individual pixels. This can be done using depth test – the classifier evaluation remains unchanged, but extra rendering passes are added which compare the intermediate accumulated sum with a given threshold and modify the depth-buffer accordingly. That means if output is below the threshold, zero is written into the depth-buffer, otherwise one is written (using `step` to avoid branching). The outputs from the classifier are rasterized on depth 1 so shaders are not executed on positions with zero depth (see Alg. 3.7).

This approach benefits from early depth-test that discards all fragments with the wrong depth (without evaluation). The limitation is that fragments modifying their depth must be evaluated so the number of the stopping decisions must be low. Therefore, training of WaldBoost classifier must include costs of the decisions.

GPU(GLSL) Approach

This section presents our experiments with an OpenGL implementation of the LRD detector, consisting of the convolution precalc module and a feature extractor. It can work on most of today's common GPU's which support OpenGL 2.0. To achieve better

Algorithm 3.7 AdaBoost shader code; *n_texture_0* is the id of the right texturing unit, *v_pixel_00* is the pixel size of that texture, *n_alphas* is the id of the alphas texturing unit, *v_alpha_pixel* is site for alphas texture, *v_block_to_*slice contains constants required for 3D texture slice from 2D texcoords (width/number of layers, convolution kernel width/number of layers, height/number of layers*convolution kernel width and slight z-offset to aid the right layer sampling), *v_selector_a00* and *v_selector_b00* are vectors selecting the right column from 3×3 grid).

```
uniform sampler3D n_texture_0;
uniform vec2 v_pixel_00;
uniform sampler2D n_alphas;
uniform vec2 v_alpha_pixel;
uniform vec4 v_block_to_slice_00;
uniform vec3 v_selector_a00, v_selector_b00;

void main()
{
  float f_result = .0; // result accumulator
  {
    // classifier 0
   }
  ...
  {
    // classifier n
  }
  gl_FragColor.r = f_result; // write output fragment
}
```

compatibility and portability, our implementation prefers the frame-buffer objects (FBO) above platform-dependent P-buffers and GLSL shading language above the Cg language.

The implementation takes a raster image in the system memory as input, then it needs to upload it to an OpenGL texture in the GPU memory, feature evaluation shaders get executed and a raster with detector responses is downloaded back to the system memory. There was no attempt for asynchronous data transfers to hide transport delay, but earlier work proved that such transfers are possible on GPU.

One implementation is already described in [73] which relies on complex, optimized image data storage. The implementation measured here is more straightforward because it is limited to sampling function dimensions 1×1, 1×2, 2×1 and 2×2. Such a limitation does not notably harm the information content extracted by the features, but significantly improves the performance. The bilinear filter (implemented in the texturing hardware of GPU) samples four pixels and assigns them weights, based on fractional texture coordinates. It is possible to simulate 1×1, 1×2, 2×1 and 2×2 pixel sums just by a texture coordinate offset.

This introduces some interesting consequences. There is no need for a pre-calculation phase; also, we just need a single texture to evaluate all weak classifiers in the WaldBoost classifier, which is important for two reasons:

1. there is no need for branching in the classifier to select the proper convolution texture

for a particular weak classifier and, therefore, there is no need to split the classifier evaluation into multiple rendering passes as in [73];

2. all textures required to evaluate the WaldBoost classifier can be bound simultaneously to available texturing units.

Now, we can evaluate all the weak classifiers in a loop in a single pass, than we had to store the classifier properties.Weak classifier properties was stored in two pixels of a RGBA texture. Once the textures with classifiers properties were generated, it was possible to evaluate the features in the fragment shader. The shader requires the data textures and the image texture as its input. For each weak classifier, the properties texture is read first so the mask can be read from the source image texture. Then it is necessary to get values of blocks a and b from the mask. In the fragment shader it is not possible to use an array referencing operator to select values from the matrix, so these needs to be masked-out using dot products. Once the values of blocks a and b are known it is straightforward to evaluate their ranks R_a and R_b. All that remains is to read the alpha texture, accumulate the classifier response and compare it with the WaldBoost thresholds. Detailed description of algorithm can be found in [32].

SSE Approach

This section presents a brief description of SSE implementation, which was introduced in [32]. It is similar to implementation described in [31], where LRP classifiers evaluation was used.

The LRD evaluation is described in Fig. 3.18. First, the data are compared to A and B vectors and masked (temporary results cmpA, cmpB). The sums of absolute differences of cmpA and cmpB are subtracted and the results for high and low parts are summed together producing the LRD value. The evaluation is much more efficient compared to CPU code without SSE since all the values are processed in parallel. The slowest step of the evaluation is the expansion of an 8- bit value to a full 128-bit SSE register.

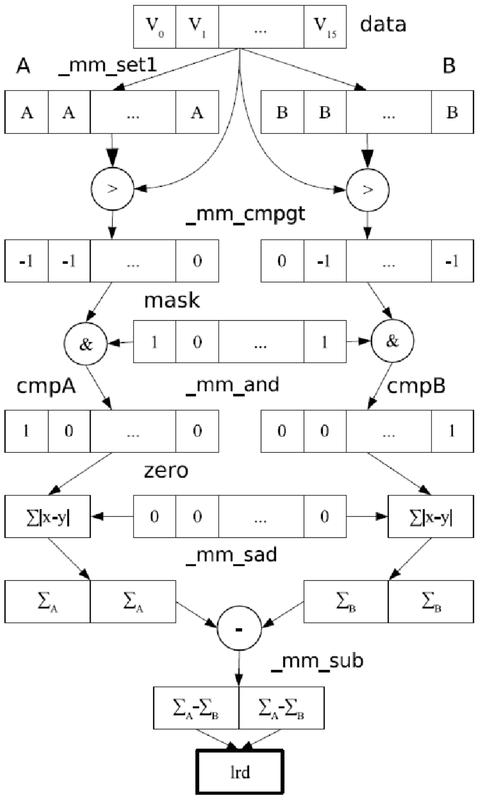


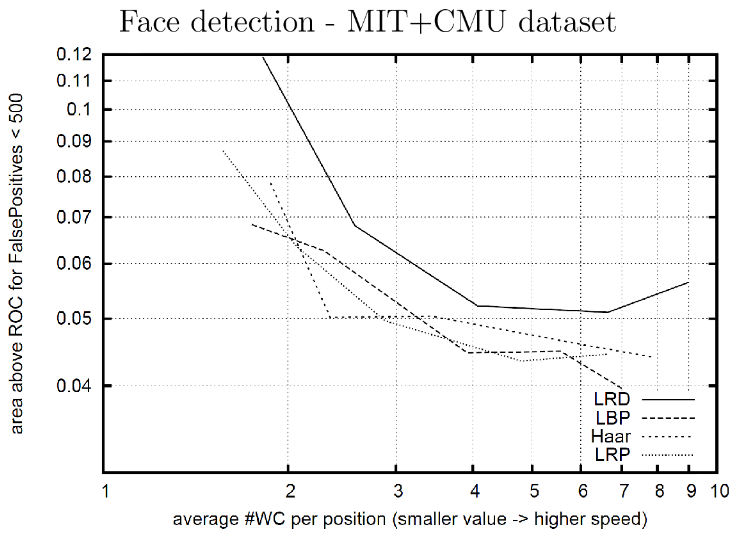
Figure 3.18: Block Scheme of the SSE Code (evaluation part only).

3.4 Detection Performance

In the context of real-time object detection, the main measurable criterion which should be used to compare individual types of features is how much useful information they can extract in a certain amount of time. The second criterion is how much are they invariant to irrelevant information. Both of these criteria have to be evaluated with respect to a certain learning algorithm. The first criterion can be directly evaluated on a training set and the second corresponds to generalization on a test set. When using some focus-of-attention mechanism, the amount of extracted useful information determines the speed of the classifier which can be then related to the precision of detection on a testing set.

We have used WaldBoost ([86]) as the learning algorithm and tested the features on two detection tasks – face detection and eye detection. We have compared the Haar-like features, LBP, LRD and LRP (all neighborhoods U^{mn} which completely fit into the samples are used). For each type of the features, classifiers for five different target error rates (1%, 2%, 5%, 10% and 20%) were created. The five target error rates resulted in five gradually faster classifiers which allowed us to explore the speed/precision trade-off provided by the features on the particular detection task. Ideally, the speed of the classifiers should be measured using some efficient implementation of the features. However, such an approach distorts the results with a different level of optimality of the individual feature implementations. To remove these, we report here the speed in average number of evaluated features per classified position.

As seen in Fig. 3.19, Haar-like features, LBP and LRP all perform very similarly on the face detection task followed by the LRD. On the other hand, clear differences can be seen on the eye detection task where LBP are the best, the second are the LRP which are followed by the LRD, while the Haar-like features are the worst. These results show that it is not possible to select a single best feature set for a variety of detection tasks. The performance of the features can be influenced by the number of the training samples, the type of distinguishing information and by the amount of intra-class variance. However, the experiments show that LRP and LRD provide in general similar detection performance as Haar-like features and LBP. Also, LRP should perform better than LRD on most tasks.



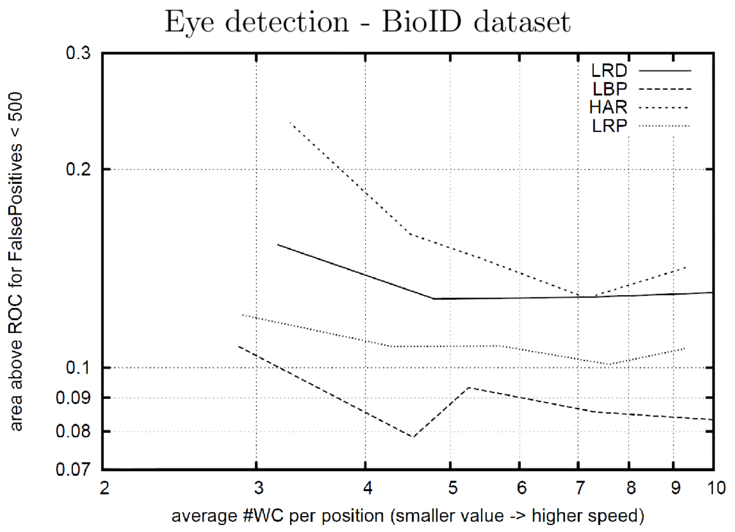


Figure 3.19: Comparison of performance of image features on face detection (top) and eye detection (bottom) tasks. The graphs show the area above ROC (integrating miss-rate over false positives) as a function of average classifier speed (lower is more precise and to the left is faster). The classifiers were created by the WaldBoost algorithm for five different target error rates (1%, 2%, 5%, 10% and 20%) for each type of feature-set. The five target error rates resulted in five gradually faster classifiers – shown as a single line. The graphs can be also used to evaluate the precision/speed trade-off for each type of feature-set for the particular task.

3.5 Performance Evaluation of LRD Implementations

LRD implementation was measured within two separate studies. The first one, with an exact number of weak classifiers (WC) was mainly focusing on the execution speed of LRD classifier features and the second one explores execution speed of real video. It is difficult to compare the two studies together, as within the first one the goal was to find out the LRD performance at different platform and compare it with Haar-like implementation, while within the second one the analysis was performed based on real data.

Research of exact number of WC observed the performance of LRD related to each WC. Real video research uses WaldBoost algorithm to speed-up the whole process of execution. WaldBoost algorithm is a combination of real AdaBoost [78] and Wald's [94] sequential probability ratio test. The thresholds are set as Wald proposes in the sequential probability ratio test, which he proves to be the fastest possible classification strategy for a given target error rate. Also, as the resulting classifier is monolithic, no information is lost.

3.5.1 Exact Number of Weak Classifiers

To evaluate the efficiency of the presented GPGPU implementation of the LRD, the following implementations were compared:

LRD on GPU Using CUDA refers to implementation in 3.3.3.
Even though Cg implementation was fairly efficient, its main disadvantage was the need of complicated control of the rendering pipeline from the CPU by issuing commands to render quads, lines or other primitives in a complex pattern that covered the searched area of the image. This disadvantage was minimized by the properties of the GPGPU philosophy. The CUDA implementation presented compared to the Cg one benefits also from some memory arrangement improvements, from improved training process and other minor advances.

LRD on GPU Using Cg Shading Language refers to implementation in 3.3.3.
An efficient memory layout was used (utilizing 3D textures and other techniques) to allow the shader to access all the nine values of the LRD mask in 3 or 6 texture look-ups. The pixel data were stored as components of the .rgba vector, and vector operations could have been used in the calculation.
For the pre-processing task, which was constituted by several passes of sub-sampling by an integer fraction (3.3.1), built-in hardware means of texture sampling were used on the GPU - see Tab. 3.1for results.

LRD on CPU Using MMX Instruction Set refers to implementation in 3.3.3.
The performance of the GPU implementation was compared to an implementation on standard Intel CPU using MMX instructions. To simplify the feature evaluation as much as possible, the convolutions of the image with the sampling function kernel were pre-computed and stored in the memory in such manner that all the results of the LRD grid could be fetched into the CPU registers through two 64-bit loads. This positively affected the evaluation that was performed in MMX CPU instructions (introduced by Intel).

Haar on CPU+GPU Using Cg Shading Language refers to implementation in 3.1.1.
Within this implementation, only the simplest (two-fold) Haar wavelet features were used (though also three-fold features are used in the object detectors, whose evaluation is slightly slower).
The Haar wavelets require normalization by the energy in the classified window - both to evaluate the energy and to evaluate the features themselves, integral images were used, which was the fastest method available to our knowledge. The calculation of the integral images constituted the preparatory phase evaluated in the comparison. Please note that (to our knowledge) there was no effective way of calculating the integral image in the shading language, and the implementation in CUDA was also not straightforward and efficient, so the preparatory phase was implemented in the CPU. The shader code evaluating the classifiers can be found in [73].

The evaluation was performed for different resolutions of the image, for different sizes of the classified window and for different amount of the weak hypotheses calculated for each classified window. Note that this evaluation was to determine the evaluation speed of the weak classifiers only, not the overall performance of the boosted classifier.

In Tab. 3.1, a coarse comparison of the performance of the pre-processing stage is given. It was difficult to compare the pre-processing for the Haar wavelets with the LRD convolutions, because the integral image calculation was difficult to implement on the GPU. Note that this is an important advantage of the LRD over the Haar wavelets, especially when in GPU implementation. The actual CUDA implementation worked without the pre-processing, because it relied on the 1×1, 1×2, 2×1 and 2×2 set of mask dimensions. As indicated by the graph in Fig. 3.14, such limited set of sampling function dimensions was still sufficient and well comparable with the commonly used Haar features.

Tab. 3.2 includes such regimes of evaluation, that were designed to correspond to real-time operation even on slower platforms, as is the C code for the CPU (it was considered slow compared to the parallel architectures as FPGA or GPU). In that table, the CUDA code did not perform excellently, but a tremendous increase of performance was observed

resol	LRD CUDA	LRD Cg	LRD CPU	HAAR CPU
320×200	—	0.72	2.52	1.22
640×480	—	1.22	9.13	10.29
800×600	—	3.51	13.80	16.41
1024×768	—	3.75	24.80	27.94
1280×1027	—	4.53	37.45	45.16

Table 3.1: Evaluation of the pre-processing stage (convolutions for the LRD, integral image for Haar wavelets); the pre-processing needs to be performed on every frame. Times are given in milliseconds. Note that pre-processing for the LRD is notably cheaper, even on CPU and performs excellently on GPU. Note also, that the presented CUDA implementation requires no pre-processing stage.

when the number of weak classifiers is increased (towards 50 in the table).

		frame-time [ms]				time-per-wc [ns]			
resol	num wc	LRD CUDA	LRD Cg	LRD MMX	Haar Cg	LRD CUDA	LRD Cg	LRD MMX	Haar Cg
320×200	**5**	13.90	0.244	17.7	0.370	43.44	0.872	55.29	1325
320×200	**10**	13.63	0.527	25.0	0.469	21.29	0.942	46.71	0.839
320×200	**50**	13.50	2.524	82.0	3.010	2.20	0.902	40.04	1.076
640×480	**5**	56.87	1.173	101.8	1.642	37.03	0.810	58.55	1.134
640×480	**10**	53.82	2.232	149.0	2.159	17.52	0.771	51.82	0.745
640×480	**50**	32.95	11.066	493.0	15.731	2.14	0.746	44.05	1.086

Table 3.2: Performance table for LRDonGPU, HAARonGPU and LRDonMMX; the table contains the times of sole evaluation of the classier, since the pre-processing for the Haar wavelets (integral image calculation), cannot be easily implemented in the GPU.

Further exploration showed that the CUDA platform (at its current version 2.0beta) exhibited relatively slow and constant load-time of the code to be executed. Also the current implementation of the boosted classier, as indicated in Tab. 3.3, consumed constant run time for wide range of increasing number of weak classifiers - though the computational load should be linearly proportional to it. This anomaly should have been further explored and may be related to some characteristic of the GPU architecture or a flaw in the compiler. However, if the boosted classier would be a standard AdaBoost [92] or similar, the number of weak classifiers would be constantly high (hundreds). In such case the CUDA implementation outperformed tremendously any other solution available to our knowledge.

		Frame time			**Feature time**		
	num of	**Load**	**Exec**	**Total**	**Load**	**Exec**	**Tot**
resol	**WC**	[ms]	[ms]	[ms]	[ns]	[ns]	[ns]
128×128	**5**	2.18	4.26	6.44	26.69	52	78.69
256×256	**5**	2.29	11.94	14.23	7	36.43	43.44
512×512	**5**	2.35	46.17	48.53	1.79	35.23	37.02
1024×1024	**5**	3.37	169.67	173.04	0.64	32.36	33
1600×1600	**5**	4.86	403.88	408.74	0.38	31.55	31.93
128×128	**40**	2.18	4.85	7.03	3.32	7.4	10.73
256×256	**40**	2.26	11.38	13.64	0.86	4.34	5.2
512×512	**40**	2.3	42.62	44.93	0.22	4.06	4.28
1024×1024	**40**	2.79	165.58	168.38	0.06	3.94	4.01
1600×1600	**40**	3.47	399.88	403.35	0.03	3.9	3.93
128×128	**160**	2.01	4.37	6.38	0.76	1.66	2.43
256×256	**160**	2.34	11.34	13.69	0.22	1.08	1.3
512×512	**160**	2.36	42.36	44.73	0.05	1.01	1.06
1024×1024	**160**	2.65	165.29	167.95	0.01	0.98	1
1600×1600	**160**	3.6	411.19	414.8	0.01	1	1.01
128×128	**640**	1.98	12.31	14.3	0.19	1.17	1.36
256×256	**640**	2.24	25.54	27.79	0.05	0.6	0.66
512×512	**640**	2.36	98.16	100.52	0.01	0.58	0.59
1024×1024	**640**	2.7	385.54	388.25	0	0.57	0.57
1600×1600	**640**	4.25	1028.21	1032.46	0	0.62	0.63

Table 3.3: Behaviour of the CUDA implementation for a range of image sizes and number of weak classifier per scanned window. Two parts of the table show the time consumed per frame and this measure divided per the number of weak classifiers in the frame. The times are structured into Load time of the program, Execution time and the sum of these both.

3.5.2 Real Video

Comparing the performance of these diverse implementations was not trivial. The most significant performance metric was probably the detector throughput in frames per second for a sufficiently long video. The processing time for one frame does not reflect the case where more frames are processed in parallel or pipelined. The processing was divided into two pipeline stages - transfer to/from the card and detection. Also with four detection engines on the Uni1p card, up to eight frames could be processed in one moment; this situation also occurs on the GPU implementation. On the other hand, the time for one frame was an important metric in situations where separate frames were processed.

The processing time can be split into several phases. The crudest division is on preprocessing and scanning. The preprocessing can be further divided into construction of the image pyramid and calculation of the convolutions. In some implementations, some of these phases did not exist at all or were interleaved. In that case, the time was measured for all interleaved phases together, since separate measurement would seriously affect the performance.

The tests were performed on a computer with:

- CPU Intel Core2 Duo E8200 at 2.66 GHz, 3 GB DDR3 RAM and ASUS NVidia ENGTX280/HTDP graphics card.

The table below shows all three partial times for one frame, together with the total frame processing time. These times are in milliseconds. The times for missing or interleaved phases are left blank, meaning the time is equal to zero. The last column shows the theoretical throughput in frames per second (only the detection phases were measured, no video reading/decoding, waiting for the camera or image displaying were counted in).

A recording of television news was used as the test data. Three experiments with differently sized video were executed:

- low resolution video (640×350px, Tab. 3.4);
- broadcasting quality video (720×576px, Tab. 3.5); and
- high resolution HD video (1920×1080px, Tab. 3.6).

LRD on GPU Using CUDA refers to implementation described in 3.3.3.

LRD on GPU Using GLSL Shading Language refers to implementation described in 3.3.3.
This was a new implementation, which substitute NVIDIA Cg shading language, which was used in previous subsection (3.5.1).

LRD on CPU Using SSE Instruction Set refers to implementation described in 3.3.3.

Simple LRD and Haar refers to straightforward CPU implementation of LRD and Haar evaluation with no special optimizations.

Note that the percentage of participation of the preprocessing and scanning phases do not have to sum up to 100 %; the rest small amount of time is overhead spent in the auxiliary parts of the program.

		Preprocessing		Scanning		Total	Throughput
		[ms]	%	[ms]	%	[ms]	[fps]
Simple	**Haar**	7.6	3.9	187.7	95.8	195.9	5.1
Simple	**LRD**	3.2	1.6	191.2	98.0	195.0	5.2
SSE	**LRD**	0.5	1.4	31.3	96.8	32.3	31.1
CUDA	**LRD**	0.2	1.1	12.1	94.5	12.8	78.7
GPU(GLSL)	**LRD**	0.1	1.0	10.0	87.3	11.5	86.9

Table 3.4: Results for Low Resolution Video (640×350px).

		Preprocessing		Scanning		Total	Throughput
		[ms]	%	[ms]	%	[ms]	[fps]
Simple	**Haar**	20.4	3.5	551.8	96.2	573.8	1.7
Simple	**LRD**	8.5	1.8	448.0	97.8	458.0	2.2
SSE	**LRD**	1.4	1.7	78.4	96.5	81.2	12.3
CUDA	**LRD**	0.5	2.8	17.2	89.7	19.2	52.1
GPU(GLSL)	**LRD**	0.3	1.4	20.4	85.0	24.0	41.6

Table 3.5: Results for Broadcasting Quality Video (720×576px).

		Preprocessing		Scanning		Total	Throughput
		[ms]	%	[ms]	%	[ms]	[fps]
Simple	**Haar**	48.2	4.3	1059.9	95.3	1111.4	0.9
Simple	**LRD**	20.2	2.5	764.3	97.0	787.9	1.3
SSE	**LRD**	3.2	2.0	153.1	96.0	159.6	6.3
CUDA	**LRD**	1.1	3.4	28.2	86.2	32.7	30.6
GPU(GLSL)	**LRD**	0.5	1.4	25.4	77.3	32.8	30.4

Table 3.6: Results for Full HD Video (1920×1080px).

3.6 Performance Evaluation of LRP Implementations

This section summarizes the two implementations (CUDA and SSE) and experiments carried out in order to optimize and evaluate the object detection architecture defined earlier in this chapter. Following subsection describes the measurements made to optimize the thread rearrangement count and their locations; and are dedicated to compare an efficient SSE implementation of the same algorithm. CUDA implementation is described more in details (by discussing the influence of block width on an overall speed of data processing, determining an optimal thread rearrangement stages and comparison to the SSE implementation).

3.6.1 Influence of Block Width

As already discussed, the height of the computed block of image defines the number of threads and its width controls the number of locations computed by each thread. Measurements shown in Fig. 3.20 illustrates the two main aspects that need to be taken into account when tuning the implementation for a target application:

- higher block width reduces the computation time, because it lowers the number of blocks necessary, and
- since the number of blocks is always integer and the blocks must share the same dimensions in CUDA, block widths that are equal or slightly higher than integer fractions of the image width are desired.

For a particular application (described among others by video resolution) a proper block width must be found according to these rules.

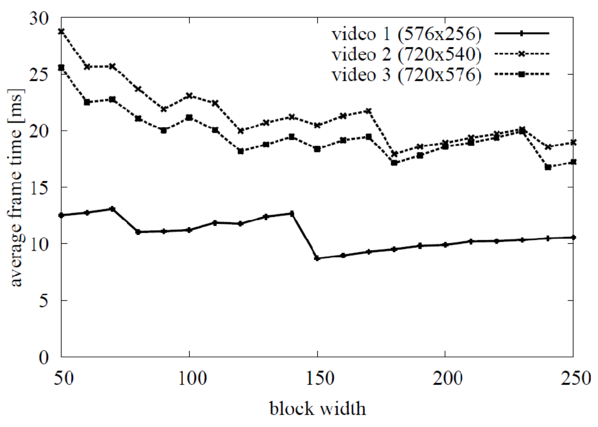


Figure 3.20: Influence of Block Width on Detector's Speed.

3.6.2 Determining Optimal Thread Rearrangement Stages

The scanning window locations need to be rearranged several times during the classifications to better use the hardware resources. We have run a number of tests to determine optimal spots for this rearranging. The tests reported that in the current set-up, no more than three rearrangements are worth doing. Fig. 3.21 summarizes the detection times for different stage of the 1st, 2nd and 3rd rearrangement.

The experiments confirmed that the 1st rearrangement matters the most, because it rearranged a large number of threads. Note that there can be a lower bound of the 1st rearrangement stage imposed by the size of the shared memory. The tests were run for six different videos (news broadcasting and movie fragments) resized to standard PAL resolution. Note the difference in the average detection times between different video contents, but rather uniform optima of the rearrangement stage. However, the optimal points for rearrangement were notably different for classifiers trained with different parameters – the shown experiment therefore did not result into fixed rearrangement spots, but rather illustrated the process of optimization for a given classifier.

3.6.3 Comparison to the SSE Implementation

This subsection gives some measurements done to compare the CUDA implementation with the SSE processor implementation. Tab. 3.7 contains the pure detection times per frame for the implementations on six videos of different content and resolution. These detection times do not include any preparatory phases (video decompression, pyramid construction, image handing, etc.), only the algorithmic detection times. Tab. 3.8 contains the total detection times; these were important for the actual use of the detectors. Fig. 3.22 visualizes the pure detection times graphically.

	I7-920 GTX280		**C2D E8200 9800GTX**	
video	**CUDA**	**SSE**	**CUDA**	**SSE**
576x256	9.4	12.6	9.0	24.2
720x540	12.5	30.5	16.8	61.1
720x576	11.1	31.4	14.9	56.5
1280x720	20.4	59.3	27.5	118.0
624x256	8.4	10.5	8.0	21.3
640x272	9.5	12.7	8.9	26.0

Table 3.7: Pure detection times [ms] (i.e. without preprocessing) on different videos, two different hardware setups, CUDA vs. SSE.

The main observations made out of these tests were:

	I7-920 GTX280		**C2D E8200 9800GTX**	
	CUDA	**SSE**	**CUDA**	**SSE**
576x256	12.0	14.5	11.6	26.5
720x540	17.5	35.2	22.7	67.1
720x576	15.9	36.1	20.3	62.0
1280x720	36.8	76.4	49.6	141.2
624x256	11.0	12.4	10.5	23.7
640x272	11.9	14.7	11.6	28.6

Table 3.8: Total detection times [ms] on different videos, two different hardware setups, CUDA vs. SSE.

- CUDA outperformed the processor implementation mainly for large videos. This can be explained by extra overhead connected with transferring the image to the GPU, starting the kernel programs, retrieving the results etc. These overhead operations consumed typically constant time independent of the problem size, so they were better amortized in high-resolution videos.

- The Intel I7 920 processor outperformed the Core2 Duo E8200 very significantly - it had twice as many cores and the computational speed was indeed twice as good.

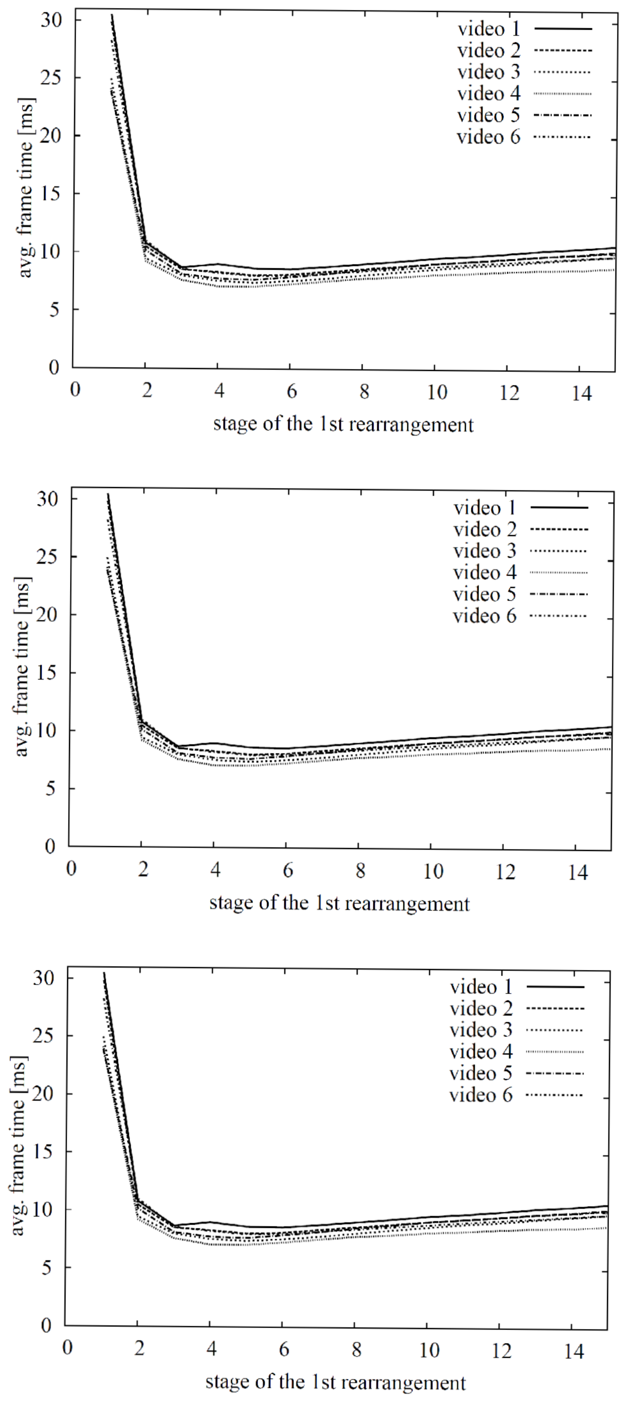


Figure 3.21: Detection time for different stages of rearrangement. The results of such measurement will be different for different classifiers.

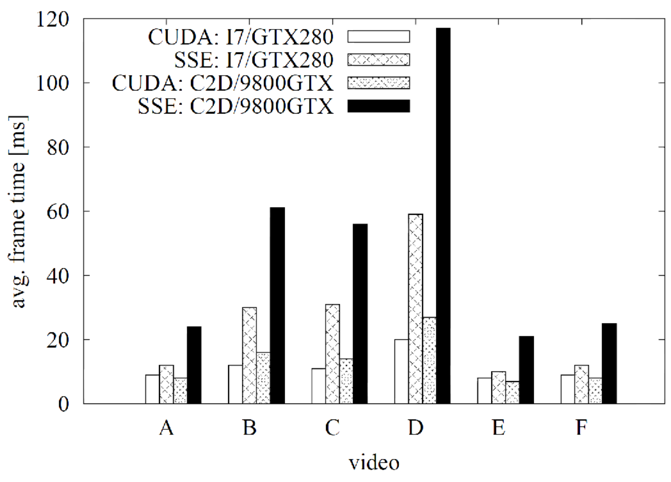


Figure 3.22: Visualization of Tab. 3.7.

Chapter 4

Spectral Image Analysis Using CUDA

This chapter presents the CUDA implementation of two different algorithms primarily targeted on spectral image analysis in real-time. The first one - Principal Component Analysis (PCA), presented in [41], explains our two different approaches to implementation - one utilizing the SSE instruction set of contemporary CPUs, and one running on GPUs. The second one is an algorithm of Non-Negative Tensor Factorization (NTF), presented in [4] that uses CUDA to run contemporary graphics processors in a GPGPU manner and uses their massive parallelism.

The exact motivation for the research presented in this section was to analyse medical surgery videos by using PCA. The topic of the problem has been revealed from the start-up project *Optical sensor technology in medical applications* of the University of Eastern Finland. This research was therefore performed within close cooperation of our colleagues from University of Eastern Finland in Joensuu, namely: Marku Hauta-Kasari and Jukka Antikainen; as well as, other research co-authors: Jiří Havel, Adam Herout and Pavel Zemčík.

4.1 Principal Component Analysis

Principal Component Analysis (PCA) is an approach that is traditionally used for analysis, simplification of large data sets, dimensionality reduction, etc. Using modern computer technology, the PCA can be used on very large data sets where its utilization has previously been unthinkable and it can also be used in real-time applications. Therefore, the computational speed of PCA, especially the speed of creation of the co-variance matrix, is critical and any improvement is appreciated. In this section, implementations using either

SSE instruction set of current processors or using a GPU are presented. These solutions are performing PCA on large data sets with relatively low dimensionality.

This research was motivated by the need of using PCA on spectral images in the context of real-time medical imaging. Accurately defined colour is shown as an important factor in many scientific and industrial purposes. Normal digital cameras, displays, and even the human vision system produce colour by using three primary colours: red, green and blue (RGB). In many cases, the representation based on three colour components cannot capture all information and spectral imaging and analysis must be used (e.g. wood analysis [76], mineral detection [23], textile industry [99] and many other interesting targets). One spectrum can contain tens or even hundreds of wavelength channels which provide a much better colour presentation than three-colour RGB.

In the case of spectral images, PCA is used mostly for dimensionality reduction [27] and feature extraction [10]. For example, if the spectral image contains 81 wavelength channels, spectral dimensionality could be reduced to 6–11, depending on the complexity of the data set, without losing any important amount of information [49]. PCA is often used for data of high dimensionalities. Generally, in the case of spectral imaging, the dimensionality of the input data is not high (commonly 6–81 channels) but the number of samples (i.e. number of pixels in image or video) is large - millions to billions. Existing solutions (e.g. [39, 38, 2, 67]) do not exactly suit this purpose and this unique situation must be covered by a particular solution. Please note that the dimensionality of the data considered in this research is relatively low, so the computation of eigenvectors – addressed by the mentioned works - is relatively cheep. It is the computation of the co-variance matrix, which is costly for the considered data, what is accelerated by the algorithms presented in this chapter.

The spectral resolution of different image sensors can vary, however, in the presented approach we suggest considering approximately 6 to 81 channels as from the human vision point of view, images starting with approximately 6 spectral channels can be considered as having enough information to accurately represent the colour information for distinct human observers with differences in colour vision. The upper boundary, 81 channels, is determined from the visible range of the human vision from 380 to 780 nm when the spectral information is captured using 5-nm steps. The step of 5 nm is generally considered to be reasonable in optical spectrum processing in order to accurately distinguish between colours/materials unless very special requirements on subtle spectral changes are required [50]. In theory, the range of channel numbers can be wider and the presented approach handles these cases as well, but the measurements were made for the dimensionalities practically interesting in spectral image processing.

A surgeon uses a surgery microscope during the operation and the video can be seen live on the display. The microscope can be equipped with a standard RGB camera or spectral camera [46] with additional spectral channels of different wavelengths - existing solutions support up to 6 channels. PCA can help in revealing information normally unseen by humans through analysis of spectral information contained in the image in the wavelengths not seen or distinguishable by the human eye. One of the possible approaches is to search for the best possible three-component vector space that can represent the spectral information in the image and then visualize the obtained information in the RGB colour space.

4.1.1 PCA in Spectral Imaging

PCA is commonly used on datasets of various dimensionalities. In the case of spectral imaging, the dimensionality is usually in the order of 6–81 components. The dimensions of the spectral image's pixels correspond to different light wavelengths. One pixel s of the spectral image is defined as:

$$\mathbf{s}(\lambda) = [s(\lambda_1), s(\lambda_2), \ldots, s(\lambda_n)]^T \tag{4.1}$$

where n is the count of wavelength channels. The spectral image – in the context of statistical colour analysis - can be perceived as a two-dimensional martix $\mathbf{S}$ where each column presents all wavelengths from one pixel of the spectral image:

$$\mathbf{S} = \begin{pmatrix} s_1(\lambda_1) & \ldots & s_m(\lambda_1) \\ \vdots & \ddots & \vdots \\ s_1(\lambda_n) & \ldots & s_m(\lambda_n) \end{pmatrix} \tag{4.2}$$

where m is the count of the pixels in the spectral image. For such an image a correlation matrix can be computed:

$$\mathbf{R} = \frac{1}{m}\mathbf{S}\mathbf{S}^T \tag{4.3}$$

From the correlation matrix $\mathbf{R}$, eigen values and eigen vectors are solved so that the following equation is fulfilled:

$$\mathbf{R}\Phi = \sigma\Phi \tag{4.4}$$

where Φ is a matrix of eigen vectors and σ is a diagonal matrix with eigen values on the diagonal. Matrix **B** is formed from the solved eigen vectors:

$$\mathbf{B} = \begin{pmatrix} b_1(\lambda_1) & \dots & b_1(\lambda_n) \\ \vdots & \ddots & \vdots \\ b_\eta(\lambda_1) & \dots & b_\eta(\lambda_n) \end{pmatrix} \tag{4.5}$$

where η is the number of wanted base vectors. Innerproduct images are calculated by using the selected base vectors **B** and the previously defined 2D matrix of pixels **S**:

$$\mathbf{P} = \mathbf{B}^T\mathbf{S} \tag{4.6}$$

4.1.2 Real-Time Implementation of PCA

From the implementational point of view, Eq. 4.3 can be reformulated as a sum of matrices of the same dimensions computed independently for all image pixels:

$$\begin{aligned} \mathbf{R} &= \frac{1}{m}\mathbf{S}\mathbf{S}^T \\ &= \frac{1}{m}\sum_i \left[s_i(\lambda_1)\dots s_i(\lambda_n)\right]^T \left[s_i(\lambda_1)\dots s_i(\lambda_n)\right] \\ &= \frac{1}{m}\sum_i \mathbf{s}_i, \end{aligned} \tag{4.7}$$

where $\mathbf{s}_i$ is a square matrix computed from each image pixel.

This idea is used in the plain-C implementation (Alg. 4.1):

Algorithm 4.1 Computation of correlation matrix – basic implementation.

```
Require: image pixels $s_i, \forall i \in \{0,\dots,m-1\}$
Ensure: correlation matrix $\mathbf{R}$
1: $\alpha[u,v] \leftarrow 0, \forall u,v \in \{1,\dots,n\}$
2: for $i \in \{0,\dots,m-1\}$ do
3:     for $v \in \{1,\dots,n\}$ do
4:         for $u \in \{1,\dots,v\}$ do
5:             $\alpha[u,v] \leftarrow \alpha[u,v] + s_i(\lambda_u)s_i(\lambda_v)$
6:         end for
7:     end for
8: end for
9: return $\frac{1}{m}\alpha$
```

which is used as the baseline in measurements and constitutes a starting point of the SSE and CUDA implementations described further in this section.

Once the correlation matrix is computed, the eigen-vectors and values are found using the standard Jacobi iterative method [21]. This algorithm is used also in the SSE and CUDA solutions described later. The target application of this article - spectral imaging - counts on a low numbers of components per pixel ($3 - 81$), so that the algorithm for finding eigen-vectors and values takes only a small fraction of the computational time and the choice of the method is not very important.

Intel SSE Instruction Set Approach

SSE offers speeding-up computations by executing instructions in a SIMD manner. Two basic approaches can be considered when using SSE for PCA computation and for similar tasks in general: either multiple channels of one pixel are processed in parallel or one operation is done for several pixels at once. Processing multiple pixels has several advantages in this case so this approach is used - see Alg. 4.2 for a pseudo-code of the implementation.

Algorithm 4.2 Correlation matrix computed by SSE.

```
Require: image pixels $s_i, \forall i \in \{0, \ldots, m-1\}, 4|m$
Ensure: correlation matrix $\mathbf{R}$
 1: $\alpha[u,v] \leftarrow 0, \forall u,v \in \{1,\ldots,n\}$
 2: for $i \in \{0,\ldots,\frac{m}{4}-1\}$ do
 3:    for $j \in \{1,\ldots,n\}$ do
 4:       for $k \in \{0,\ldots,3\}$ do
 5:          $\varphi[j,k] \leftarrow s_{(4i+k)}(\lambda_j)$
 6:       end for
 7:    end for
 8:    for $v \in \{1,\ldots,n\}$ do
 9:       for $u \in \{1,\ldots,v\}$ do
10:          $\alpha[u,v] \leftarrow \alpha[u,v] + \varphi[u]\varphi[v]$
11:       end for
12:    end for
13: end for
14: return $\frac{1}{m}\alpha$
```

A straightforward way of representing the (spectral) image in memory is an array of pixels, where each pixel is an n-tuple of values. This is the way the input data is stored and passed on to the algorithm. However, for efficient use of SSE, the image needs to be stored in a slightly modified manner. When SSE is used to process multiple pixels at once, the image must be organized as an n-tuple of pixel arrays. Generally, for efficient SSE operation, the data needs to be aligned to 16 bytes. The algorithm does not rearrange the whole image in this way but only uses a four-pixel buffer which is re-used for groups of four pixels. Steps 3–7 perform this rearrangement into buffer φ. Note that the buffer is addressed as two-dimensional in this preparatory phase, but its values are read as vectors by the SSE in Step 10.

Step 10 is the only line of the pseudo-code which fully uses the SSE vector (SIMD) instructions: vector addition (`_mm_add_ps`) and vector multiplication (`_mm_mul_ps`). This operation also uses an intrinsic function to load the 4-component float value from memory into an SSE vector register (`_mm_load_ps`). It should be noted that the loading of one argument of the multiplication can be done once for each pass of the for loop beginning in Step 8.

The computation of the inner-product image is straightforward and uses the same principles to speed-up the execution by vector multiplication and addition as in the case of the correlation matrix computation.

CUDA Approach

The **image can be represented** in the natural way - as a linear array of pixels, each composed of n chars or floats (n is the input image pixel's dimensionality, typically 3 for RGB, but higher for spectral images). This linear memory is buffered by the CUDA threads into the fast shared memory as described below.

The **correlation matrix** (Eq. 4.7) is computed as follows. Matrix $\mathbf{s}_i$ is symmetrical, so for n-dimensional input image pixels, $\frac{1}{2}n(n+1)$ values need to be calculated and summed. Each component (or several components when $n \geq 32$) of the matrix is calculated by a CUDA thread. However, to use the GPU efficiently, a minimal number of threads needs to be running in parallel within a block, so P matrices $\mathbf{s}_i$ are calculated in parallel and thus $T = \frac{1}{2}Pn(n+1)$ threads are executed in a block. The input data is buffered in the shared memory in *chunks* of C pixels for each of the P matrices computed in parallel.

Alg. 4.3 describes the computations done by one block of threads.

Each block of threads computes a part of the sum from Eq. 4.7. Shared memory is used for buffering the input pixels: Step 4 reads P chunks of C pixels into the shared memory. Each thread then processes a given chunk of pixels, computing one component of the output matrix $\mathbf{s}_i$ (Step 6) and summing it into the accumulator α. The component's coordinates within the matrix are denoted as u and v; in our implementation, these values are stored in a precomputed 1D texture and read by each thread in Step 2. Since the whole sum computed by the algorithm is subdivided into P parallel groups of threads, their partial results need to be summed by Step 9 by using the shared memory.

An important characteristic of the presented arrangement is that it can be scaled in different dimensions to perfectly fit the hardware it is executed on. The dimensionality of

Algorithm 4.3 Correlation matrix contribution of each block.

```
Require: block number $b \in \{0, \ldots, B-1\}$, input image pixels $s_i, \forall i \in \{0, \ldots, m-1\}$
Ensure: $\sum_{i=0}^{RPC} \mathbf{s}_{(bRPC+i)}$, i.e. a part of the sum in (4.7)
1: $\alpha \leftarrow 0$
2: for each thread $t \in \{0, \ldots, T-1\}$ determine:
   $u, v$ – coordinates within the matrix $\mathbf{s}_i$
   $p$ – index of matrix computed in parallel with others
3: for $r = 0$ to $R-1$ do
4:     read pixels $s_{(bRPC+rPC+i)}, \forall i \in \{0, \ldots, PC-1\}$ by $T$ available threads
5:     __syncthreads()
6:     for each thread $\alpha \leftarrow \alpha + \sum_{c=0}^{C-1} s_{(bRPC+rPC+pC+c)}(\lambda_u) s_{(bRPC+rPC+pC+c)}(\lambda_v)$
7:     __syncthreads()
8: end for
9: threads $t \in \{0, \ldots, \frac{1}{2}n(n-1)-1\}$ sum up $P$ corresponding (by pair $u, v$) accumulators
```

pixels n is given a priori by the application. Based on it, the optimal thread count can be obtained by setting an appropriate number of parallel groups of threads P – optimal number of threads for current GPU's is in the order of 128 and higher, actual measurements are given in 4.1.3.

The pixel chunk size C can be set arbitrarily to control the use of the shared memory of the CUDA multiprocessors. One logical option is to fill the whole shared memory with the buffered pixels. However, using one half, one third or other fraction of the shared memory might allow running several blocks in parallel on one multiprocessor. The number of blocks B can also be controlled by arbitrarily setting the number of repetitions R, because $N = BRPC$, where N is the total number of pixels processed (to simplify the calculation, the memory following the image data is filled with zeros to the next multiple of PC so N is slightly bigger than the actual number of pixels in the image). An optimal value of B again depends on the hardware used for the calculation – its number of multiprocessors, number of blocks runnable on one multiprocessor, etc. Measurements (refer to 4.1.3 for details) show that the number of blocks is surprisingly not a very important factor. The ideal number according to our findings is identical to the number of multiprocessors present in the graphics chip (30 for contemporary GPUs). Further parallelization by submitting more than one block to a multiprocessor does not introduce any speed-up because the limiting factor seems to be access to global and shared memory.

Each block of threads running by Alg. 4.3 produces a part of the desired sum of matrices $\mathbf{s}_i$(4.7). These B matrices have to be summed, which can be performed by a tree scheme [25] or by simple linear summation, since B never reaches high values and the summation is limited anyway by the speed of the global memory.

Since contemporary implementations of CUDA support only 512 threads, the number of components n is limited to 31, because for a higher n, the number of components in α exceed the maximal number of threads. For a higher n, more than one component of α is computed by one thread.

For **finding the eigen values**, the same function as in the C and SSE version is used, being reimplemented into CUDA for C with some small modifications. Only one thread is used in this case because the algorithm (for the practical usable problem sizes) does not consume any measurable portion of time and its parallelization would be inconvenient due to a high degree of branching.

The last step of **converting the input image into the new base** (Eq. 4.6) is rather straightforward: the whole image is processed by threads in blocks and each thread multiplies one or more pixels with the eigen-vector and the result is stored in a different location in the global memory.

4.1.3 Performance Evaluation of PCA Implementation

For performance evaluation of PCA implementation, all experiments were primarily done with respect to the target application - real-time imaging for surgical operations. However, the range of tested image dimensions and component count were wider in order to show the general usability of the PCA algorithms for spectral imaging and other applications.

For experimenting with CUDA, SSE and C versions of the algorithm, a simple framework was created which loads a video (or a set of images) from a selected source, processes the frames and reports the time durations for a selected implementation. Video input was done via DSVideoLib, a DirectShow wrapper supporting concurrent access to frame buffers from multiple threads. Time measurement was done using very accurate `QueryPerformanceCounter` WinAPI function over a number of frames, so the variability of the measured times was below 1 %.

The testing was performed on two computers:

- C2D˜E8200+9800GTX Intel Core2Duo E8200 2.66 GHz, 4 GB 2×DDR2-RAM-1066(533MHz), nVidia GeForce 9800GTX and
- i7-920+GTX280 Intel Core i7-920, 6 GB 3×DDR3-RAM-1066(533MHz), graphics: nVidia GeForce GTX280

The measurement times reported in this section include computation of the correlation matrix, eigen vectors and values, and production of one inner-product image. Especially for the CUDA algorithm, the time is “all-inclusive”, meaning that transfer of the input image data to the graphics card, the computation itself, initiation of the transfer of the inner-product image back to CPU memory, and waiting for the transfer to be complete are included in the time. Real usage of the real-time PCA computation may avoid especially the transfer of the inner-product image back because its purpose might be to be displayed using the graphics card. Also, the original image should be displayed in any case and the transfer to the GPU would be necessary even if the computation was done on CPU.

Performance on Videos of Different Resolutions

The PAL resolution (720 $\times$ 576) videos were actual surgery videos recorded from the surgery microscope (see Fig. 4.1 for an example of a frame); higher resolution videos were random videos from TV broadcasting – the algorithm contained no conditions depending on the image contents, so the actual origin of the video should not influence the measured time results.

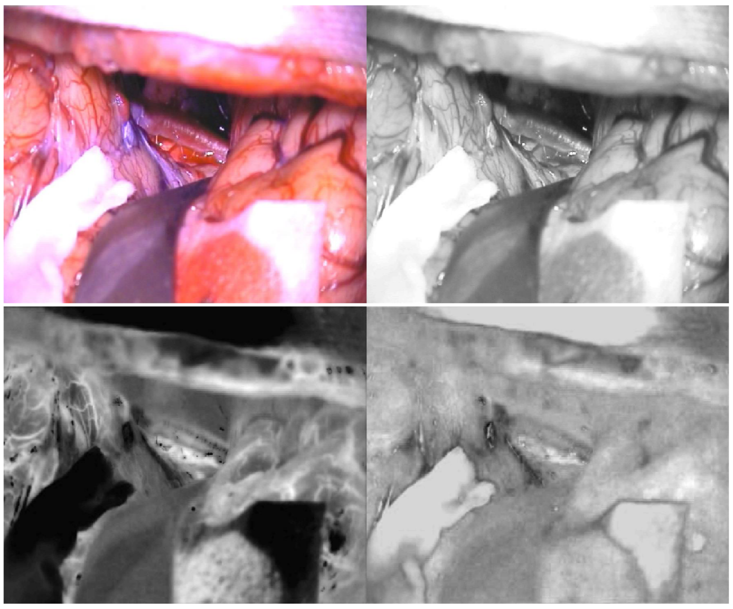


Figure 4.1: An example of a frame from a surgery video and its inner-product images.

Tab. 4.1 reports the measured time per frame by the C, SSE and CUDA implementations, running on the two above-mentioned computers. The algorithms have time complexity linearly proportional to the number of pixels. As expected, the time per-pixel (for reasonable frame dimensions) was constant for C and SSE versions of the algorithm. In the case of CUDA, the per-pixel times were slightly improving with the image resolution, which was caused by some constant overheads related to CUDA initialization, program loading, data transfer, etc. These constant overheads amortized better for larger images.

C2D E8200, 9800GTX					
resolution	**C**	**SSE**	**CUDA**	**C/SSE**	**C/CUDA**
640×480	10.5	4.9	2.3	2.1	4.6
720×756	14.4	6.9	3.1	2.1	4.7
1280×720	34.8	15.0	6.4	2.3	5.4
1920×1080	72.0	35.2	14.4	2.0	5.0
2560×1600	143.2	71.1	27.8	2.0	5.2

i7-920, GTX280					
resolution	**C**	**SSE**	**CUDA**	**C/SSE**	**C/CUDA**
640×480	4.9	2.0	0.4	2.4	11.1
720×756	6.4	2.6	0.6	2.4	11.4
1280×720	14.1	5.7	0.9	2.5	15.2
1920×1080	32.2	13.2	1.9	2.4	16.7
2560×1600	63.6	26.6	3.7	2.4	17.2

Table 4.1: Computational times per frame in milliseconds on different videos (RGB), two different hardware set-ups, C vs. SSE vs. CUDA.

Different Numbers of Spectral Channels

Surgical microscopes can contain several optical slots for cameras. These slots can be equipped with several cameras where each one of them works on individual wavelength responses. In this manner, a spectral image from the surgery can be captured in real-time. Therefore, videos with six or more spectral components were also examined. Measured times of PCA computation per frame (PAL resolution 720×576) for different numbers of channels are presented in Tab. 4.2.

Note that for 31 spectral channels, the SSE version matches the speed of CUDA. The PCA computation was demanding especially on the memory bandwidth, while the computational load was relatively low. The memory chips used both by the CPU and GPU used similar technology, so for some cases, the difference between the computational capacities became irrelevant. The CUDA solution had the disadvantage of bus communication and the need to upload the data to the graphics card and read back the inner-product image (all these actions are included in the computation times). The CUDA algorithm is surely useful in the cases when the presented measurements report superior performance to the SSE, which are cases with a lower number of spectral components. When the speed of the SSE solution matches CUDA, off-loading the computation to the graphics card still helps the medical software offer immediate responses by keeping the CPU free of computation. Also, the measurements include downloading the inner-product image back to CPU memory, which in many cases would not be useful because it is used on the GPU to be displayed on the display. On the contrary, if the computations were done on the

C2D E8200, 9800GTX					
# of ch.	**C**	**SSE**	**CUDA**	**C/SSE**	**C/CUDA**
3	14.3	7.5	3.1	1.9	4.6
4	17.4	7.8	4.0	2.2	4.3.
5	21.8	8.8	4.9	2.5	4.4
6	24.9	10.9	6.2	2.3	4.0
9	37.9	17.0	8.8	2.2	4.3
16	74.9	28.8	14.7	2.6	5.1
25	150.9	50.3	22.5	3.0	6.7
31	209.3	67.9	44.1	3.1	4.7

i7-920, GTX280					
# of ch.	**C**	**SSE**	**CUDA**	**C/SSE**	**C/CUDA**
3	5.9	2.6	0.6	2.3	9.4
4	6.9	2.7	0.7	2.6	9.6
5	8.6	3.0	0.9	2.8	9.7
6	10.0	3.4	1.1	3.0	9.1
9	14.9	4.8	1.8	3.1	8.4
16	29.1	10.3	5.0	2.8	5.9
25	52.5	18.9	11.5	2.8	4.5
31	71.5	26.1	34.5	2.7	2.1

Table 4.2: Computation times per frame in milliseconds on frames (PAL, 720×576) of different number of spectral channels; speed-ups SSE vs. C and CUDA vs. C; two different hardware set-ups.

CPU, the time to upload the resulting images to the graphics card for displaying them would need to be included.

The Optimal Set-up of CUDA Program Parameters

As mentioned in 4.1.2, one important advantage of the presented CUDA algorithm is that it can be scaled in different dimensions to fit the graphics hardware and fully use its potential. We have performed different measurements to explore the possibilities of the set-up, two most interesting of them are described below.

Fig. 4.2 shows the performance depending on the number of blocks of threads. It should be noted that GeForce 9800GTX has 16 multiprocessors, GTX280 contains 30 multiprocessors. The graph shows – as could be expected – that the number of blocks should always be a multiple of the number of multiprocessors, because the blocks performed identical actions; their duration was, therefore, identical and they were issued in groups running in parallel on the multiprocessors. What was not as obvious and expected was that for higher numbers of blocks, the algorithm performed the same or just slightly worse than for smaller numbers. For practical implementations (such as the one suggested in the following 6.2.2), choosing $B \simeq 200$ (and corresponding R) was a safe choice. For appropriate values of C (see 4.1.2, Alg. 4.3), more than one block could run on one multiprocessor at a time, which could increase the parallelism. However, since no speed-up was achieved by this arrangement, we could deduce that the limiting factor was access to the global memory with the input data and further parallelism did not help, but introduced a small overhead.

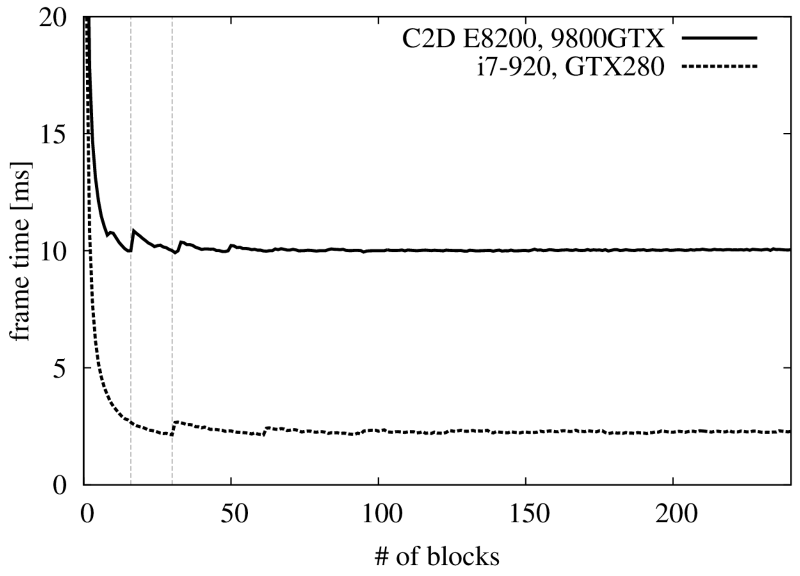


Figure 4.2: Performance of the CUDA implementation depending on the number of blocks. The best times are for 16 and 30 blocks (marked by dashed vertical lines), which correspond to the number of multiprocessors on each graphics chip.

A similar experiment has been carried out to explore the influence of the number of threads in each thread block – see Fig. 4.3. This measurement confirmed general recommendations for CUDA programs that the number of threads should be at least 128. The measurements reported that 256 threads and higher numbers were secure, from 128 the differences were barely measurable.

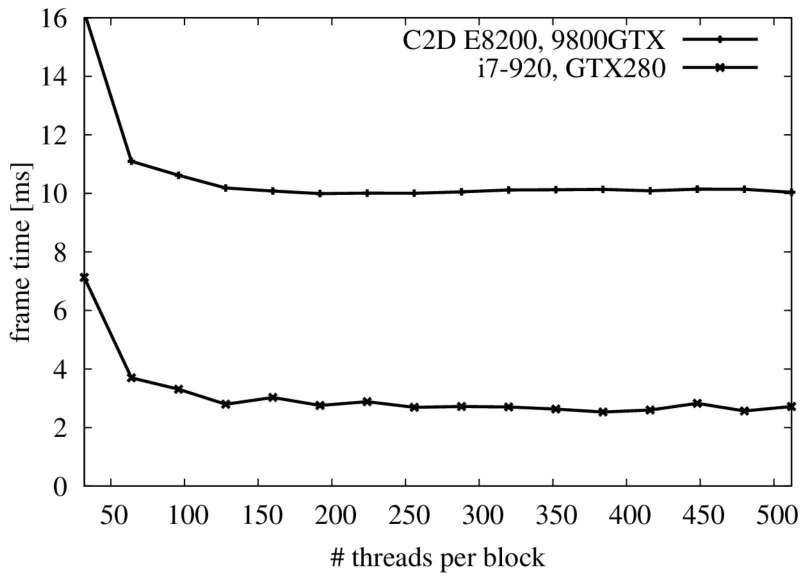


Figure 4.3: Performance of the CUDA implementation depending on the number of threads in each block.

4.2 Non-Negative Tensor Factorization

Non-Negative Tensor Factorization (NTF) can be used - in the context of spectral imaging - for image compression [3], optimal filter generation [29], and feature extraction [43]. NTF is also used in other scientific and industrial fields, such as global climate analysis, neuroscience, psychometrics, etc. [75], [6], [11], [57], [84], [48]. The dimensionality of these problems are often so high that NTF computations take hours so acceleration of this process is desirable.

This research shows an efficient GPU implementation for general iterative NTF computation by gradient descent, based on Gauss-Seidel and Jacobi methods [29], using the CUDA programming environment. The efficiency of the algorithm is compared to other available solutions. This section summarizes iterative NTF computation and fast modifications of this algorithm. Section 4.2.1 describes the proposed CUDA implementation, namely the decomposition of the problem into parts that can be calculated in parallel. Section 4.2.2 gives measurements of our implementation's performance and compares it with state-of-the-art solutions. The CUDA implementation of NTF was wrapped into a DLL, which is usable by C programs in both the Windows and Linux environments, and also by a MATLAB plugin. Information about this publicly available tool is given in 6.2.3.

In contrast to other analytical tools, such as Principal Component Analysis (PCA) or Singular Value Decomposition (SVD), NTF produces the matrix factors (basis vectors) that are always non-negative and which meet other requirements that enable real-world interpretations. In the context of spectral imaging, NTF allows the decomposition of a spectral colour into a set of filters that can be manufactured and can be used in optical systems [59].

Let $\mathbf{G} \in \mathbb{R}^{R \times S \times T}$ be a third order non-negative tensor to be analysed. Non-negative tensor factorization of $\mathbf{G}$ requires solving a non-linear minimization problem:

$$\min_{\hat{\mathbf{G}} \geq 0} \|\mathbf{G} - \hat{\mathbf{G}}\|_F^2 \tag{4.8}$$

where $\hat{\mathbf{G}}$ is the tensor of reconstructed data and $\|A\|_F^2$ is the square Frobenius norm. Also other cost functions such as α or β-divergences could be used [12]. The rank-K reconstruction is defined by sums of tensor products:

$$\hat{\mathbf{G}} = \sum_{k=1}^{K} \mathbf{u}^{(k)} \otimes \mathbf{v}^{(k)} \otimes \mathbf{w}^{(k)} \tag{4.9}$$

where $\mathbf{u}^{(k)} \in \mathbb{R}^R$, $\mathbf{v}^{(k)} \in \mathbb{R}^S$ and $\mathbf{w}^{(k)} \in \mathbb{R}^T$ are basis vectors of non-negative values. This reconstruction process is illustrated in Fig. 4.4.

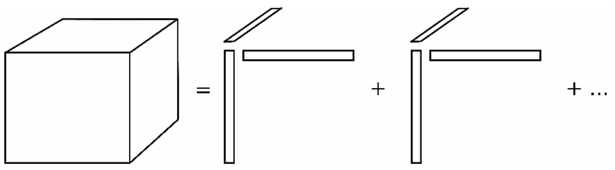


Figure 4.4: Principle of third order tensor factorization by using sums of rank-1 tensors.

The most commonly used approaches to non-negative tensor factorization are based on the Block Gauss-Seidel (BGS) method [22]. Using a combination of Gauss-Seidel and Jacobi iterative update schemes, Hazan et al. [29] derived a gradient descent that repeatedly updates $\mathbf{u}^{(k)}$, $\mathbf{v}^{(k)}$ and $\mathbf{w}^{(k)}$; these are calculated using iterative rules (Eq. 4.10–Eq. 4.12:

$$u_i^k \leftarrow \frac{u_i^k \sum_{s,t} G_{i,s,t} v_s^k w_t^k}{\sum_{m=1}^K u_i^m \langle v^m, v^k \rangle \langle w^m, w^k \rangle} \tag{4.10}$$

$$v_i^k \leftarrow \frac{v_i^k \sum_{r,t} G_{r,i,t} u_r^k w_t^k}{\sum_{m=1}^K v_i^m \langle u^m, u^k \rangle \langle w^m, w^k \rangle} \tag{4.11}$$

$$w_i^k \leftarrow \frac{w_i^k \sum_{r,s} G_{r,s,i} u_r^k v_s^k}{\sum_{m=1}^K w_i^m \langle u^m, u^k \rangle \langle v^m, v^k \rangle} \tag{4.12}$$

where $\mathbf{G}$ is the dataset and $\langle \mathbf{x}, \mathbf{y} \rangle$ denotes inner product. Usually, this iterative procedure must be repeated hundreds or even hundreds of thousands times to converge to the correct solution depending on the complexity of the dataset. Therefore, iterative NTF computation is quite time consuming, and approaches to speeding it up would be useful.

4.2.1 CUDA Implementation of NTF

Our algorithm closely follows the theoretical description in 4.2. The structure of the calculation is shown by Alg. 4.4. The first line in the algorithm initializes the vectors u, v and w by using random values between 0 and 1. The NTF problem can be divided into three sub-problems, corresponding to rules of Eq. 4.10–Eq. 4.12. Functions for their computation are named STEP in Alg. 4.4. The inner products in the equations' denominators can be calculated in advance and stored in K×K sized matrices. In Alg. 4.4, these matrices are named $\mathbf{M}_u$, $\mathbf{M}_v$, and $\mathbf{M}_w$, where $\mathbf{M}_u = \mathbf{u}^T\mathbf{u}$, i.e.:

$$\mathbf{M}_u = \begin{pmatrix} \langle \mathbf{u}^{(1)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(1)}, \mathbf{u}^{(K)} \rangle \\ \langle \mathbf{u}^{(2)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(2)}, \mathbf{u}^{(K)} \rangle \\ \vdots & \ddots & \vdots \\ \langle \mathbf{u}^{(K)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(K)}, \mathbf{u}^{(K)} \rangle \end{pmatrix} \tag{4.13}$$

and $\mathbf{M}_v$ and $\mathbf{M}_w$ are defined similarly. The function for their computation is named CMAT in Alg. 4.4. These matrices are symmetrical, so only the upper or lower triangle matrix needs to be calculated and stored.

Algorithm 4.4 Structure of the NTF algorithm.

```
Require: the input $\mathbf{G}$ (size $R \times S \times T$), the method rank $K$, and the iteration count $I$
Ensure: the output vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
 1: Init $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
 2: $\mathbf{M}_u \leftarrow \mathrm{CMAT}(\mathbf{u})$
 3: $\mathbf{M}_v \leftarrow \mathrm{CMAT}(\mathbf{v})$
 4: $\mathbf{M}_w \leftarrow \mathrm{CMAT}(\mathbf{w})$
 5: for $i \in \{0, \dots, I-1\}$ do
 6:     $\mathbf{u} \leftarrow \mathrm{STEP_u}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_v, \mathbf{M}_w)$
 7:     $\mathbf{M}_u \leftarrow \mathrm{CMAT}(\mathbf{u})$
 8:     $\mathbf{v} \leftarrow \mathrm{STEP_v}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_u, \mathbf{M}_w)$
 9:     $\mathbf{M}_v \leftarrow \mathrm{CMAT}(\mathbf{v})$
10:     $\mathbf{w} \leftarrow \mathrm{STEP_w}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_u, \mathbf{M}_v)$
11:     $\mathbf{M}_w \leftarrow \mathrm{CMAT}(\mathbf{w})$
12: end for
13: return $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
```

Calculating the numerator in the sub-problem steps is the most time consuming operation. All other calculations, including creating the correlation matrices such as Eq.

4.13, do not take a significant amount of time in comparison. The numerator calculation consists mostly of repeated summing of a large array, so it is more demanding of memory bandwidth than it is computationally intensive. The sub-problem steps only differ in the direction in which the layers of **G** are taken. Therefore, the following text will describe only the step for **w**, corresponding to Eq. 4.12; the pseudo-code for **u** and **v** is identical, except for renamed variables.

The sub-problem calculation must be divided into CUDA thread blocks. One straightforward solution could be to have each block calculate one value of **w**, so TK blocks are executed. Because **G** has T layers and K values are calculated for every layer, this division into blocks means that every layer is traversed and summed K times. To lower the number of reads from **G** (notably speeding up the computation), the calculation can be divided into T blocks, so every block calculates K values and traverses each layer of **G** only once.

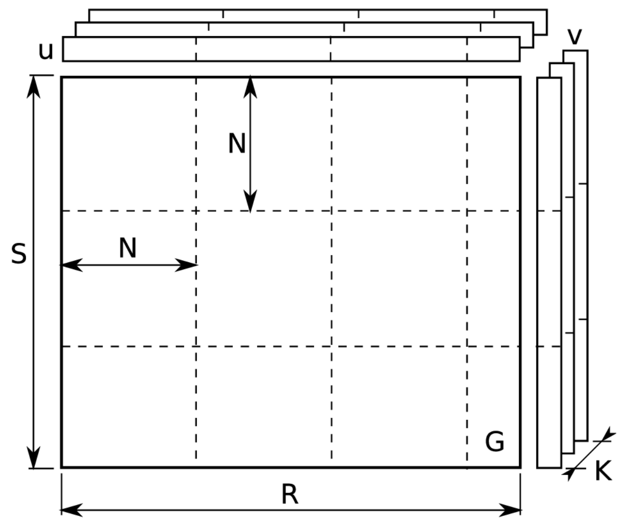


Figure 4.5: Tiling of one thread block. One slice of **G** is divided into $N \times N$ tiles computed by individual blocks of threads.

Because the number of threads in each block will often be lower than the **G** layer dimensions $R \times S$, the block must be divided into tiles. Each tile contains a number of elements equal to the number of threads in the block, which is N^2 arranged to a square matrix. Fig. 4.5 shows the tiling of **G** and the parts of **u** and **v** corresponding to each tile (for $K = 3$). Because each element of these parts is accessed N times, these parts are cached in the *shared memory* ($\mathbf{C}_u$ and $\mathbf{C}_v$). Each tile of **G** is multiplied by K corresponding parts of **u** and **v**, forming a block of size $K \times N \times N$, which is then added to a *shared memory* buffer α. After all tiles are processed, the buffer is summed via tree summation [25] to form K values. The rest of the work – calculation of the denominator and of the output value – only requires K threads. The work done by one block is shown in Alg. 4.5.

Algorithm 4.5 Computation Done by One Thread Block.

```
Require: $\mathbf{G}$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, $\mathbf{M}_u$, $\mathbf{M}_v$
    block index $t$, thread indices $i$, $j$
Ensure: new iteration $\mathbf{w}'$
 1: $\alpha[k,i,j] \leftarrow 0, \forall k \in \{0,\dots,K-1\}$
 2: for $x \in \{0,\dots,\frac{R}{N}-1\}, y \in \{0,\dots,\frac{S}{N}-1\}$ do
 3:   if $i < K$ then
 4:     $\mathbf{C}_u[i,j] \leftarrow \mathbf{u}[i,j+xN]$
 5:     $\mathbf{C}_v[i,j] \leftarrow \mathbf{v}[i,j+yN]$
 6:   end if
 7:   __syncthreads()
 8:   $e \leftarrow \mathbf{G}[i+xN,j+yN]$
 9:   for $k \in \{0,\dots,K-1\}$ do
10:     $\alpha[k,i,j] \leftarrow \alpha[k,i,j] + e\mathbf{C}_u[k,i]\mathbf{C}_v[k,j]$
11:   end for
12: end for
13: __syncthreads()
14: $\alpha[k,0,0] \leftarrow \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\alpha[k,i,j], \forall k \in \{0,\dots,K-1\}$
15: $k \leftarrow i + jN$
16: if $k \in \{0,\dots,K-1\}$ then
17:   $\mathbf{w}'[k,t] \leftarrow \frac{\mathbf{w}[k,t]\alpha[k,0,0]}{\sum_{m=0}^{K-1}\mathbf{w}[m,t]\mathbf{M}_u[k,m]\mathbf{M}_v[k,m]}$
18: end if
```

The calculation of the correlation matrices takes a negligible amount of time compared to the rest of the calculation, so no special optimizations were performed. The calculation of $K(K+1)/2$ elements of a correlation matrix (upper or lower triangular part of the matrix) is simply divided into an equal number of blocks, so every block traverses one pair of $\mathbf{w}$ rows and outputs one element of the correlation matrix.

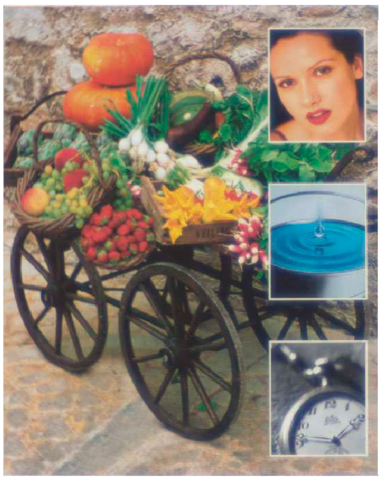
Because N^2 should be a power of two for easy tree summation, reasonable values of N can be 8 or 16. For $N = 8$, the block consists of 64 threads; for $N = 16$ it consists of 256 threads. A bigger value of N means that every element of the cache is used more times, so 16 was selected. However, a bigger N also requires larger arrays in the shared memory. The same holds for K, so for $N = 16$, 16KB of the shared memory is only sufficient for $K < 15$. If a greater value of K is needed, a lower value of N must be used. Fortunately, such cases are not common, so this limitation is mostly theoretical.

Another limitation of the implementation on current graphics chips is the maximum number of blocks per kernel, which limits the maximum of each dimension of the input tensor to 65536.

Both limits can be overcome when necessary, so the only remaining limitation of this implementation is the memory capacity of the graphics card. Storing tensors requires a huge amount of memory, so for current high-end graphics cards with two gigabytes of memory, the limit is a cube with approximately 800 elements in each direction (or a tensor with varying dimensions of equivalent volume).

4.2.2 Performance Evaluation of NTF Algorithm

In case of NTF algorithm, the computation times were measured using different sizes of spectral images (Fig. 4.6a); Fig. 4.6b shows rank 3 tensors that were calculated using NTF. The spatial resolution of the image varied between 100×100 and 1000×1000. The spectral dimensionality was either 31 or 62 (values common in spectral imaging). Channels in the 31-dimensional spectral images ranged from 420 nm to 720 nm with 10 nm steps captured by a spectral camera. The 62-channel spectral images were created by interpolating the 31-channel images to 61-channel image by using 5 nm steps. One extra channel was added by duplication into the red end to achieve double size images (62-channels) which can be compared easily to 31-channel images. Also, a simulated $600 \times 400 \times 200$ data set was used to provide a better comparison between our GPU algorithm and another recently published parallel implementation [75].



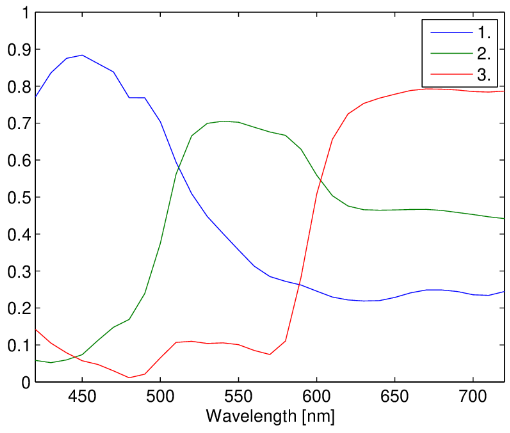


Figure 4.6: (a) RGB representation of the used spectral image; (b) calculated rank-3 factors in spectral domain.

Computational Times

All tests were performed using Intel Core i7-920 processor, 6 GB of DDR3 RAM and an NVIDIA GeForce GTX280 with 1 GB of GDDR3 memory. The GPU implementation presented in this section was compared to a standard C implementation of the same method[29] compiled by Microsoft Visual Studio 2009 (compiler ver. 15.00.30729.01). The GPU version was compiled using NVIDIA's compiler (CUDA compilation tools, release 3.0, V0.2.1221).

Fig. 4.7 shows the computation times, with 500 iterations, for both implementations. Speed-ups achieved by the GPU implementation are also displayed as a function of the spectral image's spatial size. The CPU calculations used both single and double precision floating point numbers. The GPU implementation could that time work with single precision only (fast access to the input tensor required texture lookup, that did not support double precision). The GPU implementation, which processed inputs with double precision, computed in single precision and only used fast page-locked memory for data conversion. The difference between single and double on the GPU was, therefore, the time needed for conversion from double to single precision.

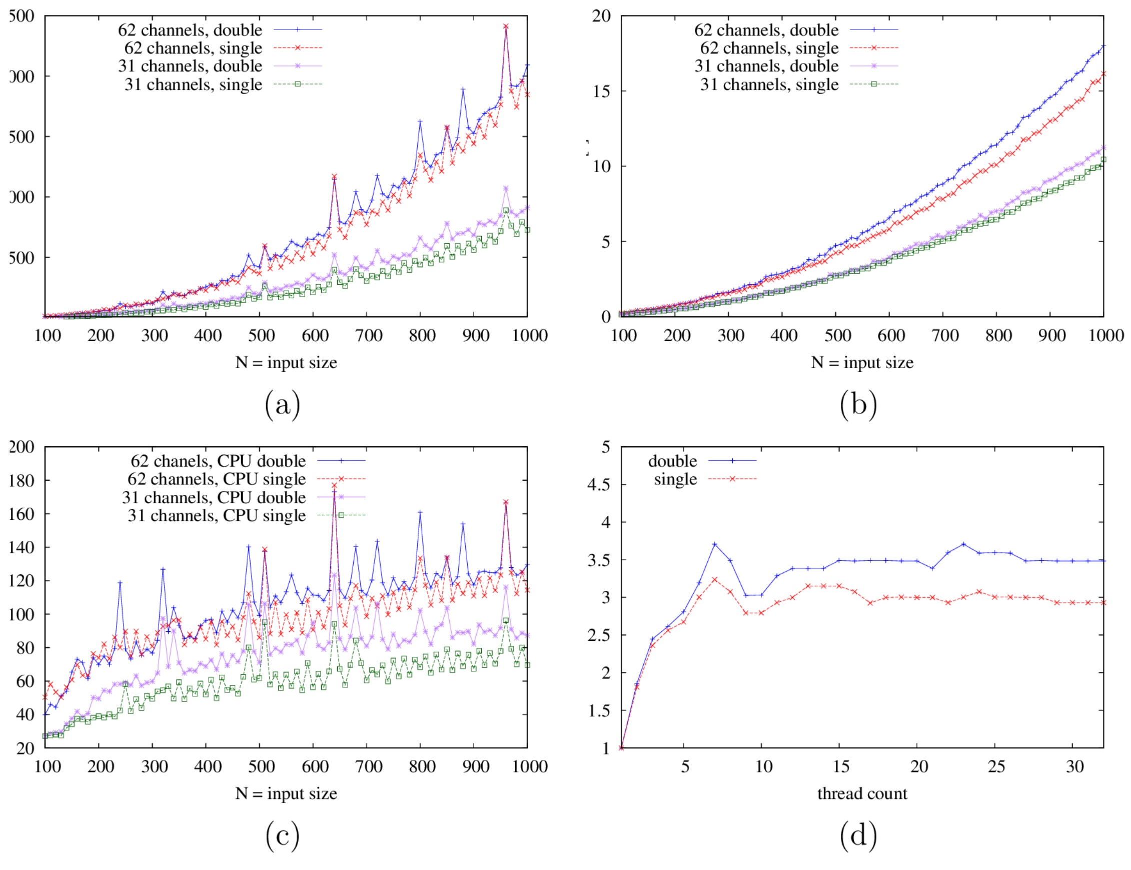


Figure 4.7: Computational times with different sizes of spectral images: (a) CPU times; (b) GPU times; (c) GPU speed-ups – CPU/CUDA; (d) CPU speed-ups for multiple cores, whereNis the spatial size of the image.

The computation times show that the speed-up factor rises logarithmically with respect to the size of the spectral image. Speed-ups for the computation times between the single-core CPU and GPU versions for practical dataset dimensions are around $60 - 100\times$.

Graph (d) of Fig. 4.7 shows the impact of using multiple threads on the CPU. The CPU used has four physical cores and eight logical cores via hyper-threading. The rest of the CPU measurements are done in the single-threaded form, the speed-ups achievable by multi-threading are limited to $3.5\times$.

The spikes in the speed-up graphs for the CPU measurements are a consequence of the cache implementation of the CPU. When the image has certain dimensions, neighbouring pixels in the x, y, and z axis directions were coherent with the alignment of the CPU cache records. Thus, some parts of the CPU cache were used heavily, while the rest of the cache memory was sparsely used or not used at all. Similar behaviour can be observed in various image processing algorithms. This behaviour could be avoided by deliberately misaligning image rows in the main memory, but this further complicated the algorithm, introduced some minor computational overhead, and slightly increased memory consumption. The CUDA environment did not suffer from similar effects.

The CPU implementation has been tried with both double and single floating-point precisions. The use of single precision might harm the precision of the results, so the GPU's effective inability to use double precision might appear limiting. Our observations, however, indicate that the floating-point implementation in modern GPUs was precise enough. In fact, the results from a GPU are comparable to the double precision CPU result and are much better than using single precision in the CPU. Errors between the single and double precisions were estimated by using root mean square error (RMSE). The RMSE difference between the CPU double precision version and the single precision version was 6.0×10^{-5}, while the error with the GPU version was only 6.6×10^{-8}. This can be explained by considering the summation strategy. When summing a large number N of items with limited (single) precision:

$$\alpha = \sum_{i=0}^{N} f(i) \tag{4.14}$$

$$\begin{aligned} \alpha &\leftarrow \alpha + f(i) \\ i &\leftarrow i + 1 \end{aligned} \tag{4.15}$$

Once i becomes high, the mantissas in the memory representations of α and $f(i)$ are overlapped by smaller and smaller numbers of bits. The accuracy can be increased by reformulating the accumulation strategy as:

$$\alpha = \sum_{k=0}^{N/K} \left(\sum_{i=kK}^{(k+1)K-1} f(i) \right) \tag{4.16}$$

Discussion Regarding Another Parallel Version of NTF

In 2009, Zhang et al. [75] introduced a multiprocessor implementation of the same NTF algorithm used as the starting point of our solution and evaluated its efficiency for a $600 \times 400 \times 200$ data set of climatic data. They used a Sun Fire X4600 M2 server for their computations.

For comparison, we measured the efficiency of both our CPU and GPU implementations using a dataset of the same size. The CPU computation time for the data set was 290 seconds for 100 iterations (with one core of the Intel Core i7 920 processor) and the equivalent GPU computation time was 2.36 seconds.

It was not possible to compare the absolute times, since the processors were different and it was impossible to imitate all circumstances of the measurement. However, Zhang et al. [75] report that their speed-up by adding further nodes was capped at about $7\times$ while our GPU algorithm achieved over $100\times$ speed-up compared to the single-core CPU version (our multi-core version on state-of-the-art processors achieved up to $3.5\times$ speed-up).

Chapter 5

Real-Time Line Detection Using CUDA

The Hough transform is a well-known and popular algorithm for detecting lines in raster images. Standard Hough transform is rather slow to be usable in real time, so different accelerated and approximated algorithms exist. This study proposes a modified accumulation scheme for the Hough transform, using a new parametrization of lines “PClines”. The algorithm discussed within this chapter is suitable for computer systems with a small but fast read-write memory, such as today’s graphics processors. It requires no floating-point computations or goniometric functions. This makes the algorithm suitable for special and low-power processors and special-purpose chips. The proposed algorithm was evaluated both on synthetic binary images and on complex real-world photos of high resolutions. The results showed that using today’s commodity graphics chips, the Hough transform can be computed at interactive frame rates, even with a high resolution of the Hough space and with the Hough transform fully computed.

This chapter presents an insight into our research on real-time line detection using Hough transform and parallel coordinates. The research presented in this chapter was performed in close cooperation with the following list of co-authors: Markéta Dubská, Adam Herout and Jiří Havel.

5.1 Line Detection Using Accelerated High-Resolution Hough Transform

The Hough transform is a well-known tool for detecting shapes and objects in raster images. Originally, Hough [34] defined the transformation for detecting lines; later it was extended for more complex shapes, such as circles, ellipses, etc., and even generalized for arbitrary patterns [5].

When used for detecting lines in 2D raster images, the Hough transform is defined by a *parametrization* of lines: each line is described by two parameters. The input image is preprocessed and for each pixel which is likely to belong to a line, voting accumulators corresponding to lines which could be coincident with the pixel are increased. Next, the accumulators in the parameter space are searched for local maxima above a given threshold, which correspond to likely lines in the original image. The Hough transform was formalized by Princen et al. [74] and described as a *hypothesis testing* process.

Hough [34] parametrized the lines by their slope and y-axis intercept. A very popular parametrization introduced by Duda and Hart [16] is denoted as $\theta - \varrho$; it is important for its inherently bounded parameter space. It is based on a line equation in the normal form: $y\,sin(\theta) + x\,cos(\theta) = \varrho$. Parameter θ represents the angle of inclination and ϱ is the length of the shortest chord between the line and the origin of the image coordinate system. There exist several other bounded parametrizations, mainly based on intersections of lines with image's bounding box [58, 17, 95]. Different properties of these intersects are used as parameters.

The majority of currently used implementations seems to be using the θ-ϱ parametrization – for example the well-known OpenCV library implements several variants of line detectors based on the θ-ϱ parametrization and none other. It is mainly because the parametrization uses a very straightforward transformation from the image space to one bounded space of parameters and because of its uniform distribution of the discretization error across the Hough space.

Several research groups invested effort to deal with computational complexity of the Hough transform based on the θ-ϱ parametrization. Different methods focus on special data structures, non-uniform resolution of the accumulation array or special rules for picking points from the input image.

O'Rourke and Sloan developed two special data structures: *dynamically quantized spaces* (DQS) [71] and *dynamically quantized pyramid* (DQP) [82]. Both these methods use splitting and merging cells of the space represented as a binary tree, or possibly a quad-tree. After processing the whole image, each cell contains approximately the same number of votes; that leads to a higher resolution of the Hough space of accumulators at locations around the peaks.

A typical method using special picking rules is the Randomized Hough Transform (RHT) [98]. This method is based on the idea, that each point in an n-dimensional Hough space of parameters can be exactly defined by an n-tuple of points from the input raster image. Instead of accumulation of a hypersphere in the Hough space for each point, n points are randomly picked and the corresponding accumulator in the parameter space is increased. Advantages of this approach are mostly in rapid speed-up and small storage. Unfortunately, when detecting lines in a noisy input image, the probability of picking two points from same line is small, decreasing the probability of finding the true line.

Another approach based on repartitioning the Hough space is represented by the Fast Hough Transform (FHT) [51]. The algorithm assumes that each edge point in the input image defines a hyperplane in the parameter space. These hyperplanes recursively divide the space into hypercubes and perform the Hough transform only on the hypercubes with votes exceeding a selected threshold. This approach reduces both the computational load and the storage requirements.

Using principal axis analysis for line detection was discussed by Rau and Chen [8]. Using this method for line detection, the parameters are first transferred to a one-dimensional angle-count histogram. After transformation, the dominant distribution of image features is analysed, with searching priority in peak detection set according to the principal axis. There exist many other accelerated algorithms, more or less based on the above mentioned approaches; e.g. HT based on eliminating of particle swarm [9] or some specialized tasks like iterative RHT [54] for incomplete ellipses and N-Point Hough transform for line detection [56]. For more information about different existing modifications of Hough transform, please see [36].

This section presents an algorithm for real-time detection of lines based on the standard Hough transform using the θ-ϱ parametrization. The classical Hough transform has some advantages over the accelerated and approximated methods (it does not introduce any further detection error and it has a low number of parameters and therefore usually requires less detailed application-specific fine-tuning). That makes the real-time implementation of the Hough transform desirable. The algorithm uses a modified strategy for accumulating the votes in the array of accumulators in the Hough space. The strategy was designed to meet the nature of today's graphics chips (GPUs).

5.1.1 Real-Time Hough Transform Algorithm

Before discussing the new real-time Hough transform algorithm, let us review the "classical" Hough transform procedure based on the θ-ϱ parametrization in Alg. 5.1 (the θ-ϱ parametrization itself is depicted by Fig. 5.1).

Points in the input image I with dimensions I_w and I_h are classified with a binary decision on line 3 (e.g. by en edge detector and thresholding). Lines 2–6 rasterize and accumulate curves into the Hough space. The function $\bar{\varrho}(\bar{\theta}, x, y)$ calculates the

Algorithm 5.1 HT for detecting lines based on the θ-ϱ parameterization.

```
Require: Input image $I$ with dimensions $I_w, I_h$, Hough space dimensions $H_\varrho, H_\theta$
Ensure: Detected lines $L = \{(\theta_1, \varrho_1), \dots\}$
1: $H(\bar{\varrho}, \bar{\theta}) \leftarrow 0, \forall \bar{\varrho} \in \{1, \dots, H_\varrho\}, \bar{\theta} \in \{1, \dots, H_\theta\}$
2: for all $x \in \{1, \dots, I_w\}, y \in \{1, \dots, I_h\}$ do
3:    if $I(x, y)$ is edge then
4:       increment $H\big(\bar{\varrho}(\bar{\theta}, x, y), \bar{\theta}\big), \forall \bar{\theta} \in \{1, \dots, H_\theta\}$
5:    end if
6: end for
7: $L = \{(\theta(\bar{\theta}), \varrho(\bar{\varrho})) | \bar{\varrho} \in \{1, \dots, H_\varrho\} \wedge \bar{\theta} \in \{1, \dots H_\theta\} \wedge$
   at $(\bar{\varrho}, \bar{\theta})$ is a high max. in $H\}$
```

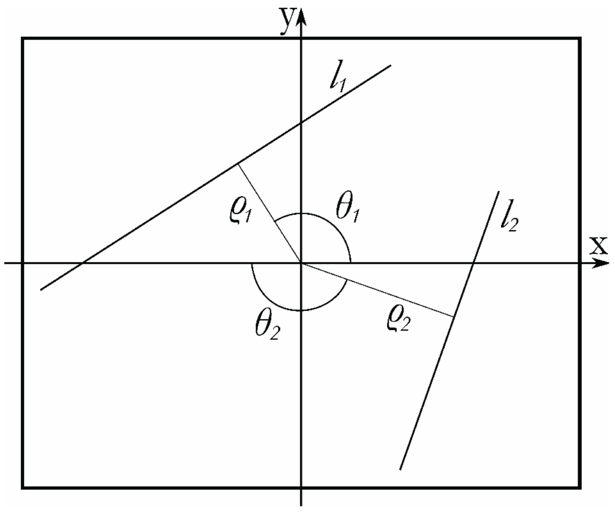


Figure 5.1: The θ-ϱ parametrization of lines in a coordinate system with origin in the centre of the input image.

corresponding $\bar{\varrho}$ for each line passing through point (x, y) at angle $\bar{\theta}$:

$$\bar{\varrho}(\bar{\theta}, x, y) = \left[\frac{H_\varrho\big((y - \frac{I_h}{2})\sin(\frac{\pi}{H_\theta}\bar{\theta}) + (x - \frac{I_w}{2})\cos(\frac{\pi}{H_\theta}\bar{\theta})\big)}{\sqrt{I_w^2 + I_h^2}} + \frac{H_\varrho}{2} \right]. \tag{5.1}$$

Line 7 detects above-threshold local maxima in the accumulated space and transforms the discretized Hough space coordinates $\bar{\varrho}$ and $\bar{\theta}$ to ϱ and θ by the following functions:

$$\begin{aligned} \varrho(\bar{\varrho}) &= \frac{\sqrt{I_w^2 + I_h^2}}{H_\varrho}\Big(\bar{\varrho} - \frac{H_\varrho}{2}\Big), \\ \theta(\bar{\theta}) &= \frac{\pi}{H_\theta}\bar{\theta}. \end{aligned} \tag{5.2}$$

Usually, a small neighbourhood (3×3 in OpenCV, 5×5 or 7×7 in cases of high resolution of the Hough space) is used for detecting the local maxima by line 7. The accumulator value must be above a given threshold to be considered for a "high local maxima". The threshold is another input parameter of the algorithm, but since it does not influence the algorithm's structure, it is used silently by line 7 for simplicity of the algorithmic notation.

The key characteristic of this algorithm is that line 4 must rasterize the half-period of the sinus curve and increment the corresponding accumulators in the Hough space. On some systems, such a large random-access read-write memory might be expensive or even not available at all.

5.1.2 CUDA Implementation

The key characteristic of Alg. 5.1 in the previous section is that steps 4 must rasterize the curves (the half-period of the sinus curve in the case of the h. parametrization) and increment the corresponding accumulators in the Hough space. In some systems, such a large random-access read-write memory might be expensive or even not available at all. This section presents an algorithm that overcomes this limitation and which is suitable for graphics processors and other special-purpose or embedded systems. The principle of these algorithms can work with other line parametrizations as well.

Hough Transform on a Small Read-Write Memory of Accumulators

The classical Hough transform accesses sparsely a relatively large amount of memory. This behaviour can diminish the effect of caching. On CUDA and similar architectures, this effect is even more significant, as the global memory is not cached. To achieve real-time performance, the memory requirements must be limited to the *shared memory* of a multiprocessor (typically 16 kB).

Alg. 5.2 shows the modified Hough transform accumulation procedure. The key difference from Alg. 5.1 is the actual size of the Hough space. The new algorithm stores only $H_\theta \times n$ accumulators, where n is the neighbourhood size required for the maxima detection. Functions $\bar{\varrho}, \theta, \varrho$, and the edge and maxima detection are identical to Alg. 5.1. First, the detected edges are stored in a set P (line 1). Then, first n rows of the Hough space are computed by lines 2–7. The memory necessary for containing the n lines is all the memory required by the algorithm and even for high resolutions of the Hough space, the buffer of n lines fits easily in the *shared memory* of the GPU multiprocessors.

In the main loop (lines 9–18), for every row of the Hough space, the maxima are detected (line 10), the accumulated neighbourhood is shifted by one row (lines 11–13) and a new row is accumulated (lines 14–17); please refer to Fig. 5.2 for an illustration of the algorithm. Thus only the buffer of n lines is being reused. The memory shift can be implemented using a circular buffer of lines to avoid data copying.

Algorithm 5.2 HT accumulation strategy using a small read-write memory.

```
Require: Input image $I$ with dimensions $I_w, I_h$, Hough space dimensions $H_\varrho, H_\theta$,
    neighborhood size $n$
Ensure: Detected lines $L = \{(\theta_1, \varrho_1), \dots\}$
 1: $P \leftarrow \{(x,y) | x \in \{1,\dots,I_w\} \wedge y \in \{1,\dots,I_h\} \wedge I(x,y)$ is an edge$\}$
 2: $H(\bar{\varrho}, i) \leftarrow 0, \forall \bar{\varrho} \in \{1,\dots,H_\rho\}, \forall i \in \{1,\dots.n\}$
 3: for all $i \in \{1,\dots,n\}$ do
 4:   for all $(x,y) \in P$ do
 5:     increment $H(\bar{\varrho}(i,x,y), i)$
 6:   end for
 7: end for
 8: $L \leftarrow \{\}$
 9: for $\bar{\theta} = \lceil \frac{n}{2} \rceil$ to $H_\theta - \lfloor \frac{n}{2} \rfloor$ do
10:   $L \leftarrow L \cup \{(\theta(\bar{\theta}), \varrho(\bar{\varrho})) | \bar{\varrho} \in \{1,\dots H_\varrho\} \wedge (\bar{\varrho}, \lceil \frac{n}{2} \rceil)$ is a high local max. in $H\}$
11:   for $i = 1$ to $n-1$ do
12:     $H(\bar{\varrho}, i) \leftarrow H(\bar{\varrho}, i+1), \forall \bar{\varrho} \in \{1,\dots,H_\varrho\}$
13:   end for
14:   $H(\bar{\varrho}, n) \leftarrow 0, \forall \bar{\varrho} \in \{1,\dots,H_\varrho\}$
15:   for all $(x,y) \in P$ do
16:     increment $H(\bar{\varrho}(\bar{\theta} + \lceil \frac{n}{2} \rceil, x, y), n)$
17:   end for
18: end for
```

In the pseudo-code, maxima are not detected at the edges of the Hough space. Eventual handling of the maxima detection at the edge of the Hough space does not change the algorithm structure, but it would unnecessarily complicate the pseudo-code. Two solutions exist – either copying the border data or rasterizing necessary parts of the curves outside of the Hough space. Both approaches perform similarly and their implementation is straightforward.

On CUDA, the threads in a block can be used for processing the set of edges P (lines 15–17 and 4–6) in parallel, using an atomic increment of the shared memory to avoid read-write collisions. In order to use all the multiprocessors of the GPU, the loop on line 9 is broken to a number (e.g. 90 is suitable for current NVIDIA GeForce graphics chips) of sub-loops processed by individual blocks of threads.

The algorithm as described above uses exactly $H_\varrho \times n$ memory cells, typically 16-bit integer values. In the case when the runtime system has more fast random-access read-write memory, this memory can be used fully, and instead of accumulating one line of the Hough space (lines 15–17 of the algorithm), several lines are are accumulated and then scanned for maxima (line 10). This leads to further speed-up by reducing the number of steps carried out by the loop over $\bar{\theta}$ (line 9).

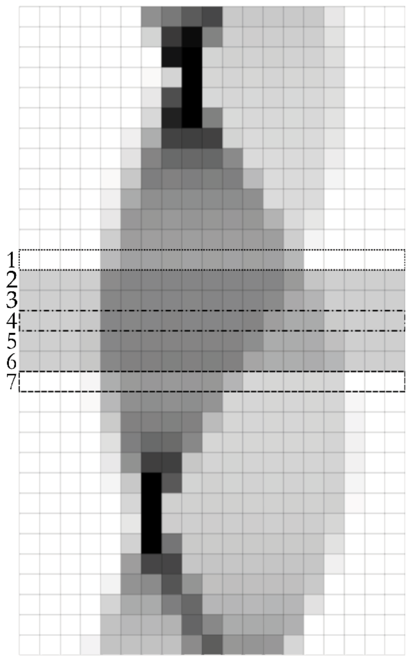


Figure 5.2: Illustration of Alg. 5.2. The grey rectangle represents the buffer of n lines. For row 4, the above-threshold maxima are detected in each step within the buffer. Then, the row 7 values are accumulated into the buffer, using the space of row 2, which will not be needed in future processing.

Harnessing the Edge Orientation

In 1976 O'Gorman and Clowes came with the idea not to accumulate values for each θ but just one value instead [66]. The appropriate θ for a point can be obtained from the gradient of the detected edge which contains this point [80]. One common way to calculate the local gradient direction of the image intensity is using the Sobel operator. Sobel detector uses two kernels, each approximates the derivation in horizontal (G_x), respectively vertical (G_y) direction. Sobel kernels for convolution are as follows: $G_x = [1, 2, 1]^T \cdot [1, 0, -1]$ and $G_y = [1, 0, -1]^T \cdot [1, 2, 1]$. Using these two values, the gradient's direction can be obtained as $\theta = \arctan(\frac{G_y}{G_x})$. To avoid errors caused by noise and rasterization, accumulators within several degrees around the calculated angle are also incremented. From experimental testing, the interval's radius equal to 20° seems suitable. This approach reduces the computation time and highlights the maxima peaks. A disadvantage of this method is its dependency of the results on another user parameter – the radius. Small radius of the incremented interval of θ can lead into discarding some maxima due to inaccurate θ location. On the other hand, too high a radius can diminish the performance benefits of the method.

This approach to utilizing the detected gradient can be incorporated to the new accumulation scheme presented in the previous section. When extracting the "edge points" for which the sinusoids are accumulated in the Hough space (line 1 in Alg. 5.2), also the edge inclination is extracted:

```
1: $P \leftarrow \{(\alpha, x, y) | x \in \{1, \ldots, I_w\} \wedge y \in \{1, \ldots, I_h\}$
           $\wedge\ I(x, y)$ is an edge with gradient slope $\alpha\}$.
```

$P \leftarrow \{(\alpha, x, y) | x \in \{1, \ldots, I_w\} \wedge y \in \{1, \ldots, I_h\} \wedge I(x, y)$ is an edge with gradient slope $\alpha\}$.

Then, instead of accumulating all points from set P (lines 4–6), only those points which fall into the interval with radius w around currently processed θ are processed and accumulated into the buffer of n lines:

```
4: for all $(\alpha, x, y) \in P \wedge\ i - w < \bar{\alpha} < i + w$ do
5:     increment $H(\bar{\varrho}(i, x, y), i + \lfloor \frac{n}{2} \rfloor)$
6: end for
```

and similarly for lines 15–17:

```
16: for all $(\alpha, x, y) \in P \wedge\ \bar{\theta} + \lfloor \frac{n}{2} \rfloor - w < \bar{\alpha} < \bar{\theta} + \lfloor \frac{n}{2} \rfloor + w$ do
17:     increment $H(\bar{\varrho}(\bar{\theta} + \lfloor \frac{n}{2} \rfloor, x, y), n)$
18: end for.
```

Please, note that the edge extraction phase (line 1) can sort the detected edges by their gradient inclination α, so that loops on lines 15–17 and 4–6 do not visit all edges, but only edges potentially accumulated, based on the current $\bar{\theta}$ (line 9 of Alg. 5.2). For (partial) sorting of the edges on GPU, an efficient prefix sum can be used [26].

5.1.3 Performance Evaluation of Hough Transform

This section evaluates the speed of the newly presented line-detection algorithm, which is explained more in detail within 5.1. Two groups of experiments were made:

- the first one was focused on the speed-up in the case when ϱ was calculated for each θ (see 5.1.2, Alg. 5.2);
- the second test evaluated the situation when the Sobel operator was used for detection of edge orientation and only an interval of the sinusoid curves was accumulated to the Hough space (see 5.1.2).

Each test compared the computation time of 4 implementations:

- ASUS nVIDIA GTX480 graphics card (1.5GB GDDR5 RAM) running the new algorithm;

- ASUS nVIDIA GTX280 graphics card (1GB GDDR3 RAM) running the same code;
- an OpenMP parallel CPU implementation of the presented algorithm (Intel Core i7-920, 6GB 3×DDR3-1066(533MHz) RAM – the same machine was used for evaluating the GPU variants);
- and an OpenMP parallel "standard" implementation running on the same machine. As the "standard" implementation, the code based on OpenCV functions was used and optimized by parallelization.

Synthetic Binary Images

As the dataset for this experiment we used automatically generated black-and-white images. The generator randomly placed L white lines and then inverts pixels on P different positions in the image. The evaluation was done on 36 images (resolution 1600×1200): images 1–6, 7–12, 13–18, 19–24, 25–30, 31-36 were generated with $L = 1, 30, 60, 90, 120, 150$ respectively, with increasing $P = 1, 3000, 6000, 9000, 12000, 15000$ for each L. The parameters of the experiments were $H_\varrho = 960$ and $H_\theta = 1170$ (resolution of the Hough space) and the threshold for accumulators in the Hough space was 400.

Fig. 5.3 reports the results of the four implementations. Please note that the CUDA version is several times faster than the commonly used OpenCV implementation (parallelized to utilize the 8 cores of the processor) and achieves real-time or nearly real-time speeds.

Real-Life Images

The images used in this test were real-world images depicted by Fig. 5.4. For possibility of comparison with previous test, resolution of Hough space was same; i.e. $H_\varrho = 960$ and $H_\theta = 1170$; the threshold for accumulators in the Hough space was dependant on the input image resolution (one fourth of the diagonal); this corresponds to the shortest possible line detected by Hough transform); the radius of the accumulated interval (see 5.1.2) was 20°.

Fig. 5.5 contains the measured results. The results indicate that even for complex real-world images and high-resolution Hough space, the proposed algorithm implemented on commodity graphics hardware can detect lines at interactive frame rates. Contrary to the version that works with the whole sinusoids in the Hough space (see 5.1.3), the speed of the CPU implementation of the presented algorithm is about as fast as the standard CPU version. This can be explained by better cache coherency when only fractions of the sinusoids are rasterized. However, for efficient implementation on CUDA and similar architectures, the presented algorithm is required.

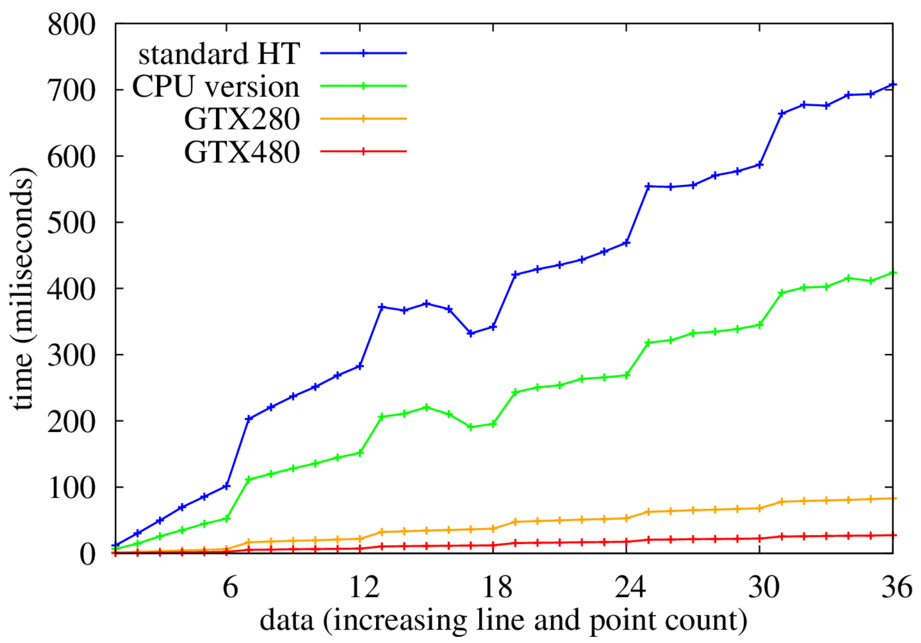


Figure 5.3: Performance Evaluation on Synthetic Binary Images. Red: GTX480, Orange: GTX280, Green: Striped algorithm on the CPU, Blue: Standard HT accumulation.

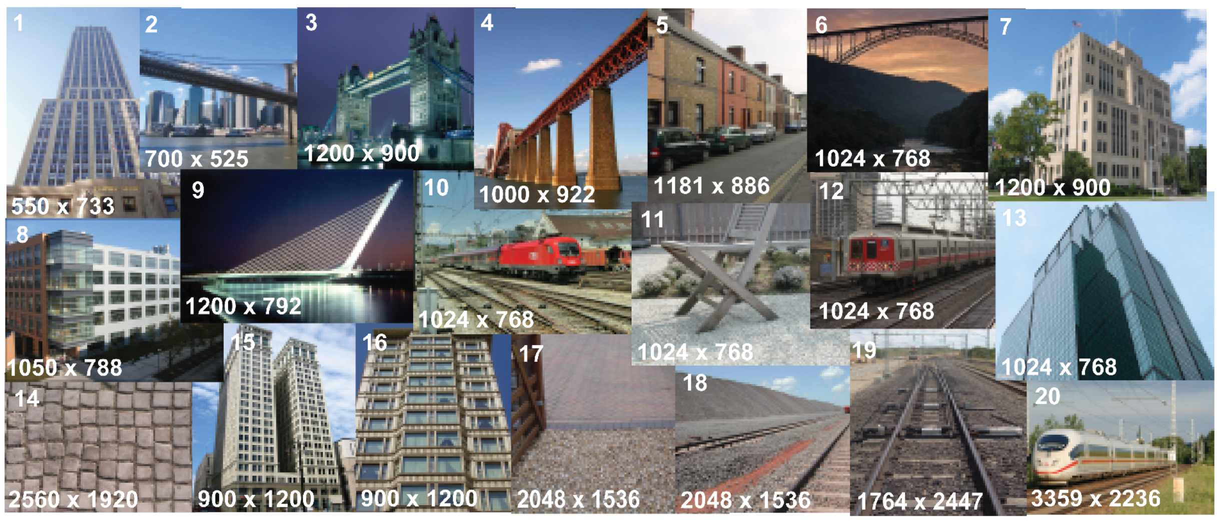


Figure 5.4: Images used in the test. The number in the top-left corner of each thumbnail image is the image ID – used on the horizontal axis in Figure 5.5. The bottom-left corner of each thumbnail states the pixel resolution of the tested image.

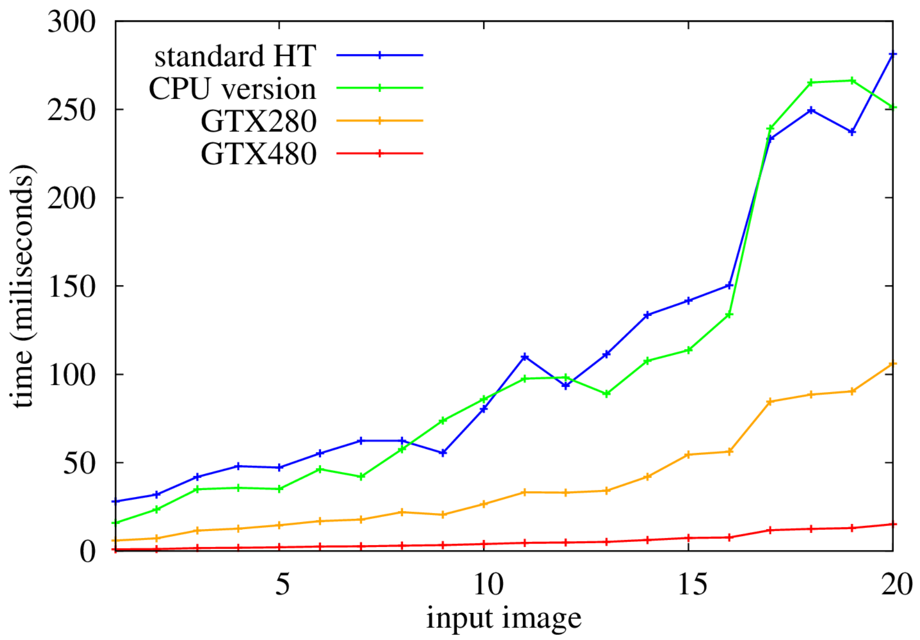


Figure 5.5: Performance evaluation on real-world images (see Fig. 5.4) using the Sobel operator and only accumulating intervals of the sinusoids. Red: GTX480, Orange: GTX280, Green: Striped algorithm on the CPU, Blue: Standard HT accumulation.

5.2 Line Detection Using Parallel Coordinates

The following section reviews existing parametrizations of lines suitable for a fast and/or precise detection of lines. The classical Hough transform (based on any parametrization) has some advantages over the accelerated and approximated methods (it does not introduce any further detection error and it has a low number of parameters and, therefore, usually requires less detailed application-specific fine-tuning). This makes the real-time implementation of the Hough transform desirable.

This study presents an algorithm for real-time detection of lines based on the PClines parametrization of lines. The algorithm discussed within this section used a modified strategy for accumulating the votes in the array of accumulators in the Hough space. The strategy was designed to meet the nature of today's graphics chips (GPUs) and other special-purpose computational platforms. The implementation achieves real-time performance at executing the "full" Hough transform on the GPU.

Parallel coordinates (PC) were invented in 1885 by d'Ocagne [13] and they were further studied and popularized by Inselberg [37]. The coordinate system used for representing geometric primitives in parallel coordinates is defined by mutually parallel axes. Each N-dimensional vector is represented by (N - 1) lines connecting the axes (see Fig. 5.6). In this thesis, we will be using an Euclidean plane with a u–v Cartesian coordinate system to define positions of points in the space of parallel coordinates. For defining these points, a notation $(u, v, w)_{\mathbb{P}^2}$ will be used for homogeneous coordinates in the projective space $\mathbb{P}^2$ and $(u, v)_{\mathbb{E}^2}$ will be used for Cartesian coordinates in the Euclidean space $\mathbb{E}^2$:

In the two-dimensional case, points in the x–y space are represented as lines in the space of parallel coordinates. Representations of collinear points intersect at one point– the representation of a line (see Fig. 5.7).

Based on this relationship, it is possible to define a point-to-line mapping between the original x–y space and the space of parallel coordinates. For some cases, such as line $\ell : y = x$; the corresponding point ' in the parallel coordinates lies in infinity (it is an ideal point) and the points on this line are represented by the parallel horizontal lines. Projective space $\mathbb{P}^2$ (contrary to the Euclidean $\mathbb{E}^2$ space) provides coordinates for these special cases. A relationship between line $\ell : ax + by + c = 0$ (denoted as $[a, b, c]$) in Cartesian coordinates and its representing point $\overline{\ell}$ in parallel coordinates can be defined by mapping:

$$\ell : [a, b, c] \rightarrow \ell : (db, -c, a + b)_{\mathbb{P}^2} \tag{5.3}$$

where d is the distance between parallel axes *x0* and *y0*.

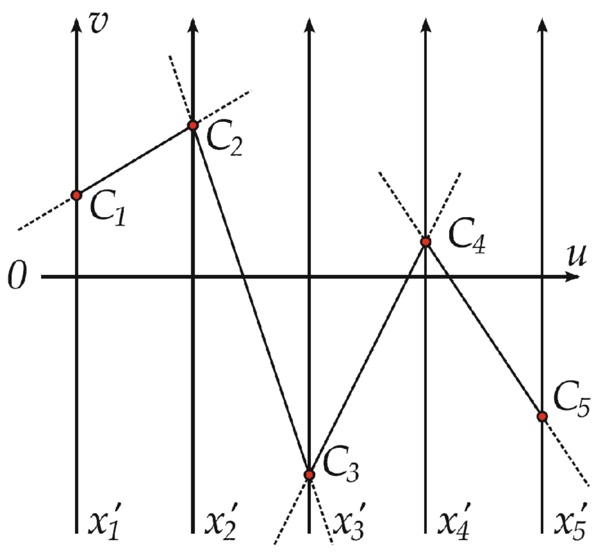


Figure 5.6: Representation of a 5-dimensional Vector in Parallel Coordinates. The vector is represented by its coordinates $C_1, \dots, C_5$ on axes $x'_1, \dots, x'_5$, connected by a complete poly line (composed of 4 infinite lines).

Parametrization "PClines" for Line Detection

This section gives an overview of the "PClines" parametrization introduced by Dubská et al. [15]. The text is kept very concise; for more information, the original paper should be consulted. In the following section, we will use the intuitive slope-intercept line equation $y = mx + b$ where m defines the slope of the line and b the y-coordinate of an intersection between the line and y-axis. Using this parametrization, the corresponding point $\overline{\ell}$ in the parallel space has coordinates $\left(d, b, 1-m\right)_{\mathbb{P}^2}$. The line's representation $\overline{\ell}$ is between the axes x' and y' if and only if $-\infty < m < 0$. For $m = 1$, $\overline{\ell}$ is an ideal point (a point in infinity). For $m = 0$, $\overline{\ell}$ lies on the y' axis, for vertical lines ($m = \pm\infty$), $\overline{\ell}$ lies on the x' axis. The system defined by parallel axes x', y' is further referred as straight ($\mathcal{S}$) space.

The representations of the lines with a positive slope lie in an infinite area outside the space between axes x', y'. To enclose also these representations to a finite part, we propose a *twisted* ($\mathcal{T}$) system x', $-y'$, which is identical to the straight space, except that the y'-axis is inverted. In the twisted space, $\overline{\ell}$ is between the axes x' and $-y'$ if and only if $0 < m < \infty$. By combining the straight and twisted spaces, the whole $\mathcal{TS}$ plane can be constructed, as shown in Fig. 5.8.

Fig. 5.8 (left view) shows the original x–y image with three points A, B, and C and three lines ℓ_1, ℓ_2, and ℓ_3 coincident with the points. The origin of x–y is placed into the middle of the image for the convenience of the figures and the right view depicts the corresponding $\mathcal{TS}$ space. It should be noted that a finite part of the u–v plane sufficient

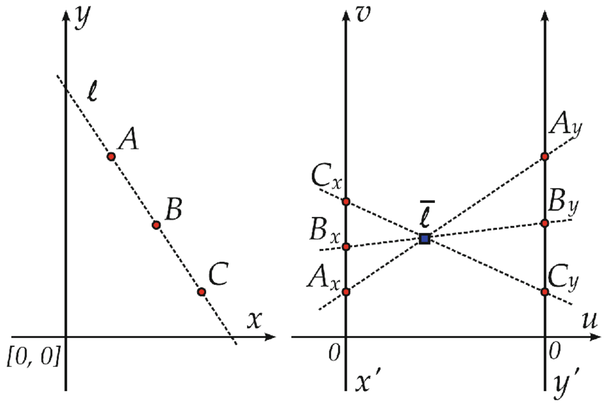


Figure 5.7: Three collinear points in parallel coordinates: (left) Cartesian space and (right) space of parallel coordinates. Line ℓ is represented by point $\overline{\ell}$ in parallel coordinates.

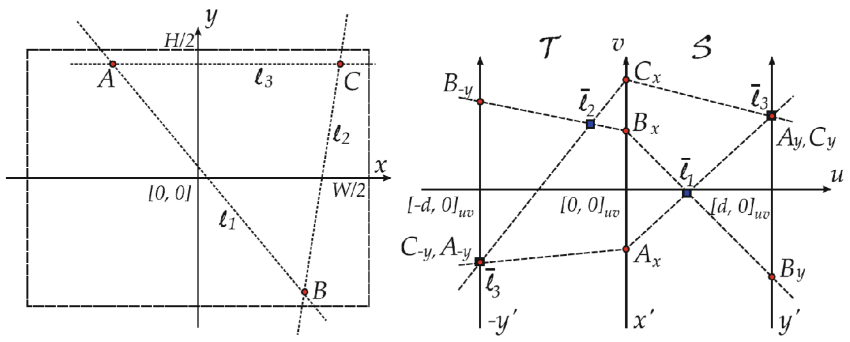


Figure 5.8: Left Original x–y space and right its PClines representation, the corresponding $\mathcal{TS}$ space.

for representing all possible lines in the bordered input image is defined as follows:

$$\begin{gathered} -d \leq u \leq d \\ -max\left(\frac{W}{2}, \frac{H}{2}\right) \leq v \leq max\left(\frac{W}{2}, \frac{H}{2}\right) \end{gathered} \tag{5.4}$$

where W and H are the width and height of the input raster image, respectively.

Any line $\ell : y = mx + b$ is now represented either by point $\overline{\ell}_{\mathcal{S}}$ in the *straight* space or by $\overline{\ell}_{\mathcal{T}}$ in the *twisted* space of the u–v plane:

$$\begin{aligned} \overline{\ell}_S &= (d, b, 1 - m)_{\mathbb{P}^2} & -\infty \leq m \leq 0 \\ \overline{\ell}_T &= (-d, -b, 1 + m)_{\mathbb{P}^2} & 0 \leq m \leq \infty \end{aligned} \tag{5.5}$$

Consequently, any line ℓ has exactly one image $\overline{\ell}$ in the $\mathcal{TS}$ space; except for cases that $m = 0$ and $m = \pm\infty$, when $\overline{\ell}$ lies in both spaces either on y' or x'-axis. That allows the $\mathcal{T}$ and $\mathcal{S}$ spaces to be "attached" one to another. Figure 3 illustrates the spaces attached along the x'axis. Attaching also the y' and $-y'$axes results in an enclosed Mobius strip.

Eq. 5.5 defines line-to-point mapping which can be used as a parametrization for the Hough transform. In this case, the $\mathcal{TS}$ space is used as an accumulator space, as depicted in Alg. 5.3.

Algorithm 5.3 Detection of Lines Using Parallel Coordinates

```
Require: Input image $I$ with dimensions $W$, $H$
Ensure: Detected lines $L = \{(m_1, b_1), \ldots\}$
 1: $S(u,v) \leftarrow 0, \forall u \in \{-d, \ldots, d\}, v \in \{v_{min}, \ldots, v_{max}\}$
 2: for all $x \in \{1, \ldots, W\}, y \in \{1, \ldots, H\}$ do
 3:    if $I(x,y)$ is an edge then
 4:       rasterize line in the $\mathcal{S}$ space
 5:       rasterize line in the $\mathcal{T}$ space
 6:    end if
 7: end for
 8: $L \leftarrow \{\}$
 9: $L = \big\{(m(u), b(u,v))|u \in \{-d, \ldots d\} \wedge$
        $v \in \{v_{min}, \ldots, v_{max}\} \wedge S(u,v)$ is a high local max.$\}$
```

The space $\mathcal{TS}$ is discretized directly according to Eq. 5.4; other discretizations (denser or sparser) would be possible by just linearly mapping the u and v coordinates used in the algorithm. The condition used in step 3 is application specific and it typically involves an edge detection operator and thresholding. The lines rasterized in Steps 4 and 5, in fact, constitute a two-segment polyline defined by three points: $(-d, -y) - (0, x) - (d, y)$; where $(-d, -y)$ and $(0, x)$ are vertices of the line accumulated in the $\mathcal{T}$ half and $(0, x)$ and (d, y) are vertices of the line accumulated in the $\mathcal{S}$ half. Step 9 scans the space of accumulators $\mathcal{S}$ for local maxima above a given threshold-this is a standard Hough transform step. The line's parameters m–b are computed by the functions $m(u)$ and $b(u, v)$ based on the u and v coordinates of the point in the $\mathcal{TS}$ space using Eq. 5.3; any other parametrization of lines can be the output of the algorithm.

Step 9 of the pseudo-code looks for local maxima above a given threshold in the $\mathcal{TS}$ space. Usually, a small neighbourhood (3×3; 5×5 or 7×7 in cases of high resolution of the Hough space) is used for detecting the local maxima. The accumulator value must be above a given threshold to be considered for a "high local maxima". The threshold is another input parameter of the algorithm, but since it does not influence the algorithm's structure, it is used silently by Step 9 for simplicity of the algorithmic notation.

5.2.1 CUDA Implementation

The key characteristic of Alg. 5.3 in the previous section is that steps 4 and 5 must rasterize the lines in the $\mathcal{T}$ and $\mathcal{S}$ spaces (or the half-period of the sinus curve in the case of the h. parametrization) and increment the corresponding accumulators in the Hough space. In some systems, such a large random-access read-write memory might be expensive or even not available at all. It builds upon an algorithm recently published by the authors of this article [42].

As already discussed, algorithm for rasterization into Hough space in 5.1.2, a principle of Hough transform algorithm remains same. The difference is only in a way of parametrization, originally $\theta - \varrho$, into $\mathcal{TS}$ parametrization [28].

Harnessing the edge orientation is also based on 5.1.2 with difference in $\mathcal{TS}$ space modification.

5.2.2 OpenGL Implementation as a Reference

Contrary to the "standard" $\theta - \varrho$ parametrization where sinusoids need to be rasterized into the accumulator space, in the case of PClines, for each edge point detected in the input image, two-line segments were rasterized. Rasterization of line segments (and blending the rasterized pixels into a frame buffer) is a natural task for the graphics chips. There is a separate paper published on OpenGL implementation of the PClines [14]. The whole process was done by the graphics chip, programmed in OpenGL and GLSL:

- *Edges are extracted* by a geometry shader which accesses a texture with the input image and, for each pixel in the input image, it emits zero, two, or three endpoints of a poly-line to be rasterized into the $\mathcal{TS}$ space;
- *Line segments are rasterized* by OpenGL and blended into the frame buffer;
- *The $\mathcal{TS}$ space is searched* by another geometry shader which emits the parameters of detected lines.

This implementation using OpenGL and GLSL will be used as a reference and referred to as "PClinesGL" in the charts. For more information on the algorithm and its implementation, please refer to the original paper [14].

5.2.3 Performance Evaluation of Parallel Coordinates

This section presents the experimental evaluation of the proposed algorithm of PC, which are explained in 5.2 and briefly describes a PClines-based algorithm for OpenGL that was used as a reference in the measurements. 5.2.3 contains the results achieved by a CUDA implementation of the PClines-based algorithm presented in this section compared to other implementations.

The following hardware was used for testing (in bold face is the identifier used later on in this text):

- **GTX480**: NVIDIA GTX 480 in a computer with Intel Core i7-920, 6 GB 3×DDR3-1066 (533 MHz) RAM;
- **GTX280**: NVIDIA GTX 280 in a computer with Intel Core i7-920, 6 GB 3×DDR3-1066 (533 MHz) RAM;
- **HD5970-1**: AMD Radeon HD5970 (single core used) in a computer with Intel Core i5-660, 4 GB 3×DDR3-1066 (533 MHz) RAM;
- **HD5970-2**: AMD Radeon HD5970 (both cores used) in a computer with Intel Core i5-660, 4 GB 3×DDR3-1066 (533 MHz) RAM; and
- **i7-920**: Intel Core i7-920, 6 GB 3×DDR3-1066 (533 MHz) RAM—the same computer is used for testing the GTX 480 and GTX 280.

An evaluation of the accuracy of the PClines line parametrization can be found in paper where the PClines parametrization was introduced in [15]. The measurements reported that PClines are equal or more accurate than the "standard" $\theta - \varrho$ parametrization.

Real-Life Images

Two datasets were used for measuring the performance of different algorithms. The first one was a set of real photographs with different amounts of edge points and different dimensions (Fig. 5.9).

The images are sorted according to the number of edge points detected by the Sobel filter. Only this limited set of images is selected for the graphs to be readable. The images were selected randomly from a large set of images and they well represent the behaviour of the algorithms for all images we have observed.

The presented algorithm (referred to below as **PClines-CUDA**) was compared to different alternatives:

- Software implementations of the PClines based on a Hough transform implementation taken from the **OpenCV** library and parallelized by OpenMP and slightly optimized;
- A CUDA implementation of the standard $\theta - \varrho$. parametrization (**ThetaRho-CUDA**). The arrangement of the algorithm is very similar to the presented PClines-based one;
- The OpenGL implementation of PClines (**PClines-OpenGL**) as described in 5.2.2.

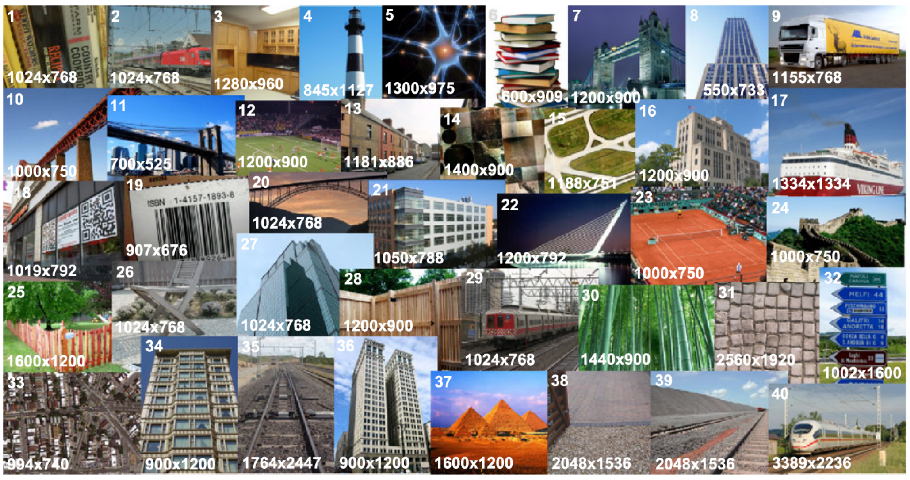


Figure 5.9: Images used in the test. The number in the top-left corner of each thumbnail image is the image ID used on the horizontal axis in Figs.5.10 and 5.11. The bottom-left corner of each thumbnail image states the number of edge points and pixel resolution of the tested image.

The results are shown in Fig. 5.10. The measurements verify that the computational complexity is linearly proportional to the number of edge points extracted from the input image and the edge-detection phase is linearly proportional to the image resolution. The GPU-accelerated implementations are notably faster than the software implementation.A detailed comparison of the GPU-accelerated implementations is shown in Fig. 5.11.

Synthetic Binary Images

The second dataset consisted of automatically generated black-and-white images. The generator randomly places L white lines in an originally black image and then inverts pixels on P random positions in the image. The evaluation is done on 36 images (resolution 1600×1200): images 1–6, 7–12, 13–18, 19–24, 25–30, 31–36 are generated with $L = 1$, 30, 60, 90, 120, 150, respectively, with increasing $P = 1$, 3000; 6000; 9000; 12000 for each L. The suitable parameters for images of these properties were $H_\varrho = 960$ and $H_\theta = 1170$ (resolution of the Hough space) and the threshold for accumulators in the Hough space was 400. The purpose of this test was to accurately observe the dependency of processing time on the number of lines in the image and on the number of pixels processed as edges. These two quantities determine the number of repetitions in critical parts of the algorithm.

Fig. 5.12 shows the results of the four implementations; Fig. 5.13 contains a selection of the graphs-only the hardware-accelerated methods. Once again, it should be noted that all the accelerated versions are several times faster than the commonly used OpenCV implementation and achieve real-time or near real-time speeds even for high-resolution

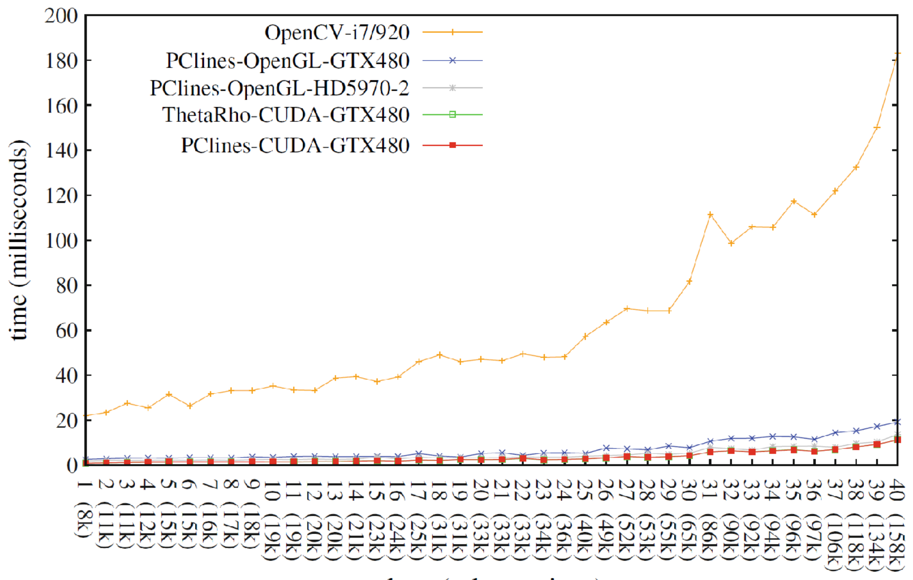


Figure 5.10: Performance evaluation on real-world images (see Fig. 5.9) using the Sobel operator and only accumulating an interval on the u-axis (see 5.2.1).

inputs.

On current graphics chips, the algorithm presented here (PClines–CUDA) and the previously published algorithm (ThetaRho-CUDA) perform equally fast (it should be noted that, in Fig. 5.11 and 5.13, their curves totally overlap). On special, embedded, and low-power architectures, the PClines-based version may perform much better or can be the only feasible one, because it requires no floating-point computations and no goniometric functions (which are cheaply available on the GPUs). The only advantages of the PClines-based algorithm on GPU is, therefore, its better accuracy [15] and its ability to directly detect parallel lines and sets of lines coincident with one point.

Fig. 5.11 and 5.13 show that, on the pre-Fermi NVIDIA card (GTX280), the OpenGL version of the PClines-based Hough transform performs better than CUDA. That is because the atomic increment operation (atomicInc) in the shared memory is not optimized on this generation of the graphics chips. Very good results also come from recent Radeon graphics chips (with the OpenGL version). Fig. 5.11 and 5.13 also show that the OpenGL algorithm by Dubská et al. [14] scales well on the dual-core graphics card Radeon HD5970. When executed on both the cores, the speed is almost doubled compared to the single-core version. A comparable scaling is achieved also on the CUDA version of the algorithm. However, on CUDA, the problem must be “manually” divided into an appropriate number of blocks within the kernel. Such a division is discussed in 5.2.1.

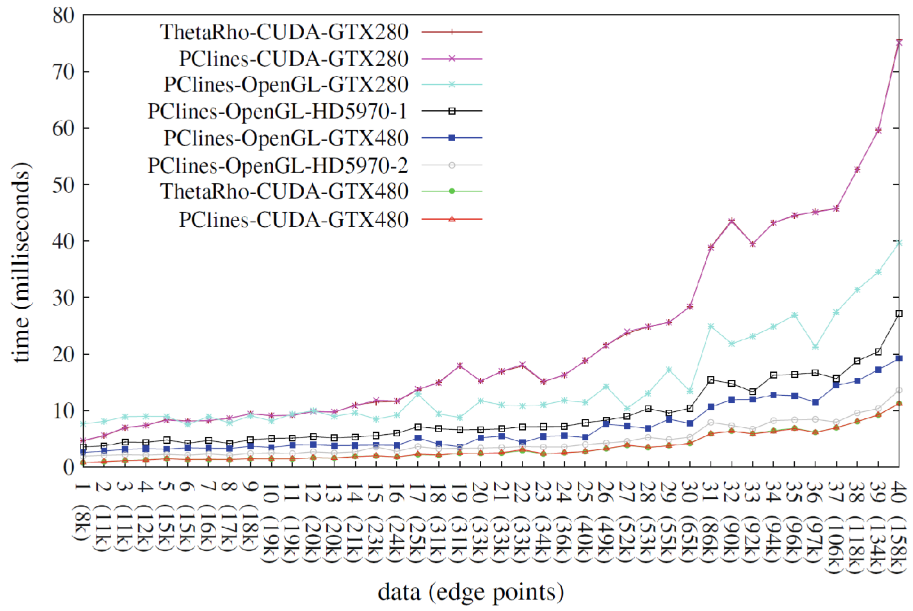


Figure 5.11: Performance evaluation on real-world images (see Fig. 5.9) using the Sobel operator and only accumulating an interval on the u-axis (Sect. 5.2.3). Only the hardware-accelerated methods are shown here for better clarity.

Discussion

The Fermi architecture (compared to the previous generation) speeded up the algorithm in the OpenGL version just the amount which can be expected from the increase in the number of the streaming multiprocessors. However, the CUDA version presented in this study speeded up notably more (about 4 times) on the Fermi architecture. This can be explained by the improved atomic operations in the shared memory, involving the new design of the L2 cache on the GTX480 [[20]]. Attribution of the performance boost between the GTX280 and GTX480 to the atomic instructions was verified by running the algorithm with the non-atomic equivalents of the increment/add instructions (Fig. 5.14). For weaker graphics chips (low-power, mobile, etc.), the OpenGL version of the PClines-based algorithm might be the right choice.

We have evaluated several different configurations of the shared memory as it is used by the algorithm. Namely, different number of columns can be allocated for the circular buffer of columns, as noted in the last paragraph of 5.2.1. We allocated varying numbers of these columns and observed the results in Fig. 5.15. Different configurations of the shared memory also illustrate the performance of the algorithm in terms of being computation/memory bound. We measured instructions per cycle (1/CPI) and the effective bandwidth in Fig. 5.16. These measurements indicate that the algorithm is mostly computation bound and using the whole shared memory helps in accessing the

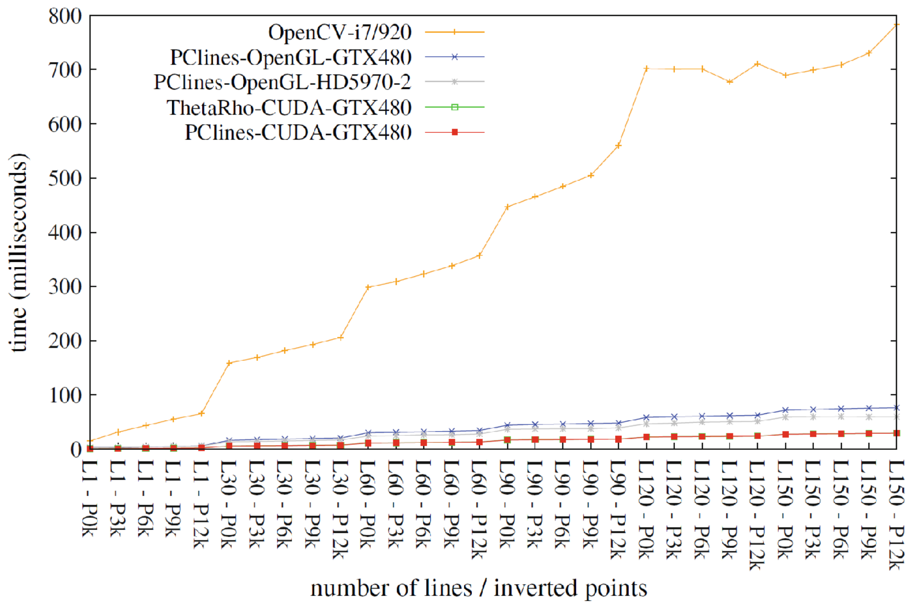


Figure 5.12: Performance Evaluation on Generated Data

global memory more efficiently. This behaviour reflects the nature of the algorithm which was designed to be using memory efficiently by processing the data in stripes. This access strategy helps serialize and minimize the accesses to the global memory.

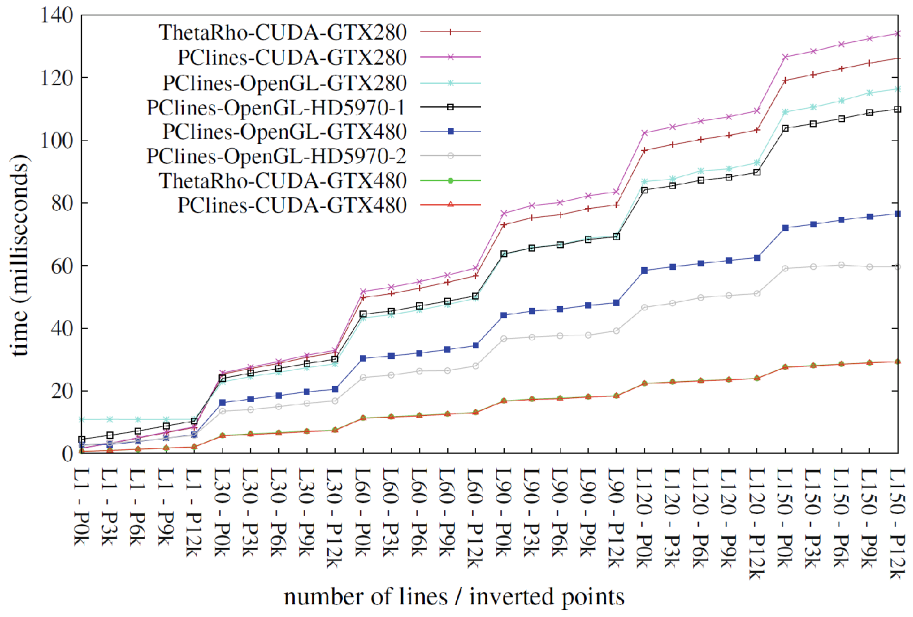


Figure 5.13: Performance evaluation on generated data. Only the hardware-accelerated methods are shown here for better clarity.

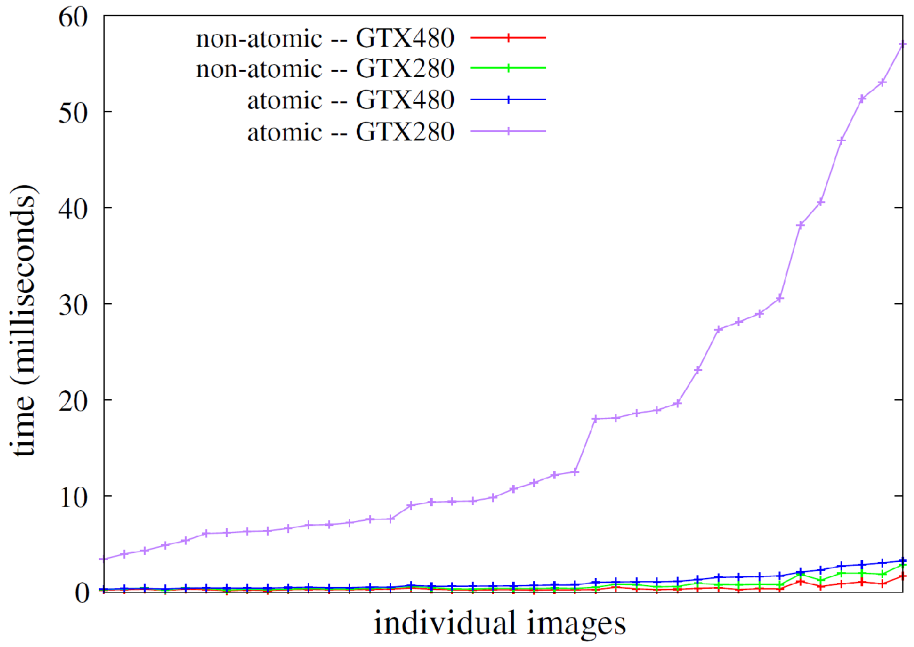


Figure 5.14: Comparison of the speed on graphics cards of two different generations: GTX480 and GTX280. In the case of GTX480, execution without atomic instructions (atomic add and inc were replaced by non-atomic equivalents) is about three times faster (*blue*, *red*). However, in the case of GTX280 (*magenta*, *green*), the performance when using atomic instructions is about 259 slower. It should be noted that this includes only the edge-detection part of the algorithm. This part is the most time-consuming one and more importantly it is much more prone to the speed of atomic instructions. The rest of the algorithm is severely affected by the incorrect results produced by non-atomic operations and thus their timing was omitted.

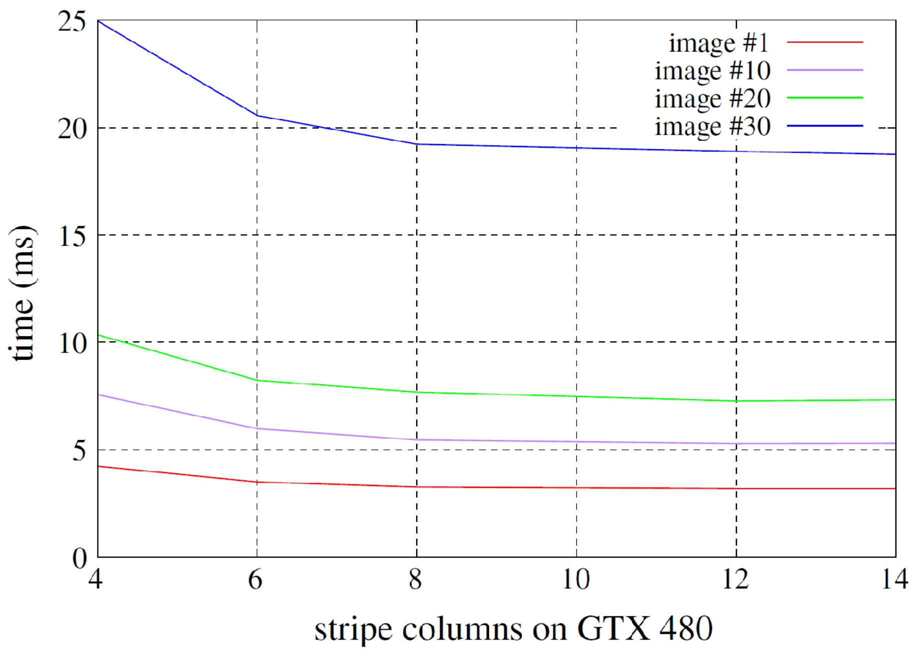


Figure 5.15: Time performance for several selected images from Fig. 5.9 for different configurations of the shared memory usage (i.e., number of spare columns used by the algorithm). Note that as expected in the algorithm design, using the whole shared memory for the accumulation buffer indeed speeds the computation up. However, for high number of blocks within the kernel, the impact of this improvement is diminished and also, very large shared memory would not help notably any more (as illustrated in Fig. 5.16). Time performance for several selected images from Fig. 5.9 for different configurations of the shared memory usage (i.e., number of spare columns used by the algorithm). Note that as expected in the algorithm design, using the whole shared memory for the accumulation buffer indeed speeds the computation up. However, for high number of blocks within the kernel, the impact of this improvement is diminished and also very large shared memory would not help notably any more (as illustrated in Fig. 5.16).

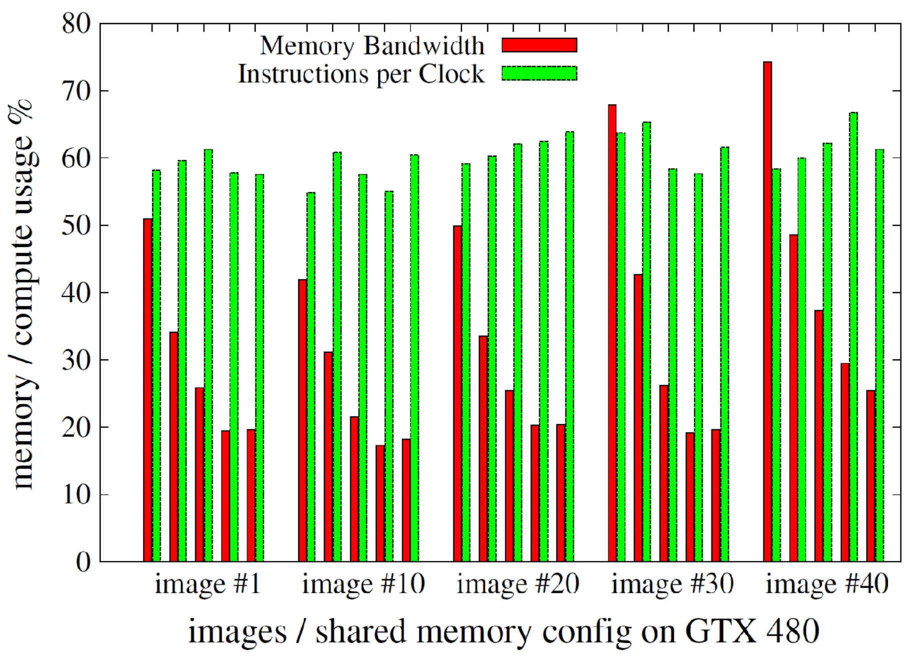


Figure 5.16: Usage of the graphics chip in terms of memory and computation percentual load compared to theoretical limits. Green boxes represent percentual usage of the computational power of the graphics board (CPI/theoretical maximum). Red boxes reflect the usage of the theoretical memory bandwidth (effective bandwidth/ theoretical max). The graph shows five series of measurements on five different images (selected from Fig. 5.9); a single measurement within the series represents one shared memory configuration, equally as in Fig. 5.15.

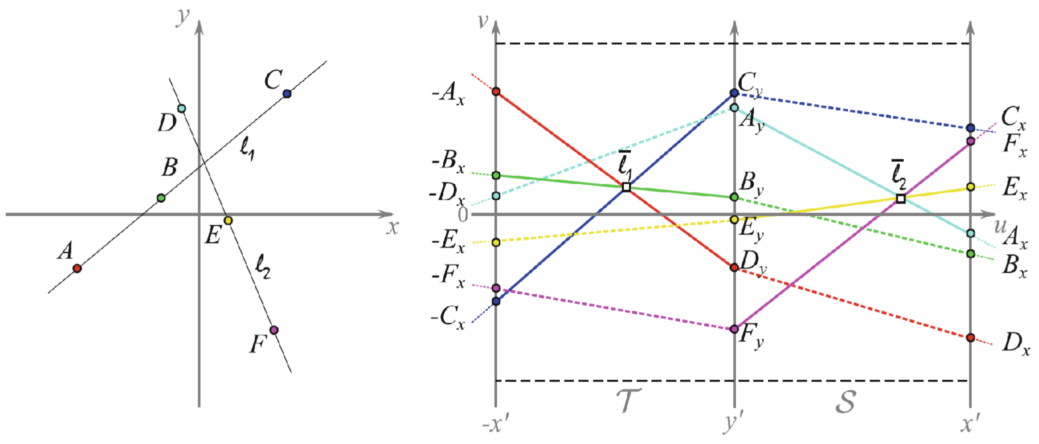


Figure 5.17: Original x-y space (left) and its PClines representation the corresponding $\mathcal{TS}$ space (right).

Chapter 6

Research Achievements

As all the research activities presented in this thesis were performed as a collective work, this chapter aims to conclude the research achievements of me, as an author, and my asset to this thesis. First section provides an overview of work I have done, across all the research areas corresponding with my overall area of focus (CUDA implementation and optimization). Second section concludes and presents products whose development was supported by our research outputs. All those products are however not an outcome of single individual, but are the result of the group of people, all those, that have been participating on the research.

6.1 Author's Achievements and Contribution to Research

This section is divided into three parts in accordance with all researched areas, where each of them points out the areas that were a subject of my responsibility. Any other details to research are available in previous chapters.

6.1.1 Real-Time Object Detection Performance Boost

Real-time object detection and boosting its performance, is a very costly task from the computational resources point of view. As stated in 1.1.1, there was a high demand for efficient object detection methods and implementations. My inputs to this research were two proposed CUDA implementations (see 3.2.1 and 3.3.3) that were promising to be more efficient from various points of view such as portability, maintenance, speed-ups, and time consumption during the development. It was then compared to, except others, shader solution (which was the closest solution to CUDA). This was shown to have complicated drawing of geometric primitives on the "screen" to control the object detection process.

To be able to perform both implementations, the knowledge of weak classifiers was beneficial. Knowledge of weak classifier cascade enabled me to conform distribution of computation capacity between different parts of GPU, what is explained further in this section.

CUDA implementations brought memory arrangement improvements in comparison with Cg shader. Another improvement was that it was working without pre-processing phase, because it relied on the 1×1, 1×2, 2×1 and 2×2 set of mask dimensions. As indicated by the graph in 3.14, such limited set of sampling function dimensions was still sufficient and well comparable with the commonly used Haar features.

In Tab. 3.2, the CUDA code did not perform excellently, but a tremendous increase of performance was observed when the number of weak classifiers is increased (towards 50 in the table). However, if the boosted classier would be a standard AdaBoost [92] or similar, the number of weak classifiers would be constantly high (hundreds). In such case the CUDA implementation outperformed tremendously any other solution available to our knowledge.

The following paragraphs describes in detail particular functional blocks of algorithms that I was focusing on. Initially I have taken into account two facts:

- the classifier was operating on one fixed-size window; and that
- the execution of the classifier on different locations of the input image was parallel.

Loading and Representing the Classifier Data

I was experimenting with placement of the classifier data in shared, constant and texture memory; and tried to balance all access of whole algorithm into units of texture memory and constant memory.

The placement into the shared memory required pre-loading it upon start of each block from another location, what made this solution the least efficient solution. Two other options (texture memory or constant memory) seemed to be performing equally well, so storing the classifier in constant memory was preferred in order to offload the texturing units which were used for accessing the pyramidal image (see 3.2.1).

Although the access would have been slightly simpler if the data was stored in texturing memory of CUDA environment; the experiments showed that the overall detection times are better when the classifier data is stored in the constant memory. This was mainly because the image was stored in texturing memory and was heavily accessed, so off-loading the access to classifier data to the constant memory relieved a system bottleneck.

The constant memory (as well as texturing memory) was cached and the referencing to the classifier data exhibited a large locality of reference – all the threads were typically processing the same weak classifier.

Input Image Pre-Processing

To be able to detect the object in different scales, the image must have been scanned in multiple resolutions. The common approach benefited from the ability of Haar wavelets calculated using the integral image to be evaluated in arbitrary scales in constant time. The LRF features could have been evaluated in a similar manner as well, but experiments showed that especially on the graphics card, it was notably more efficient to construct a multi-resolution pyramid from the input image, and scan it by the detector. See Fig. 3.6 for the illustration of how the pyramid was built. I used constant colour filling to eliminate empty spaces by classifier itself.

Also, I didn't need to pre-process image for various feature sizes, because I choose to rely on the combinations 1×1, 1×2, 2×1 and 2×2 of the sampling function, what allowed nice performance improvements. Thanks to built-in texture sampling with bilinear interpolation (Alg. 3.4) on the usable graphics cards, sums of 2 neighbouring pixels in vertical or horizontal direction or sum of four neighbouring pixels consumed the same amount of time as sampling just one source pixel.

Object Detection

My main goal in this subtask was to divide whole work into small tasks for threads as efficiently as possible. Threads were consuming hardware resources: registers and shared memory what was limiting the number of threads that could have been efficiently executed in a block (both the maximal and minimal number of threads).

One thread could also perform the task of smaller granularity (e.g. one or more weak classifiers), but that would imply too much the inter-thread communication. Image pixels (or window locations, more precisely) were therefore divided into groups, which were calculated by the threads. The final solution divided image into rectangular tiles, which were solved by different thread blocks. Experiments showed that the suitable number of threads per block was around 128 (detailed measurements were done and are presented within 4.1.3).

However, executing blocks for only 128 pixels of the image would not have been efficient, so we chose than one thread will calculate more that one pixel - a whole line of pixels in the rectangular tile (Fig. 3.10). One thread was computing one or more locations of the scanning window in the image. The tile could extend over the whole width of the image, or just a part of it. Because of the thread rearrangement described in 3.2.1, the total number of pixels processed by one thread block was limited proportionally to the size of the shared memory (see 6.1.1).

When object was recognized at window position, the coordinates were written to the global memory. To avoid collisions of concurrently running threads and blocks, atomic increment (`atomicInc()`) of one shared word in the global memory was used for synchronization.

I have also studied the influence of CUDA block width size (see 3.6.1). As shown from measurements in Fig. 3.20, bigger block reduced the computation time, because it lowered the number of blocks necessary, and since the number of blocks is always integer and the blocks must share the same dimensions in CUDA, block widths that were equal or slightly higher than integer fractions of the image width were desired. For a particular application a proper block width must have been found in accordance with these rules.

Thread Rearrangement

In case of branching, the threads were split into groups in accordance to the variant of code they were executing, and the groups of identical execution paths were run separately from other groups. Threads were organized into warps and remained in a warp until their end.

The weak classifier cascade thresholds were set as Wald proposed in the sequential probability ratio test, which he proved was the fastest possible classification strategy for a given target error rate. Due to desired focus-of-attention capability of WaldBoost, some threads terminated with negative decision earlier than others (Fig. 3.9), but the warp continued to evaluate until the very last thread terminated. This led to relatively low utilization of the hardware resources.

To address this issue, I proposed thread rearrangement: at some stage of the classifier, all locations in the image that have not been classified as negative were written into a memory block shared between the threads, and another phase of the classification was started (that processed only these locations). This rearrangement could have been performed several times during the whole classification process. See Fig. 3.11 for an illustration of two rearrangements.

The intermediate positive (more accurately not-yet-negative) samples were stored into the shared memory of the multiprocessor similarly as the final detections were written to the global memory, as described above. The exact count and locations of the rearrangement steps needed to be determined experimentally.

Generally, the major influence of the rearrangements was during the beginning of the classifier, because the most of the locations were dropped out very early (Fig. 3.9) and only a small fraction of computational load remained to the further stages. Determining optimal thread rearrangement stages must have been done experimentally based on knowledge of classifier discrimination characteristic. Basically, scanning window locations needed to be rearranged several times during the classifications to better use the hardware resources. In our environment, no more than three rearrangements were worth doing. My experiments (Fig. 3.21) confirmed that the 1^{st} rearrangement matters the most, because it rearranged a large number of threads. The optimal points for rearrangement were notably different for classifiers trained with different parameters – the shown experiment therefore did not result into fixed rearrangement spots, but rather illustrated the process of optimization for a given classifier.

There are many efficient image processing CUDA implementations that use the shared memory for storing the processed image. The shared memory is very fast and is dozens of kilobytes large – tiles of the processed image can be loaded into it, and processed by thread blocks. I have tried variants of this arrangement and experiments I have performed showed that using the texture memory was more efficient. The texturing units performed bilinear interpolation between neighbouring pixels, which could have been used for evaluation of LRP. Most importantly, when using the texturing memory, the execution was as fast as when using shared memory (apparently because the bottleneck was in the calculation, not memory access), and the shared memory remained spared for other helpful purposes, as was the thread rearrangement described above.

I have also tried several arrangements, where the threads were assigned the work dynamically, so that when the evaluation at one location terminated, the thread "asked for" another location in the image and processed it. The idea was that the work unit would not be one location in the image, but one weak classifier. The control required by this arrangement, and especially the need to synchronize the threads seemed to be too complex and these attempts were much slower than the finally achieved solution with the thread rearrangement (although some threads were still idle).

6.1.2 Spectral Image Analysis Performance Boost

My research achievements in the field of spectral image analysis boosting and optimization were described into details in sections 4.1.2 and 4.2.1. The research performed in those sections was fully covered by me, but also with the great support of my colleagues. As the problem research was too complex, following paragraphs are just a brief description.

Principal Component Analysis

The topic of the problem has been revealed from the start-up project *Optical sensor technology in medical applications* of the University of Eastern Finland.

Using modern computer technology, the PCA can be used on very large data sets where its utilization has previously been unthinkable, and it can also be used in real-time applications.

This research was motivated by the need of using PCA on spectral images in the context of real-time medical imaging.

Generally, in the case of spectral imaging, the dimensionality of the input data was not high (commonly 6–81 channels) but the number of samples (i.e. number of pixels in image or video) was large - millions to billions. Existing solutions (e.g. [39, 38, 2, 67]) did not exactly suit this purpose and this unique situation must have been covered by a particular solution.

My research assumed that the dimensionality of the data was relatively low, so the computation of eigenvectors, addressed by the mentioned works, was relatively cheap. It was the computation of the co-variance matrix, which was costly for the considered data, and my goal was to accelerate the algorithms presented in this part of the research.

In the presented approach I was considering spectral dimensionality from 6 to 81 channels (see 4.1). My goal was to search for the best possible three-component vector space that could represent the spectral information in the image, and then visualize the obtained information in the RGB colour space.

Result of my work was effective computation of the correlation matrix (Eq. 4.7). I had to consider minimal number of CUDA blocks and also the minimal number of CUDA threads for best usage of available GPU resources. The number of CUDA blocks and its usage was not such a problem to overcome, as the number of CUDA blocks should be the same as number of multiprocessors in GPU. Bigger problem was the arrangement of threads when spectral image didn't have so many recorded wavelengths and we needed at least ~ 100 threads to run [64]. To overcome this problem I came with a solution where threads were divided into groups – chunks p (Fig. 6.1) and each group processed another part of s_i (Eq. 4.7). Threads in the same group iterated and accumulated results in one chunk of pixels (Alg. 4.3 Step 6) for pre-computed $[u, v]$ coordinates. These pre-computed coordinates also reflected symetricity of the output matrix.

In the initialization phase of each repetition (Alg. 4.3 Step 4), all threads loaded all chunks of pixels, which they will process, to shared memory. After initialization and synchronization, processing phase began with thread arrangement mentioned above (each thread processed specified coordinates $[u, v]$, threads were divided into groups, and all threads traversed over specified number of pixels C (see 4.1.2)). Another problem was that we could not load enough pixels of s_i into the shared memory and utilize each CUDA thread block as much as possible. To overcome this problem, I made algorithm to repeat with another set of pixels r (Fig. 6.1).

At the end of CUDA block algorithm, we needed to summarize all threads which have the same $[u, v]$ coordinates, but different groups of pixels s_i. To resolve this problem, I used tree summation.

This approach helped us to utilize GPU to maximum and as we measured the results, we found that the biggest issue in this case was the speed of memory.

Details to the whole proposed algorithm, with measurements of performance impact for various CUDA thread and block arrangements, can be found in 4.1.2.

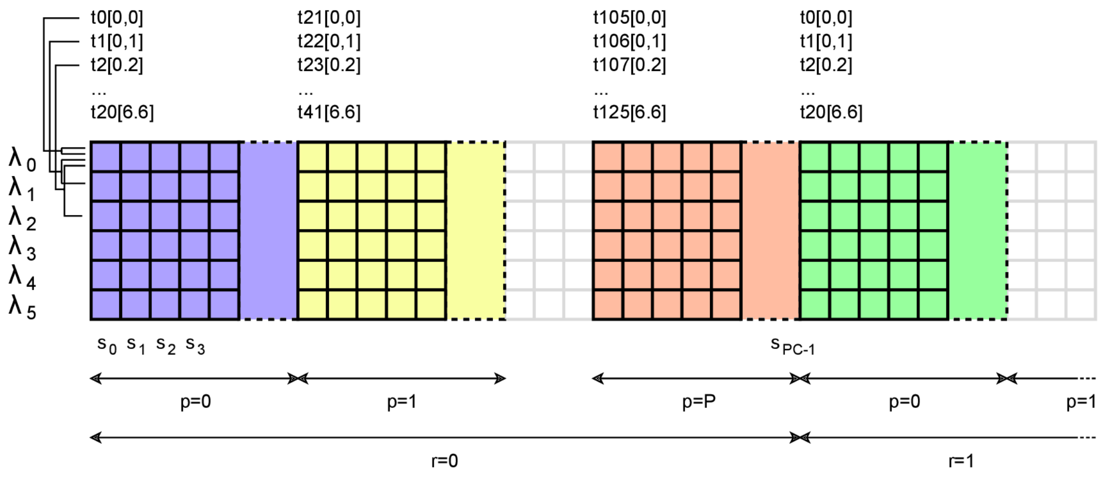


Figure 6.1: Example of one CUDA block thread arrangement for PCA correlation matrix computation.

Non-Negative Tensor Factorization

NTF have various fields of usage, but the dimensionality of these problems is often so high that NTF computations takes hours, so the acceleration of this process was desirable. My NTF research was focused on the efficient GPU implementation for general iterative NTF computation by gradient descent, based on Gauss-Seidel and Jacobi methods [29], using the CUDA programming environment. The aim was to decompose the problem into parts that can be calculated in parallel. Details on algorithm and design can be found in 4.2.

As the baseline for my algorithm (Alg. 4.4) I have used Hazan's et al. [29] iterative rules (Eq. 4.10 – Eq. 4.12). My goal was to divide those rules/equations to smaller tasks, which could be parallelized. The first opportunity for parallelization were temporary matrices $\mathbf{M}_u$, $\mathbf{M}_v$, and $\mathbf{M}_w$ (Eq. 4.13), created by inner product of vectors u, v and w.

The second one was numerator of Eq. 4.10 – Eq. 4.12, which was the most significant time-consuming part of the whole NTF computation. The numerator calculation consisted mostly of repeated summing of the large array, so it was more demanding for memory bandwidth than computationally intensive.

After analysis of the iterative formulas, I came with an effective division of the numerator summing part for threads (Alg. 4.5). Instead of calculating each value of vector **u** and **v** resp. **w** independently and after that traverse all **K** layers in the same manner, which will cause memory bandwidth problems, I calculated whole set of values for each layer in vector in one pass. High demand for memory bandwidth was solved by lowering the number of reads from **G** matrix.

The algorithm depicted on Fig. 6.2 starts with a straightforward solution, where each CUDA block computes one $\mathbf{u}_i$ (resp. $\mathbf{v_i}$, $\mathbf{w_i}$) value from Eq. 4.10, Eq. 4.11 or Eq. 4.12 for whole set of layers K. Than the calculation was divided into independent tiles of **G**, so every tile was covered with $N \times N$ threads (8×8 or 16×16 for better tree summation), which calculated one summation per one vector layer **k**, and stored it in array of accumulators α. This traversed **G** only once, and reduced whole needed bandwidth. In the next step the whole set of threads moved to next tile, and accumulated new sums to α of each thread.

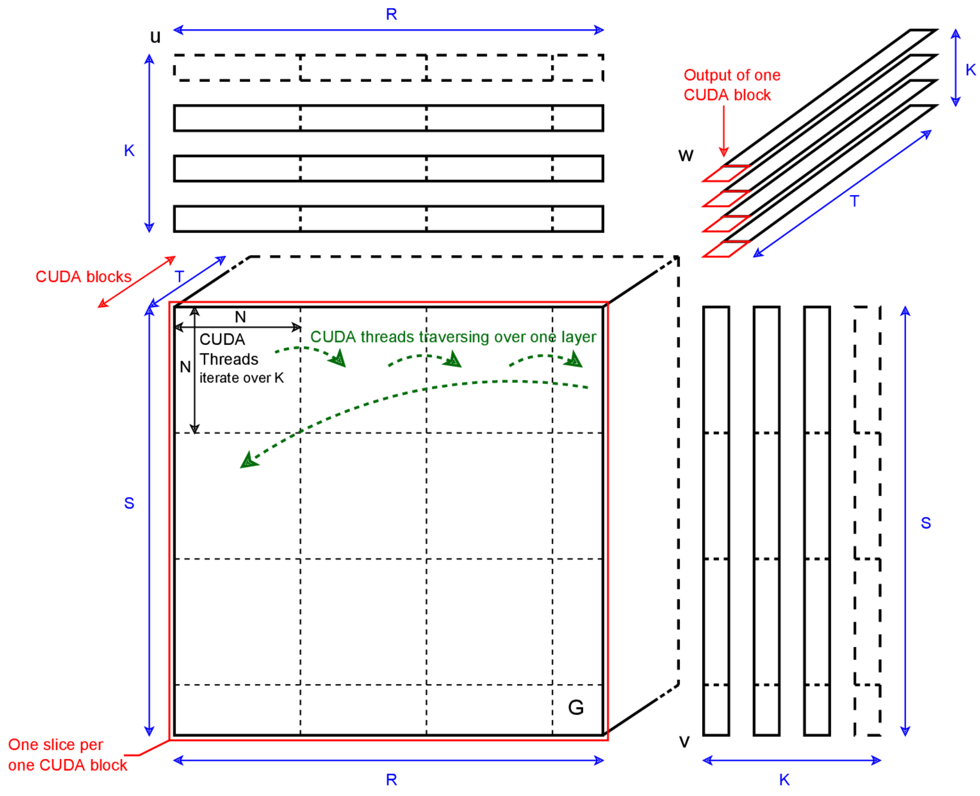
Parts of vector **u** and **v** resp. **w**, corresponding to the working tile, were cached in the shared memory. This gave us a big performance speed-up, because each element of these cached parts was accessed many times. The reason why tiling is performed is that it was not possible to fit whole vectors with all layers into fast shared memory.

After traversing all tiles, tree summations were used for final result and then summed by tree summations [25] to form K values. After all tiles are processed by all CUDA blocks, the whole set of values for output vector is formed.

Details of whole algorithm design can be found in 4.2.1. With this design of algorithm, up to 100× speed-up was achieved.

6.1.3 Real-Time Line Detection Performance Boost

Standard Hough transform was known to be too slow to be usable in real time. My task within this part of the research was again the optimization and implementation of the proposed algorithm discussed within Chapter 5, suitable for computer systems with a small but fast read-write memory, such as today's graphics processors. As we knew that currently available algorithm was working with large amount of data, what was hard (or almost impossible) to be processed in real-time in GPUs, we needed to design an algorithm that would suit these limited but fast resources.



R
u
K
Output of one CUDA block
K
w
T
CUDA blocks
T
N
CUDA Threads iterate over K
N
CUDA threads traversing over one layer
S
S
G
One slice per one CUDA block
v
R
K

Figure 6.2: NTF algorithm overview.

To achieve real-time performance, the memory requirements must have been limited to the *shared memory* of a multiprocessor. Following sections are concluding my main achievements within the area of CUDA boosting.

CUDA version proposed by me was several times faster (Fig. 5.3, 5.5, 5.10, 5.11, 5.12, 5.13) than the commonly used OpenCV implementation (parallelized to utilize the 8 cores of the processor) and achieved real-time or nearly real-time speeds. The real-life image test showed that the proposed algorithm implemented on commodity graphics hardware could detect lines at interactive frame rates.

Small Read-Write Memory of Accumulators

The first part of my idea was storing just a small part of Hough space. My goal was to fit Hough space into small *shared memory* of a multiprocessor. I have observed that just a small part of Hough space would be enough for maxima detection performed in next steps.

The new algorithm (Sec. 5.1.2) stored only $H_\theta \times n$ accumulators (see Fig. 6.3), where n was the neighbourhood size required for the maxima detection. The memory necessary for containing the n lines was all the memory required by the algorithm and even for high resolutions of the Hough space, the buffer of n lines fitted easily in the *shared memory* of the GPU multiprocessors. Whole scheme worked on principle of shifts by one or more rows, where the new row/rows were accumulated. Thus only the buffer of n lines was being reused. The memory shift was implemented using a circular buffer of lines to avoid data copying (Alg. 5.2).

In the case, when the runtime system had faster random-access read-write memory, this memory could be fully used, and instead of accumulating one line of the Hough space, several lines were accumulated and then scanned for maxima. This led to further speed-up by reducing the number of steps carried out by the loop over $\bar{\theta}$.

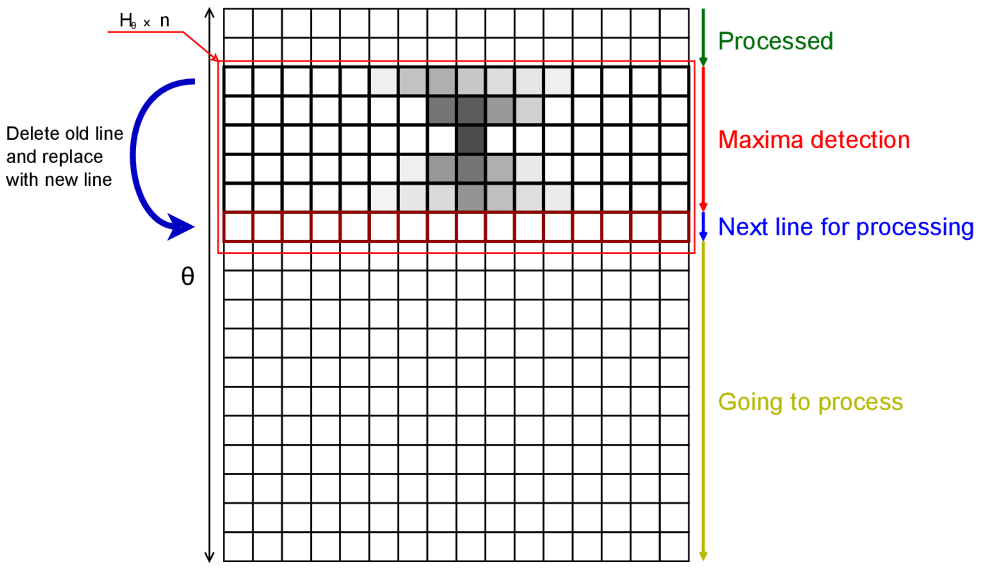


Figure 6.3: Small Read-Write Memory of Accumulators

Harnessing the Edge Orientation

The second part of my idea was special edge orientation harnessing. This special approach, described in 5.1.2, to utilize the detected gradient could have been incorporated to the new accumulation scheme presented above and in the 5.1.2.

Instead of accumulating all points from set P (see 5.1.2), only those points which fell into the interval with radius w around currently processed θ were processed and accumulated into the buffer of n lines.

The edge extraction phase sorted the detected edges by their gradient inclination θ, so that loops did not visit all edges, but only edges potentially accumulated, based on the current $\bar{\theta}$. This basically increased the efficiency of point look-up.

First of all I have detected the edges and their orientation (Tab. 6.1a). Consequently I have had to sort the edges and for each group of them, count the number of edges that fell into that particular group (Tab. 6.1b). Groups were set to be split into specified width. Width of each group was based on our Hough space θ resolution.

For rough sorting of the edges on GPU, an efficient prefix sum was used (Tab. 6.1c) [26]. Based on these prefix sums I have allocated the buffer, and this buffer was then filled with edges in accordance with their orientation (Fig. 6.4). When the buffer was prepared, it was used for filling $H_\theta \times n$ accumulators. Finally, the rest of the algorithm was left in the original manner.

X	Y	θ	G
20	120	15º	200
53	165	126º	151
48	975	78º	54
158	304	26º	186
624	546	105º	76
297	89	5º	42
352	805	8º	94
245	312	19º	115
...	...	...	...

(a) Sobel operator output

θ range	Count
0º-10º	64
10º-20º	81
20º-30º	75
30º-40º	124
40º-50º	106
...	...

(b) Edge Counts for α-ranges

θ range	Count
0º-10º	0
10º-20º	64
20º-30º	145
30º-40º	220
40º-50º	344
...	...

(c) Prefix-sums for α-ranges

Table 6.1: Example of Harnessing the Edge Orientation

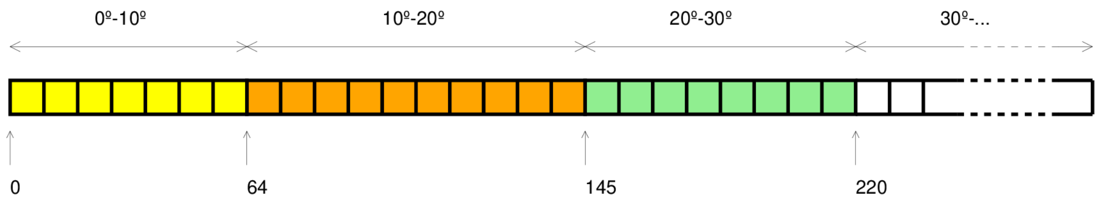


Figure 6.4: Example of Sorted Edge Buffer

6.2 Developed Products

The research described within this thesis, contributed in development of four products that are further described in this section. As already mentioned in the beginning of this chapter, the following products are not an outcome of single individual, but are the result of the group of people, all those, that have been participating on the research.

6.2.1 Object Detection Framework

Object detection framework software, developed in 2009, contains general framework for object detection by classification. It uses classifiers trained by WaldBoost algorithm (see Chapter 3). The framework contains several modules that implements feature extraction from image on various architectures (CPU+SSE, GPU, CUDA). The package could be used in various applications, while the most prominent of them are detection of faces and facial features. The other possible application include detection of persons, licence plates, cars and others. The package itself contains runtime framework, classifiers and videos, it is free of charge and is available on the following link: www.fit.vutbr.cz - products - Object Detection Framework.

6.2.2 CUDA PCA Plug-in

PCA (see Chapter 4) dynamically linked library and MATLAB plug-in, developed in 2009, can be used for any purposes – both research or industrial. It is free of charge, and the library is available on the following link:
www.fit.vutbr.cz - products - CUDA PCA Matlab Library. Researchers using the library are asked to kindly cite this article in works using the library. A detailed guide to using the library is a part of the package, however, this section gives a brief summarization of the library's design and a sketch of its usage.

The dynamically linked library exports several functions `cudaPCA_char`, `cudaPCA_float`, `cudaPCA_double`, `ssePCA_char`, `ssePCA_float`, and `ssePCA_double`. Since CUDA supports only single precision (CUDA does support also double, but it is much slower and generally hardly usable), the `cudaPCA_double` version only uses page-locked memory for faster conversion to float and back.

One function prototype is:

```
cudaPCA_char(unsigned char *pixels,
  unsigned width, unsigned height, unsigned n,
  float *eigen_vec, float *eigen_val);
```

and prototypes of the other functions are analogous. The image data is stored in `pixels`, which contains `width*height*n` bytes of image data, row alignment padding is not supported for simplicity, though modification of the algorithm to support it is possible.

Identifiers `eigen_vec` and `eigen_val` are output variables, with preallocated buffers for the eigen vectors and values.

The function in the MATLAB plug-in takes arguments corresponding to the above mentioned function and returns the eigen vectors and values. The plug-in internally calls the `cudaPCA_char` or `cudaPCA_double` functions.

6.2.3 CUDA NTF Plug-in

NTF (see Chapter 4) dynamically linked library and MATLAB plug-in, developed in 2009, can be used for any purpose – both research or industrial. It is free of charge and is available on the following link: www.fit.vutbr.cz - products - CUDA Nonnegative Tensor Factorization Library. Researchers using the library are asked to kindly cite this article in works using the library or its source code. A detailed guide to using the library is part of the package.

The library interface is designed to be very simple. The dynamically linked library exports only two functions `cudantfFloat` and `cudantfDouble`.

Since CUDA supports only single precision, the variant processing double inputs computes in single precision and only uses page-locked memory for faster conversion to float and back. The prototype of the `cudantfFloat` function is:

```
cudantfResult cudantfFloat(ds
  const float * G, // float[R*S*T]
  unsigned R, unsigned S, unsigned T,
  unsigned rank, unsigned iterations,
  cudantfInit init,
  float * U, // float[R*rank]
  float * V, // float[S*rank]
  float * W  // float[T*rank]
);
```

The double precision function `cudantfDouble` is similar. Identifiers `G`, `U`, `V`, and `W` correspond to the same names in Eq. 4.10 – Eq. 4.12; `R`, `S` and `T` are the input tensor dimensions. The initial values of `U`, `V`, and `W` can be randomly generated or supplied in the output arrays. This behaviour is specified by the `init` parameter.

The function in the Matlab plug-in takes three parameters: the input tensor, the method rank, and the iteration count. It returns the `U`, `V`, and `W` vectors. The plug-in internally calls the `cudantfDouble` function with randomly generated initial values.

6.2.4 GStreamer AdaBoost Plug-in

This implementation was developed in 2010 and enables the integration of the AdaBoost algorithm to any program as a component of the GStreamer framework. The application is executable both on Linux OS and the embedded devices with operating system Maemo Linux. The plug-in is available on the following link: www.fit.vutbr.cz - products - GStreamer Adaboost plugin.

Chapter 7

Conclusions

Research performed on CUDA architecture gave us lot of chances for algorithm improvements. Evaluations done within research assignments presented in this thesis showed us the real performance benefits. Graphic cards capable of GPGPU operations (using CUDA framework) are nowadays common equipment available in research laboratories, so the solutions proposed in this thesis does not require any special supercomputer-like investments.

Gained speed-up was not as high as could have been expected from the rough computational power of the GPU in comparison with CPU, but this was mainly due to nature of the algorithms, which did not match the requirements of CUDA and GPU environment in general.

As demonstrated by the measurements carried out within the research, a computer equipped with one or more graphics boards with powerful GPUs, can process a multiple video signals in high resolution in real-time. Using the GPU technology would therefore find its application in surveillance and other real-world industrial tasks. After all, the next section provides an overview of impact of our outcome, and therefore lists several possible applications of proposed solutions.

Eight articles in total - evaluating performance of LRD, LRP, PCA, NTF, Hough transform and parallel coordinates algorithms - have been produced during the research, together with four products in form of dynamically linked library and MATLAB plug-ins. Those have been developed by the group of my colleagues participating on this research.

The experimental implementation of the Local Rank Functions (namely LRD) image feature using CUDA GPGPU environment , its comparison to other approaches such as CPU implementation and Haar-like features on the GPU leaded to the conclusion that the LRD is a vital low-level image feature set, which outperforms the commonly used Haar wavelets (especially in case of higher resolutions) in several important measures, and that fast implementations of object detectors and other image classifiers, should consider the LRD as an important alternative. Hardware-accelerated implementations speeded-up the baseline LRD implementations more than by order of magnitude. Measurements have also shown that the performance on the GPUs was equal for CUDA and GLSL programming.

Optimized algorithm of PCA computation , primarily targeted on spectral image analysis in real-time, achieved speed-ups that allow processing of high-resolution images with several colour channels (both common RGB and spectral images) in real-time. PCA algorithms presented in this thesis allows also many other useful applications where fast computation is needed, and can help to solve some problems where real-time image segmentation and pattern recognition tasks are used.

Research of optimized implementation of an efficient NTF algorithm for GPGPU computation achieved around 60× - 100× speed-up compared to a C implementation compiled by an optimizing compiler running on a state-of-the-art computer. This results were considered to be outstanding, when taking into account that Zhang et al. [75] reported their speed-up by adding further nodes was capped at about 7×. This speed-up value is attractive in this field, since computation of NTF for typical problems in spectral image analysis takes hours.

Other positive results were achieved in study of modified algorithm for line detection using the Hough transform based on $\theta - \varrho$ parametrization. This algorithm was designed to be intensively using a small read-write memory; what made it suitable for execution on recent graphics processors. The experiments showed that on commodity graphics hardware, the algorithm can operate at interactive frame rates even on high-resolution real-life images, while accumulating to a high-resolution Hough space to achieve accurate line detections. While the algorithm was designed for GPU processing, it outperformed the standard HT implementation even on the CPU, thanks to better cache usage of the new accumulation scheme.

Finally, the last, but not least significant improvement was achieved in study of an algorithm based on the PClines parametrization, which allowed real-time computation of the "full" Hough transform on high-resolution images. Measurement showed that the GPU-accelerated algorithm achieved interactive (or faster) detection times even for images of really high resolutions. Other proposed usage of the algorithm were low-power and embedded devices, as well as designing specialized circuity such as FPGA, as it requires no floating-point calculation or goniometric functions.

Considering the fact that CUDA is much more intuitive and compatible to standard C language programming, CUDA was a good selection for exploiting graphics hardware for non-rendering tasks, such as object detection, spectral image analysis or line detection.

7.1 Research Impact

During our research, eight papers (publications) have been published in different journals, or as a book chapter. Following section provides the citation analysis, in order to assess our research impact, and gauge the extent of publication's influence on the academic field.

7.1.1 Citation Analysis

Citation analysis is distinguished as internal, self-citing or reciprocal citations by our colleagues – where at least one author is already an author of the publication itself; and external citations refers to authors also from our university, but not in relation with our research group, or international authors from different academic fields all over the world. Detailed list of both internal and external citations is not a part of this thesis, but is available on request, and is also one of the supplements of the thesis. Within this section, mostly external citations are analysed.

Table 7.1 counts the number of times each article has been cited in general, both internally and externally. Data were collected via Google Scholar tool. All citations have been analysed and divided into internal and external citations. Some of them were published twice (duplicates). Those external have been analysed further, and if available, the purpose of citation was defined. The purposes were defined as follows:

- Attribution of ideas/research (A) if the citation is in the manner of confirming or illustrating a point; in the manner of disputing, correcting or questioning; or in the manner of the use of methods, tools, design, definition or data which are one of the outputs of our research;
- Providing proof that position is well-researched (P) by providing holistic view of research, literature review, or using our publication as a primary source;
- Helping to disseminate useful knowledge (H) in the manner of demonstrating other points of view, or referring to our publication as a source of supplementary information;
- Giving a formal credit for research (G) as a normative research practice. This purpose of citation is not present in our analysis.

7.1.2 Research Impact Conclusion

Our outcomes of real-time object detection using CUDA were primarily used as a reference providing supplementary information to continuing researches on:

Publication	Number of citations	Internal	External	Duplicates
Local Rank Differences Image Feature Implemented on GPU	10	6	3	1
GP-GPU Implementation of the “Local Rank Differences” Image Feature	9	4	4	1
Low-Level Image Features for Real-Time Object Detection	11	8	2	1
Real-Time Line Detection Using Accelerated High-Resolution Hough Transform	9	5	4	0
Real-time object detection on CUDA	18	1	16	1
Non-Negative Tensor Factorization Accelerated Using GPGPU	8	0	8	0
Real-Time PCA Calculation for Spectral Imaging (using SIMD and GP-GPU)	8	0	6	2
Real-Time Detection of Lines Using Parallel Coordinates and CUDA	2	1	1	0

Table 7.1: List of author publications.

- Weak classifier applications, increasing accuracy of AdaBoost classifiers;
- Development of real-time image processing system for anomaly detection;
- Medical imaging applications such as foreground/background classification, 3D pose detection, and boundary delineation;
- Improvement of performance, and reduction of power consumption in many image processing applications – using different approach than we have been using during our research;
- Corner point detection;
- Highly optimized Haar-based face detector that works in real-time over high definition videos;
- Studying different parallelization strategies of image-filtering algorithms;
- Video photo mosaics;

- Implementation of Haar-Classifier for face detection and tracking based on the Haar-Features on System on Chip (SoC) to be used in a human machine interface and action interpretation;
- Recognition of the scene presented in an image with specific application to scene classification in field sports video;

LRF algorithm novel [32] also impacted the development of mobile application allowing user to discover the information about a given building or landmark. The outcome of Real-time object detection on CUDA [30] supported the consequent research on parallel algorithm of face detection on images for GPU architecture, using different approach that the one used during our research.

Our research on spectral image analysis using CUDA was used as a source of information during consequent researches and some of them are:

- Military applications - target detection surveillance using hyperspectral remote sensing, demanding real-time or near real-time processing;
- Signal processing - calculating the overall covariance matrix by accumulating a group of partial covariance matrices;
- Steelworks - parallel dynamic solidification model development;
- Analysis of metabolomics and transcriptomics data;
- Large scale data processing using MapReduce;
- Sclera Vein Recognition;
- Non-negative multiple tensor factorization;

And our last subject of research – real-time line detection using CUDA produced outcomes that were used as source of information during following different implementations of Hough transform both on FPGA and GPU, and subsequent research on line detection both internally and externally.

References

[1] Tinku Acharya and Ajoy K Ray. *Image processing: principles and applications*. John Wiley & Sons, 2005. 3.1.3

[2] M Andrecut. Parallel GPU implementation of iterative PCA algorithms. *Journal of Computational Biology*, 16(11):1593–1599, 2009. 1.1.2, 4.1, 6.1.2

[3] Alexey Andriyashin, Jussi Parkkinen, and Timo Jaaskelainen. Illuminant dependence of PCA, nmf and ntf in spectral color imaging. In *19th International Conference on Pattern Recognition (ICPR 2008), December 8-11, 2008, Tampa, Florida, USA*, pages 1–4, 2008. 1.1.2, 4.2

[4] Jukka Antikainen, Jiri Havel, Radovan Josth, Adam Herout, Pavel Zemcik, and Markku Hauta-Kasari. Nonnegative tensor factorization accelerated using GPGPU. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1135–1141, 2011. (document), 4

[5] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. pages 714–725, 1987. 1.1.3, 5.1

[6] Michael W. Berry, Murray Browne, Amy N. Langville, Paul V. Pauca, and Robert J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, September 2007. 1.1.2, 4.2

[7] S Charles Brubaker, Matthew D Mullin, and James M Rehg. Towards optimal training of cascaded detectors. In *Computer Vision–ECCV 2006*, pages 325–337. Springer, 2006. 3.1

[8] Jiann-Yeou Chen and Liang-Chien Rau. Fast straight lines detection using hough transform with principal axis analysis. *Journal of Photogrammetry and Remote Sensing*, 8(1):5–34, 2003. 1.1.3, 5.1

[9] HD Cheng, Yanhui Guo, and Yingtao Zhang. A novel hough transform based on eliminating particle swarm optimization and its applications. *Pattern Recognition*, 42(9):1959–1969, 2009. 5.1

[10] Xuemei Cheng, Yang Tao, Yud-Ren Chen, and Xin Chen. Integrated PCA-fld method for hyperspectral imagery feature extraction and band selection. In *Biomedical Imaging: Nano to Macro, 2006. 3^{rd} IEEE International Symposium on*, pages 1384–1387, April 2006. 4.1

[11] Andrzej Cichocki and Shun-Ichi Amari. *Adaptive Blind Signal and Image Processing.* Wiley, 1 edition, June 2002. 1.1.2, 4.2

[12] Andrzej Cichocki, Rafal Zdunek, Seungjin Choi, Robert Plemmons, and Shun ichi Amari. *Novel Multi-layer Non-negative Tensor Factorization with Sparsity Constraints*, volume 4432, pages 271–280. Springer Berlin / Heidelberg, 2007. 4.2

[13] Maurice d'Ocagne. *Le calcul simplifié par les procédés mécaniques et graphiques: Histoire et description sommaire des instruments et machines á calculer, tables, abaques et nomogrammes.* Gauthier-Villiars, 1905. 5.2

[14] Markéta Dubská, Jiří Havel, and Adam Herout. Real-time detection of lines using parallel coordinates and opengl. In *Proceedings of the 27th Spring Conference on Computer Graphics*, pages 149–155. ACM, 2011. 5.2.2, 5.2.3

[15] Markéta Dubská, Adam Herout, and Jirí Havel. PClines—line detection using parallel coordinates. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1489–1494. IEEE, 2011. 5.2, 5.2.3, 5.2.3

[16] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972. 5.1

[17] Ulrich Eckhardt and Gerd Maderlechner. Application of the projected hough transform in picture processing. In *Pattern Recognition*, pages 370–379. Springer, 1988. 5.1

[18] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995. 3.1

[19] Michael Garland. Designing a unified programming model for heterogeneous machines. 2.3.2

[20] Peter N Glaskowsky. Nvidia's fermi: the first complete gpu computing architecture. *White paper*, 2009. 2.3.2, 5.2.3

[21] H. H. Goldstine, F. J. Murray, and J. von Neumann. The jacobi method for real symmetric matrices. *Journal of the ACM*, 6(1):59–96, 1959. 4.1.2

[22] L. Grippo and M. Sciandrone. On the convergence of the block nonlinear Gauss-Seidel method under convex constraints. *Operations Research Letters*, 26(3):127–136, 2000. 4.2

[23] Olli Haavisto. *Reflectance spectrum analysis of mineral flotation froths and slurries.* PhD thesis, Helsinki University of Technology, Espoo, Finland, 2009. 4.1

[24] Mark Harris. Cuda debugging and profiling tools, 2010. 2.2.3

[25] Mark Harris et al. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, 2:45, 2007. 4.1.2, 4.2.1, 6.1.2

[26] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007. 5.1.2, 6.1.3

[27] Markku Hauta-Kasari, Juha Lehtonen, Jussi PS Parkkinen, and Timo Jaeaeskelaeinen. Spectral image compression for data communications. In *Photonics West 2001-Electronic Imaging*, pages 42–49. International Society for Optics and Photonics, 2000. 4.1

[28] Jiří Havel, Markéta Dubská, Adam Herout, and Radovan Jošth. Real-time detection of lines using parallel coordinates and CUDA. *Journal of Real-Time Image Processing*, pages 1–12, 2012. (document), 5.2.1

[29] T. Hazan, S. Polak, and A. Shashua. Sparse image coding using a 3D non-negative tensor factorization. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 50–57Vol. 1, October 2005. 1.1.2, 4.2, 4.2, 4.2.2, 6.1.2

[30] Adam Herout, Radovan Jošth, Roman Juránek, Jiří Havel, Michal Hradiš, and Pavel Zemčík. Real-time object detection on CUDA. *Journal of Real-Time Image Processing*, 6(3):159–170, 2011. (document), 3.2, 7.1.2

[31] Adam Herout, Radovan Josth, Pavel Zemcík, and Michal Hradis. Gp-GPU implementation of the ?local rank differences? image feature. In *Computer Vision and Graphics*, pages 380–390. Springer, 2009. (document), 3.3, 3.3.3

[32] Adam Herout, Pavel Zemcík, Michal Hradiš, Roman Juránek, Jiří Havel, Radovan Jošth, and Martin Žádník. Low-level image features for real-time object detection. *IN-TECH Education and Publishing*, page 25, 2009. (document), 3.2, 3.3.3, 3.3.3, 7.1.2

[33] Adam Herout, Pavel Zemcik, Roman Juránek, and Michal Hradis. Implementation of the” local rank differences” image feature using simd instructions of cpu. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP’08. Sixth Indian Conference on*, pages 450–457. IEEE, 2008. 3.2.1

[34] Paul V. C. Hough. Method and means for recognizing complex patterns, December 1962. U.S. Patent 3,069,654. 1.1.3, 5.1

[35] Michal Hradiš, Adam Herout, and Pavel Zemčík. Local rank patterns – novel features for rapid object detection. Technical report, 2008. 3.1.3, 3.2, 3.2, 3.3, 3.3.2

[36] John Illingworth and Josef Kittler. A survey of the hough transform. *Computer vision, graphics, and image processing*, 44(1):87–116, 1988. 5.1

[37] Alfred Inselberg. *Parallel coordinates.* Springer, 2009. 5.2

[38] Edward J. Jackson. *A User’s Guide to Principal Components (Wiley Series in Probability and Statistics).* Wiley-Interscience, September 2003. 1.1.2, 4.1, 6.1.2

[39] I. T. Jolliffe. *Principal Component Analysis.* Springer, 2nd edition, October 2002. 1.1.2, 4.1, 6.1.2

[40] Stephen Jones. Introduction to dynamic parallelism, 2012. 2.3.2

[41] Radovan Jošth, Jukka Antikainen, Jiří Havel, Adam Herout, Pavel Zemčík, and Markku Hauta-Kasari. Real-time PCA calculation for spectral imaging (using SIMD and gp-GPU). *Journal of real-time image processing*, 7(2):95–103, 2012. (document), 4

[42] Radovan Jošth, Markéta Dubská, Adam Herout, and Jiří Havel. Real-time line detection using accelerated high-resolution hough transform. In *Image Analysis*, pages 784–793. Springer, 2011. (document), 5.2.1

[43] A. Kaarna, A. Andriyashin, S. Nakauchi, and J Parkkinen. *Multiresolution Approach in Computing NTF*, volume 4522, pages 334–343. Springer Berlin / Heidelberg, 2007. 1.1.2, 4.2

[44] Timor Kadir and Michael Brady. Saliency, scale and image description. *International Journal of Computer Vision*, 45(2):83–105, 2001. 3.3

[45] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011. 2.3.2

[46] Ohsawa Kenro, Ajito Takeyuki, Komiya Yasuhiro, Haneishi Hideaki, Yamaguchi Masahiro, and Nagaaki Ohyama. Six-band HDTV camera system for spectrum-based color reproduction. *The Journal of Imaging Science and Technology*, 48(2):85–92, March 2004. 4.1

[47] Khronos OpenCL Working Group. The OpenCL specification, October 2009. 2.2.2

[48] Minje Kim and Seungjin Choi. Monaural music source separation: Nonnegativity, sparseness, and shift-invariance. In *ICA*, pages 617–624, 2006. 1.1.2, 4.2

[49] Hannu Laamanen, Tuija Jetsu, Timo Jaaskelainen, and Jussi Parkkinen. Weighted compression of spectral color information. *JOSA A*, 25(6):1383–1388, 2008. 4.1

[50] Juha Lehtonen, Jussi Parkkinen, and Timo Jaaskelainen. Optimal sampling of color spectra. *JOSA A*, 23(12):2983–2988, 2006. 4.1

[51] Hungwen Li, Mark A Lavin, and Ronald J Le Master. Fast hough transform: A hierarchical approach. *Computer Vision, Graphics, and Image Processing*, 36(2):139–161, 1986. 1.1.3, 5.1

[52] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–900. IEEE, 2002. 3.1.1, 3.3

[53] Paul Lilly. From voodoo to geforce: The awesome history of 3d graphics, May 2009. 2.3.1, 2.3.1, 2.3.2, 2.3.2

[54] Wei Lu and Jinglu Tan. Detection of incomplete ellipse in images with strong noise by iterative randomized hough transform (IRht). *Pattern Recognition*, 41(4):1268–1279, 2008. 5.1

[55] Philipp Michel, Joel Chestnutt, Satoshi Kagami, Koichi Nishiwaki, James Kuffner, and Takeo Kanade. GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 463–469. IEEE, 2007. 3.3

[56] Yoshihiko Mochizuki, Akihiko Torii, and Atsushi Imiya. N-point hough transform for line detection. *Journal of Visual Communication and Image Representation*, 20(4):242–253, 2009. 5.1

[57] Morten Mørup, Lars K. Hansen, Christoph S. Herrmann, Josef Parnas, and Sidse M. Arnfred. Parallel Factor Analysis as an exploratory tool for wavelet transformed event-related EEG. *NeuroImage*, 29(3):938–947, February 2006. 1.1.2, 4.2

[58] Frank Natterer. *The mathematics of computerized tomography.* Number iSBN 9780471909590. Springer, 1986. 5.1

[59] Ken Nishino, Mutsuko Nakamura, Masayuki Matsumoto, Osamu Tanno, and Shigeki Nakauchi. Imaging of cosmetics foundation distribution by a spectra difference enhancement filter. In *Proceedings of the 5th European Conference on Colour in Graphics, Imaging, and Vision, CGIV*, pages 275–281, June 2010. 4.2

[60] NVIDIA Corporation. *CUDA C Programming Guide 2.0*, 2008. 2.3.2, 2.3.2, 3.3

[61] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi. *Whitepaper*, 2009. 2.3.2

[62] NVIDIA Corporation. Dynamic parallelism in cuda, 2012. 2.3.2

[63] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Kepler gk110. *Whitepaper*, 2012. 2.3.2

[64] NVIDIA Corporation. *CUDA C Programming Guide 6.0*, Feb 2014. 2.5, 2.2.1, 2.6, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.3.2, 6.1.2

[65] NVIDIA Corporation. NVIDIA GeForce GTX750 Ti. *Whitepaper*, 2014. 2.3.2

[66] F. O'Gorman and M.B. Clowes. Finding picture edges through collinearity of feature points. *IEEE*, C-25(4):449–456, 1976. 5.1.2

[67] Julius Ohmer, Frederic Maire, and Ross Brown. Implementation of kernel methods on the GPU. In *Digital Image Computing: Techniques and Applications, 2005. DICTA'05. Proceedings 2005*, pages 78–78. IEEE, 2005. 1.1.2, 4.1, 6.1.2

[68] Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. Gray scale and rotation invariant texture classification with local binary patterns. In *Computer Vision-ECCV 2000*, pages 404–420. Springer, 2000. 3.1.2, 3.3

[69] General-Purpose Computation on GPUs. available as of 04-2008 at http://www.gpgpu.org. 3.3

[70] OpenGL ARB Group. Opengl api transferd to khronos group, July 2006. Accessed: 10 Aug 2014. 2.1

[71] Joseph O'Rourke. Dynamically quantized spaces for focusing the hough transform. In *IJCAI*, pages 737–739. Citeseer, 1981. 1.1.3, 5.1

[72] Constantine P Papageorgiou, Michael Oren, and Tomaso Poggio. A general framework for object detection. In *Computer vision, 1998. sixth international conference on*, pages 555–562. IEEE, 1998. 3.1.1

[73] Lukáš Polok, Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, and Radovan Jošth. ?local rank differences? image feature implemented on GPU. In *Advanced Concepts for Intelligent Vision Systems*, pages 170–181. Springer, 2008. (document), 2.3.2, 3.3, 3.3.2, 3.3.3, 3.3.3, 1, 3.5.1

[74] J. Princen, J. Illingowrth, and J. Kittler. Hypothesis testing: A framework for analyzing and optimizing Hough transform performance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(4):329–341, 1994. 5.1

[75] Brian T. Lamb Qiang Zhang, Michael W. Berry and Tabitha Samuel. *A Parallel Nonnegative Tensor Factorization Algorithm for Mining Global Climate Data*, volume 5545, pages 405–415. Springer Berlin / Heidelberg, 2009. 1.1.2, 4.2, 4.2.2, 4.2.2, 7

[76] James B Reeves, Gregory W McCarty, David W Rutherford, and Robert L Wershaw. Mid-infrared diffuse reflectance spectroscopic examination of charred pine wood, bark, cellulose, and lignin: Implications for the quantitative determination of charcoal in soils. *Applied spectroscopy*, 62(2):182–189, 2008. 4.1

[77] Ashu Rege. An introduction to modern gpu architecture, 2008. 2.1.1, 2.2, 2.3.1, 2.3.2, 2.3.2, 2.11

[78] Robert E Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Machine learning*, 37(3):297–336, 1999. 3.1, 3.1, 3.5

[79] Henry Schneiderman and Takeo Kanade. Object detection using the statistics of parts. *International Journal of Computer Vision*, 56(3):151–177, 2004. 3.1

[80] Linda Shapiro and George C Stockman. Computer vision. *Prentice Hall*, 2001. Tom Robbins. 5.1.2

[81] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. GPU-based video feature tracking and matching. In *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, volume 278, page 4321, 2006. 3.3

[82] Kenneth R Sloan Jr. Dynamically quantized pyramids. In *International Joint Conference on Artificial Intelligence*, pages 734–736. Kaufmann, 1981. 1.1.3, 5.1

[83] Tim Smalley. Nvidia ceo reveals fermi architecture. available as of 20.1.2010 at http://www.bit-tech.net/news/hardware/2009/09/30/huang-reveals-fermi-architecture/1. 2.3.2

[84] Age Smilde, Rasmus Bro, and Paul Geladi. *Multi-way Analysis: Applications in the Chemical Sciences*. John Wiley and Sons, New York, 2004. 1.1.2, 4.2

[85] Jan Sochman and Jiri Matas. Inter-stage feature propagation in cascade building with adaboost. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 1, pages 236–239. IEEE, 2004. 3.1

[86] Jan Sochman and Jiri Matas. Waldboost-learning for time constrained sequential detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 150–156. IEEE, 2005. 1.1.1, 3.1, 3.1.3, 3.4

[87] Jan Sochman and Jiri Matas. Learning a fast emulator of a binary decision process. 2007. 3.1, 3.3

[88] Alan Tatourian. Nvidia gpu architecture and cuda programming environment, Sep 2013. 2.10, 2.3.2

[89] Paul Thurrott. Microsoft, sgi turn up the heat on opengl, Dec 1997. Accessed: 10 Aug 2014. 2.1

[90] Jeremiah van Oosten. Introduction to opengl and glsl, Feb 2014. Accessed: 10 Aug 2014. 2.1, 2.1, 2.1.1

[91] Jan Verschelde. Introduction to supercomputing - evolution of graphics pipelines, Mar 2014. Accessed: 10 Aug 2014. 2.1

[92] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001. 1.1.1, 3.1, 3.1, 3.1.1, 3.3, 3.3.2, 3.5.1, 6.1.1

[93] Matas J. Šochman J. Waldboost - learning for time constrained sequential detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, 2005. 3.3, 3.3.3

[94] Abraham Wald et al. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945. 3.1, 3.5

[95] R. Wallace. A modified hough transform for lines. In *CVPR*, pages 665–667. Carnegie-Mellon University. Robotics Institute and Wallace, RS, 1985. 5.1

[96] Scott Wasson. Nvidia's 'fermi' GPU architecture revealed. available as of 20.1.2010 at http://techreport.com/articles.x/17670. 2.3.2

[97] Rong Xiao, Long Zhu, and Hong-Jiang Zhang. Boosting chain learning for object detection. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 709–715. IEEE, 2003. 3.1

[98] Lei Xu, Erkki Oja, and Pekka Kultanen. A new curve detection method: randomized hough transform (rht). *Pattern recognition letters*, 11(5):331–338, 1990. 1.1.3, 5.1

[99] T. Yotsuda, T. Yamamoto, H. Ishizawa, T. Nishimatsu, and E. Toba. Near infrared spectral imaging for water absorbency of woven fabrics. In *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, volume 3, pages 2258–2261Vol.3, May 2004. 4.1

[100] Lun Zhang, Rufeng Chu, Shiming Xiang, Shengcai Liao, and Stan Z Li. Face detection based on multi-block lbp representation. In *Advances in biometrics*, pages 11–18. Springer, 2007. 3.1.2

Nomenclature

AGP	Accelerated Graphics Port
AMD	Advanced Micro Devices
API	Application programming interface
ARB	Arechitecture Review Board
ASIC	Application Specific Integrated Circuit
BGS	Block Gauss-Seidel
CPI	Clock Per Instruction
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DQP	Dynamically Quantized Pyramid
DQS	Dynamically Quantized Spaces
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FBO	Frame-Buffer Objects
FHT	Fast Hough Transform
FPGA	Field-Programmable Gate Array
GLSL	OpenGL Shading Language
GPC	Graphics Processing Clusters
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HD	High-Definition
HLSL	High Level Shader Language

HT	Hough Transform
LBP	Local Binary Patterns
LD/ST	Load/Store
LRD	Local Rank Differences
LRF	Local Rank Functions
LRP	Local Rank Patterns
MMX	Multimedia Extension
MTIU	Multi Threaded Instruction Unit
MVP	Model View Projection
NTF	Nonnegative Tensor Factorization
OpenCV	Open Source Computer Vision
OpenMP	Open Multi-Processing
PAL	Phase Alternating Line
PC	Parallel Coordinates
PCA	Principal Component Analysis
PHIGS	Programmer's Hierarchical Interactive Graphics System
RGB	Red Green Blue
RGBA	Red Green Blue Alpha
RHT	Randomized Hough Transform
RMSE	Root Mean Square Error
ROC	Receiver Operating Characteristi
SFU	Special Function Unit
SGI	Silicon Graphics, Inc.
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SLI	Scalable Link Interface
SM	Streaming Multi-Processor

SMM	SMX naming for Maxwell
SMX	Next Generation Streaming Multiprocessor
SOC	System On Chip
SP	Stream Processor
SSE	Streaming SIMD Extensions
SVD	Singular Value Decomposition
TandL	Transformation and Lightening
TMU	Texture Map Unit
TSV	Through Silicon Vias
ViRGE	Virtual Reality Graphics Engine
WC	Weak Classifiers

Appendix A

Publications

1. Jošth Radovan. *Real time atmosphere rendering for the space simulators.* Proceedings of the 9th Central European seminar on computer graphics, CESCG, 2005.

2. Polok Lukáš, Herout Adam, Zemčík Pavel, Hradiš Michal, Juránek Roman, Jošth Radovan. *"Local Rank Differences" Image Feature Implemented on GPU.* In Advanced Concepts for Intelligent Vision Systems, pages 170–181. Springer, 2008.

3. Herout Adam, Jošth Radovan, Zemčík Pavel, Hradiš Michal. *GP-GPU implementation of the "Local Rank Differences" image feature.* In Computer Vision and Graphics, pages 380–390. Springer, 2009.

4. Herout Adam, Zemcík Pavel, Hradiš Michal, Juránek Roman, Havel Jiří, Jošth Radovan, Žádník Martin. *Low-level image features for realtime object detection.* IN-TECH Education and Publishing, page 25, 2009.

5. Jošth Radovan, Dubská Markéta, Herout Adam, Havel Jiří. *Realtime line detection using accelerated high-resolution hough transform.* In Image Analysis, pages 784–793. Springer, 2011.

6. Antikainen Jukka, Havel Jiri, Josth Radovan, Herout Adam, Zemcik Pavel, Hauta-Kasari Markku. *Nonnegative tensor factorization accelerated using GPGPU.* Parallel and Distributed Systems, IEEE Transactions on, 22(7):1135–1141, 2011.

7. Herout Adam, Jošth Radovan, Juránek Roman, Havel Jiří, Hradiš Michal, Zemčík Pavel. *Real-time object detection on CUDA.* Journal of Real-Time Image Processing, 6(3):159–170, 2011.

8. Jošth Radovan, Antikainen Jukka, Havel Jiří, Herout Adam, Zemčík Pavel, Hauta-Kasari Markku. *Real-time PCA calculation for spectral imaging (using SIMD and gp-GPU).* Journal of real-time image processing, 7(2):95–103, 2012.

9. Havel Jiří , Dubská Markéta, Herout Adam, Jošth Radovan. *Realtime detection of lines using parallel coordinates and CUDA.* Journal of Real-Time Image Processing, 2014, vol. 2014, no9, pages 205–216, ISSN 1861-8200.

Appendix B

Products

Products, that have been developed during our research are more in detail in 6.2. Section below provides the list of them.

Product list:

1. Antikainen Jukka, Havel Jiří, Herout Adam, Jošth Radovan: *CUDA Nonnegative Tensor Factorization Library*, software, 2009
2. Antikainen Jukka, Havel Jiři, Herout Adaḿ, Jošth Radovanl: *CUDA PCA Matlab Library*, software, 2009
3. Beran Vítězslav, Havel Jiří, Herout Adam, Hradiš Michal, Jošth Radovan, Juránek Roman, Polok Lukáš, Zemčík Pavel: *Object Detection Framework*, software, 2009
4. Mlích Jozef, Juránek Roman, Zemčík Pavel, Jošth Radovan, Hradiš Michal, Herout Adam, Havel Jiří: *GStreamer Adaboost Plugin,* software, 2010