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 Bc. PAVEL MACENAUER
AUTHOR

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

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
April 22, 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.

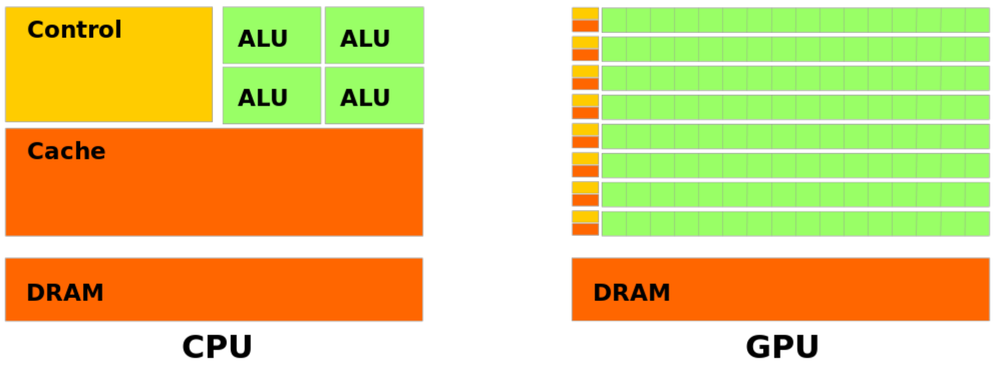


Figure 1.1: CPU and GPU architecture comparison ([1])

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.

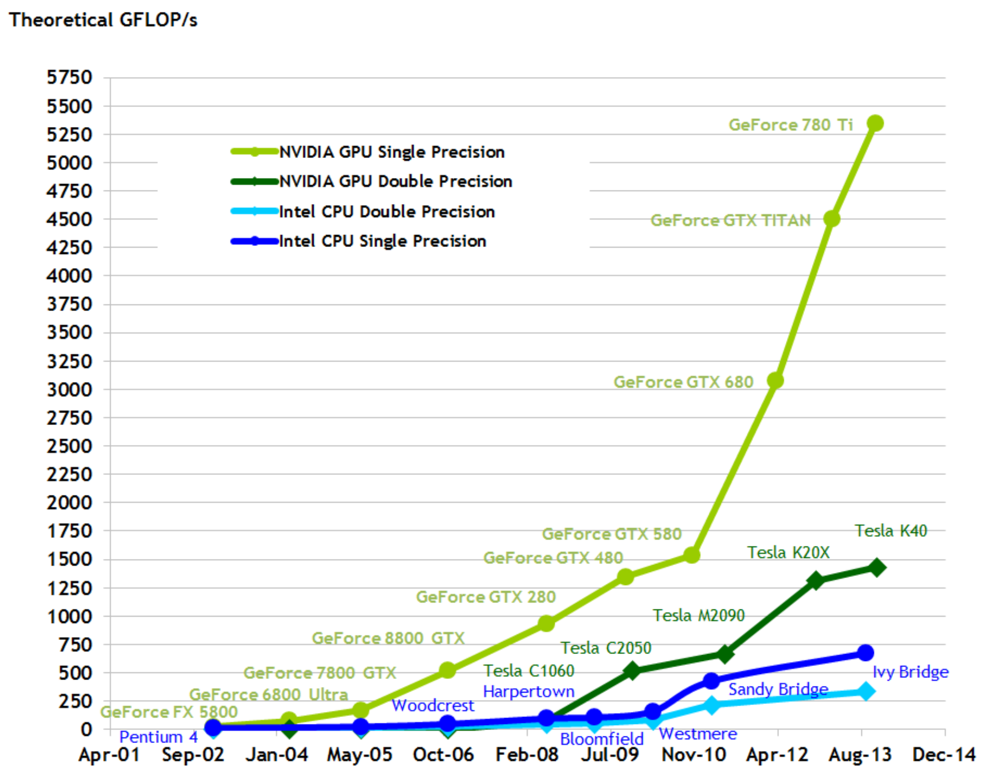


Figure 1.2: Floating-Point operations per second for the CPU and GPU ([1])

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 is 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 per block. The same kernel code is run by the whole grid. Code run by the kernel is called to device code, whereas the code run outside of the kernel is called the host code.

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 be synchronized only by running a new kernel.

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.

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`.

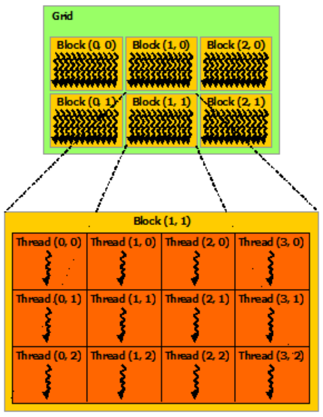


Figure 1.3: A grid of blocks and threads run by a kernel ([1])

```
// 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])

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

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:

Global memory is accessible by all threads in a grid and allows both read and 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.

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 commonly subjected 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 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.

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

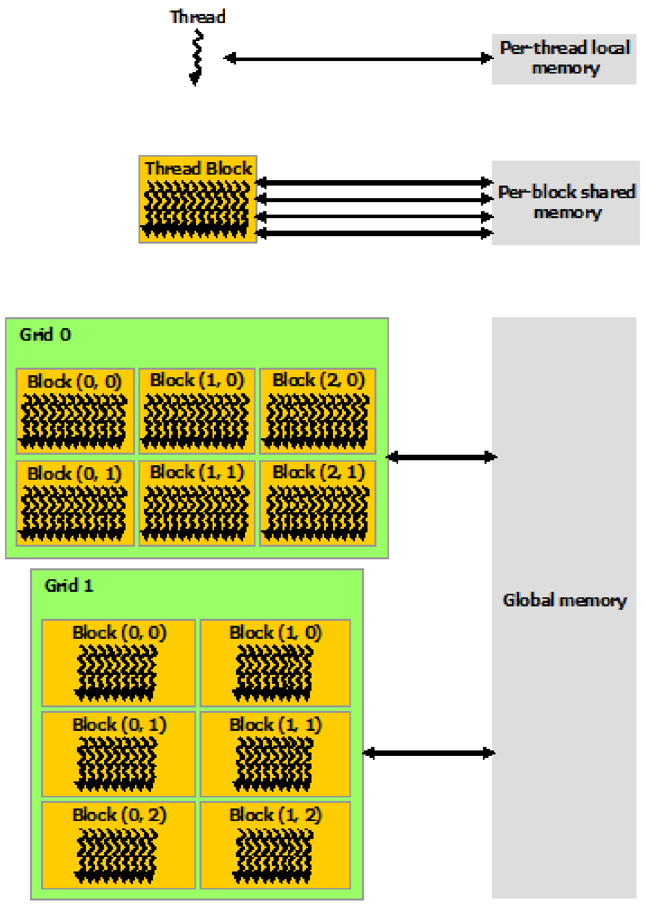


Figure 1.5: Memory hierarchy ([1])

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:

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

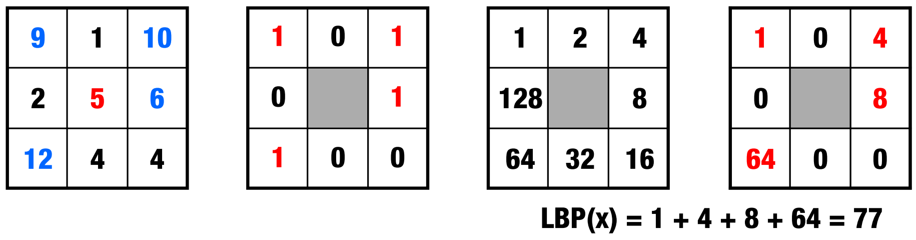


Figure 2.1: LBP feature

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

3.1 Object detector

The object detector 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.

The project is designed to be used by programmers without the knowledge of waldboost object detection or with a basic knowledge depending on the module used. There are two modules:

- WaldboostDetector class - a simple interface for using the detector without any additional knowledge.
- A waldboost detector library - contains all the object detection functionalities both for the gpu and the cpu.

3.2 WaldboostDetector class

The first option, the WaldboostDetector class, is designed to simplify the detection as much as possible. It can be used 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 in **??**. On the other hand when not set, these are set optimally based on results obtained by measurements on different GPUs.

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

The living cycle of the detector is a bit different for a dataset, than for a video, but both use the following steps.

1. Initialization with an image - allocates appropriate memory and setups the detector based on the passed image (represented as OpenCV matrix)
2. Image setup - loads the image into the detector memory
3. Detection - runs the object detection process
4. Free memory - frees allocated memory

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.

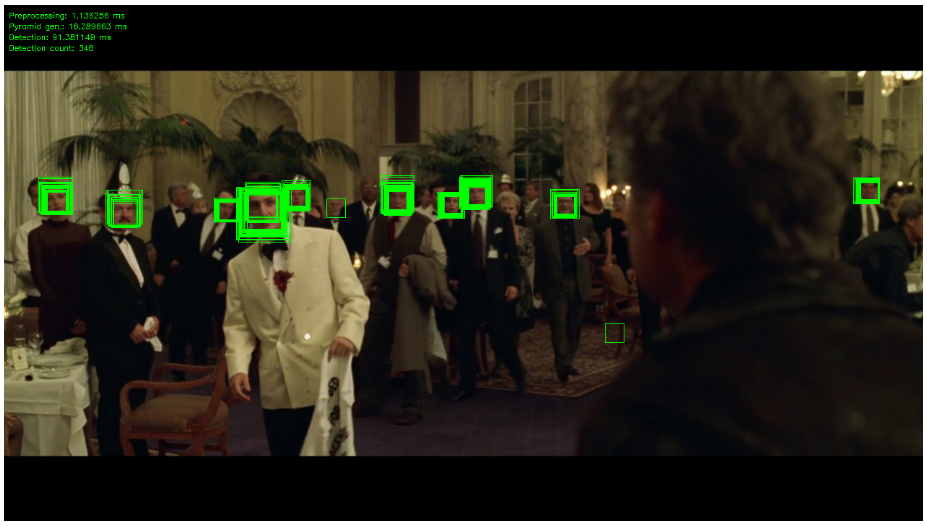
For a dataset, every image can be different and so the whole process must be run multiple times for every loaded image.

3.3 wbd library

The second option is the library itself, which is used by the WaldboostDetector class, but can also be used on its own. It contains kernels to preprocess the image, build a pyramidal image and different implementations of GPU-accelerated waldboost detection using LBP features.

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

- simple - a simple `C++` implementation
- gpu - a GPU implementation using CUDA
  - pyramid - pyramidal image generation
  - detection - detection processing
- waldboost detector - a class wrapper for simple detector usage



```
Preprocessing: 1.136256 ms
Pyramid gen.: 16.289663 ms
Detection: 91.381149 ms
Detection count: 346
```

Figure 3.1: A sample output

3.4 Program structure

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

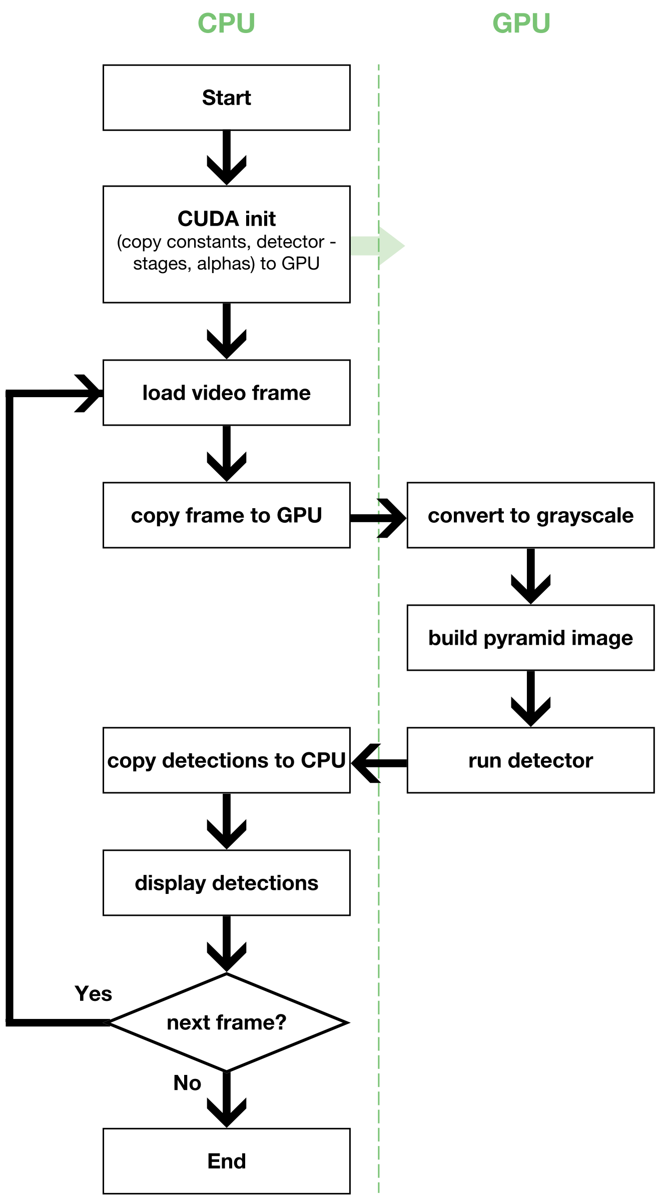


Figure 3.2: Application pipeline

3.4.1 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 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.

3.4.2 Kernels

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

- Preprocessing
- Pyramidal image
- Detection

3.4.3 Preprocessing

First the initial loaded image has to be preprocessed. There are 2 operations to be done - conversion from the initial integers to a float-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 and this is done many times while creating the pyramidal image.

Conversion to grayscale is a simple image processing operation described by the formula (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 saved as a texture.

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

3.4.4 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 [?]).

This has several advantages. One of them being the slight overhead (up to 1.5 $\mu s)bybindingandunbindi$

The main advantage in our case is, that texture objects can be passed as arguments and therefore 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 3.4.5.

3.4.5 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 all the features inside the window somehow describe an object of similar size as the window. Objects such as faces are usually much larger, therefore we have to create many sub-sampled images and hope, that we find at least one among them, where the object fits inside the window.

In order to do this, the straight forward method is to create a pyramidal image.

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.

As shown on 3.4.5 when sub-sampling an image below half its original width, pixels are left out and a sampling error is created.

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

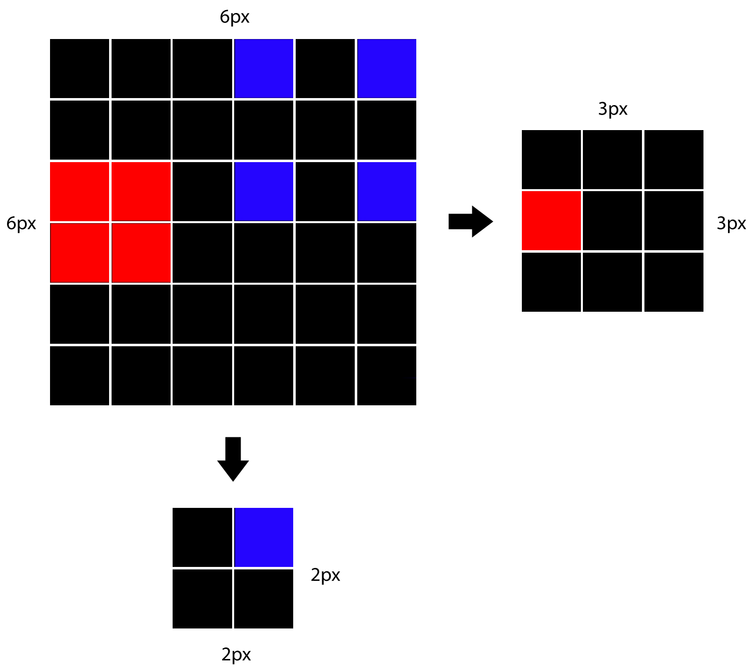


Figure 3.3: Error when sub-sampling an image below twice the original width using bilinear interpolation

of images in an octave, every image is $2^1/number_o f_i mages$ 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.

The implementation itself uses 8 levels and 4 octaves. This results in the smallest image being 16x smaller than the original. By reducing the number of octaves or levels, better performance can be gained, but it will be get reflected in the number of detections.

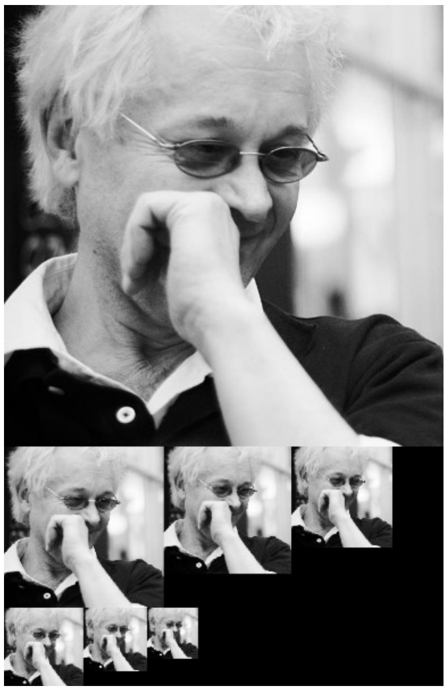


Figure 3.4: Pyramidal image

3.4.6 Detection

A Waldboost detector consists of several parts. First we will discuss the general idea of a waldboost detector and the memory organization of the different structures. Then we will focus more on the details of the implementation.

- α coefficient table
- stages
- final threshold

To detect an object, the detector has to successfully evaluate a given number of stages, which are processed sequentially. In our case 1024, this number can be reduced, but it will have an impact on the number of detections. Every stage the algorithm processes, a response is given by the α-table and sample's LBP. It is then added to the accumulated response. The sample is discarded in case the accumulated value falls below the threshold `thetaB`.

```
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 3.5: Stage structure

The detector uses a 26x26 pixel-wide window, where `x` and `y` are offsets inside the window and `width` and `height` describe the size of the feature. `alphaOffset` is an offset inside the α-table corresponding to appropriate LBP values. `thetaB` is the accumulated threshold value.

```
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

3.5 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-table** - 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 as described in 3.4.5. It is highly optimezd for random read-only access, which is exactly what we want.

3.6 Thread allocation

A GPU thread is allocated for every sample, therefore every pixel. Based on [5] only a fraction of samples (around 1%) is still processed by the classifier after only 10 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.

[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 reorganized to continue within fewer warps and the other resources to be freed.

The detector uses several ways of reorganizing surviving threads. They are summarized below by the types of functions they use and where surviving threads are stored:

- atomic functions / global memory
- atomic functions / shared memory
- atomic functions / hybrid global-shared
- prefix sum / global memory

It also uses 3 functions, which are implemented as functions within a kernel or kernel functions on their own. This depends whether the kernel can by synchronized without having to run it again.

- `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.

3.6.1 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 based on the value of the global counter and atomically incrementing it. It then saves the information about the sample (3.6.1) to global memory with the ID as an index.

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

Figure 3.6: Stage structure

Threads cannot be synchronized across the whole grid (between all the threads), only across a block, therefore a new kernel must be launched. Functions themselves are implemented as kernels.

3.6.2 Atomic functions and shared memory

The global memory method has two logical downsides. One being the use of a global memory counter, which might become a serious bottleneck on an image with lots of detections. The other being the synchronization, when a whole new kernel has to be run.

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.

Threads can be synchronized within a block and so the 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 classifier stages the surviving sample is assigned an ID within its block based on the value of the shared memory counter and atomically increments it. Threads are then synchronized and only the ones having ID smaller than the value of the counter continue.

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.
- 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 due to threads being packed at the start of the shared memory.
- An overhead of syncing threads, resetting the counter and checking for surviving threads is added.

For example if there would be only 32 surviving threads, the global memory method would only need a single warp. On the other hand the shared memory variant would probably need more warps, because the surviving threads would be distributed across more blocks.

The overhead of rerunning a kernel is quite low and so a third method was proposed.

3.7 Hybrid method using both shared and global memory

The hybrid method removes the global memory counter bottleneck and still preserves the efficiency of organizing survivors in global memory. Like in the shared memory implementation, it uses a counter stored in shared memory. Each surviving thread is assigned an ID based on this counter, after which the threads are synchronized. An offset in global memory is also stored. The current value of the offset is stored by the thread and the value of the counter is atomically added to it. The stored value is then used as an index of survivor information in global memory (`shared memory ID + global memory offset`). Because an atomic add is used, in case another block wants to save its survivors, it already uses the new offset value and again atomically adds its counter.

The disadvantage is a slight overhead, where all threads inside the block have to be synchronized before the counter can be added to the offset and so all threads must wait in case any of them survives. Also both an atomic increment and an atomic add are used.

Chapter 4

Performance measurements and profiling

This chapter evaluates performance results obtained by measurements on different GPUs and discusses them based on results obtained from profiling. The following GPUs were used:

- NVidia GeForce GTX 980
- NVidia GeForce GTX 780Ti
- NVidia Quadro K1000M

The technology used in the implementation is supported only by Kepler architecture and newer (CUDA 5.0+). Because of this the cutting edge 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 chosen to test the performance in more standard user conditions.

Tests were done for all the implementations (prefixsum, hybrid, global, shared and simple) and the following settings:

- 720x480 video
- 1280x720 video
- 1920x1080 video
- BioID face dataset with 1512 gray level images

The videos used were both a movie with up to 100 faces and a video from a webcam with 1-2 faces.

-	NVidia GeForce GTX 980	NVidia GeForce GTX 780Ti	NVidia Quac
Architecture	Maxwell	Kepler	Kepler
CUDA Cores	2048	2880	192
Base Clock (MHz)	1126	875	850
Memory (MB)	4096	3072	2048
Memory bandwidth (GB/s)	224	336	28.8

Table 4.1: Parameters of graphics cards used for performance measurements

4.1 Detection time

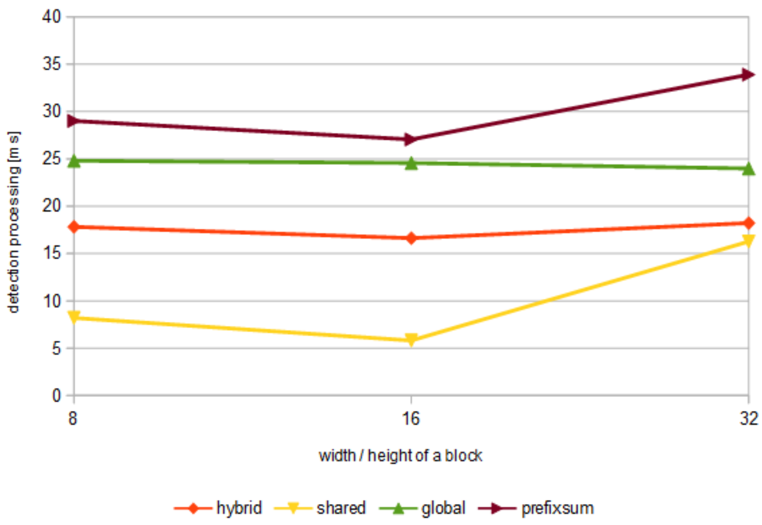


Figure 4.1: Comparison of detection times of different implementations on GeForce GTX 980 (Maxwell)

- Best block setup generally seems to be 16x16.
- Shared memory implementation seems to be the fastest on both 780Ti and 980, where as on the K1000M the fastest implementation is the one using global memory.
- Shared memory is dramatically faster than hybrid shared memory, which is dramatically faster than global memory implementation on Maxwell, whereas on Kepler all three implementations are comparable.

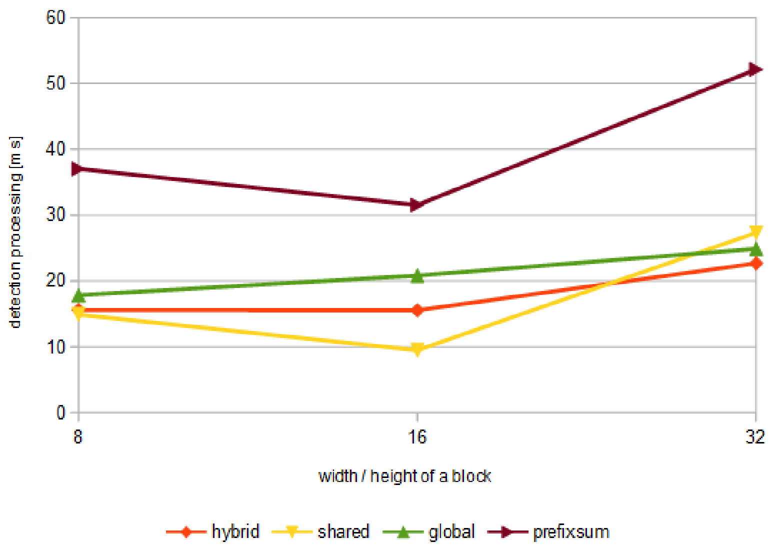


Figure 4.2: Comparison of detection times of different implementations on GeForce GTX 780Ti (Kepler)

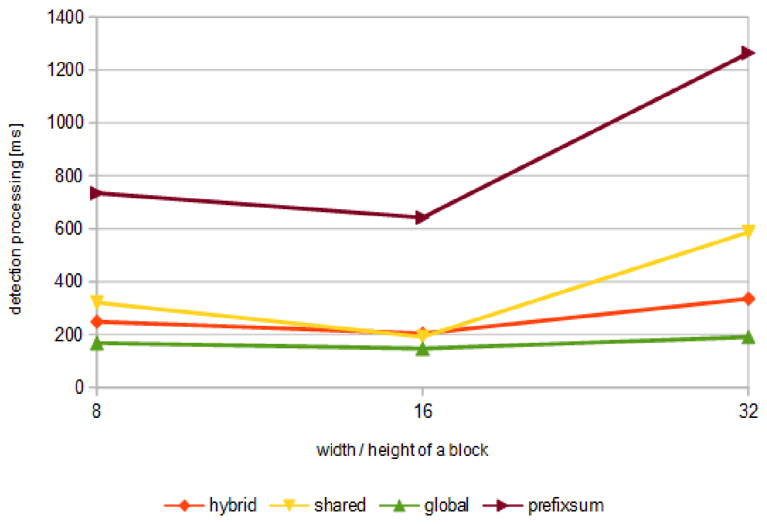


Figure 4.3: Comparison of detection times of different implementations on Quadro K1000M (Kepler)

4.2 Summary

As of April 22, 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.

4.3 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.