BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
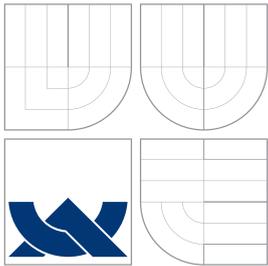
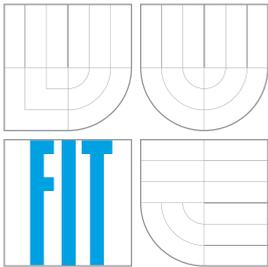# OBJECT DETECTION ON GPU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. PAVEL MACENAUER

BRNO 2015

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

OBJECT DETECTION ON GPU
DETEKCE OBJEKTŮ NA GPU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR   Bc. PAVEL MACENAUER
AUTOR PRÁCE

SUPERVISOR   Ing. ROMAN JURÁNEK, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2015

Abstract

This thesis addresses the topic of object detection on graphics processing units. As a part of it, a system for object detection using NVIDIA CUDA was designed and implemented, allowing for realtime video object detection and bulk processing. Its contribution is mainly to study the options of NVIDIA CUDA technology and current graphics processing units for object detection acceleration. Also parallel algorithms for object detection are discussed and suggested.

Abstrakt

Tato práce se zabývá detekcí objektů pomocí grafických procesorů. Jako její součást byl navržen a naimplementován nástroj pro detekci objektů na technologii NVIDIA CUDA, umožňující detekovat objekty ve videu v reálném čase nebo zpracovávat velké množství fotografií. Jejím cílem je prozkoumat aktuální možnosti technologie NVIDIA CUDA vzhledem k detekci objektů a navrhnout, jak by se daly akcelerovat.

Keywords

Object detection, Classifier, WaldBoost, Local Binary Patterns, CUDA, NVidia, Graphics Processing Unit, Realtime object detection

Klíčová slova

Detekce objektů, klasifikátor, WaldBoost, Local Binary Patterns, CUDA, NVidia, grafický procesor, detekce objektů v reálném čase

Citation

Pavel Macenauer: Object Detection on GPU, master's thesis, Faculty of Information Technology, BUT, Brno 2015

Object Detection on GPU

Declaration

I hereby declare, that this thesis is my own work and has been created under the supervision of Ing. Roman Juránek, Ph.D. All other sources of information, that have been used, have been fully acknowledged.

<div align="right">

. . . .
Pavel Macenauer
May 25, 2015
</div>

Acknowledgment

I would like to thank Ing. Roman Juránek, Ph.D. and Ing. Michal Kůla for support and technical consultations provided during the work on this thesis.

Contents

Chapter 1

GPGPU

With high demand for real-time image processing, computer vision applications and a need for fast calculations in the scientific world, general-purpose computing on GPU (Graphics Processor Units), also known as the GPGPU, has become a popular programming model to accelerate programs traditionally coded on the CPU (Central Processing Unit).

Until the last decade or so, when technologies for GPGPU became available, the GPU was used mostly to render data given to it by the CPU. This has changed in a way, that the GPU, with its massive parallel capabilities, isn't used only for displaying, but also for computation. The approach to this is to transfer data bidirectionally between the CPU and the GPU, which on one hand brings the overhead of copying the data, but on the other, it enables to do the calculations much faster due to the architecture of the GPU. As shown in figure 1.1, many more transistors are dedicated to data processing instead of cache or control, which leads to a higher memory bandwidth.

Figure 1.1: CPU and GPU architecture comparison (taken from [15])

GPUs are also designed with floating-point calculations in mind, which can be taken advantage of, in applications such as object detection, where most of the math is done using single-precision floating-point arithmetic.
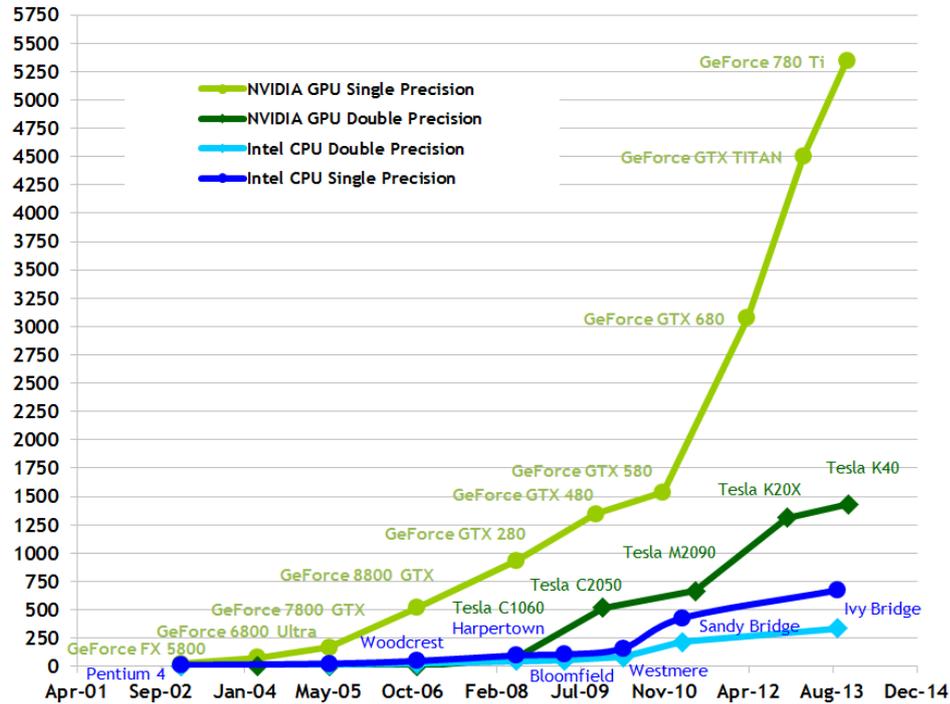
Figure 1.2: Floating-Point operations per second for the CPU and GPU (taken from [15])

1.1 Parallel computing platforms

In November 2006 the first parallel computing platform - CUDA (Compute Unified Device Architecture) was introduced by NVIDIA. Since then the world of GPUs and computer graphics have changed significantly. The GPU stopped to be viewed only as a tool for displaying, but became another processing unit. By introducing CUDA to programmers, suddenly its programming model and memory model became available for use.

The most widely used GPGPU platforms are the following.

- CUDA - NVIDIA

- OpenCL - Khronos Group

- C++ AMP - Microsoft

- Compute shaders - OpenGL

- DirectCompute - Microsoft

All of the technologies above allow access to the GPU computing capabilities. The first two - CUDA and OpenCL work on a kernel basis. As a programmer, you have access to low-level GPU capabilities and have to manage all the resources yourself. The standard approach is the following:

1. Allocate memory on the GPU

4

2. Copy data from the CPU to the allocated memory on the GPU

3. Run a GPU based kernel (written in CUDA or OpenCL)

4. Copy processed data back from the GPU to the CPU

C++ AMP is a more higher-level oriented library. Introduced by Microsoft as a new C++ feature for Visual Studio 2012 with STL-like syntax, it is designed to accelerate code using massive parallelism. Currently it is supported by most GPUs, which have a DirectX 11 driver.

The last two - Compute shaders and DirectCompute also work in a more high-level fashion, but quite differently from C++ AMP. They are not a part of the rendering pipeline, but can be set to be executed together with other OpenGL or DirectX shaders.

1.1.1 Evolution of the graphics pipeline

This change towards a more general-purpose use of GPUs can be also viewed in the current OpenGL pipeline, with which most graphics programmers are familiar. In OpenGL 2.0 (2004) shader programs were introduced enabling to partially program the pipeline itself. With OpenGL 3.* (2008-2010) the shader language had improved significantly and geometry shaders were introduced. Later on in 2010 another programmable stage was introduced with OpenGL 4.0 - the tesselation shader, but still these were set stages with a predefined purpose. The most significant year for the GPGPU was 2012 with OpenGL 4.3 and the Compute Shaders.

Compute shaders are very specific. They don't have a predefined number of inputs, outputs or a place in the pipeline. It is up to the programmer to specify these and also to specify a space on which they operate, like per-pixel or per-vertex basis, by fetching the data itself. This means that even the graphics pipeline, traditionally used for visualization, changed in a way, that it can be programmed without the use of traditional shaders, with compute shaders only, and a more general-purpose use in mind.

1.2 NVIDIA CUDA

NVIDIA CUDA is a programming model enabling direct access to the instruction set and memory of NVIDIA GPUs. The other main competitor to CUDA, on the market, is OpenCL. Both of these models are designed with direct access to the GPUs capabilities in mind compared to other parallel platforms. OpenCL has the advantage of not being platform specific. It is supported by both major GPU vendors AMD and NVIDIA. CUDA is only usable on NVIDIA cards, but generally offers more options to the programmer and a better debugging environment.

1.2.1 Programming model

CUDA uses a language called CUDA C, which is an extension to C and uses NVCC compiler to generate code for the GPU. It also allows to write C-like functions called kernels. A kernel is defined by the `__global__` declaration specifier and is executed using a given configuration wrapped in `<<< ... >>>`. The configuration is specifies a grid and takes as parameters the number of blocks and the number of threads per block. The same kernel code is run by the whole grid. Code run by the kernel on the GPU is called the device code, whereas the code run outside of the kernel by the CPU is called the host code.

Thread hierarchy

Threads are a basic computational unit uniquely identified by 3-dimensional indexes `threadIdx` (an index within a block) and `blockIdx` (an index of a block).

Blocks are groups of threads, where every block resides on a single processor core, therefore a kernel can be run with the maximum of 1024 threads. Each block is uniquely identified by a 3-dimensional index `blockIdx`. Threads within a block can be synchronized by a `__syncthreads` call, but blocks themselves can't be synchronized, and so a new kernel must be run to do so.

The code is executed in a SIMT (Single Instruction, Multiple Thread) fashion, which is very similar to the commonly known SIMD (Single Instruction, Multiple Data) architecture. In reality the code is executed in groups of 32 threads, referred to as warps and a single Kepler or Maxwell multiprocessor supports up to 64 active warps. The important thing to mention about warps is, that when a single thread within a warp is active, the whole warp stays active and so performance-wise it is important to optimize the code to occupy warps as much as possible.
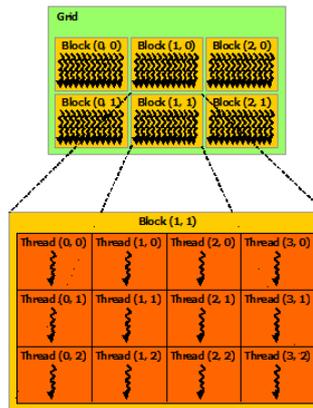
Figure 1.3: A grid of blocks and threads run by a kernel (taken from [15])

Kernel configuration parameters can be passed as integers or `dim3` structures. `dim3` specifies the number of threads or blocks in every dimension, therefore a `dim3 threadsPerBlock(4,4,1)` would run a kernel with 16 threads per block, where `threadIdx.x` would range between 0 and 3 and the same for `threadIdx.y`.

Example 1.4 shows how to add 2 arrays in parallel using N threads and 1 block.

Scheduling

After a kernel is launched, the grid is organized in blocks, grids and threads, which are described in section 1.2.1. Depending on the architecture, blocks can have only up to certain number of warps and threads. For every architecture, these values are summarized in table 1.2.

Blocks are assigned between multiprocessors, where for modern GPUs, their count ranges between a single multiprocessor for average GPUs and 16 for the high-end. Threads

```
// 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);
    ...
}
```

Figure 1.4: Example of vector addition in CUDA (taken from [15])

within a block are then divided into warps. Warp execution is controller by a warp scheduler using a prioritized scheduling policy.

All threads within a warp execute the same instruction at the same time. In case the instruction is not active for the given thread, an active mask is used and as soon as all the threads within the warp finish or are stalled by synchronization, a new warp is assigned. This is shown in figure 1.5, where as soon as an instruction finishes, another instruction, processed by a different warp, is assigned.
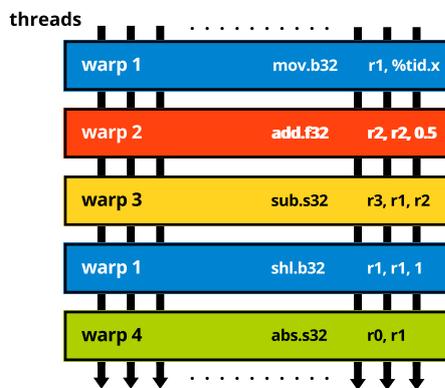
Figure 1.5: As soon as an instruction has its operands ready, it is scheduled by the warp scheduler for execution.

1.2.2 Memory model

Another important aspect of CUDA are the types of memories, which can be used. They range from locally accessible registers with extremely fast access to globally accessible global memory, which takes hundreds of cycles to read or write to or from. They are summarized by the following table:

Memory	Keyword	Scope	Access	Lifetime
Registers	-	Thread	Read/Write	Kernel
Local memory	-	Thread	Read/Write	Kernel
Shared memory	`__shared__`	Block	Read/Write	Kernel
Global memory	`__device__`	Grid	Read/Write	Application
Texture memory	-	Grid	Read-only	Application
Constant memory	`__constant__`	Grid	Read-only	Application

Table 1.1: CUDA Memory types

Global memory is accessible by all threads in a grid and allows both reading and writing. It is also the slowest memory type. Its access is the bottleneck for most applications with access latency ranging from 400 to 800 cycles. There are several strategies for it to be fast like coalescing access with 32-, 64-, 128-byte transactions, described in section 1.2.3.
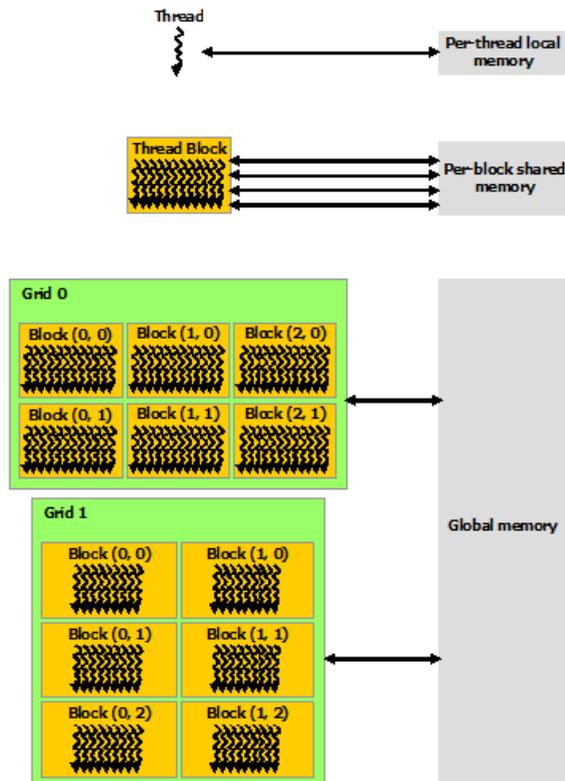
Figure 1.6: Memory hierarchy ([15])

Texture memory can be regarded similarly to global memory. Cache is optimized for 2D spatial access pattern and address modes or interpolation can be used at no additional cost.

Constant memory is the third memory type, which can be accessed by all threads and is typically used to store constants or kernel arguments. It doesn't bring any speed-up compared to global or texture memory, but it is optimized for broadcast, which means that it is optimal to use it, when the same data is accessed by all the blocks.

Shared memory can be accessed by all threads within a block. It is much faster than the other types, but is commonly subjected to bank conflicts. It is mainly used for acceleration.

Unified memory is a memory type introduced in CUDA 6.0 (version of CUDA Toolkit). It enables to use the same memory addresses both in host and device code, which simplifies writing code. On the other hand, as of spring 2014, there doesn't seem to be any hardware support and performance-wise the unified memory performs very similar to global memory.

Local memory is a part of global memory, where everything which doesn't fit into registers is stored. For devices with Compute Capability 2.x there are 32768 32-bit registers.

1.2.3 Optimization methods

Global memory coalescing

Using global memory is generally slow, but it is unavoidable to use, because it is the only type of GPU memory, which can be copied to from the CPU. On the other hand, since Compute Capability 1.1, global memory loads and stores can be optimized by coalescing, thus making it much faster.

GPU can access memory in 32-, 64- or 128-byte transactions. Therefore, in case the memory access is aligned, multiple banks can be read or written to in a single transaction. On the other hand when it is not, more transactions have to be initialized, making the memory bandwidth much lower. Examples of coalesced (top) 1.7 and uncoalesced (bottom) 1.8 access with stride of 4 bytes are shown in figure 1.9.

```
tmp = globalmem[blockIdx.x * blockDim.x + threadIdx.x];
```

Figure 1.7: Example of coalesced global memory load.

```
tmp = globalmem[blockIdx.x * blockDim.x + threadIdx.x + stride];
```

Figure 1.8: Example of uncoalesced global memory load with a stride.

It is a common optimization technique, to use shared memory in cases, where strided access is needed. Data is copied between global memory and shared memory in a coalesced fashion, but then strided access is used on shared memory without a penalty.

Thread synchronization and stalling

Threads cannot be synchronized across the whole grid, but only across a block. Therefore, if a result of a certain computation is needed by all the threads, a whole new kernel has to
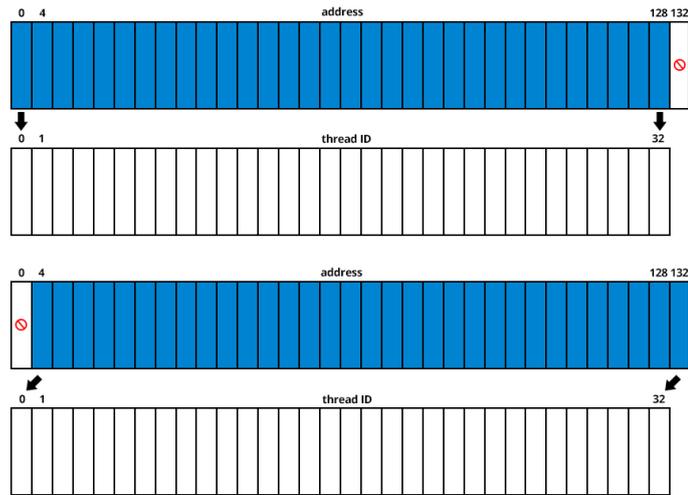
Figure 1.9: Coalescing access to global memory. Top part shows coalesced access, whereas bottom part shows strided uncoalesced access.

be launched. Synchronization across a block is done by the `__syncthreads` call. When a thread reaches this call, it waits idly until all the threads within the block also reach the call.

In certain scenarios, it might take a lot longer for a certain thread to process its code, then for another. An example is the prefix-sum, as described later on in section 4.6.4, where the number of iterations of a for-cycle depends on the thread ID. The problem is, that after the for-cycle finishes, threads must be synchronized. Most of the threads in this case are stalled and cannot be freed until all of them finish to run another warp.

When writing a kernel, it is important to keep this in mind and try to minimize the use of `__syncthreads` calls, where this would lead to stalling most of the threads.

1.2.4 Compute capability

NVidia GPUs vary not only by their parameters, but also by their architecture. There are 4 architectures supporting CUDA: Tesla (November 2009) [10], Fermi (March 2010) [12], Kepler (March 2012) [13] and Maxwell (March 2014) [14]. Another way to describe the architecture is its compute capability, which is 1.x for Tesla, 2.x for Fermi, 3.x for Kepler and 5.x for Maxwell. The higher the compute capability, the more features such GPU supports, also all of the compute capabilities have different parameters summarized in table 1.2.

1.2.5 Maxwell improvements

In march 2014 the new Maxwell architecture was released and it brought several improvements, which affect application performance and are taken heavy use of in the implementation described in chapter 4. These are summarized below.

Compute Capability	2.x	3.x	5.x
Architecture	Fermi	Kepler	Maxwell
Max. number of blocks per SM	8	16	32
Max. number of warps per SM	48	64	64
Max. number of threads per SM	1536	2048	2048
Max. number of threads per block	1024	1024	1024
Local memory per thread	512 KB	512 KB	512 KB
Max. shared memory per SM	48 KB	48 or 112 KB	64 or 96 KB
L2 cache size	768 KB	512 KB	2048 KB

Table 1.2: Limits of devices for every architecture. SM stands for Streaming Multiprocessor.

Larger shared memory

Shared memory, described in section 1.2.2, is increased from 64 KB to 96 KB per multiprocessor. Shared memory is the faster memory type, because it resides on the multiprocessor itself. This means, that it is available only for the given block. On the other hand it's by far the faster memory type and is used mainly for performance optimization. Increasing its size makes it more available.

Larger L2 cache

The size of L2 cache is increased $4\times$ from 512 KB to 2048 KB. This allows for performance benefits for bandwidth-bound applications such as image processing.

Shared memory atomic operations

Maxwell implements native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions with reduced overhead.

The difference can be seen in PTX assembly, where atomicAdd compiled for Kepler shown in figure 1.10 is made up of 4 instructions, where the same code for Maxwell shown in figure 1.11 is made up of a single instruction.

```
/*00c8*/ LDSLK P0, R2, [RZ];
/*00d0*/ @P0 IADD R2, R2, 0x1;
/*00d8*/ @P0 STSCUL P1, [RZ], R2;
/*00e0*/ @!P1 BRA 0xc0;
```

Figure 1.10: atomicAdd compiled with nvcc -arch=sm_35 (Kepler) (taken from [1])

```
/*00f0*/ @!P0 ATOMS.ADD RZ, [RZ], R4;
```

Figure 1.11: atomicAdd compiled with nvcc -arch=sm_50 (Maxwell) (taken from [1])

New multiprocessor design

The new Maxwell multiprocessor is called SMM (multiprocessor for Kepler was called SMX). All the different parameters are described in table 1.2. Apart from that, most of the other improvements were made with improved occupancy without increasing the parallelism, in mind, such as dependent arithmetic instruction latency being reduced, warp schedulers having dedicated CUDA cores or having more active warps per SMM.
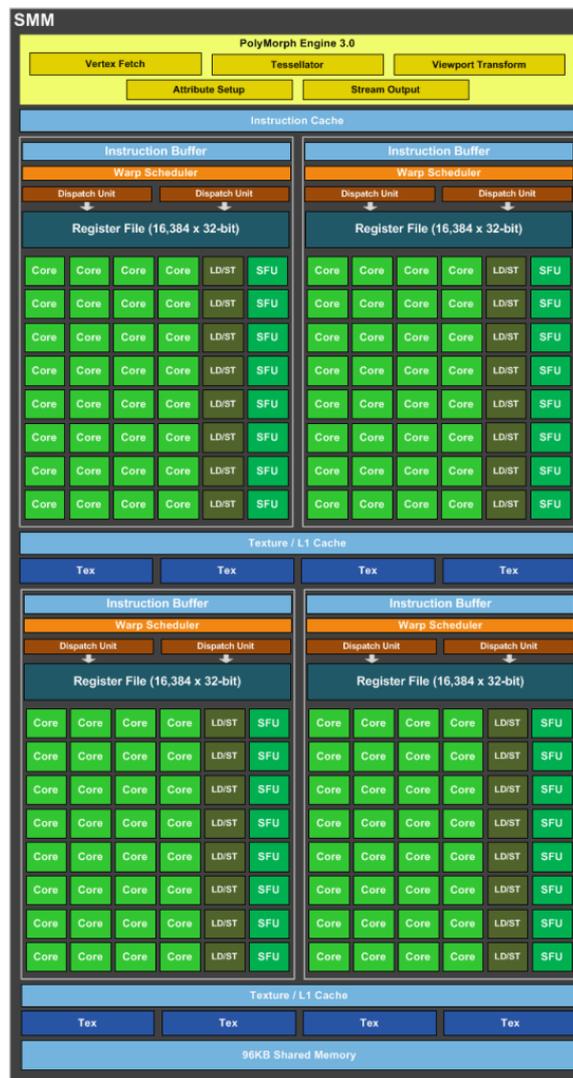
Figure 1.12: Maxwell's multiprocessor, SMM. Taken from [14].

Chapter 2

Object detection

Object detection is the process of localizing an object in input image data. The type of object depends on which data the detector was trained for. Typical applications are human faces, pedestrians, cars, traffic signs and others. A more detailed description of such use cases is discussed in section 2.7.

In this chapter, we will briefly discuss different approaches to object detection and in more detail, the implemented method, which is WaldBoost, a meta-algorithm used to train classifiers and classify samples.

2.1 Approaches to object detection

Objects are mostly classified using a features, which were first introduced by Viola in [22]. To explain them more thoroughly, let's take a face as an example. Despite the differences such as lighting, color of eyes or skin or the length of hair, we as humans, can identify we are looking at another human face based on certain similarities. For example a pair of eyes, a nose and a pair of ears in their corresponding positions. These similarities can be called a set of features. To a computer, on the other hand, they are still too abstract and cannot be enumerated, and so a lot of different feature methods were devised.

Classifying an object by a feature-based method can be approached using different strategies. One such classification of object detection was summarized by Yang, Kriegman, Ahuja [8].

Knowledge-based methods. These methods encode knowledge of a certain object and capture relationships between its features. For a face this means capturing relationships between the positions of eyes, mouth, nose or ears.

Feature invariant approaches. These algorithms try to find similarities which don't depend on lighting, viewpoint or pose.

Template matching. Patterns to describe an object as a whole are stored. Correlations between the input image and the patterns are then computed to detect the object.

Appearance-based methods. Models are learned from a set of training images, which capture a variability of the object. Such set of training images contains positive and negative samples.

2.2 Appearance-based methods

The most exploited approach to object detection and mainly face detection is the appearance-based. In general, these rely on methods from machine learning or statistical analysis to develop a model for the sought object. Such methods can be characterized from a statistical point of view, where a sample from an image is classified by a Bayesian classification as an object or a non-object. In reality more advanced methods are used.

2.3 AdaBoost

One such algorithm to analyze a sample is AdaBoost, an algorithm originally introduced in 1995 by Viola, Jones [23]. The main idea is combining several weak classifiers into a single strong classifier. This means, that every weak classifier decides to which class the given sample belongs to. In object detection this is usually a yes or no decision, and after processing all of the weak classifiers a final decision is made, also a called a strong classifier. The strong classifier is evaluated as sum of decisions made by the weak classifiers.

2.4 WaldBoost

Another algorithm and one, which we will discuss in more detail is WaldBoost, because it is used for the implementation. WaldBoost is an algorithm, which combines AdaBoost and Wald's Sequential Propability Ratio Test (SPRT). SPRT is a strategy to determine what class a sample belongs to, based on a series of measurements. It's main advantage over AdaBoost, which was briefly explained in section 2.3 is, that it doesn't have to process all of the weak classifiers, but can make a decision throughout the process.

$$SPRT = \begin{cases} +1 & \text{if } R_m \leq B \\ -1 & \text{if } R_m \geq A \\ \# & \text{else take another measurement} \end{cases}$$

R_m is the likelihood ratio and A, B are constants to compute the wanted false negatives α and false positives β ratios as follows:

$$R_m = \frac{p(x_1, ..., x_m | y = -1)}{p(x_1, ..., x_m | y = +1)} \tag{2.1}$$

$$A = \frac{1 - \beta}{\alpha}, B = \frac{\beta}{1 - \alpha} \tag{2.2}$$

As mentioned in [19] with face detection in mind, the positive rate β can be set to 0 and the required false negative rate α to a small constant. As such the equations can be simplified to

$$A = \frac{1 - 0}{\alpha} = \frac{1}{\alpha}, B = \frac{0}{1 - \alpha} = 0 \tag{2.3}$$

R_m is always positive and therefore the algorithm will only classify the sample as a face when it finishes its training cycle or discard it as a background when the ratio gets greater than the given constant A. Therefore the whole strategy can be simplified to the following.

$$SPRT = \begin{cases} -1 & \text{if } R_m \geq \frac{1}{\alpha} \\ \# & \text{else take another measurement} \end{cases}$$

2.5 WaldBoost as an object detector

Now we have an idea of how AdaBoost introduced in section 2.3 and mainly WaldBoost explained in section 2.4 work, we may explain this in terms of object detection.

The detector scans an image using a scanning window, usually the size of 24×24 or 26×26 pixels, for every possible location and searches for the object. Such scanned region of interest is called a sample and to make a decision, whether it's the given object or not, it is processed by a strong classifier (produced by the WaldBoost algorithm).

The algorithm processes weak classifiers and for each one of them compares them to the constant A mentioned in equation 2.4. Such weak classifiers are trained based on a given method such as LBP, LRD and so on (these methods are covered later on in section 2.6.

After processing all of the weak classifiers, the sample is classified as an object or gets discarded throughout the process. A more detailed explanation is given for the implementation in section 4.1.

2.6 Image features

What we haven't discussed yet is how these image features, which are statistically processed are described. A method of extracting such features from a texture or an image has to be devised. Fortunately, there are many such methods, where the choice heavily affects the performance of the detector. Examples of such features are summarized below.

- Haar-like features [23]

- Local Binary Patterns [16]

- Local Rank Differences [6]

- Local Rank Patterns [7]
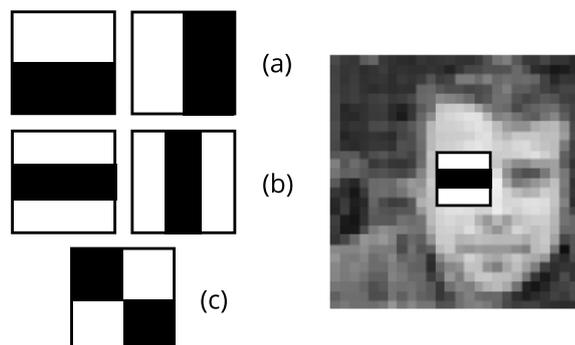
Figure 2.1: Haar-like features. (a) Edge feature (b) Line feature (c) Four-rectangle feature.

2.6.1 Haar-like features

A method originally used by Viola and Jones [23] in their object detection framework. Haar-like features are described in figure 2.1, where the sum of the pixels in the black area is subtracted from the sum of the pixels in the white area. The calculation can be accelerated by using integral images, which store the image as already precalculated sums.

2.6.2 Local Binary Pattern

Another method to describe image features, which is used in the implementation, are Local Binary Patterns (LBP) [16]. They are based on encoding local intensities of an image with 8-bit codes. In their elementary form, they take a 3×3 area as an input and compare intensity values of all the pixels with the central one as shown in figure 2.2. Mathematically LBP evaluation is summarized below.

$$compare(p_{middle}, p_i) = \begin{cases} 1 & \text{if } p_i \geq p_{middle} \\ 0 & \text{else} \end{cases} \tag{2.4}$$

$$lbp(p_{middle}) = \sum_{i=0}^{7} 2^i compare(p_{middle}, p_i) \tag{2.5}$$

9	1	10
2	5	6
12	4	4

1	0	1
0		1
1	0	0

1	2	4
128		8
64	32	16

1	0	4
0		8
64	0	0

LBP(x) = 1 + 4 + 8 + 64 = 77
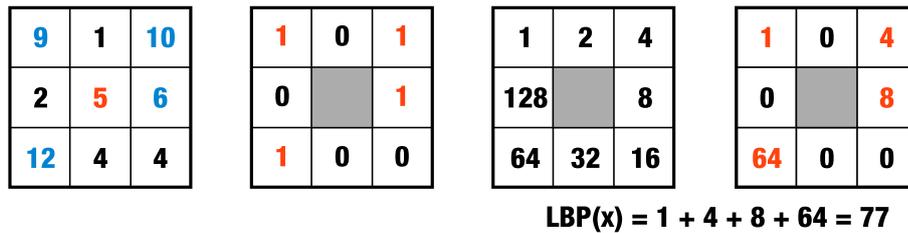
Figure 2.2: Evaluation of the LBP feature.

LBPs can be extended to be used not only for single pixels and thus 3x3 areas, but also for larger areas. When comparing larger areas, the sum of the areas is compared instead of single pixel values. For example a 2x2 LBP would sum up areas of 4 pixels and compare them with the sum in the middle.

LBP features are invariant to lighting changes, because even though the image is lighter or darker, the sign of intensity differences stays the same. On the other hand they are not invariant to geometrical transformations such as scale or rotation.

As mentioned later on in section 4.4, in reality the sum of the areas doesn't have to be used and an average can be used instead. This is then exploited on the GPU using texture memory.

2.7 Uses cases of object detection

Object detection isn't only used on its own, but also is a part of use cases. These range from uses in security surveillance, to traffic analysis or biometrics. Object detection doesn't have to be bound to humans, there are works researching detection on dogs [9]. A short summary of such applications is given below.

2.7.1 Face tracking and recognition

In order to recognize a face and track it afterwards, it has to be detected at first. Recognition is done usually by comparing the detected region of interest with a reference and deciding whether it's similar enough. This can be done by different techniques such as comparing image histograms. Face tracking is a similar use case, but for a video, which compares detections in consecutive frames.

2.7.2 Biometrics

An extension to face recognition is the use of biometrics. These are metrics describing human characteristics. Examples can be an iris, retina, fingerprints, palm veins or a face. They are used mainly for authentication, and as such these metrics have to be detected first. Apart from authentication, they these are starting to be used in modern vehicle systems, which detect whether a person is tired or sleeping and try to wake him or her up.

2.7.3 Traffic analysis

A very important application of object detection is traffic analysis. A trivial use case is vehicle speed measurement, where in order to measure the car's speed, it has to be detected in consecutive frames of a video, after which the speed can be calculated based on framerate and distance traveled.

In traffic analysis, object detection actually has many more uses. Traffic congestion can be analyzed and afterwards data sent to inform traffic control or an intersection can be monitored in order to analyze how dangerous it is by measuring the angle or speed at which vehicles enter it. Example of such a work is described in [20].

2.7.4 Other uses

There are many more uses, not only those described above. Other examples can be video stabilization, vehicle parking systems, image panoramas, detecting parts of an image in order to perform more complicated object detection. A smart device GUI can be developed, which adapts itself based on where or at which angle the user is looking.

Chapter 3

Analysis

In this chapter, the general design of an object detector is discussed and later on tools for measuring its performance and profiling. It does not cover implementation specifics. These are discussed later on in chapter 4.

The idea of implementing a WaldBoost object detector with the use of LBP features on a parallel platform such as CUDA is not a new one and mainly for face detection, it is well researched. On the other hand, since it was first proposed, both OpenCL and CUDA have undergone many changes, and so research done in this area tends to get out of date.

The implementation of a WaldBoost detector on a CPU, where every pixel is processed sequentially isn't complicated. The general idea, as explained in section 2.4, is that a sample, in our case a pixel, is processed by a series of weak classifiers (also called stages). These stages accumulate a response, always adding or subtracting a value based on how the LBP feature is evaluated. In case the accumulated response gets greater than a certain value, the sample is discarded, the program skips the rest of the classifiers and continues with the next pixel.

Image processing tasks are generally suitable for parallel acceleration. On the other hand it is not as simple. The number of pixels is huge and as shown in figure 3.1, most of the samples get discarded as background after few initial stages. On a parallel platform, such as the GPU, this presents a problem, because most of the processors have to wait for the few samples, that are still being processed and thus wasting a lot of resources.
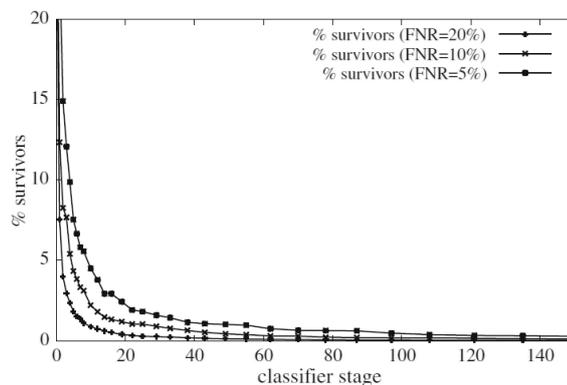
Figure 3.1: Surviving percentage of samples gets below 5% after few initial stages. Taken from [5].

18

Most of the current implementations of WaldBoost detectors are based on a work proposed by Herout [5], which is summarized in sections 3.1 and 3.2.

3.1 Pyramidal image

The detector uses a window for scanning the image, usually 24x24 or 26x26 pixels. In order to detect the object in such window, a pyramidal image was proposed. Such image contains many sub-sampled images stored in a pyramid fashion. The structure of the pyramidal image and the way how to construct it is discussed in section 4.3.2.

3.2 Thread rearrangement

As shown in figure 3.1, only a few samples are being processed after the few initial stages. Without a thread rearrangement scheme, these few surviving samples compute, meanwhile the others, which were already discarded, wait for these to finish. As such, they occupy GPU resources and most of the computational power and parallelism is wasted. The rest of the samples still occupies GPU resources and waits for these to finish. To free these resources and use them accordingly, a method of thread rearrangement has to be proposed.

At the time of writing the paper, detectors with 512 or 1024 stages were used (this implementation uses a detector with 2048 stages). Herout proposed [5], that every sample can be processed up to a set number of stages, after which thread rearrangement is done and the process continues. This is described in figure 3.2.
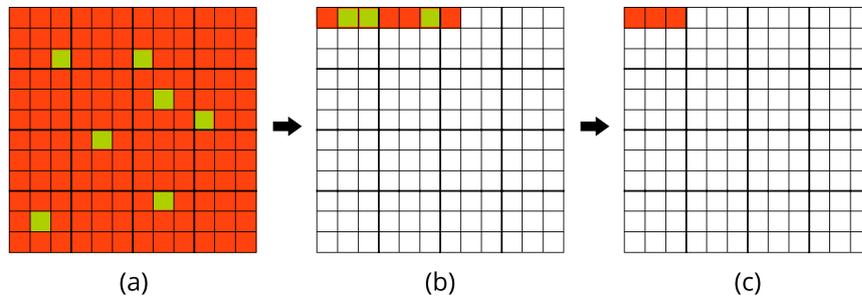
Figure 3.2: Surviving threads (green) after each thread rearrangement. Every thread processes a single sample. (a) 144 initial threads, 7 survive (b) Threads reorganized into 7 threads, 3 survive. (c) Threads reorganized into 3 threads, none survive.

Implementation details on how threads are rearranged are covered in section 4.6.

3.3 Debugging and profiling tools

One of the main advantages of CUDA over OpenCL are its debugging and profiling environments, which are important for writing optimized and error-free code. There are several CUDA-specific options to use. CUDA Profiler API also provides functions to help with

profiling. Only specific parts of the code can be profiled by using `cudaProfilerStart` and `cudaProfilerStop` calls. This allows to collect data only for important parts of the code.

3.3.1 Visual Profiler

The first option to profile and optimize code is the Visual Profiler. As the name itself says, it displays a visual representation of all the GPU calls, where statistics such as memory bandwidth (shared, local, global, ...), instruction throughput, occupancy and much more can be displayed. It is well documented [1] and provides a simple interface to use (shown in figure 3.3).
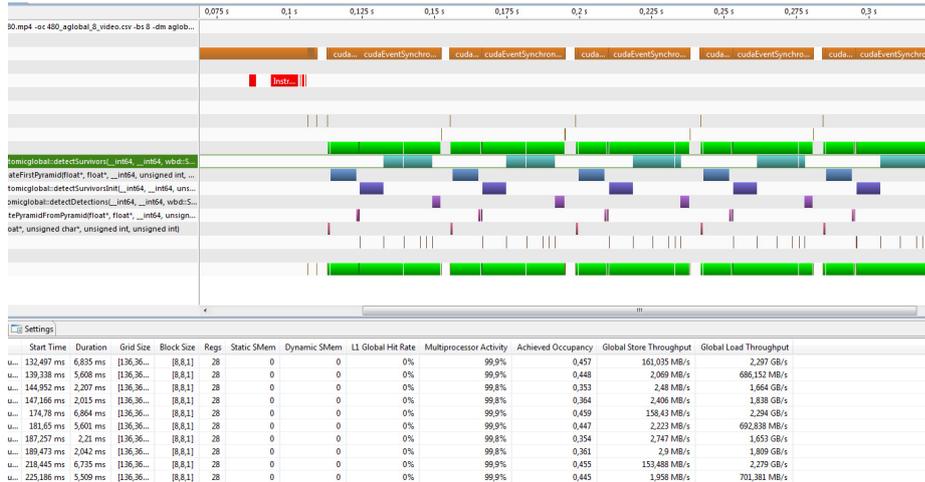
Figure 3.3: Visual profiler interface.

To profile a program, only four things need to be set up: binary file to run, CUDA toolkit version, working directory and command line arguments. The profiler allows to export the results to a CSV file.

3.3.2 nvprof

Nvprof has basically the same options as the Visual Profiler, with the difference, that it is a command-line tool. As such it can be used as a part of a script, which can automate the whole process of profiling.

---

[1] http://docs.nvidia.com/cuda/profiler-users-guide/#visual-profiler

```
nvprof --analysis-metrics -o  myprog-analysis.nvprof ./myprog --benchmark -i=1
```

Figure 3.4: An example how to run nvprof to output profiling data.

3.3.3 NVIDIA Nsight

Another handy tool to use is NVIDIA Nsight, which is a set of tools developed for Microsoft Visual Studio [2] and Eclipse [3] allowing to debug and analyze GPU and graphics applications.

It consists of two main modules and allows for simple profiling. On the other hand dedicated tools such as visual profiler, described in section 3.3.1, or nvprof, described in section 3.3.2, are better for the job.

CUDA Debugger allows to set breakpoints inside GPU code and view the local variables, inspect the memory or perform other debugging tasks such as memory checks.

Graphics Debugger allows to debug vertex shaders, pixel shaders and view the current state of the graphics pipeline.

3.4 Metrics

Apart from the choice of profiling tools, there is also a lot of metrics and events to be measured. Below, a short introduction will be given to those, which will be discussed and measured later on in section 5.2.

3.4.1 Achieved occupancy

Achieved occupancy is the number of active warps per clock cycle divided by the maximum warps per streaming multiprocessor. In other words, it is the percentage of used up GPU computational resources. It is one of the key factors influencing every GPU application, because as long as there are resources available, they can be used. Maximum occupancy doesn't always lead to the best results, because other areas of optimization, such as memory access, atomic operation stalls or cache hit rate may outperform it. Typical case of slow kernels with full occupancy are kernels with large blocks (for example 1024 threads).

3.4.2 Memory load/store throughput

Memory throughput is the amount of data transferred by the kernel per second, either by writing into the memory (store) or reading from it (load). It is not a metric, which can be directly optimized in code, but gives the programmer an insight into performance of the application.

3.4.3 Atomic and L2 throughput (for atomic requests)

As of Visual Profiler 6.5, there isn't a good metric to measure the performance of atomic operations, or at least in a way, which would be useful to our implementation. However a general insight into their performance can be gained by either their throughput or the L2 cache throughput for atomic requests. It doesn't measure how the kernel gets stalled, when multiple threads are writing to a single location using atomic instructions, but it is directly affected, which means the higher the throughput, the lower the stalls.

---

[2] https://www.visualstudio.com/
[3] https://eclipse.org/

3.4.4 L2 hit rate (texture reads)

The last measured metric is the L2 hit rate for texture reads, which is the percentage of texture reads, which were found in the L2 cache. L2 cache is located in between global memory and the multiprocessor as shown in figure 1.12. All data which is read from the global memory or the texture memory goes through this cache. Texture memory is generally as fast as global memory. On the other hand it utilizes one thing, that the global memory doesn't - spatial data locality. As such, a lot of data can be fetched from the cache and greatly speed up the application. A metric to measure the percentage of data fetched from the cache is the L2 hit rate.

Chapter 4

Implementation

The object detector is implemented in C++ with dependencies on OpenCV[1] - an open source computer vision library with C and C++ interfaces and CUDA[2] - a library for writing NVIDIA GPU code with a CUDA C interface, which is an extension to C compiled by the NVCC compiler.

It is implemented as an easy-to-use class, which takes a dataset or a video on the input and outputs detections. Current implementation uses an open-source face detector trained by R. Juranek[3] under Brno University of Technology license. As such, it can be used as a part of an application such as face tracking or face recognition, but not only faces have to be processed. Depending on what detector is trained on, anything from traffic signs to humans can be detected.

Figure 4.1: A sample output with debug information shown.

---

[1] http://www.opencv.org
[2] http://www.nvidia.com/object/cuda_home_new.html
[3] http://medusa.fit.vutbr.cz/LBPDetector/

A CMake[4] project is included, therefore it can be built on multiple platforms. As for the project itself, it is logically separated into the following modules and sub-modules (in code this is done using C++ namespaces):

- wbd - wrapper around the whole detector

- wbd::simple - CPU implementation of the waldboost detector

- wbd::gpu - WaldboostDetector class, an interface to use the detector and most of the methods used for GPU acceleration

- wbd::gpu::pyramid - pyramidal image generation on the GPU

- wbd::gpu::detection - object detection on the GPU

4.1 Detector

The detector can be used to detect any object it is trained to, not only faces. Most detectors are stored as XML files, which brings the overhead of reading and parsing a file and so an exported version of an XML detector to a C++ header file is used. As such 2 header files (`wbd_detector.h`, `wbd_alphas.h`) must be compiled with the detector.

These headers are be obtained by parsing a XML detector [5]. The detector consists of two main parts, the stages or also called the weak classifiers, which are loaded into a structure described in 4.2 and an α-table, which is stored as an array of floating-point values. Every stage points to a specific range inside the α-table, where responses for all the LBP values are stored. Training of a detector is not the subject of this thesis and as such it will not be discussed.

```
struct Stage {
    uint8 x, y; // X, Y offsets
    uint8 width, height; // width, height of the feature
    float thetaB; // threshold
    uint32 alphaOffset; // alpha table offset
};
```

Figure 4.2: Stage structure

The detector uses a 26x26 pixel-wide scanning window, where `x` and `y` are offsets inside the window and `width` and `height` describe the size of the feature. It is the size of 1x1, 1x2, 2x1 or 2x2, because of this knowledge, hardware provided bilinear interpolation can be used. `alphaOffset` is an offset inside the α-table corresponding to appropriate LBP values. `thetaB` is the threshold value, which decides if the sample gets discarded or not in the current stage.

---

[4]http://www.cmake.org/
[5]https://github.com/mmaci/classifier-export

4.2   Interface

The object detector is designed to be used using a class WaldboostDetector, to simplify the detection as much as possible it is able to be run only by loading a video or a dataset and then running the `run()` method, which returns a list of detections. Optionally the programmer can set specific settings described such as the method used ir block configuration. On the other hand when not set, these are set optimally based on results obtained by measurements on different GPUs.

The whole interface is described in the added Doxygen[6] documentation.

4.3   Program cycle

The detector processes both videos and image datasets, which are passed to the detector as a filename. For a dataset, a text file is used with filenames of the images separated by a newline. OpenCV is used to load both images and videos and as such it supports most common formats.

The application pipeline (described in figure 4.3) of the detector is a bit different for a dataset, than for a video, but both use the following steps as described in sections 4.3.1 to 4.3.4.

For a video, the detector can be initialized only once with the initial frame. All the other frames have the same size and so the same memory structures can be used. Image setup and the Detection are then run multiple times for every frame. After the video is processed, memory is freed.

For a dataset, every image can be different and so the whole process must be run multiple times for every loaded image.

4.3.1   Initialization

The initialization of the detector can be separated into two parts. The first being the tasks, that have to be done only once for both datasets and videos. This means copying the detector into GPU memory (the detector contains weak classifiers/stages and an α-table). This is done in the constructor of the WaldboostDetector class.

The other part being allocating GPU memory for the image itself, which is allocated dynamically due to dependence on the size of the input image. For a video this can also be done only once, because every frame has the same size and as such, everything is allocated based on the initial frame. On the other hand a dataset can have completely different image sizes and so this must be done for every photo separately.

4.3.2   Setting an image

After the initialization is complete, an image or a frame is loaded by OpenCV and then copied over to the GPU. This must be done both for every image or every frame. It is copied over to the global memory, there it is preprocessed, as discussed in section 4.3.2, and set as a texture. From the texture, a pyramidal image is generated, as described in section 4.3.2. After that, everything is ready for object detection.

---

[6]http://www.doxygen.org
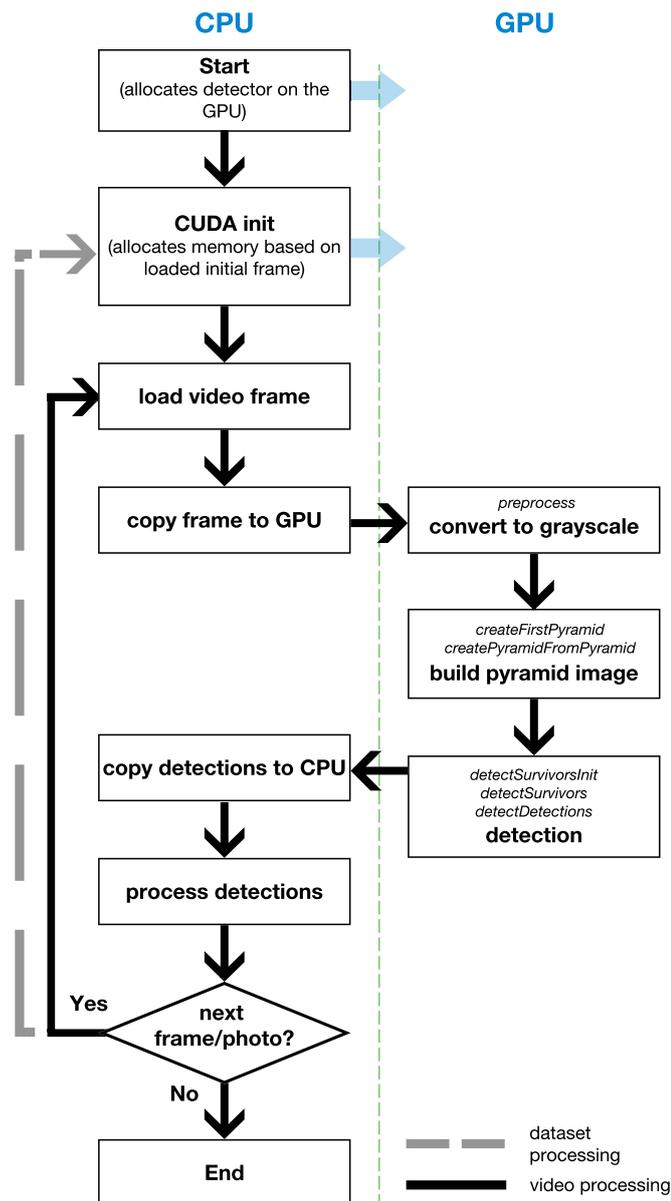
Figure 4.3: Application pipeline. On the GPU side, kernel methods are mentioned.

Preprocessing

The first kernel to be run is the preprocessing kernel. There are 2 operations to be done - conversion from the initial integers to a floating-point representation and then a conversion to grayscale.

Floats are needed in order to work with textures. Texture memory enables hardware implemented bilinear interpolation for subsampling images, which is done mainly in creating the pyramidal image. Even though bilinear interpolation should only be supported for floating-point values, due to official documentation, it is not the case. On the other hand it might be a bug, which might get fixed in the future and it would be wise not to be used.

The detector itself is trained on grayscale images, and so the input must also be in

grayscale. After the kernel finishes, the result is saved as a texture.

Dynamic texture and texture objects

As of CUDA 5.0 and Kepler GPUs textures don't have to be defined globally as static textures, but they can be used dynamically using Texture Objects (`cudaTextureObject_t` class API [2]).

This has several advantages. One of them being the slight overhead (up to $1.5\mu s$) by binding and unbinding static textures during kernel launch, which is eliminated. Even though this is not our case and doesn't sound as much, it might be quite a significant overhead while launching large quantities of fast kernels.

The main advantage in our case is, that texture objects can be passed as arguments, set to store variable image sizes and therefore easily be used as a part of a library. Also multiple textures can be created using given parameters, which is exploited in pyramidal image creation and explained in section 4.3.2.

Pyramidal image

All the pixels are processed using a 26x26 pixel-wide window. The size again depends on how the detector is trained. The basic idea is that for the given object, all the features describing it fit inside this window. On the other hand objects, such as faces, are usually much larger in the initial image, therefore we have to create many sub-sampled images and run the detection on every one of them. In order to do this a the second kernel to be run is the pyramid kernel.

The straight forward method is to create a single pyramidal image and store it as a texture on which the detection is run.

As mentioned previously, the implementation uses hardware bilinear interpolation provided by the texture memory for sub-sampling the image. It has to be kept in mind, that bilinear interpolation has some negative side-effects, one of them being the sub-sampling of an image below half its original width.
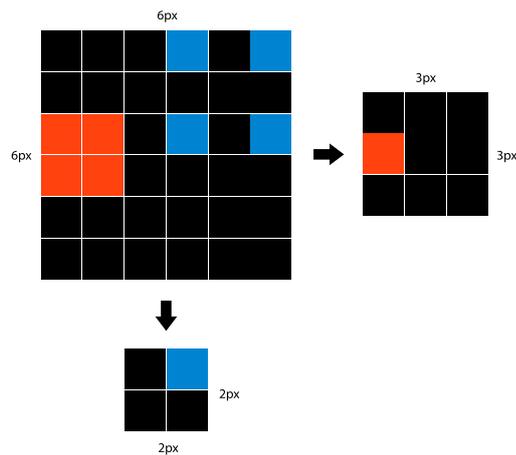
Figure 4.4: Error when sub-sampling an image below twice the original width using bilinear interpolation

As shown in figure 4.3.2, when sub-sampling an image below half its original width, pixels are left out and a sampling error is created. Other methods, such as Lanczos resampling, solve this at the cost of a computational overhead, because a software implementation has to be used instead of the hardware-based provided by using a floating-point 2D texture.

The image is generated in octaves. An octave is a structure of several images, where the smallest image has half the width/height of the original image. Depending on the number of images in an octave, every image is $2^1/number_of_{images}$ smaller than the previous. The following octave is then sub-sampled from the previous octave.

Two methods were tested. The first one using a single texture for both writing a reading the individual octaves. This proved to be very costly and therefore another method had been implemented and is described below:

1. A pyramidal image of N images is generated, where each image is $2^1/N$ smaller then the previous. Every image is sub-sampled from the original image.

2. Generated pyramidal image is stored as a dynamic texture. Simultaneously the pyramid is being written inside a final image used for detection.

3. A pyramidal image is generated, by sub-sampling the previously generated pyramid. Width and height of the sub-sampled pyramid are twice smaller than that of the original.

4. Generated pyramidal image is saved as a dynamic texture.

5. Steps 3 and 4 get repeated for a set number of octaves.

Figure 4.5: Pyramidal image

The implementation itself uses 8 levels and 4 octaves. This results in the sides of the smallest image being 16× smaller than the original. By reducing the number of octaves or levels, better performance can be gained at a cost of the quality of the detections.

4.3.3 Detection

To detect an object, the detector has to successfully evaluate a given number of stages (weak classifiers) described in section 4.1, which are processed sequentially. In our case 2048. This number can be reduced, but it will have an impact on the number of detections. Every weak classifier the algorithm processes, a response is given. The response is a number read from the α-table, from the address given by the stage offset and the calculated LBP value. It is then added to the accumulated response. The sample is discarded in case the accumulated value falls below the stage threshold `thetaB`.

for *every pixel (a GPU thread is created)* **do**
    for *every stage* **do**
        1. compute LBP coefficient
        2. add response for the given LBP to the accumulated response
        if *accumulated response $\geq$ stage threshold thetaB* **then**
            discard sample
        end
    end
end

Algorithm 1: Object detection algorithm simplified

4.3.4 Freeing the memory

After everything has been processed, the GPU memory can be freed. This also must be done for every image in case of a dataset, but only once in case of a video.

4.4 Memory organization

The use of GPU memory is one of the most important parts of programming on GPU architectures. The types of CUDA memories are described in section 1.2.2.

Below we will discuss, how the most important parts of the detector are stored and why.

- **Stages/Weak classifiers** - constant memory
  Stages are stored in the constant memory. Even though it's not as fast as let's say shared memory, its capability to broadcast simultaneously accessed data is ideal. Every thread processes a single image position, for which it loops through a for-cycle of stages. Every read from the constant memory is then not only broadcast to a half-warp (a group of 16 threads), but also cached. The only problem can be the size, which is limited to 64 KB. The detector uses 2048 stages, where each stage is 12 B. This leads to 24 KB, which is enough, but has to be accounted for when storing other data in the constant memory.

- **α-table** - texture memory
  There are 256 coefficients for every stage and every coefficient is stored as a float. This

leads to $256 * 2048 * 4 = 2MB$ and by far exceeds the memory available for constant memory. Another reason the constant memory cannot be used is that the access is random, due to the fact, that we are likely to get different LBP values for every pixel. In this case we wouldn't utilize the broadcast and it would even be slower.

- **Original image and pyramidal image** - texture memory
  Both are stored in the texture memory. Original image is used to create a pyramidal image using hardware accelerated bilinear interpolation for creating down-sampled images as described in section 4.3.2. It is highly optimized for random read-only access, which is exactly what we want.

4.5 Thread allocation

A GPU thread is allocated for every sample, therefore every pixel/sample. Based on [5] only a fraction of samples (around 1%) is still processed by the classifier after the few initial stages. On the other hand the GPU organizes threads in warps - groups of 32 threads, which are organized in blocks across with they can be synchronized. The problem is, that when only a single thread within a warp is active, the other threads have to wait for it until it finishes, thus wasting GPU resources.

4.6 Thread rearrangement

Herout in [5] proposed, that every few stages, threads can be checked if they are still evaluating the classifier stages or they have been dropped. Surviving threads can then be rearranged and the process continues only with the remaining samples.

The detector uses several ways of reorganizing surviving threads. Every one of them works better under different conditions. To discuss and measure such conditions is one of the goals of this thesis. All the implementations are summarized below using the types of functions and memory used.

- atomic functions / global memory

- atomic functions / shared memory

- atomic functions / hybrid global-shared

- prefix-sum / global memory

The whole implementation is based around three functions, which are implemented as `__device__` functions in case of the shared memory implementation, otherwise they are kernels on their own. The shared memory is specific by the fact, that it operates only within shared memory space and doesn't use global memory at all and as such, it doesn't have to be synchronized by rerunning the whole kernel, but only using `__syncthreads` calls.

- `detectSurvivorsInit` - processes all samples for a given number of stages and outputs the surviving threads. Called at the beginning.

- `detectSurvivors` - processes surviving samples for a given number of stages and outputs the surviving threads. Called multiple times.

- **detectDetections** - processes remaining surviving threads and outputs detections. Called at the end.

4.6.1 Atomic functions and global memory

The first proposed method uses a counter stored in global memory to count the number of surviving threads within a block and global memory for storing the surviving sample information. After processing a given number of stages of a classifier, every sample which wasn't discarded by the classifier is assigned an ID in global memory by reading the current value of the surviving sample counter and incrementing it. It then saves the information about the surviving sample (structure is described in figure 4.6.1) to global memory based on the assigned ID.

```
struct SurvivorData {
    uint32 x, y; // X, Y coordinates of the sample
    float response; // current response
};
```

Figure 4.6: Stage structure

Threads cannot be synchronized across the whole grid, which means, that it is not possible for all the samples to stop at a given stage and write down the surviving samples. In other words, it is possible, that a thread processing survivors after, let's say 512 samples, would write to memory at the same time, as another thread processing survivors after 256 stages, therefore a new kernel must be launched.

Advantages

- Samples are tightly packed together after synchronization, using up as little memory (and warps) as possible.

Disadvantages

- A global memory counter might become a serious bottleneck on an image with a lot of detections, because threads will get stalled waiting for the atomic increment to be processed.

- The use of global memory is generally slow, even though it can be optimized by its coalescing, explained in section 1.2.3.

- An overhead of running a new kernel.

4.6.2 Atomic functions and shared memory

To solve these issues a method exploiting the use of shared memory was proposed. All the survivors are stored in shared memory and so is the counter, which only counts surviving threads within a block. This means, that all the threads are processed locally within their block.

Functions are not implemented as kernels, like in the previous case, but only as **__device__** functions. The kernel doesn't have to be run again, in this case.

After processing a given number of stages the surviving sample is assigned an ID based on the value of the counter, which is a number ranging from 0 to max threads per block and atomically increments it. Threads are then synchronized and evaluation of the stages continues. This leads to warps being freed.

Advantages

- Shared memory is much faster than global memory (the bandwidth of shared memory is 1.7TB/s compared to that of global memory which is 150GB/s.

- Counters and atomic increments are distributed across shared memory removing the single counter bottleneck, as threads increment their counter separately based on the block they are in.

- Only a single kernel can be run with the use of `__syncthreads` to synchronize threads inside a block every given stage.

Disadvantages

- More warps are being run, because as long as there is at least a single sample being processed a warp has to be scheduled and the same happens for the whole block. As long as there is a single warp surviving inside a block, the block occupies a multiprocessor.

- An overhead of synchronizing threads, resetting the counter and checking for surviving threads is added.

4.6.3 Hybrid method using both shared and global memory

The overhead of rerunning a kernel is quite low and so a third method was proposed, which would solve the issue with having a lot of resources wasted on only a few threads occupying whole warps or blocks.

The hybrid method removes the global memory counter bottleneck and still preserves the efficiency of organizing survivors in global memory. It uses two counters, a local counter stored in shared memory, which assigns a local ID the same way as in described in section 4.6.2 and a global counter, which gets increased by the total number of its survivors. Adding to the global counter can happen only when a block is synchronized, because a final number of survivors within a block must be known. It is used as an offset to the global memory array for the block of surviving samples, whereas the local counter is used an offset within this block of memory.

Basically this method can be looked at as a compromise between the global memory implementation, explained in section 4.6.1 and the shared memory implementation explained in section 4.6.2.

Advantages

- Distributed use of counters. The maximum number of atomic increments inside a block is the size of the block and the maximum number of atomic adds to the global counter is the number of blocks.

- An efficient use of global memory and GPU resources by tightly storing the surviving samples.

Disadvantages

- Synchronization of a block is needed before the global counter can be added to, which stalls a large portion of the threads inside a block.

- An overhead of running new kernels.

4.6.4 Prefix-sum

The last proposed method tries to remove the use of a counter altogether and thus removing a possible bottleneck. A parallel prefix-sum is an algorithm commonly used for such tasks. It allows to sum up values on a parallel architecture with a logarithmic complexity.

The main idea behind is, that it produces an array of values, where every element is the sum of the previous values. For an array of 0's and 1's, where 1 is a surviving thread and 0 is a discarded thread, it not sums up all the survivors in the last element, but also assigns a unique id for every thread. This is done using shared memory, where the size of the memory is the same as the number of threads in a block.

The algorithm can be viewed as a tree and be represented as an array in shared memory. It consists of two phases an up-sweep phase and a down-sweep phase. The tree is a binary tree with the values (in our case 1's and 0's) as leaves.

In the up-sweep phase, the tree is traversed from its leaves to the root, where in every stage the values are summed up in their parent as shown in figure 4.7.
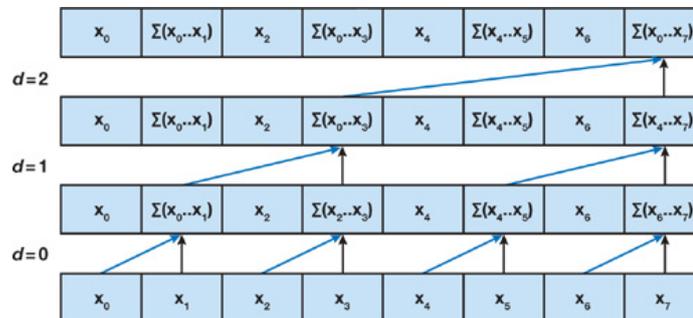
Figure 4.7: Up-sweep phase of the prefix sum algorithm (taken from [4])

At the beginning half of the threads in a block are used to calculate the corresponding sums. The next iteration the number of threads used is halved and so on, until the root is reached, in which the total sum of all the values is stored.

In the down-sweep phase as shown in figure 4.8, the root is zeroed and traversed back to the leaves. Using the partial sums a sum of all the previous values for every leaf is gained.

Every surviving thread is then assigned an ID based on its value in the prefix-sum array and the total number of surviving threads inside the block is the last value inside the array.
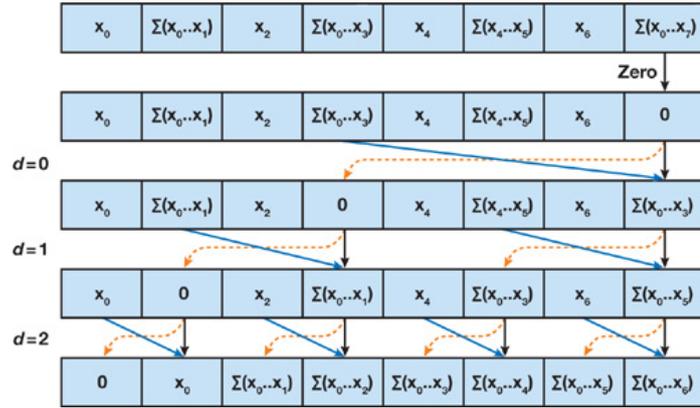
Figure 4.8: Down-sweep phase of the prefix sum algorithm (taken from [4])

4.7 Setting the thread-rearrangement stages

Thread rearrangement is a costly process, but greatly enhances the performance, therefore a balance between rearranging threads and running the detection has to be found. Every detection function call has a starting stage and an ending stage. For the initial function call, the stating stage is set to 0 and for the final stage, it's set to the total number of stages, which is in our case 2048. It is important to set these optimally, because it will have large impact on the overall performance.
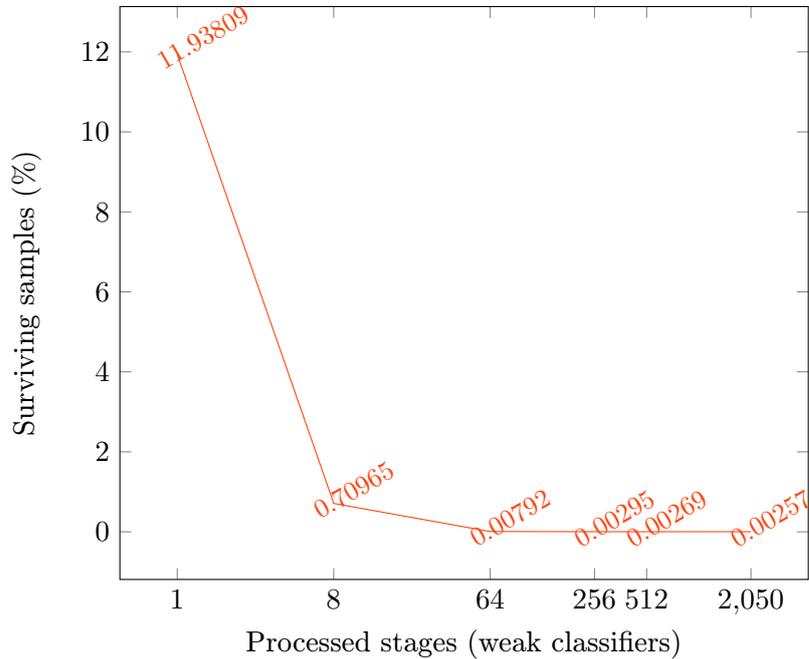

Figure 4.9: Surviving samples after processed number of stages.

The implementation allows to output the number of surviving threads after each thread rearrangement or in another words after returning from the kernel. A few experiments were conducted and a single strategy had to be devised to be able to measure the performance

34

and profile the implementations. The best results were on average obtained by the configuration 1, 8, 64, 256, 512, which are the numbers of processed stages, after with the thread rearrangement occurred.

Figure 4.9 shows the number if surviving samples after each rearrangement stage or in another words after each kernel run. It can be argued, that the last 2 rearrangement stages aren't needed, because only around 10% of the surviving samples get discarded. It's important to note in these cases, that there is a lot of stages being processed afterwards, and so every active warp affects the performance much more, than during the previous kernels.

Thread rearrangement strategies are specific to every video or dataset, therefore it can't be said, there there is only one, which is the best. They must be thoroughly measured for every use case.

Chapter 5

Performance measurements and profiling

This chapter evaluates performance, which means the time it takes for the detector to process an image. The results obtained by measurements were not only measured, but also profiled on different GPUs.

-	NVidia GeForce GTX 980	NVidia GeForce GTX 780Ti	NVidia Quadro K1000M
Architecture	Maxwell	Kepler	Kepler
SM's[1]	16	15	1
CUDA Cores[2]	2048	2880	192
Base Clock (MHz)	1126	875	850
Memory (MB)	4096	3072	2048
Memory bandwidth (GB/s)	224	336	28.8

Table 5.1: Parameters of graphics cards used for performance measurements

The technology used in the implementation is supported only by the Kepler architecture and newer (CUDA 5.0+). Because of this the high-end Kepler and Maxwell GPUs were chosen (GTX 980 for the Maxwell architecture and GTX 780Ti for the Kepler architecture) to test the limits of the implementation. Also an average notebook GPU (Quadro K1000M) was used to test the performance in more standard user conditions. Specifications of the GPUs can be found in table 5.1. The driver version used was 340.62 and for the CPU Intel(R) Core(TM) i7-3610QM CPU @ 2.30 GHz.

The detector can process both image datasets and videos. As such the following tested use cases were chosen.

- 720×480 video

- 1280×720 video

- 1920×1080 video

- BioID face dataset with 1512 grayscale images

As a video, a section of the movie „The Game" (1997) was chosen, with a larger group of people, having 10-50 faces per frame. The detector also has many configurations. One of them being the different implementations, which were discussed in section 4.6 and the second being the block sizes. For a general overview, they are summarized below.

Implementations (based on the CUDA feature they exploit)

- Global memory

- Hybrid-shared memory

- Shared memory

- Prefix-sum

- CPU

Block sizes (number of threads per block)

- 8×8 (64 threads)

- 16×16 (256 threads)

- 32×32 (1024 threads)

The most important things, that depend on the size of a block are the size of the shared memory and the number of threads, which are synchronized using the `__syncthreads` call. Only 2^N sizes of the blocks were tested, because specifically the prefix-sum implementation works only for those.

5.1 Detection time

In order for the detector to be a real-time object detector, it must process a single frame of a video in under $1/framerate$ seconds, and so as a unit of performance, detection time was measured.

5.1.1 Global and shared memory usage

The most interesting thing to note is that, for the Quadro K1000M, the fastest implementation is the global memory implementation. Originally it was thought, that the fastest implementations would be the shared, explained in section 4.6.2 or the hybrid-shared method, explained in section 4.6.3, because of a single counter in global memory, which is accessed by all the threads.

There seem to be two bottlenecks here, on being the use of a single counter, accessed by all the threads and tje second being the single multiprocessor, which in case of the shared memory implementation doesn't process the blocks fast enough, as there is many synchronization calls. This can be seen in figure 5.1, where the global memory implementation works the best for all the configurations.
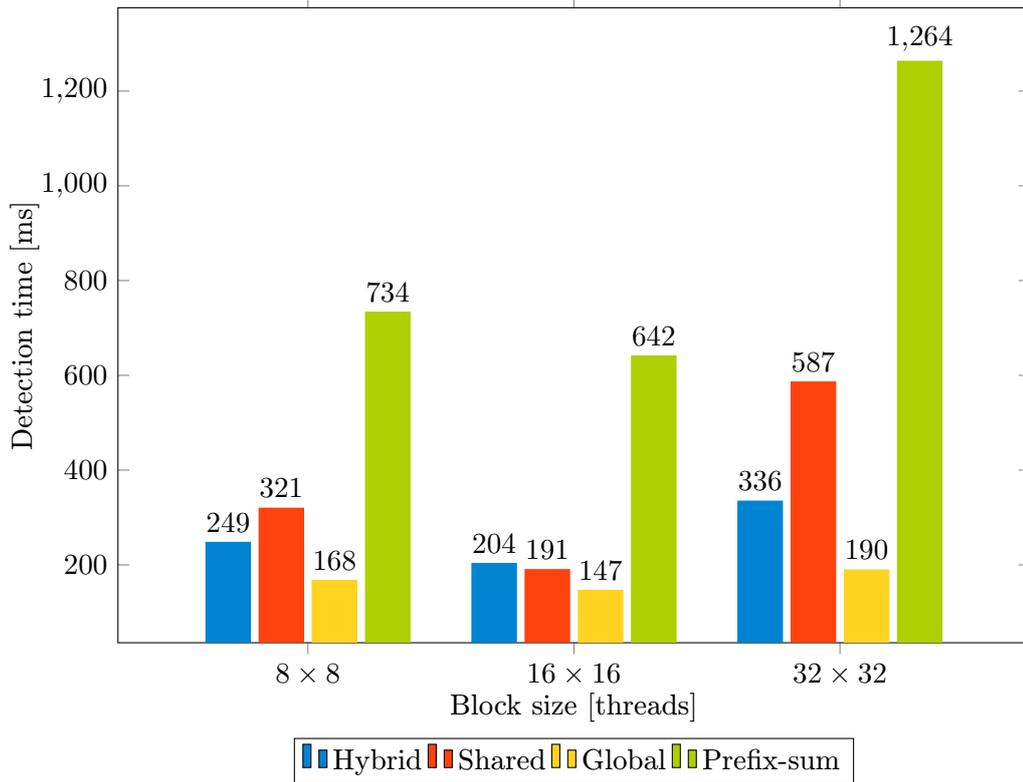
Figure 5.1: Comparison of detection times of different implementations on Quadro K1000M (Kepler)

5.1.2 Selecting the optimal block configuration

In all of the measurements in figures 5.1, 5.3 and 5.2, the best results were obtained for the 16×16 configuration, with the exception of the global memory implementation, which wasn't affected all that much by the configuration. For the Maxwell architecture, the best results were obtained by the 32×32 configuration.

The main reason being, that mainly on the high-end GPUs, the bottleneck is the counter in global memory and it doesn't depend that much on how threads are divided into blocks, but rather on usage of atomic increment for the counter and the occurring stalls.

5.1.3 Shared memory and block size

The size of the block affects the shared memory implementation the most, because the samples within a block share a counter, therefore the larger the number of threads within a block, the more stalls by the atomic increments of the counter. This would lead to the conclusion, that the smaller the block, the better.

It is however not the case, because of thread occupancy. The Kepler and Maxwell architectures have their limits of blocks per multiprocessor or warps per block, as mentioned in table 1.2. As such, even though the 8×8 configuration performs better, than the 32×32, and in terms of stalls even better, than the 16×16, it is not occupied to its maximum. Occupancy is discussed later on in section 5.2.2.
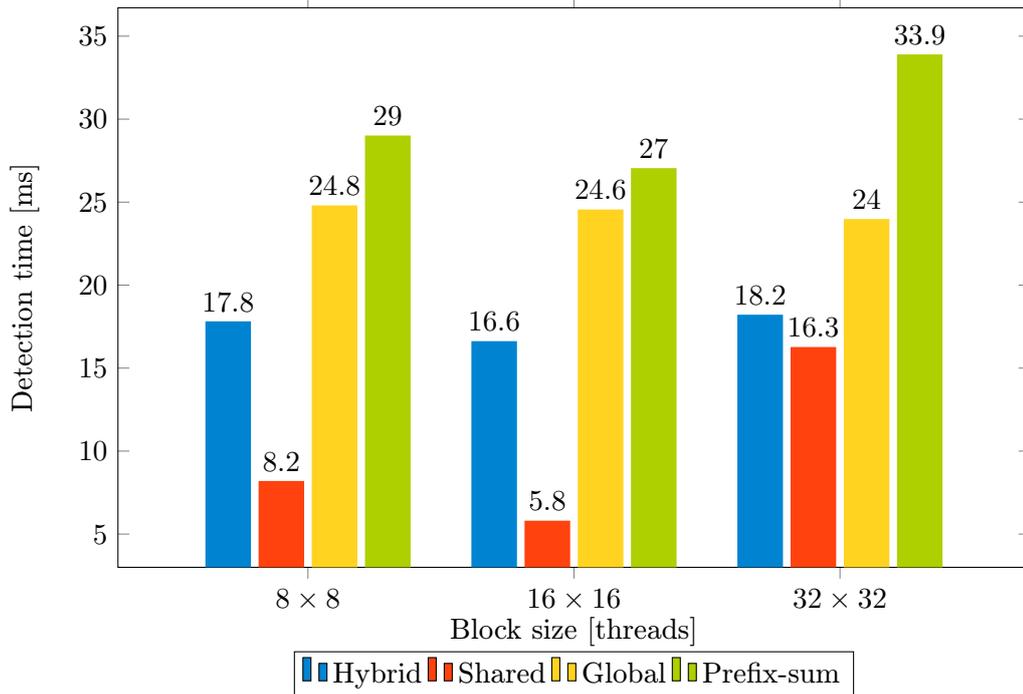
Figure 5.2: Comparison of detection times of different implementations on GeForce GTX 980 (Maxwell)

5.1.4 Hybrid-shared memory method

The hybrid-shared memory method, which was originally thought to be the most effective, didn't perform that well at the end. For the high-end GPUs, it performed worse in most cases, than the shared memory implementation. A huge difference between the two can be seen for the Maxwell architecture in figure 5.2. This seems to be because of the great improvement of shared memory atomic operations in Maxwell.

For the Kepler architecture GPUs it performed slightly worse for the GTX 780Ti and better for the Quadro K1000M, than the shared memory method. This is a result of the GTX 780Ti having 16 streaming multiprocessors, which get occupied by the larger number of blocks with the shared memory method, whereas for the Quadro K1000M, its single multiprocessor becomes the bottleneck.

5.1.5 Prefix-sum

The prefix-sum implementation performed worse in terms of detection time, than all the other implementations using atomic methods. This is due to the algorithm design, where all the threads inside a block have to be synchronized and wait for a single thread to read the thread offset. All the warps inside a block are therefore stalled until the very end, where the number of surviving threads is finally calculated.

5.1.6 Summary

In terms of the best implementation for a given GPU, the implementation was tested on high-end Kepler and Maxwell GPUs, with 16 and 15 multiprocessors and an average Kepler
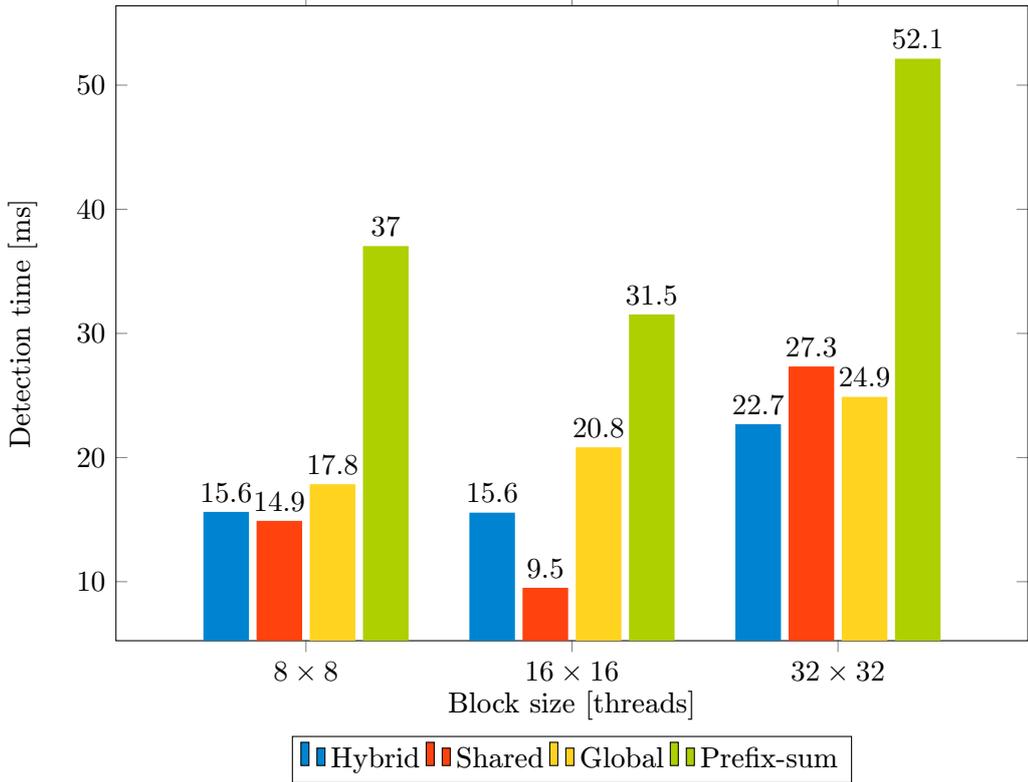
Figure 5.3: Comparison of detection times of different implementations on GeForce GTX 780Ti (Kepler)

GPU with a single multiprocessor. The proposed methods in terms of their performance are summarized below in table 5.2.

Category	Proposed method
high-end Maxwell	16×16 shared memory implementation
high-end Kepler	16×16 shared memory implementation
Average Kepler	16×16 global memory implementation

Table 5.2: Proposed implementation for every GPU category

5.2 Profiling

To further analyze the performance of all the implementations, metrics measurements introduced in section 3.4 will be discussed. Compared to other GPU technologies, CUDA allows for great profiling tools described in section 3.3, and as such a thorough analysis of the implementations can be made. The main focus will be not only on the best implementations performance-wise, and why they performed so well, but also on why some implementations performed worse, where are their problems, why are they not viable for the given implementation and how these issues can be solved.

5.2.1 L2 hit rate (texture reads) for global and shared memory implementations

GPU	L2 Hit Rate (Texture Reads) (%)
GTX 980 (Maxwell)	93.263
GTX 780Ti (Kepler)	94.37
Quadro K1000M (Kepler)	85.804

Table 5.3: L2 hit rate (texture reads). Shared memory implementation, 16×16 block size, 1920×1080 video.

Results obtained for the L2 hit rate (texture reads) for the shared memory implementation are summarized in table 5.3. These values are averaged across all the thread rearrangement stages, because the implementation only uses a single kernel. It is interesting to note, that both of the high-end GPUs, also have almost 10% bigger hit rate. First, this supports the claim, that it is better to use the shared memory implementation on high-end GPUs. Second, and more importantly, the values for the first two GPUs are close to 100%. This means, that the implementation only accesses texture memory for the initial reads and later on, the data is mostly read from the cache. For Quadro K1000M it's not as high and it's also one of the reasons another implementation should be used instead.
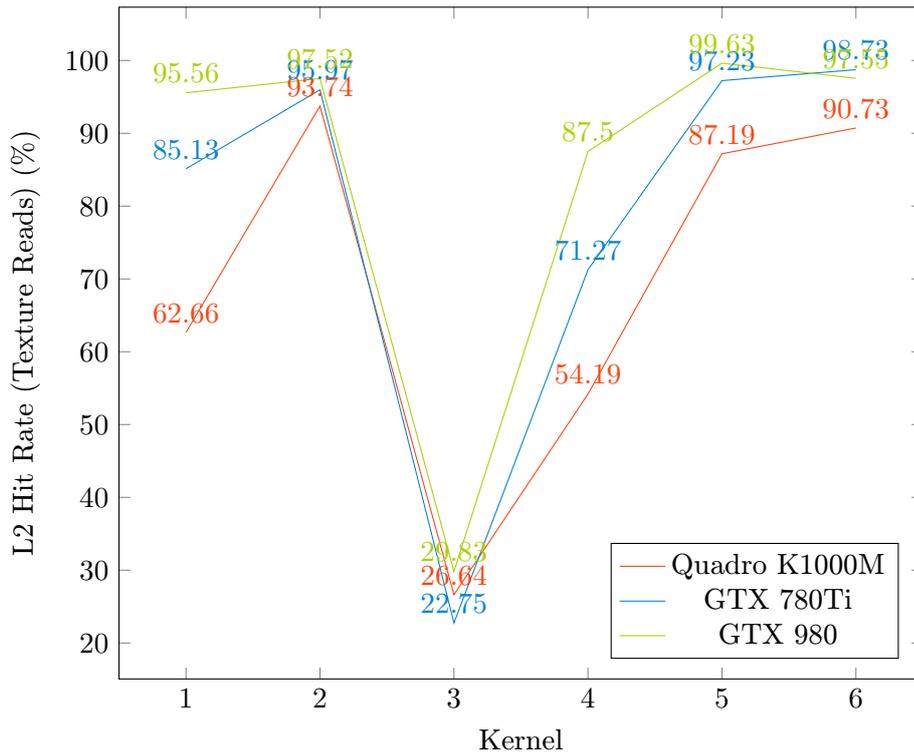

Figure 5.4: L2 Hit Rate (Texture Reads) for all three GPUs throughout the computation. Global memory method, 16×16 block size, 1920×1080 video. Values on the X-axis enumerate kernels in the order, they a run. 1: detectSurvivorsInit, 2-5: detectSurvivors, 6: detectDetections.

The hybrid-shared method and the global memory method both use global memory to

reorganize the surviving samples and after each rearrangement stage, they run a new kernel. Thread rearrangement stages used were 1, 8, 64, 256, 512, which means that the first kernel ran only for the first stage (weak classifier), the second processed all stages from the 2nd until the 8th and so on. Figure 5.4 shows L2 hit rate for the global memory method. The hybrid-shared implementation uses the same way of reorganizing the threads, so the results are very similar. The critical part to note is during the 3rd kernel (stages 8-64), where the hit rate significantly decreases, to around 20-30% for all the GPUs and then it starts to increase again.

The sudden decrease in hit rate can be explained by the behavior of the detector. The first kernel utilizes the cache very well on Maxwell and a bit worse on Kepler, because it has 4× bigger cache size. The kernel only runs for a single stage of the detector. After this only 10-15% of the samples survive, as seen in figure 4.9. Most of the surviving samples are in clustered areas, where data locality is utilized very well and so the texture reads are still cached. On the other hand, during the 3rd kernel, which runs after processing 8 stages, there is a huge decrease in hit rate, to around 20-30%. This is because after the 2nd thread rearrangement only around 1% of the samples survive, most of them being scattered all over the pyramidal image and not being clustered anymore. These samples don't utilize the data locality very well and the hit rate is very low. It improves later on, when only a few candidate samples remain, and so a lot of them simply fall in the cache. This is shown in figure 4.9, that after processing 64 stages, 30% of the processed samples survive until the end. After 256 stages, it's almost 90%. These values correlate with the increase in hit rate.

There are two proposed improvements in order to improve L2 cache hit rate. The first being adding more thread rearrangement stages, and therefore kernels. Experiments with this lead to a decrease in performance, because adding kernels leads to an additional overhead of loading, storing a reorganizing surviving samples. Second improvement can be made by setting the rearrangement stages differently. This is specific to the videos or datasets being used and should be statistically measured for the given case. The current implementation enables to output the number of surviving samples to conduct such experiments.

5.2.2 Achieved occupancy

Global memory implementation

There are several things to note in figure 5.5, which shows the achieved occupancy for Quadro K1000M and global memory implementation. The first is the low occupancy of the 8 × 8 configuration, which varies between 30.9% and 46.5%. This can actually be predicted from parameters mentioned in table 1.2, where the maximum number of blocks per streaming multiprocessor for compute capability 3.0 is 16, but the maximum number of threads is 2048. This means, that to reach full or almost full occupancy, at least 128 threads per block are needed. In case of the 8 × 8 configuration, there are only 64 threads and so only half of the maximum occupancy can be reached.

Another fact to note is that for the last 3 kernels the occupancy is lower for all 3 configurations anyway, because the number of surviving samples is lower than the maximum occupancy and so this doesn't matter much anyway. Lower occupancy doesn't always mean worse performance. In the case of the global memory implementation, even though the achieved occupancy of the 8 × 8 configuration is twice smaller than that of the 32 × 32 configuration, it still performs much better as shown in figure 5.1

The significant downside of this implementation is the low occupancy for the last 3
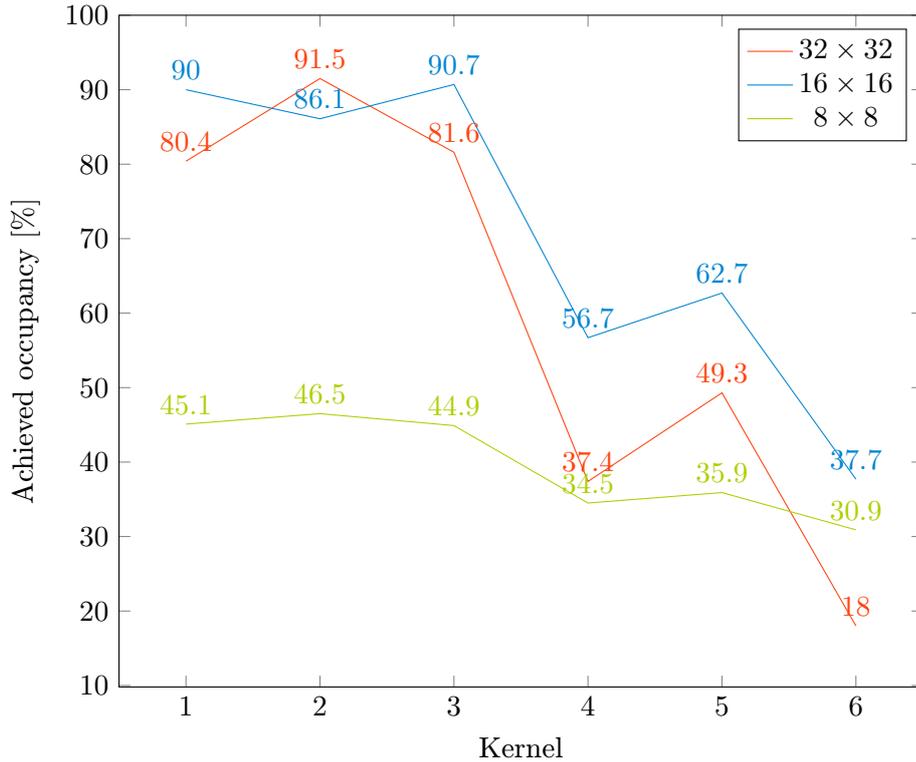
Figure 5.5: Achieved occupancy on Quadro K1000M (Kepler) for the global memory method. Values on the X-axis enumerate kernels in the order, they are run. 1: detect-SurvivorsInit, 2-5: detectSurvivors, 6: detectDetections.

kernels. When we look at figure 4.9, this means stages 64 to 2048. For better GPUs, the global memory method is not viable as shown in figures 5.3 and 5.2 and was profiled only for the 16×16 block configuration. The achieved occupancy for these cards gets even lower.

A proposed improvement to this issue would be to implement a method to process the future stages using the unused resources and utilize them as a cache in a way, that the surviving samples would use the preprocessed stages instead of processing them on their own.

Shared memory implementation

GPU	L2 Achieved occupancy [%]
GTX 980 (Maxwell)	62.1
GTX 780Ti (Kepler)	46.6
Quadro K1000M (Kepler)	47.5

Table 5.4: Achieved occupancy. Shared memory implementation, 16×16 block size, 1920×1080 video.

Achieved occupancy for the shared memory implementation, summarized in the table 5.4, is generally low. Thread rearrangement in this case works similarly to the global or hybrid-shared methods, only it utilizes a single kernel and synchronization across blocks.

This means, that even when the overall number of samples decreases, as long as there is a single thread within a warp running, that warp would stay active, and so there should be a slight increase in the number of active warps.

An increase in occupancy can be seen for the Maxwell architecture. This is a result of a change mentioned in section 1.2.5, that Maxwell can have more active warps per multiprocessor (32 compared to 16 for Kepler), which helps in cases like this, where blocks with a small number of threads are run.

As mentioned previously, at the end of section 5.2.2, a proposed improvement would be to process future stages and utilize GPU in a better way.

5.2.3 Global memory store, load and atomic throughput

Global memory implementation

The load and store global memory throughput and the atomic throughput all correlate with each other, because the main bottleneck here is actually the use of atomic operations before the global memory stores are processed, and so the store throughput is only as high as the atomic throughput.
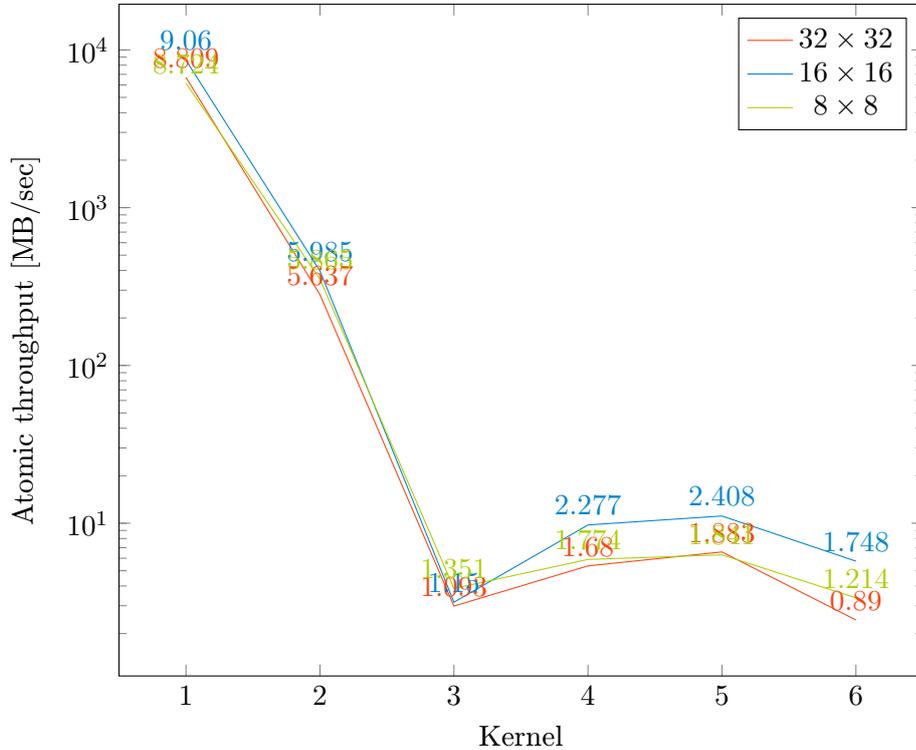
Figure 5.6: Atomic throughput on Quadro K1000M (Kepler) for the global memory method. Values on the X-axis enumerate kernels in the order, they are run. 1: detectSurvivorsInit, 2-5: detectSurvivors, 6: detectDetections.

The 8×8 configuration has a lower overall achieved occupancy as shown in figure 5.5 and the 32×32 on the other hand creates more conflicts when issuing atomic operations, this leads to the 16×16 being the best compromise between these three configurations. This is also shown in figure 5.5. The low atomic throughput for the kernels 4-6 is the result

of a low number of samples and thus lower atomic operation. For kernel 3 it's also a result of longer detection time due to lower L2 hit rate mentioned in section 5.2.1.

The atomic operation bottleneck is also one of the reasons, why the global memory implementation cannot be used with high-end GPUs. Their increased computational capacity allows them to process more threads at a time, which results in more stalling caused by waiting for atomic operations and so this resource advantage isn't used to its maximum.

Shared memory atomic throughput

Maxwell architecture doesn't allow to measure atomic throughput and so L2 throughput for atomic requests was measured. As mentioned in section 1.2.5, Maxwell architecture implements native shared memory atomic operations compared to the previous architectures (Kepler and Fermi), where shared memory atomic operations were implemented in a software fashion using lock-modify-unlock pattern.
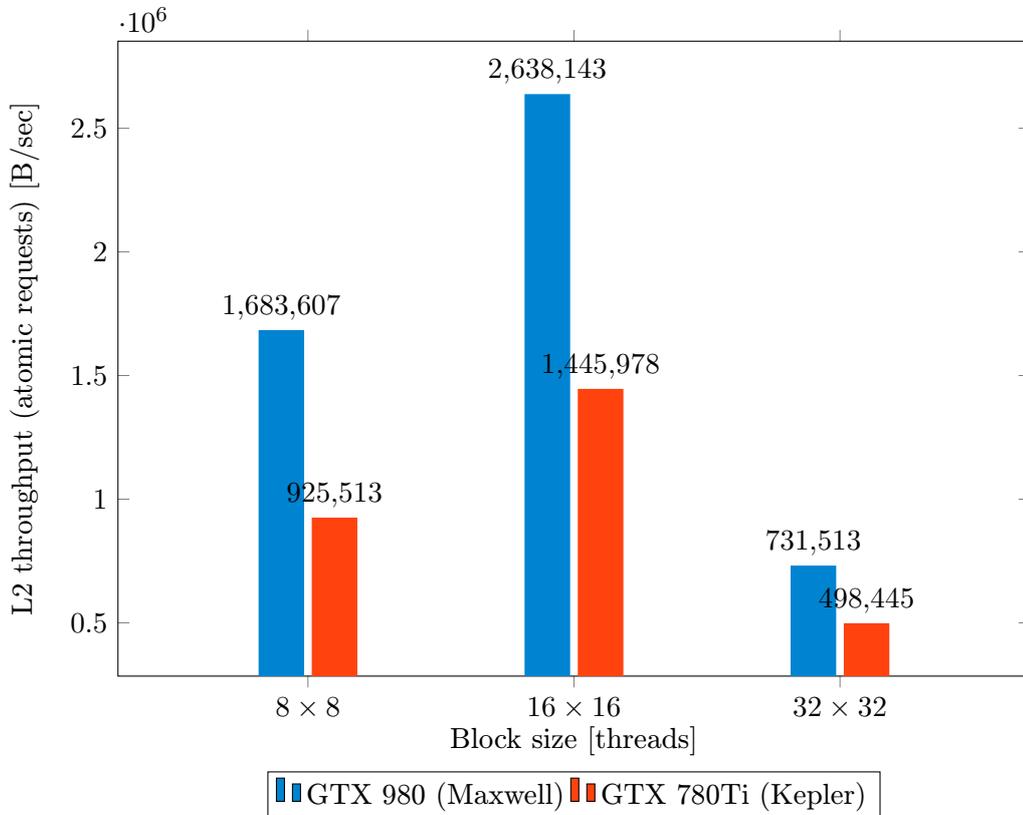
Figure 5.7: L2 throughput for atomic requests.

The shared implementation relies heavily on the use of shared memory atomic instructions to update a single location in memory, which turned out to be very expensive in case of the Kepler/Fermi architecture, but in case of the Maxwell architecture, it emphasized its advantages. The results in figure 5.7 show, that the performance on GTX 980 is almost 2×, than that of GTX 780Ti for L2 atomic throughput.

5.3 Summary of proposed improvements

Throughout the thesis different methods to the problem of object detection on GPU were implemented, measured and finally profiled. Every implementation always came with some conclusions about its usability, possible problems and raised ideas how to handle these issues.

Measurements and profiling on more GPUs. It was clear, that for the high-end GPUs the best results were obtained by the shared memory implementation and for the opposite extreme - a single multiprocessor GPU, the best results were measured for the global memory implementation. The hybrid-shared memory method didn't perform all that well, but it performs better on single multiprocessor GPUs, than the shared memory implementation and better on high end GPUs, than the global memory implementation. Based on this, it can be predicted, that it might perform better than both of these on more average GPUs, such as GTX 960, which has 8 streaming multiprocessors.

Exploit parallelism. Even on a GPU with a single streaming multiprocessor, the occupancy decreased below 50% for most of the stages and GPU resources were left unused. These resources might be used to process future stages and use them in a similar fashion as a cache. A way of mapping such stages, which were preprocessed to the ones being processed can be discussed.

Prefix-sum. The prefix-sum method should be researched more, if it can be optimized to discard unused threads throughout the computation.

Dataset initialization overhead. When processing a dataset, GPU memory is allocated and freed specifically for every image. This brings a lot of overhead to the whole process, an overhead almost as large as the detection itself. A single texture can be allocated to copy the images to. The size of such has to be discussed, as it will stay the same throughout processing the whole dataset.

Minimize size of the block for shared memory implementation. We found out, that even though 16×16 configuration performed the best, 8×8 configuration had good results, even though it had very low occupancy. Based on the architecture, a strategy can be devised for the size of the block, to be as small as possible and still fully occupy the GPU.

Pyramidal image. Pyramidal image doesn't have to be used. A 1D texture containing pixel values can be used instead, which enables much smaller textures. On the other hand, this approach doesn't allow for hardware accelerated bilinear interpolation and doesn't utilize L2 cache as much for spatial data locality.

Chapter 6

Conclusion

The goal of this thesis was to design a high-performance application or a library for object detection on the GPU. The topic of WaldBoost object detection on the GPU using LBP is well researched, on the other hand there is much less research done on the implementations themselves, and so the main focus was to research how different options of the NVidia CUDA Toolkit affect object detection.

Different approaches to implement the detector were proposed described in section 4.6, concentrating on the use of specific features of the GPU programming. Every approach had different bottlenecks and was thoroughly measured and profiled in chapter 5. Based on this research a framework, called WBD, was implemented, which allows to run a specific implementation on a large range of devices, based on their parameters. The framework itself has 4 CUDA C implementations, which are optimized for specific GPUs and a simple C++ implementation to be run on non-CUDA devices.

The detector[1] can process both image datasets, independent on the image size and videos, outputting detections as regions of interest inside an image. As such, it doesn't have to be used on its own, but can also be used as a part of another application, which uses object detection to greater benefits.

The two main contributions of this thesis are, first, the implementation itself, which doesn't have to be used on its own, but can also be used as a part of another application, which uses object detection to greater benefits. Second, new ways of looking at GPU implementations of WaldBoost detectors, which can be further optimized and bring even better results.

---

[1]Latest version is available at: https://github.com/mmaci/vutbr-fit-waldboost-detector

Bibliography

[1] Andrew V. Adinetz. First experiences with maxwell gpus, 2014.

[2] M. Clark. Cuda pro tip: Kepler texture objects improve performance and flexibility, 2013.

[3] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.

[4] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[5] 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.

[6] Adam Herout, Pavel Zemčík, Roman Juránek, and Michal Hradiš. Implementation of the „local rank differences“ image feature using simd instructions of cpu. In *Proceedings of Sixth Indian Conference on Computer Vision, Graphics and Image Processing*, page 9. IEEE Computer Society, 2008.

[7] Michal Hradiš, Adam Herout, and Pavel Zemčík. Local rank patterns - novel features for rapid object detection. In *Proceedings of International Conference on Computer Vision and Graphics 2008*, number 12 in Lecture Notes in Computer Science, pages 1–12. Springer Verlag, 2008.

[8] Ming hsuan Yang, David J. Kriegman, and Narendra Ahuja. Detecting faces in images: A survey. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 24(1), 2002.

[9] Roman Juránek. Detection of dogs in video using statistical classifiers. In *Proceedings of International Conference on Computer Vision and Graphics 2008*, Lecture Notes in Computer Science, page 11. Springer Verlag, 2008.

[10] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro*, 28(2):39–55, 2008.

[11] J. Luitjen and S. Rennich. Cuda warps and occupancy, gpu tech conf. gtc express 2011.

[12] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi, whitepaper, 2009.

[13] NVIDIA. Nvidia's next generation compute architecture: Kepler gk110, whitepaper, 2012.

[14] NVIDIA. Nvidia geforce gtx 980, whitepaper, 2014.

[15] CUDA Nvidia. Cuda c programming guide. *Cuda Toolkit*, 2015.

[16] Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. Multiresolution grayscale and rotation invariant texture classification with local binary patterns. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 24(7):2002, 2002.

[17] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. *Computer graphics forum*, 26(1):80–113, 2007.

[18] J. Sochman and J. Matas. Adaboost with totally corrective updates for fast face detection. In *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on*, pages 445–450, May 2004.

[19] Jan Sochman and Jiri Matas. Waldboost - learning for time constrained sequential detection. In *CVPR (2)*, pages 150–156. IEEE Computer Society, 2005.

[20] Jakub Sochor. Fully automated real-time vehicles detection and tracking with lanes analysis. In *Proceedings of CESCG 2014*, pages 59–66. TU Vienna, 2014.

[21] J. Verscheldem. Memory coalescing techniques, 2014.

[22] Paul Viola. Feature-based recognition of objects. In *in Proceedings of the AAAI Fall Symposium Series: Machine Learning in Computer Vision*, pages 60–64, 1993.

[23] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features, 2001.

[24] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010.

[25] C. Woolley. Gpu optimization fundamentals, 2013.

[26] Pavel Zemčík, Roman Juránek, Martin Musil, Petr Musil, and Michal Hradiš. High performance architecture for object detection in streamed videos. In *Proceedings of FPL 2013*, pages 1–4. IEEE Circuits and Systems Society, 2013.

Appendix A

DVD contents

Below, the structure of the DVD is described. Before using the binary, please read the README.md file.

bin   Win32 binary including sample videos and images.

docs/doxygen   Doxygen generated documentation in HTML and pdf.

docs/poster   A poster for presentation.

docs/thesis   The thesis itself in TeX and PDF.

measurements   Measurements taken throughout the work on the thesis.

src   Source codes with a CMake file.

README.md   A README file containing information about building a binary or usage of the command-line tool.

LICENSE.md   MIT License.

Appendix B

Command-line tool description

-D –dataset [filename]   Input textfile containing dataset filenames.

-V –video [filename]   Input video.

-c –csv   CSV output.

-v –verbose   Verbose output.

-t –timer   Measure detection time.

-o –visualoutput   Shows a visual output.

-s –survivors   Outputs surviving threads every stage of the classifier.

-d –visualdebug   Shows a visual output including pyramidal and preprocessed image.

-b –blocksize [8/16/32]   Sets block size. Should be set to 8, 16 or 32 for (64, 256 or 1024 threads)

-l –limitframes [number]   Process only a given number of frames.

-m –detectionmode [global/shared/prefixsum/hybrid/cpu]   Sets the specific implementation to use.