VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

OBJECT DETECTION ON GPU

MASTER'S THESIS
MASTER'S THESIS

AUTOR PRÁCE Bc. PAVEL MACENAUER
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DETEKCE OBJEKTŮ NA GPU

OBJECT DETECTION ON GPU

MASTER'S THESIS
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. PAVEL MACENAUER

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

BRNO 2015

Abstrakt

Tato práce se zabývá detekcí objektů pomocí grafických procesorů. Jako její součást byl navržen a naimplementován systém pro detekci objektů na technologii NVIDIA CUDA, umožňující detekovat objekty ve videu v reálném čase. Jejím přínosem je prozkoumání aktuálních možností NVIDIA CUDA a stávajících grafických karet k akceleraci detekce a navržení způsobů jak dále tyto výpočty akcelerovat pomocí paralelních algoritmů.

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. 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.

Klíčová slova

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

Keywords

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

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.

. . . .
Pavel Macenauer
January 16, 2015

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 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) using the data-parallel processing powers of the GPU.

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 traditional approach 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 enables to do the calculations many times faster due to the architecture of the GPU. As shown on 1.1 many more transistors are dedicated to data processing instead of cache or control, which leads to a higher memory bandwidth.

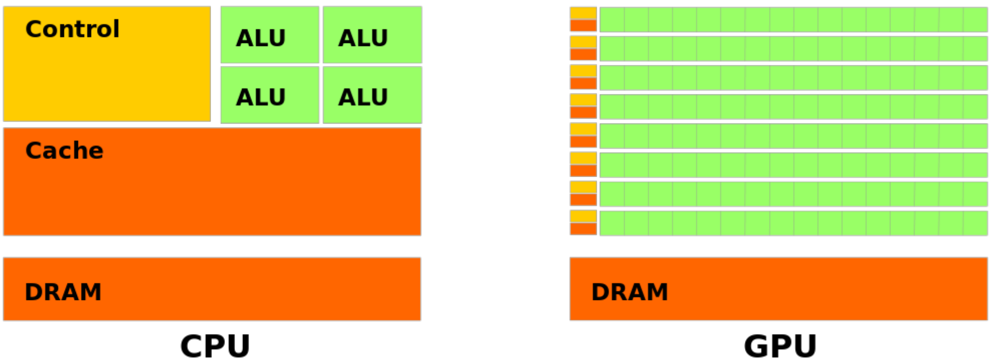

Figure 1.1: CPU and GPU architecture comparison ([1])

GPUs are also designed with demand for floating-point capabilities in mind, which can be taken advantage of in applications such as object detection, where most of the math is done in single-point arithmetic.

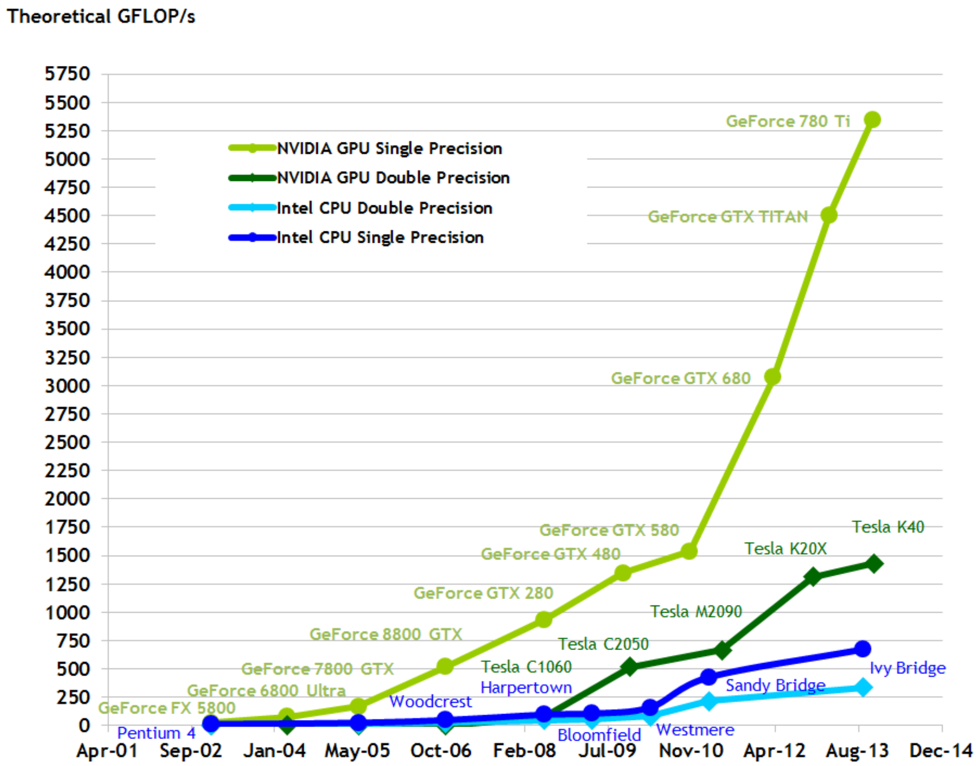

Figure 1.2: Floating-Point operations per second for the CPU and GPU ([1])

1.1 Parallel computing platforms

In November 2006 the first parallel computing platform - CUDA (Compute Unified Device Architecture) was introduced by NVIDIA. Since then several others were created by other vendors:

- 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
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 also quite differently from C++ AMP. They are not a part of the rendering pipeline, but can be set to be executed among other OpenGL or DirectX shaders. The difference between compute shaders and other shaders is, that they don't have specified input or output. These must be specified by the programmer. Theoretically it is then possible to write the whole rendering pipeline using compute shaders only.

1.2 NVIDIA CUDA

NVIDIA CUDA is a programming model enabling direct access to the instruction set and memory of NVIDIA GPUs.

1.2.1 Programming model

CUDA C extends 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 executed using a given configuration wrapped in `<<< ... >>>`. The configuration is called a grid and takes as parameters the number of blocks and the number of threads. The same kernel code is run by the whole grid. Also code run by the kernel is called to device code, where as the code run outside of the kernel is called the host code.

Threads are a basic computational unit identified by a 3-dimensional id `threadIdx`, which is typically used to index arrays.

Blocks are groups of threads, where each block resides on a single processor core, therefore a kernel can be run with the maximum of 1024 threads.

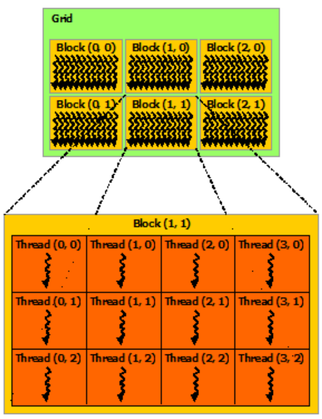

Figure 1.3: A grid of blocks and threads run by a kernel ([1])

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.

```
// 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 ([1])

1.2.2 Memory model

CUDA threads may access the following types of memories:

Global memory is accessible by all threads in a grid and allows read-write. 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 32B, 64B, 128B transactions.

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.

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: Memory types

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.

Shared memory can be accessed by all threads within a block. It is much faster than the other types, but is subject to bank conflicts.

Unified memory is a memory type introduced in CUDA 6.0. It enables to use the same memory addresses both in host and device code, which simplifies writing code. On the other as of spring 2014, there doesn't seem to be any hardware support [2] and 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.

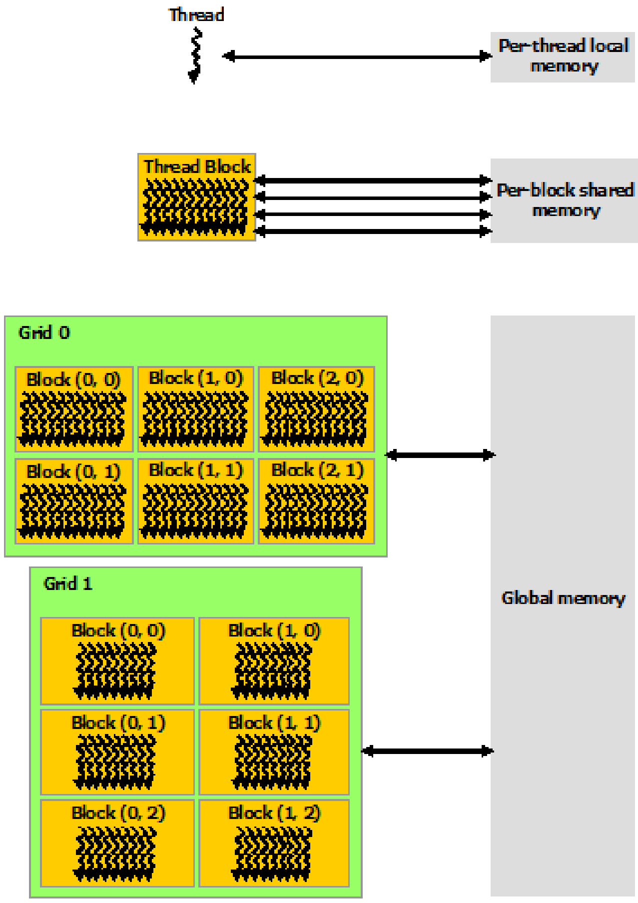

Figure 1.5: Memory hierarchy ([1])

Chapter 2

Object detection

2.1 Introduction

Object detection is a computer technology with the capability of localizing an object in input image data. The type of object depends on which data the detector was trained. Typical applications are human faces, pedestrians, cars, traffic signs and others.

Detector used by the implementation is a frontal-face human detector, which tries to identify a human face within an image. The implementation is therefore optimized for human faces, which means, that the software can be used with other detectors, but it might effect its performance due to specific optimizations. Combined with the capabilities of a GPU, the aim is to produce an object detector capable of real-time object detection on videos.

2.2 Features

There are several methods how to access the topic of object detection. In the following sections we will discuss feature-based object detection.

Let's take a frontal human face as an example. Despite the differences such as lighting, color of eyes or skin, the length of hair, we as humans, can identify we are looking at a human face based on similarities, for example - a pair of eyes, a nose, a pair of ears and so on. These similarities can be called features, but to a computer, they are still too abstract and cannot be enumerated.

2.2.1 Local Binary Pattern

One of the feature methods to describe an image are local binary patterns (LBP). They are based on encoding local intensities of an image with 8-bit codes. In their elementary form they take a 3x3 area as an input and compare intensity values of all the pixels with the central one.

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

LBP value is then evaluated as follows:

$$lbp(p_{middle}) = \sum_{i=0}^{7} 2^i compare(p_{middle}, p_i) \tag{2.1}$$

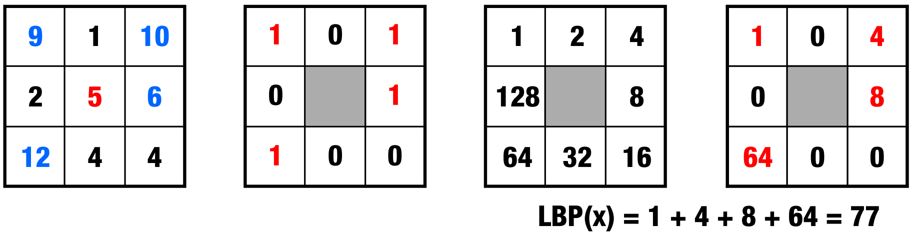

Figure 2.1: LBP feature

LBPs can be extended to be used not only for single pixels and thus 3x3 areas, but larger for larger ones. For example, when you compare 2x2 areas instead of single pixels, you compare the sum of a middle 2x2 area with the surrounding 2x2 areas.

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

2.3 Waldboost

Only one feature to describe a face is not enough, so a meta-algorithm to process a series of such weak classifiers is needed.

One such algorithm is WaldBoost, 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.

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

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:

$$R_m = \frac{p(x_1, ..., x_m | y = -1)}{p(x_1, ..., x_m | y = +1)} \tag{2.2}$$

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

As mentioned in [3] 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

$$A = \frac{1 - 0}{\alpha} = \frac{1}{\alpha}, B = \frac{0}{1 - \alpha} = 0 \tag{2.4}$$

and the whole strategy to

$$SPRT = \begin{cases} +1 & \text{if } R_m \leq 0 \\ -1 & \text{if } R_m \geq \frac{1}{\alpha} \\ \# & \text{else take another measurement} \end{cases}$$

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.

Chapter 3

Implementation

Application is implemented in `C++` with dependencies on OpenCV - an open source computer vision library with `C` and `C++` interfaces and CUDA - a library for writing NVIDIA GPU code with a CUDA C interface, which is an extension to C. OpenCV is used to load and process separate video frames. There are also many libraries based on CUDA, which might be used in the future to enhance performance.

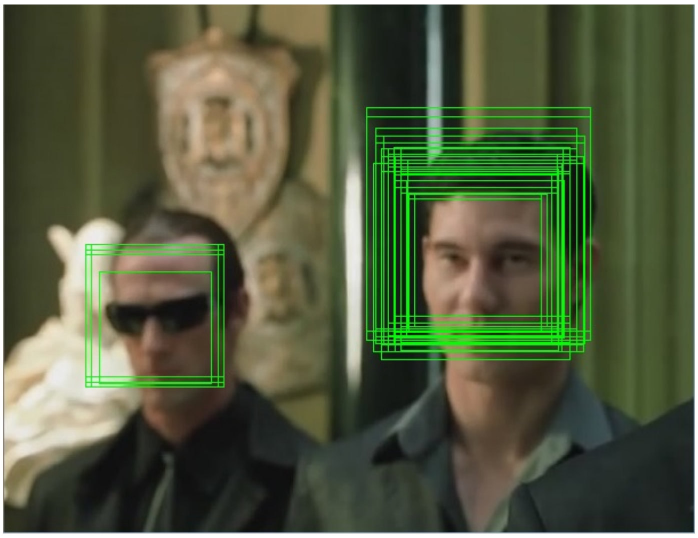

Figure 3.1: A sample output

3.1 Program structure

The basic outline of the application pipeline can be described by 3.2.

3.1.1 CUDA initialization

In this phase all the constants and the detector itself are copied to the GPU. Constants account for data like image width and height, classifier width and height, α count, stage

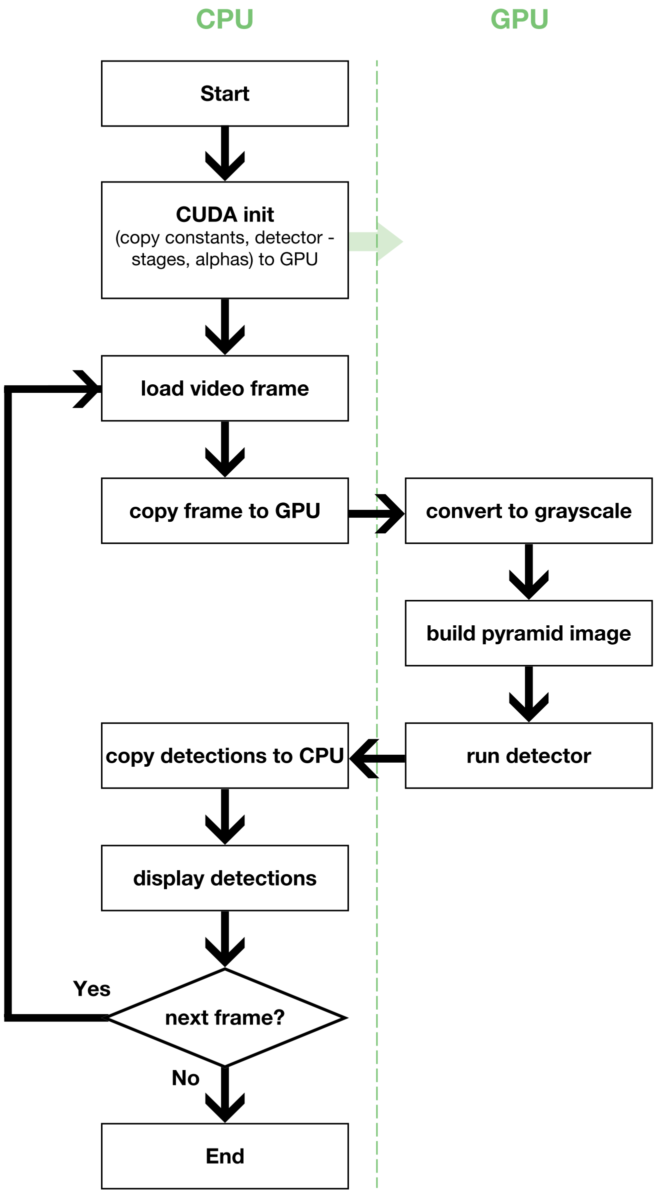

Figure 3.2: Application pipeline

count and so on. The α coefficients and stages of the detector as described in 2.3 are stored in separate header files generated from an XML file.

3.1.2 Kernels

After loading a video frame and copying the frame to the GPU there are 3 types of kernels to be run.

- Grayscale conversion
- Pyramidal image build
- Detection

Grayscale conversion kernel

Conversion to grayscale is a simple image processing operation described by the formila (3.1). The detector itself is trained on grayscale images, and so the input must also be in grayscale. After the kernel finishes, the result is converted to a texture.

$$Y = 0.2126R + 0.7152G + 0.0722B \tag{3.1}$$

Pyramidal image kernel

The pyramidal image is created from the grayscale image saved in texture memory. All the features are processed inside a 26x26 pixel-wide window. The size again depends on how the detector is trained. The basic idea is that all the features inside this window somehow describe the object, so we have to create a sub-sampled image, where the object is similar or the same size as the scanning window.

As mentioned in **??** the grayscale image is stored as a texture. The advantage of this is, that it enables us to use bilinear interpolation at no performance cost. It has to be kept in mind though, that bilinear interpolation has some negative side-effects, one of them being, that sub-sampling an image below half its original size can cause errors.

Figure 3.3: Pyramidal image

While the previous kernels can be viewed as preprocessing, the core of the application is the detector kernel.

Detection kernel

The detector consists of several parts, which will also be discussed later in 3.2 and how they are stored in the GPU memory.

- α coefficient table
- stages
- final threshold

Before we discuss detection processing, let's first make clear what we mean by a stage. As mentioned in 2.3 a WaldBoost algorithm processes a sample sequentially and in our case determines it is an object we are looking for only after processing all the measurements or also called stages. Every stage the algorithm processes, a response is given and added the accumulated response. If it rises above a given threshold, the sample is discarded.

Such stage is described by the following structure:

```
struct Stage {
    uint8 x, y;
    uint8 width, height;
    float thetaB;
    uint32 alphaOffset;
};
```

Figure 3.4: Stage structure

The detector uses a 26x26 pixel-wide window, where `x` and `y` are offsets inside this window and `width` and `height` describe the size of the feature. Using this data, the LBP coefficient is calculated resulting in an 8-bit code. This code is then added to the `alphaOffset`, which points to a specific α, which is then added to the accumulated response. The accumulated response is then compared with `thetaB` whether to discard the sample or continue processing. This can be summarized by the algorithm 1.

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

3.2 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 1.2.2.

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

- **Stages** - 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 **coefficients** - texture memory
  Texture memory not only has read-only properties, but also there are 256 coefficients for every stage. Every coefficient is stored as a float, which leads to $256 * 2048 * 4 = 2MB$ and by far exceeds the memory available for constant memory. Also the access is random, because we are likely to get different LBPs for every pixel.

- **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, which is a feature of the texture memory. Another feature is its read-only access.

Chapter 4

Results

4.1 Summary

As of January 16, 2015 the detector contains a working GPU and CPU implementations. The CPU version is available for comparison measurements and the GPU version is unoptimized with only a few GPU acceleration features. The latest version is available at: https://github.com/mmaci/vutbr-fit-object-detection.

- Memory usage is likely to stay similar, as there aren't many more viable options. The only other option is unified memory in CUDA 6.0, which seems to be more of a programmer convenient, than performance feature and shared memory, which might be used for local optimizations.
- Bilinear interpolation using texture memory is used for image down-sampling, instead of a software implementation.
- LBP for 2x1, 1x2 and 2x2 features is calculated using texture memory bilinear interpolation, which leads to a speed-up due to the fact, that sum of the intensity values isn't needed and an average is used instead.

4.2 Future work

Some of the ideas and key features yet to be implemented are:

- It is generally known, that most of the samples get discarded by the WaldBoost algorithm at the beginning as background. This leads to a large number of threads waiting for the few ones, that still compute. A measurement has to be taken to statistically determine the waiting-thread count and rearrange threads in a way to increase the percentage of running threads.
- Other methods of interpolation, such as Lanczos interpolation should be explored and measured compared to the current bilinear interpolation.
- The success rate and performance of the detector is also highly dependent on the pyramid image build, therefore other ways to build an optimized pyramid should be explored or if mipmaps can be used instead and thus the whole software based interpolation omitted.

- The CPU version should exactly match the algorithm used for the GPU version and also be optimized to provide a valid comparison.

Bibliography

[1] http://docs.nvidia.com/cuda/cuda-c-programming-guide. Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[2] http://www.acceleware.com/blog/nvidia-cuda-60-unified-memory-performance. Nvidia cuda 6.0 unified memory performance. `http://www.acceleware.com/blog/nvidia-cuda-60-unified-memory-performance`.

[3] Sochman J. ; Matas J. *WaldBoost - learning for time constrained sequential detection*. IEEE, 2005. ISBN 0-7695-2372-2.

[4] Zemcik P. ; Juranek R. ; Musil P. ; Musil M. ; Hradis M. *High performance architecture for object detection in streamed videos*. IEEE, 2013.

[5] Herout A. ; Josth R. ; Juranek R. ; Havel J. ; Hradis M. ; Zemcik P. *Real-time object detection on CUDA*. Springer-Verlag, 2011. ISSN 1861-8219.