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

ACCELERATING FACE ANTI-SPOOFING ALGORITHMS USING GP-GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ONDŘEJ BEŇUŠ
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

AKCELERACE OBLIČEJOVÝCH ANTI-SPOOFING ALGORITMŮ POUŽITÍM GP-GPU

ACCELERATING FACE ANTI-SPOOFING ALGORITHMS USING GP-GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ BEŇUŠ
AUTHOR

VEDOUCÍ PRÁCE — Ing. KAREL VESELÝ
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se specializuje na akceleraci algoritmu z oblasti obličejově zaměřených anti-spoofing algoritmů s využitím grafického hardware jakožto platformy pro paralelní zpracování dat. Jako framework je použita technologie OpenCL která umožňuje použití od výkoných stolních počítačů po přenosná zařízení, od různých akcelerátorů jako grafické čipy, či ASIC až po procesory typu x86 bez vazby na konkrétního výrobce či operační systém. Autor předkládá čtenáři rozbor a akcelerovanou implementaci široce používaného algoritmu a dopadu urychlení výpočtu.

Abstract

This thesis is specializes on algorithm acceleration from the field of face-based anti-spoofing. Graphics hardware is used as platform for data-parallel processing. As framework, the OpenCL is used. It allows execution on devices such as powerful desktop computers or hand-held devices as well as usage of different kind of processing units such as GPU, ASIC or CPU without any bound to hardware vendor or operating system. Author presents to reader analysis and accelerated implementation of widely used algorithm and impact of such improvement in execution time.

Klíčová slova

GPGPU, OpenCL, biometrie, rozpoznávání obličeje, anti-spoofing, LBP, LBP-TOP

Keywords

GPGPU, OpenCL, biometry, face recognition, anti-spoofing, LBP, LBP-TOP

Citace

Ondřej Beňuš: Accelerating face anti-spoofing algorithms using GP-GPU, bakalářská práce, Brno, FIT VUT v Brně, 2015

Accelerating face anti-spoofing algorithms using GP-GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Miguela Bordallo z University of Oulu, jako formální vedoucí za FIT práci zaštiťoval Ing. Karel Veselý

. . . . . .
Ondřej Beňuš
July 29, 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Despite fast development in the field of IT software and hardware, there are problems computationally too complex to calculate in real-time.

Since the beginning of computer science, programmers tried to find ways how to shorten the time of computation. Common approach is writing programs or its parts in assembler. With spreading of new technologies and processors came new instructions being able to process multiple data in one cycle. SIMD instructions such as MMX and SSE often improved performance.

With introduction of multi-core processors, the trend of algorithm parallelization was set. There are numerous problems (synchronization, race conditions, etc.) writing algorithms in way, that unleash nearly full potential of all cores.

In last few years the computation units in graphic accelerators become general enough to run general-purpose programs and powerful enough to process huge amounts of data much faster than CPU. It is achieved by thousands of cores processing same instructions over multiple data.

The rate of CPU innovations significantly decreased over few past years, but the rate of performance gain of GPU is constant or slightly increasing. Together with wide use of modern GPGPUs from all PCs and laptops to most of hand-held devices, GPU is ideal platform for accelerating computations over big data-sets such as image processing and recognition, voice synthesis and recognition, machine learning and others.

This work is focused on implementing GPGPU algorithm widely used in face based anti-spoofing and evaluating it's use.

In the second chapter, face-based anti-spoofing is described as the field that will be used to demonstrate possible target for accelerating algorithms using GPGPU.

Next chapter describes potential target for acceleration and presents several possible frameworks for GPGPU.

Implementation of serial and parallel algorithm is explained in chapter 4.

Fifth chapter is focused on testing this implementation on low-power device as ideal future target of face-based anti-spoofing techniques.

Last chapter promote benefits of implementing algorithms on GPGPU.

Chapter 2

Face-based anti-spoofing

Humans and animals are using body characteristics such as face, voice or body structure to recognize each other for thousands and millions of years. The system of using a number according to body measurements was invented in the mid-19th century. While the idea was gaining popularity, more important discovery was made. It was discovered, that fingerprints are unique to every living person. The idea that some of the body features can be unique when measured properly and then using it to identify person revolutionized criminology.

2.1 Background

There are numerous body characteristics, but only few of them satisfy these requirements:

- *Universality* each individual should have the specified characteristic.
- *Distinctiveness* any two individuals are different enough to distinguish them using only this characteristic.
- *Permanence* the characteristic is invariant enough to be recognized over the period of time.
- *Collectability* the characteristic is measurable quantitatively.

There are numerous other requirements for biometric systems that are used in everyday life:

- *Performance* is relation between the ability to achieve sufficient accuracy and resources needed to achieve recognition in specified time.
- *Acceptability* indicates how many people are willing to provide biometric information in everyday use. After the information of government programs to collect personal information leaked, many security questions arised.
- *Circumvention* reflects how easy it is to make false acceptance by spoofing methods.

2.1.1 Biometrics and access systems

In the past, biometrics was developed mostly due to extensive use in law enforcement to identify criminals. Nowadays it is still more often used in person recognition systems in a large number of commercial applications.

A biometric system is pattern recognition system. First step is to acquire data from person. Second step is extracting a feature set from acquired data. Last step is comparing against template dataset in database. Depending on application, system works in verification or identification mode.

- *Verification mode* In this mode, system compares extracted features against one template dataset in database. Template selection is based on other identifying factor as ID card or username.
- *Identification mode* System performs one-to-many comparison in order to find matching template for extracted features. It is also possible to introduce negative recognition to prevent user using multiple identities.

2.2 Spoofing techniques

2.2.1 Definition of spoofing

According to online dictionary *dictionary.reference.com* spoofing means:

4 to fool by a hoax; play a trick on, especially one intended to deceive.

For our paper, we will extend definition to better reflect spoofing in face recognition systems.

- A spoofing is, when stolen and/or forged biometric traits are used to gain illegitimate access to system or resource.

2.2.2 Types and examples of spoofing

In face-based authentication systems, spoofing attacks are usually performed using photographs, videos or masks. In specific cases it is also possible to use make-up or plastic surgery. However due to extensive use of social networks as Facebook or Youtube, there is a lot of resources (especially photographs and videos) that can be used for spoofing.

- *Image* attacks are the easiest option. It is surprisingly efficient when used on unprotected system [5]. Even non-skilled attacker needs only printer (in some cases also mono printer is sufficient) and photograph, that can be easily found on internet.
- *Video* is presented on screen of a hand-held device. It is more sophisticated method than image spoofing and can achieve higher rate of false acceptance than image attacks.
- *Mask* attacks are the harder way to perform an attack, but even with sophisticated systems it is more efficient than other types of attacks when properly executed [15].

2.3 Anti-spoofing techniques

Most of the work in area of face-based anti-spoofing is done in research labs or in small and highly controlled environments, however attempts to spread this technology and provide widely usable solutions are already in progress [13].

When using 2D face recognition system, anti-spoofing methods can be classified in three categories: motion analysis, texture analysis and liveness detection.

2.3.1 Motion

Motion detection use the difference of motion vectors detected in videosequence to detect spoofing. Head usually has different movement vector than near surroundings and background. It can be also used for detection of small differences directly on face, because of 3D nature of face. This type of anti-spoofing methods are usually effective against photographs and other purely 2D methods.

2.3.2 Texture analysis

Texture analysis is usually focused on texture-patterns caused by printing or by screen projection. It can also detect blur or overall color balance.

Micro-texture analysis focus on light scattering and reflection or color unbalances on local level.

Texture analysis usually uses Local Binary Patterns (LBP) as a mean of feature extraction.

2.3.3 Liveness detection

This method focus on individual parts of face and by computing movement vectors tries to determine, whether the biometrics data were acquired from live user that is physically present. As a point of interest can be used eyes, eyelids (blinking), lips or even cheeks. Algorithm calculate trajectory of specified part of the face using simple optical flow and then it is explored by heuristic classifier.

2.4 Anti-spoofing resources

Researchers usually have quite specific requirements of software. One of those requirements is environment allowing fast prototyping and testing of new ideas and at the same time also easy implementation of final piece of software and easy access to large data-set databases. This requirement is noticeable especially in the field of multimedia and its applications. Next very important requirements for researchers is self-explanatory and clear code with great documentation.

Lot of packages developed for machine learning and/or signal processing are available. Among others [1], mlpack [2], OpenCV [3] or numl [4]. However, just few of them provide complete set of functionality for managing research experimentation such as database interfaces, alternative API for accelerated versions of algorithms or scriptable drawing utilities. For

[1] https://github.com/josephmisiti/awesome-machine-learning
[2] http://mlpack.org/
[3] http://opencv.org/
[4] http://numl.net/

example, OpenCV is developed primarily in C++ and lacks API for database access or build-in analysis utilities. Among many choices, it seems that only two frameworks are well-equipped for research works. Scikit-learn [5] and BOB [6]. Scikit-learn provides dataset APIs and machine learning algorithms, but lacks basic signal processing functionality and a clean C++ API. Researchers using this platform might deal with lack of convenient environment for speeding up algorithms.

2.4.1 The BOB System [3]

Biometrics Group at Idiap Research Institute in Switzerland developed BOB framework to satisfy all requirements stated above. BOB is free signal processing and machine learning toolbox. Since then, growing community have created many modules to extend functionality. BOB provides Python interface for rapid development and C++ parts speeding up identified bottlenecks. This approach is designed to meet the needs of researchers by reducing development time and fast and efficient processing large data-sets.

Python and C++ part is seamlessly integrated ensuring easy of use and great extensibility. BOB also provides reproducible research result technique through integrated experimental protocols for several databases. It currently provides API as interface to several database protocols. Several protocols for well-known face-based databases are integrated to provide way of making reproducible and comparable results with other scientific publications. Also one of the main goals is code clarity, documentation and extensible testing ensuring easy adoption for new developers. Code readability and maintainability is preferred over optimization.

Bob's abilities and the scope in the field of signal processing is rather big including covering fields as computer vision and audio processing. It also covers many activities in field of machine learning, to name a few: dimensional reduction, clustering, generative modeling, and discriminative classification.

Most importantly, Bob was designed to be highly extensible. Bob library system was designed to easily prototype ideas and algorithms in friendly Python environment, then later to rewrite it in C++ to gain speed. Bob contains this layer connecting clean Python and clean C++ environment. Researchers can quickly develop ideas in Python while seamlessly exploiting rich possibilities of C++ codebase provided by Bob.

Multidimensional data arrays are extensively used to represent various kind of digital signals. At Python level, it is ensured using NumPy [7] and at C++ level it is handled by Blitz++ [8]. In order to exchange data between Python and C++, Boost [9] template library is used.

[5] http://scikit-learn.org/
[6] https://idiap.github.io/bob/
[7] http://www.numpy.org/
[8] http://blitz.sourceforge.net/
[9] http://www.boost.org/

Chapter 3

Acceleration of anti-spoofing techniques

3.1 Motivation

Due to the fact, that optical sensors (cameras) has improved significantly over few past years and number of pixels has increased considerably [6], complex algorithms for image processing became too slow to execute in real-time [9, p. 11-12] on low-power devices. On the other hand, still more and more complex spoofing methods require more complex anti-spoofing techniques to be used.

As a result, everyday CPU doesn't have enough power to run anti-spoofing techniques being able to reveal basic spoofing attacks in acceptable time. Mobile devices, as the most common target for face-spoofing attacks, are despite quick evolution years from being able to process enough data in real-time to have seamless user experience.

3.1.1 Why is it important?

Face-based anti-spoofing techniques are essential part of face authentication systems as it can prevent false hit and thus compromising whole system. In case of successful attack on mobile phone, attacker can access list of contacts, conversations, personal files and also mostly other accounts as email accounts or Facebook account and in some cases also to online banking account information. These information can be then used for phishing or directly as invoice information. Being able to correctly recognize and authenticate user is crucial. No image based authentication system should be used without some kind of anti-spoofing technique.

Using anti-spoofing methods results in higher requirements for processing power. Development of CPU technology can not satisfy in near future this demand, but leveraging GPU power can increase easily usable processing power of device more than hundred times. This processing power can be used to reduce algorithms execution time or to increase algorithm's complexity and prevent more sophisticated attacks.

3.1.2 Real-time anti-spoofing

For seamless user experience, face-based authentication should not last more than few seconds. As today's devices does not have enough computing power to offer this experience with robust algorithms preventing false identification, many companies started to offer their

own cloud-based solutions [2, 14]. This approach can be used when authenticating online banking transfers or logging into online applications. In case of loosing internet connection (outlying areas, tunnels, roaming or wifi-only devices), this approach can not be used. In such cases, the mobile device has to make all computations in acceptable time.

Embedded systems managing physical access such as entrance tourniquets with many incoming people in short period of time has even higher demands for reducing latency between obtaining video sequence of person and successful authentication. In ideal state this latency is just few seconds, but the exact time also depends on the type of access system. This short time can ensure seamless experience for all users.

3.2 Parallel processing and GPUs

Architecture of GPUs allows executing many threads slowly rather than one thread quickly. In addition, execution units are organized into groups, that can run only the very same program as the other units in the same group [1]. It allows very high density of cores on the chip and effective execution of data-parallel algorithms.

3.2.1 Different alternatives for acceleration

Due to big computation potential in GPUs and accelerators, many companies and organizations created through time their own way, how to utilize the power of accelerators. We will focus on three well known and wide-spread frameworks that are commonly used to accelerate algorithms through data-parallelism.

CUDA [10]

CUDA is hardware and software interface for programming GPU, created by NVIDIA and is used only for graphic IP of this company. Its aim is to make it easier to create GPU computing programs by avoiding using shader languages. First version was introduced in 2006 and first SDK was released a year later for G80 architecture. With development of graphic architecture new versions were released adding new functionality, or extending compute capabilities. Every version is tighted to corresponding architecture which results in non forward compatible features. The latest version in time of writing (2Q 2015) is 7.0.

Simplified programming procedure can be described as loading data to host (CPU), initializing GPU device, copying data from host memory to device memory, running computation and finally copying result back from device to host.

Programming can be done through C language with additional library functions and syntax extensions to express parallelism. CUDA SDK also provides extensions for C++ and Fortran. Third party extensions also provides CUDA functionality to Python, Perl, Java, Ruby, Lua, Matlab, etc.

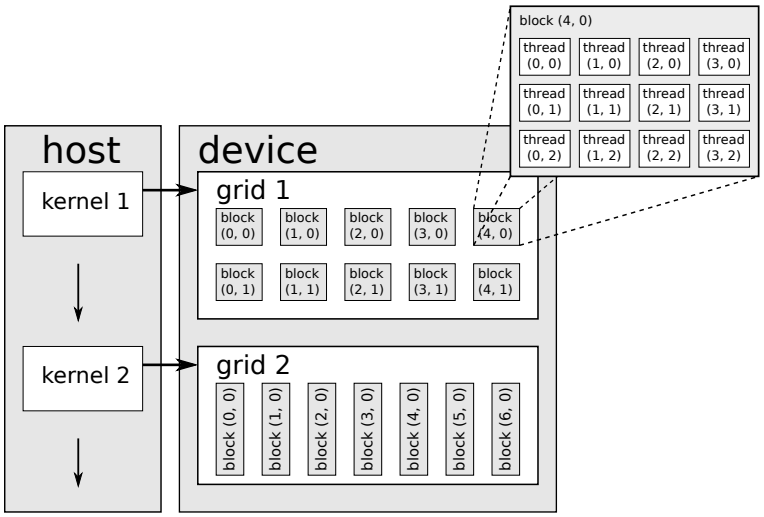
There are two main models in CUDA programming.

Execution Model

CUDA program consist of functions capable running on host (CPU) or the device (GPU). Functions running on GPU (kernels) can be executed asynchronously from the host and across multiple processors in the same time. To decide what function should run on host

or on device, qualifiers as *__device__* , *__global__* or *__host__* are used in source program. Note, that *__host__* qualifier is usually omitted, as it is default qualifier.

Figure 3.1: Kernel executed on GPU is organized as a grid of parallel blocks that contains parallel threads.



Grids and blocks can be divided to up to three dimensions. In order to obtain the best performance boost, size of grids and blocks should be tuned for every architecture. Following example shows the computation of grid size and block size:

```
dim3 gridSize(imageX/blockX , imageY/blockY , 1);
dim3 blockSize(blockX , blockY , 1);
someKernel <<<gridSize , blockSize>>>(param1 , param2);
```

Configuration parameters for individual kernels are specified in triple angular brackets. First parameter specifies number of blocks in grid and the second one specifies number of threads in every block as can be seen in figure 3.1. Both parameters can be composed of 1, 2 or 3 dimensions. These dimension sizes can be accessed from kernel through built-in variable *gridDim* and *blockDim*. Individual threads can communicate through shared memory and barrier synchronization. Each thread is then identified by *threadIDx* built-in variable and *blockIdx* built-in variable. Unique identifier can be then calculated from two dimensions as follows:

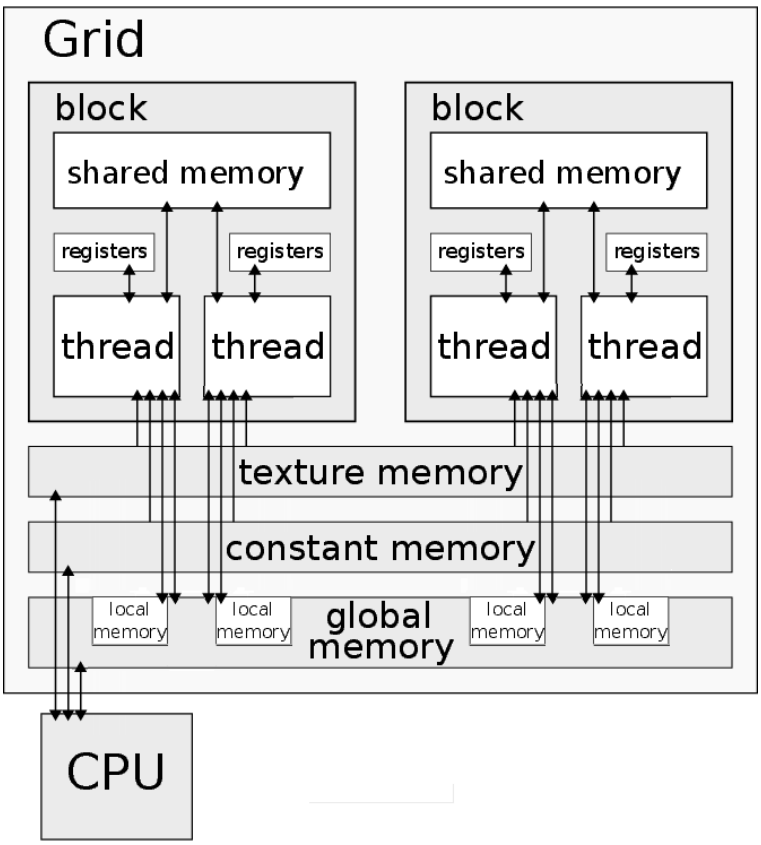
```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int id = x + y * gridDim.x;
```

Memory Model

Very important part, that needs to be considered when creating GPU compute programs, of CUDA platform is it's memory model. It describes GPU memory access and hierarchy. In addition to large high-latency memory (global memory), model describes also caches, registers, shared memory, constant memory and texture memory.

CPU chips has limited number of cores and caches associated with those cores. Size of those caches can be relatively large, due to it's limited number. GPU chips usually has hundreds or thousands of cores and every core has it's own cache. Due to this large amount of caches, every cache has very limited size.

Figure 3.2: Memory hierarchy in CUDA system



In traditional CPU programming, all caches are transparent to programming model and programmer can not easily directly modify it's content. In GPU programming many of caches and different types of memory can be, and should be, controlled by programmer.

In case, that some cores needs to exchange data with other core and due to parallel access, synchronization techniques must be introduced.

Shared memory is relatively small and low-latency memory within every SM. This memory is divided into memory blocks called banks, that can be accessed in parallel. In case, that multiple threads tries to access the same memory bank in the same time, access is serialized to prevent data inconsistency. Shared memory is defined by programmer and it's purpose is to allow communication between individual threads and to exchange often used data.

Constant and texture memory is not located on-chip, but off-chip (on device) together with global and local memory. The constant memory is used to contain constant variables used by many threads (global variables) and texture memory is mostly used to store graphic data. Because of the nature of those data types stored in this type of memory, access can be highly cached, as data flow is usually one-directional (read-only from device). Storing data in this type of memory usually result in faster access, than accessing data in global memory. texture memory also benefits from hardware units performing interpolation when accessing data. Hardware units can also perform clamp and wrap memory access to prevent out-of-bounds access.

CPU can access GPU's global, constant and texture memory, but access is usually done

through PCI-Express bus and is considered as bottleneck in CUDA programming, because the limited bandwidth introduces latency. Due to this limitations, transfers between host and device should be limited.

Programmers specifies variable location by using keyword *__device__* for global memory, *__shared__* for shared memory and *__constant__* for constant memory. If there is need to allocate memory on device, functions cudaMalloc() and cudaFree() can be used to reserve or to free memory respectively. Allocated memory can also be copied by using function cudaMemcpy() in combination with a keyword specifying data location of source and target. Possible data locations can be host –> host, host –> device, device –> host, or device –> device.

OpenACC [11, 12]

Low level APIs such as CUDA and OpenCL provide rich set of features enabling fine control of GPU hardware. Managing every aspect of execution model and memory model can however be difficult for programmer. To simplify GPU programming, NVIDIA, The Portland Group, CAPS and Cray created simpler way, using compiler directives, to program accelerators such GPU.

The OpenACC standard was first released in November 2011 as version 1.0 and API consist of compiler directives, library functions and variables. Version 2.0 was released in June 2013.

The OpenACC API and its directives can be used within C, C++ and Fortran programs. Compilers are available across wide range of operating systems and hw accelerators. In April, 2015, GNU GCC 5.1 was released and it includes a preliminary implementation of the OpenACC 2.0a specification. OpenACC is maintained and developed by OpenACC corporation, which includes (in addition to founding members) organizations such as Swiss National Supercomputer Center, Georgia Tech, Allinea or Oak Ridge National Lab.

Figure 3.3: OpenACC parallel loop

```
int main () {
        double var = 0.0;
        # pragma acc kernels
        for( long i =0; i < N ; i ++) {
                double r = ( double )(( i +0.5)/ N );
                var += 1.618/(1.0 + r*r);
        }
}
```

OpenACC consist of compiler directives used to describe what loop or what portion of code should be executed in parallel on accelerator. In this simple scenario, compiler will decide what optimizations will be used and create target code. Programmer itself does not have to need know anything about memory and execution model. However, there is still need to recognize algorithms and it's ability to run in parallel. OpenACC also has features for more explicit control over program execution.

OpenACC can be similar in many ways to OpenMP standard, due to the fact, that many developers of OpenACC standard take also part in OpenMP Architecture Review Board and long term of OpenACC is to become something like 'OpenMP for accelerators'

Figure 3.4: OpenMP parallel loop with reduction

```
int main () {
        double var = 0.0;
        # pragma omp parallel for reduction (+: var )
        for( long i =0; i < N ; i ++) {
                double r = ( double )(( i +0.5)/ N );
                var += 1.618/(1.0 + r*r);
        }
}
```

rather than separate and independent standard. This effort resulted in publishing 'OpenMP for accelerators' [4] and implementing some of the proposed ideas in OpenACC standard version 4.0 published in July 2013.

As can be seen in both examples (3.4 and 3.3), C code uses *#pragma* compiler directive followed by the name of standard and then followed by standard-specific command.

OpenACC standard defines only API and it is up to developers to implement OpanACC functionality into compilers. From beginning there were available only commercial compilers such as those form PGI and CAPS, but as mentioned earlier, GCC in version 5.1 has preliminary implementation of standard OpenACC 2.0a and there is also accULL, The OpenAcc research implementation developed by University of La Laguna. Most of implementations of OpenACC standard are based on translating code into CUDA or OpenCL.

Execution Model

OpenACC is mostly used for heterogeneous systems with host processor and accelerator, but when translating into OpenCL, also multicore processor can be used. On such systems, program is executed in serial until the point of intensive data-parallel section, usually containing one or more loops that can be distributed between multiple threads (work-items) and processor offload such execution to accelerator for fast parallel processing. After execution on accelerator is done, results are transferred back to host processor. This behavior is known as host-directed execution model.

The process of execution is following: host allocate device memory, initiate data transfer, transfer code to the device, pass required arguments, wait for the execution to complete, transfer results back to host memory and deallocate device memory. Vast majority of this steps are done automatically by OpenAcc compiler, but there are also ways how to influence those steps by programmer.

Parallel code can be executed synchronously using *#pragma acc kernels* directive or asynchronously (eg. more loops) using *#pragma acc parallel* directive.

Memory Model

Memory is usually separated in systems with host CPU and attached accelerator device, such as GPU. These heterogeneous systems are primarily focused by OpenACC.

All data allocation and data movement between host and device must be managed by host through library calls. OpenACC also has notion about private and shared memory, where shared memory is usually software cache-like memory and private memory is used as hardware cache. Many of memory constraints and data location changes are handled automatically by compiler based on directives specified by programmer or specified directly, eg. that memory should be allocated on device, that data should be copied only in one direction or that data already is on device.

In a typical application with parallel region, data is copied to the device from the host at the beginning of each parallel region and copied back from device to host memory in the end of such region. When there is some form of pipeline, compiler may not recognize it and move data from device to host and back again. This data copying is redundant and cause slowing down whole algorithm. Programmer can specify region by *#pragma acc data* to prevent such behavior and keep data on GPU between multiple kernel runs. Compiler also automatically creates barrier constructs to prevent simultaneous access to the same memory block, that might result in memory inconsistency.

3.3 The OpenCL framework

OpenCL was developed by Apple Inc., and then under the auspices of Khronos group published as industry standard. Since then many hardware and software vendors adopted OpenCL as standard for heterogeneous computing. OpenCL programs, kernels, can run on various devices such as x86 CPUs, various GPUs, FPGAs, ARM under Windows, Mac, Linux, FreeBSD and Android. Programming model is based on separate programs running on accelerating device called kernels, written in language based on C99 and host code written in C/C++. There are also wrappers for many languages like Java, Python, Haskell etc.

Low level access to system allows simultaneous execution of kernels on more devices and improving performance by user-managing multi-level memory. Cooperation and memory sharing between OpenCL and other graphics APIs such as OpenGL improve usability and performance. Specification specifies „Full profile“ that is common in PCs and „Embedded profile“ that is common in mobile devices. Main difference is floating point accuracy and support for 3D images (optional), writing to 2D image array or limitations to channel data type operations. OpenCL is an API that defines access and control of OpenCL-capable devices and it includes a C99-based language that allows implementation of kernels on them. OpenCL simplifies execution of task-based and data-based parallel tasks on sequential and parallel processors. Currently, there are OpenCL implementations on CPU and GPU. However, several efforts have been made to port the code into other processors and platforms, such as application specific multi-cores or multicore DSP.

OpenCL (Open Computing Language) is essentially an open standard for parallel programming of heterogeneous systems. It consist of an API for coordinating parallel computation across different processors and cross-platform programming language with a well-specified computation environment. It was conceived by Apple Inc., which holds trademark rights, and established as standard by Khronos Group in cooperation with others and is based on C99. The purpose is to recall OpenGL and OpenAL, which are open industry standards for 3D graphics and computer audio respectively, to extend the power of the GPU beyond graphic facilitation General Purpose computation on Graphic Processing Units.

In the OpenCL model, the high-performance resources are considered as Compute Devices connected to a host. the standard supports both data and task based parallel programming models, utilizing a subset of ISO C99 with extension for parallelism. OpenCL is defined to efficiently inter-operate with OpenGL and other graphics APIs. The current supported hardware ranges from CPUs, GPUs and DSPs to mobile CPUs such as ARM with Mali graphic core.

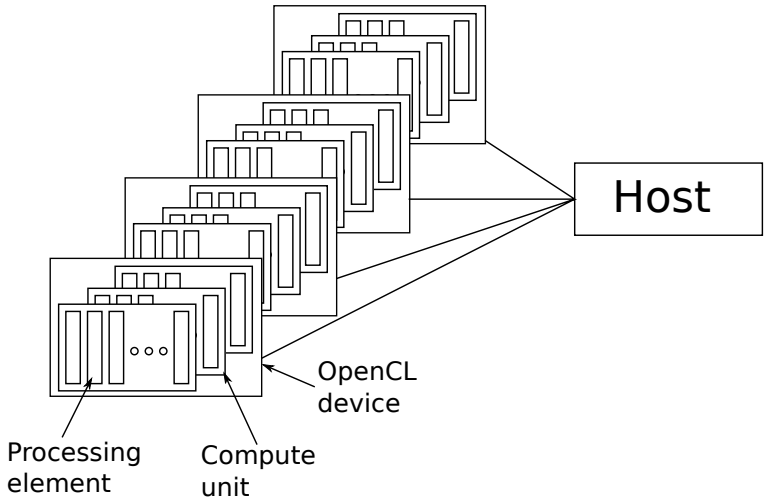
To be able to use OpenCL on such platforms and devices, the standard defines four abstract models (layers).

- Platform model specifies one host and one or more devices running OpenCL code
- Execution model defines, how is OpenCL set on host and how it will be executed on device
- Memory model defines memory hierarchy used by OpenCL kernels
- Programming model describes, how is specified model mapped to physical device

Platform model

The OpenCL Platform model defines a high-level representation of any heterogeneous platform compatible with OpenCL. This model is shown in the fig. 3.5. The host can be connected to one or more OpenCL devices (GPU, CPU, DSP, FPGA, ...). Kernel execute on the device. Device is composed of CUs (compute units) and those are divided into PEs (processing elements). Computation is made in PEs. Each PE executes SIMD.

Figure 3.5: Platform model specifies host and connected devices composed of compute units having processing units.



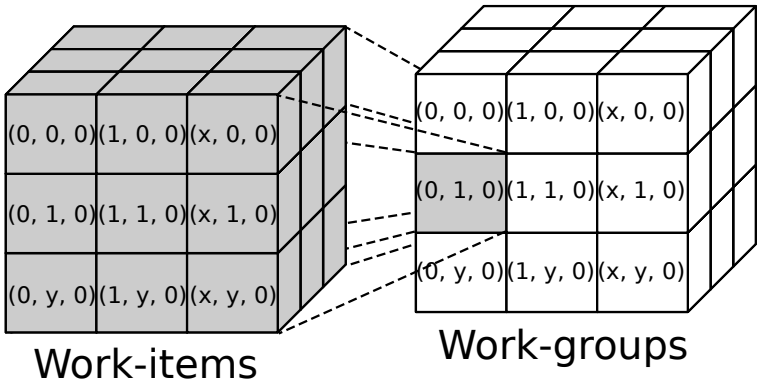
Execution model

Execution is composed of two parts. First part is host program, the second one is one or more kernels. Due to OpenCL layer, the exact steps of kernel execution on various devices (CPU, GPU, ...) are abstracted away. Kernels are executed on devices transforming input memory objects into output memory objects. There are two types of kernels, OpenCL kernels and Native kernels.

Kernel execution can be described by following points:

- Host defines kernel.
- Command is issued by host to execute kernel on device
- OpenCL runtime creates an index space
- Instance of kernel is called work-item, work-items are organized in work-groups. This division is shown on figure 3.6

Figure 3.6: Work-items (instance of kernel) are organized into work-groups and all work-groups represent execution of kernel. Organization structures can be 1D, 2D or 3D



Memory model

The OpenCL memory hierarchy model is structured in way, that abstract and/or mimic most common memory configuration of NVIDIA/AMD/ATI hardware. There are some differences since every company defines their memory hierarchy differently. Abstraction was possible since most of companies somehow defines top memory (global memory) and local memory per work-group. Also every work-item has small private memory in form of registers.

Communication and synchronization of individual work-items in work-groups is done through shared memory, but access of every work-item to shared memory is independent of other work-groups. The OpenCL framework defines relaxed consistency between individual work-groups. Memory access of work-items is not protected in any way. Reading or writing out-of-scope does not result in error.

3.4 Discussion of advantages and disadvantages

As discussed previously, face-based algorithms are required to operate on wide range of devices. OpenCL is open standard currently working on many devices. Compilers are available for platforms from hand-held devices to supercomputers. CUDA is limited only on hardware solutions produced by nVidia company. While having big share in PC and laptop market, only few hand-held devices based on Tegra K1 and Tegra X1 were launched. OpenACC is open standard, but rather few compilers support this standard.

Table 3.1: Framework properties

Framework	OpenCL	CUDA	OpenACC
SW platform	Windows, Linux, MacOS, Android	Windows, Linux, MacOS, Android	OpenCL & CUDA
HW platform	AMD, ARM, Intel, Nvidia	Nvidia	AMD, ARM, Intel, Nvidia
Programming complexity	high	medium	low
Performance gain	high	high	medium

On the other hand, OpenAcc and CUDA provide faster and easier way how to leverage

GPU potential. However, low level approach enables wide variety of possible optimizations, some of that automated compiler can not make from high level code.

3.5 Accelerating anti-spoofing techniques

Image analysis is composed out of two groups of operations. Image to feature extraction and making decisions based on those features. The core of image-based anti-spoofing techniques is usually based on computation with the extracted features. However algorithms used to operate with extracted features are very hard to parallelize and the computation is usually not that long. Higher potential to benefit from acceleration is in accelerating image based algorithms.

3.5.1 Identifying bottlenecks

There are two major bottlenecks when accelerating image based algorithms. Memory bottleneck and computation bottleneck.

Memory bottleneck

This type of bottleneck is usually seen on algorithms involving very simple computation algorithms with many pixels. Accelerator then has to read lot of data, perform simple computation and write back relatively lot of data. In this case acceleration is beneficial only in case that data is already on device.

Platforms that share memory between host and accelerator and the only bottleneck is memory bottleneck can not reduce computation time by offloading computation to device. For example addition of two vectors is memory bound algorithm. After fetching data from memory follows very simple operation of addition and writing data to memory back. ALU is underutilized and memory subsystem is fully utilized. In such cases, GPU can not perform operation faster than CPU.

Platforms that do not share memory between host and device are even slower in performing memory bound algorithms due to data round-trip from host to device and from device to host. The only case when it is beneficial to offload computation to device is pipeline processing when data are already on the device and after computation data will be further processed. In those cases algorithms benefits from high bandwidth interface present on device. While DDR3 memory at 2.4GHz has bandwidth only 19.2 GiB/s [8], HBM memory used in AMD's Fury series has bandwidth 512 GiB/s. The bandwidth of GPU's planned for next year is 1TiB/s.

Computation bottleneck

This bottleneck is seen with algorithms that use little of data and perform complex operations with lot of arithmetic calculations. Memory subsystem of host is not fully utilized, whereas ALU or/and FPU units are usually busy. Host CPU usually has only few ALU and FPU whereas GPU has thousand computation units. In this case, data can flow to device faster due to higher computation ability of accelerator device.

Example of this bottleneck is matrix multiplication. Processor has to perform a lot of arithmetic operations (mainly multiplication and addition) over relatively small chunk of data.

3.5.2 Analysis of parallelization opportunities

There are many different algorithms used for anti-spoofing in face-recognition systems. To benefit most, image-based algorithms with medium or high complexity should be accelerated. Due to fact, that face-based anti-spoofing is still beginning to emerge and is undergoing rapid development, no standard image algorithms are established.

Often used algorithm within BOB system is LBP-TOP. This is image based algorithm with video sequence with some additional parameters on input side and with comparably less data in form of histograms on output. Algorithm is described more in section 3.5.3 and 4.1.

3.5.3 Justification of the choices

OpenCL is used in this work because CUDA is not hardware multi-platform and OpenACC doesn't have fully functional compilers for platforms that are functional with OpenCL. In order to run OpenCL programs, nothing else than appropriate drivers is required. Even though that many devices are OpenCL capable, serial C++ code is provided as reference implementation and as fall-back for platform not supporting any of required technologies of OpenCL runtime.

Even though, that OpenCL 2.0 was announced more than 2 years ago [7], very little device on market implement this standard. For that reason OpenCL 1.2 is used in this work. There are no OpenCL 1.2 only features used, but OpenCL 1.2 C++ wrapper fix several bugs with respect to the version 1.1. However, all algorithms developed within this work can be rather easily rewritten to use OpenCL 1.1 that is widely supported across many hardware and software platforms.

As the algorithm to accelerate we choose LBP-TOP. This algorithm is pixel independent with medium algorithm complexity and takes long time in overall process to execute. Those are signs of good candidate for acceleration.

Chapter 4

Implementation

4.1 Algorithmic description

LBP (Local Binary Pattern) is efficient, yet simple algorithm for feature extraction from images.

Algorithm is very simple in it's nature and the simplest form can be described as follows:

Figure 4.1: The simplest form of LBP algorithm

```
1. Mark every pixel as center pixel and do following:
   a) Compare neighbor pixels with center pixel in
      counter-clockwise manner.
   b) If the value of center pixel is higher than
      compared neighbor record "1", else record "0".
   c) Merge all recorded values for center pixel into
      single binary string.
2. When computed all the binary strings, create histogram
   of all binary strings.
3. Optionally normalize histogram.
```

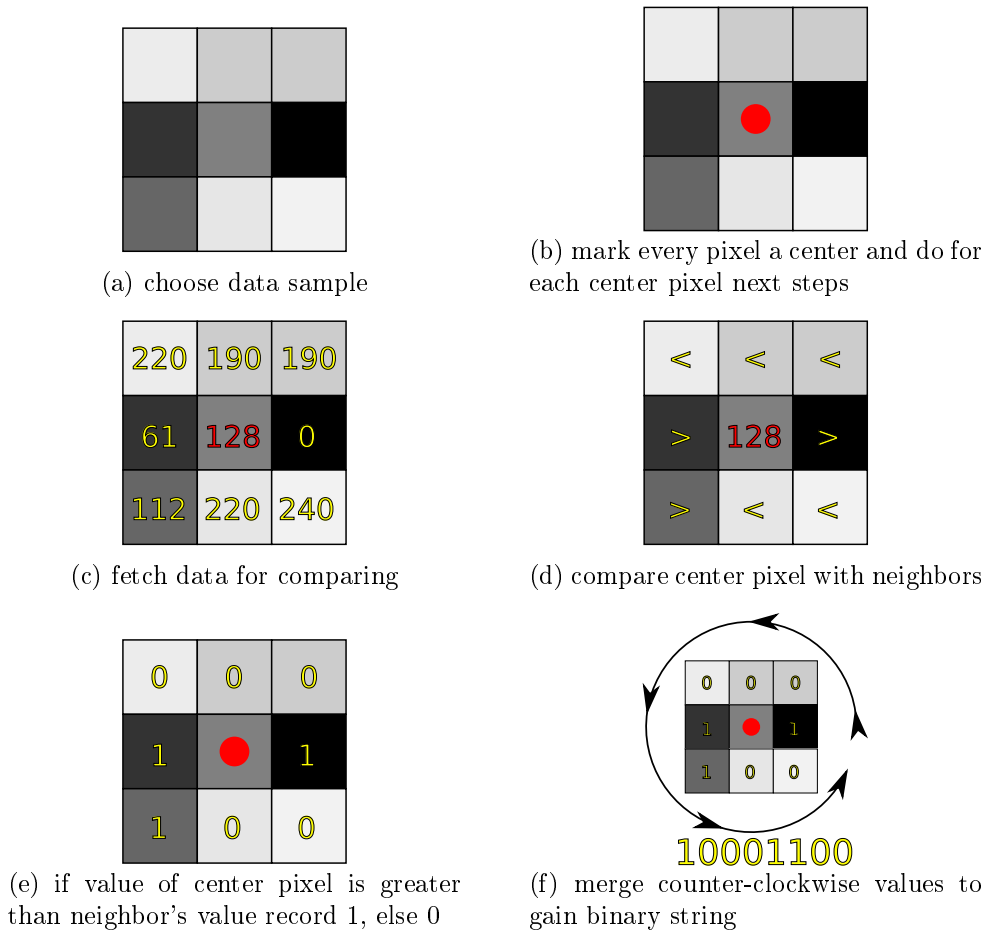
To fully understand principle of this algorithm, figure 4.2 presents usage on data sample for one pixel with 8 neighbors.

Implemented algorithm however is not the simplest case presented. It is variation called LBP-TOP (Local Binary Pattern on Three Orthogonal Planes) [16], which also includes 3rd dimension. LBP then has to be done for every plane (x-y plane, x-t plane and y-t plane) separately gaining three different histograms. Algorithm used in BOB computing LBP-TOP for video sequence also allows to specify additional parameters. Version for Matlab is declared as follows:

4.2 Reference implementation

```
function Histogram = LBPTOP(VolData, FxRadius, FyRadius,
                            TInterval, NeighbourPoints,
                            TimeLength, BorderLength,
                            bBilinearInterpolation,
```

Figure 4.2: LBP applied on random sample of image data



(a) choose data sample

(b) mark every pixel a center and do for each center pixel next steps

(c) fetch data for comparing

(d) compare center pixel with neighbors

(e) if value of center pixel is greater than neighbor's value record 1, else 0

(f) merge counter-clockwise values to gain binary string

Bincount , Code)

- **VolData** keeps brightness of individual pixels in three dimensions
- **FxRadius**, **FyRadius** and **TInterval** specify radius interval along X, Y and T axis. Only values 1, 2, 3 and 4 are accepted, values 1 and 3 are recommended. Pay attention to parameter **TInterval**, where TInterval*2 must be smaller than length of the video sequence.
- **NeighbourPoints** represents number of neighboring pixels to central pixel. This input is in form of three item array specifying number of neighbor pixels in planes XY, XT and YT. Only values 4, 8, 16 and 24 are accepted and recommended value is 8 for every plane, [8 8 8] as a argument.
- **TimeLength** and **BorderLength** are parameters for cropping center pixels in space and time. TimeLength is usually equal to TInterval and BorderLength to the greater one of FxRadius and RyRadius.
- **bBilinearInterpolation** is switch enabling bilinear interpolation when getting value of neighbor pixel in circle. When set to 1, bilinear interpolation is used, when set to 0, nearest interpolation is used.

- **Bincount** represents, how many values will be used in resulting histogram. If Bincount is set to 0, then number of resulting values is computed as 2^{YTNP} where YTNP is number of neighbor points in YT plane.
- **Code**: when Bincount is not 0, then this if transformation function for distributing $2^{NeighbourPoints}$ values into Bincount values.

Full implementation in Matlab can be seen in appendix B.

4.3 Parallel implementation

Parallel implementation in OpenCL was done twice. First implementation is naive and tries to mimic serial code as much as possible. Second implementation is done using knowledge of how is GPU architecture build and how certain optimizations can help improve performance.

Implementation consists of two parts, host code and kernel code. Host code is written in C++ with C++ OpenCL wrapper provided by Khronos Group. This wrapper significantly reduce boilerplate code and allows quicker development reducing cost and deploy time on devices.

4.3.1 Differences between naive and optimized kernel

Due to mobile architecture of development platform, the amount of possible optimizations is limited.

Passing arguments as constants at compile time

This optimization is to pass as many variables as possible as constants when building kernels. Optimizer can then unroll loops, optimize arithmetical operations and eliminate dead code if any.

Following parameters that can be passed at compile time.

- VALUES Number to indicate how many values will be produced by LBP. Used to specify size of local histograms.
- NEIGHBOURXY, NEIGHBOURXT, NEIGHBOURYT Values to be passed into loops. Optimizer can unroll the loops and boost performance

Host code changes only minimally and only in code snippet building kernel and passing arguments. Arguments that are passed as constants do not have to be passed again to kernel through parameters. Excess code for argument passing can be eliminated and kernel declaration can be simplified. Only thing to change in host code is building options string. For example instead of empty string passing $''-DVALUES = 256''$ ensures defining macro named $VALUES$ to have value 256. In kernel code then instead of using parameter $values$, $VALUES$ is used.

Using every work-item for computation

In naive kernel implementation every pixel of input video sequence refers to one work-item. In optimized kernel implementation every pixel producing output values refers to one work-item. In naive implementation border pixels that do not output values do not pass condition

and end without any useful output. In optimized version border pixels are ignored and only pixels with useful output are considered.

For video sequence 30 frames long, 640px wide and 480px high with 1px borders, over 7% of all work-items does not produce useful output in naive implementation. When considered same video sequence with 4px borders, over 27% of work-items does not produce useful output.

Size of work-group depends on dimensions of the problem. Work-group size in each dimension has to divide the size of problem in the corresponding dimension without any remainder. While work-group size in naive implementation depends only on video sequence size which is usually easily dividable by higher numbers, the work-group size in optimized implementation depends also on border size. As a result work-group size in optimized version is usually smaller than in naive implementation leaving some of work-items not utilized.

Global computation of shared values

Optimization is based on not computing trigonometric functions for every work-item separately, but to compute it once using CPU and distribute this information to all work-items.

Relative position of neighbors is same for all pixels throughout whole problem. Computing all values on CPU and then distributing table of all necessary coordinations is in this case faster than computing coordinates for every work-item separately.

Caching global data

This optimization is not implemented in optimized kernel, because of target platform does not have local memory on chip. Local memory is substituted by reserved memory are in global space. Utility clinfo shows this situation as followed:

- Local memory type: Global
- Local memory size: 32768

Host code doesn't have to change (except compilation options, discussed in section 4.3.1). Kernel code has to have initialization and data loading to cache, synchronization point and every read has to be altered to read from cache. Cache can be created as local array of values. In OpenCL 1.2 C language is limitation, that all local arrays has to be of constant size known at compilation time. This constants can be passed as compilation arguments, this is discussed in section 4.3.1.

Cache filling is heavily dependent on the type of application and will not be shown here, but strategy is to divide reading from global memory between all work-items evenly.

Kernel caching

Compilation of kernel take some time, especially when various optimizations are performed. Caching kernel and not compiling it on every run is way how to gain performance. This performance boost can be observable mainly with smaller data sets, when computation times are rather short.

For caching is possible to use static variables in simple cases or even more complex caching libraries taking into account also input parameters.

Figure 4.3: Code snippets of cache initialization and usage

```
//create local cache
__local float cache[AREAX][AREAY][AREAZ];

//synchronize work-items in work-group
barrier(CLK_LOCAL_MEM_FENCE);

//read from cache
float CenterVal = cache[coords.x][coords.y][coords.z];
```

Adjusting arithmetic operations

As every graphic architecture works in little bit different way, this optimization method heavily depends on accelerator architecture. On the architecture of the testing machine is code 4.5 slower than 4.4 despite the fact that is written on more lines.

Figure 4.4: Optimized accumulation of bits

```
if (currentVal >= CenterVal)
    BasicLBP += 2^8;
BasicLBP = BasicLBP >> 1;
```

Figure 4.5: Non-optimized accumulation of bits

```
if (currentVal >= CenterVal)
    BasicLBP += pow((float) 2, p);
```

The trick here is, that with uniform distribution of input data, function *pow*() is called in 50% of cases. In optimized version, shift is done always, but only by one place. Function *pow* receives arguments 2 and number from 0 to 7 for 8 neighbors. Average number of places to shift is 3.5 then.

Chapter 5

Experiments

Figure 5.1: Testing machine with mobile system architecture

All experiments are done on low-power architecture very similar to today's architectures of hand-held devices. Not by processor's ISA when experimental setup supports only AMD64 instruction set and most of mobile devices are ARM type, but by it's shared memory subsystem and low-power constraints of speculative execution in form of branch prediction and memory prefetch. Most of mobile devices also has shared memory between CPU and GPU. Similarly also experimental setup has shared memory in form of reserved separate memory space for GPU.

Full specification of device can be found in table 5.1

5.1 Experimental setup

All experiments were done in the very same environment with strictly set input data. Video sequence of faces was chosen as testing data. Data was converted from webm format to 8bit gray-scale as appropriate input to program. Width of video sequence is 512px and height 344px. For the sake of experiments, length of sequence was limited in source code to 10 frames.

Number of neighbors was set to 8 in every orthogonal plane as recommended. This creates histograms with 256 values, but also some other possible values were tested.

Radius in all orthogonal planes is set to 1px and thus borders are also set to 1px in every plane. Setting radius and borders to same value ensures, that algorithm will not read

Table 5.1: Experimental setup

OS:	Lubuntu 15.04
Kernel:	3.19.0
Device:	Lenovo G585
Processor:	AMD E1-1200 @ 1400MHz
Memory type:	DDR3 @ 1066 MHz effectively
Memory size:	4GB (GPU 384MB)
Graphics:	AMD Radeon HD 7310
Pipelines:	80 @ 500 MHz
Driver:	14.501
Compiler:	GCC 4.9.2

out of bounds. Reading out of bounds is safe on GPU returning border color, color of pixel in mirrored image or few other possibilities, but on CPU it could result in memory rules violation and ending program by OS.

Bilinear interpolation is enabled to show full potential of running algorithms on GPU, as it has electronic blocks dedicated to perform bilinear interpolation with almost no performance impact, whereas while computing bilinear interpolation on CPU, whole algorithm gets noticeably more complex.

Experiment was done ten times and table 5.2 shows arithmetic average of all values to eliminate OS timing and other differences between individual computations that can not be directly influenced.

5.2 Performance measurements

Optimization of arithmetic operations (4.3.1), optimal use of work-items (4.3.1), using constants (4.3.1) and pre-computed values (4.3.1) improved speed boost of order of magnitude against serial code. Kernel caching itself added speed boost twenty times.

Table 5.2: Measured speedup

Method	C++	Naive	Optimized
Total time [s]	341.836	14.037	6.968
Speedup	1	24.3525	49.058

In table 5.3 can be seen influence of kernel caching. First run of optimized code is faster than naive code due to all other optimizations, but kernel cache was just filled. Second and all following runs (in appendix C) of optimized code are much better from the perspective of performance. From the records can be determined, that kernel loading and compilation took about 400ms. It is relatively long time in comparison to total time of execution. Over 36% of time is accounted to kernel loading and compilation.

As table 5.3 shows, when caching take place, overall performance boost with using optimized techniques is more than twice the performance of unoptimized code and more than fifty times faster than serial code.

Table 5.3: Influence of kernel caching

i	**C++** [ms]	**Naive [ms]**	**Optimized [ms]**
0	38005	1637	1138
1	37966	1580	726

Chapter 6

Conclusion

The goal of this work was to explore and demonstrate algorithm acceleration in the area of face-based anti-spoofing. During research one convenient area of acceleration was discovered and it is image based operations. Image based operations are not specific to face-based anti-spoofing techniques, but accelerated algorithm LBP-TOP was designed to be used in this area.

It was shown on the architecture of GPU how data-parallel algorithms can be accelerated as well as what is needed to be done in order to achieve it. Utility written in C++ and OpenCL was developed as proof of concept and to measure performance boost. It is very useful utility showing developers, whether accelerate image based algorithms on various platforms or not. On systems with strong CPU and relatively weak GPU it might not be beneficial, but as shown low-power architectures should benefit.

6.1 Implications

Great performance boost gained from accelerating algorithms on GPU, as shown in this work, can be used to approach to ideal state of real-time face-based anti-spoofing, to make current systems smaller and cheaper or to allow more complex and secure algorithms to take place in production environment.

Due to really significant difference, developers should consider changing frameworks and tools to reflect heterogeneous systems better. Then it would be possible to develop algorithms directly using CUDA, OpenCL, OpenACC etc. or re-implement already developed algorithms to use data-parallel accelerators.

6.2 Utility, advantages, limitations

Utility was developed to measure performance gain from accelerated version of algorithms. Even though that one of the most important acceleration technique (caching global data 4.3.1) was not convenient to implement, performance gain is high. If this technique would be implemented, it would cause high performance gain on more advanced architectures with on-chip local memory, but due to low-power architecture and placing local memory into global memory, it would cause slowing down the algorithm.

It was not convenient to implement this accelerated algorithm into BOB framework due to nature of script-based systems and the fact, that initialization of program takes more time, than simple computation. In table C.1 is this relation visible. First run is

very slow and all other consequential runs are much faster. However, this effect of first slow run also observable after few minutes of activity other than OpenCL computation. This unpredictability needs to be solved before integrating OpenCL algorithms into BOB framework.

In the end, the developed tool is more 'proof of concept' than integrated solution, which limits its potential for practical use. We do not support any video processing library and thus data needs to be prepared to specific format to use. It is also convenient for video sequence to have dimension of $2^n + border * 2$ in every dimension for work-groups to scale optimally.

6.3 Future directions

The trend of accelerating algorithms is set and not only from this work can be seen why. Many professional programs already offer some kind of acceleration [1] [2] and many other are in stage of implementation [3].

This trend will lead to smaller form factors, higher computing capabilities, longer battery life and richer user experience.

Relatively new standard, HSA (heterogeneous system architecture) can further improve performance, decrease power consumption and enable new technologies like GPU OS.

Face-based authentication will play a big role in future and even now we can see first attempts[4] of deploying this technologies in practice. However this technique is cloud based with potential security risks.

MasterCard solution of security according to Valuewalk [5] is:

> The facial recognition software will also create a code which will be sent to MasterCard via the Internet
>
> Security experts at the company think that blinking is the best way to prevent fraud involving people showing photos of the card owner to a smartphone, and tricking the system.

Even though these measures ensures some level of security, real-time anti-spoofing would be great asset in such applications as banking or personal information sharing.

[1] https://blogs.adobe.com/premierepro/2012/05/opencl-and-premiere-pro-cs6.html
[2] http://www.macinchem.org/applications/gpuScience.php
[3] http://wiki.blender.org/index.php/OpenCL
[4] http://timesofindia.indiatimes.com/tech/tech-news/Your-selfie-will-soon-verify-online-payments/articleshow/47954441.cms
[5] http://www.valuewalk.com/2015/07/mastercard-selfie-app-verifies-online-payments/

Bibliography

[1] White Paper Advanced Micro Devices. COMPUTE CORES. http://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf , January 2014.

[2] Inc Animetrics. Authentication and Face-Recognition Services. http://animetrics.com/cloud-face-recognition-services/ , Retrieved 2015-03-10.

[3] A. Anjos, L. El Shafey, R. Wallace, M. G „unther, C. McCool, and S. Marcel. Bob: a free signal processing and machine learning toolbox for researchers. In *20th ACM Conference on Multimedia Systems (ACMMM), Nara, Japan*. ACM Press, October 2012.

[4] J. Beyer, E. Stotzer, A. Hart, and B. de Supinski. Openmp for accelerators. *OpenMP in the Petascale Era*, page 108–121, 2011.

[5] Ivana Chingovska, André Anjos, and Sébastien Marcel. On the effectiveness of local binary patterns in face anti-spoofing. September 2012.

[6] Jessica Dolcourt. Camera megapixels: Why more isn't always better. http://www.cnet.com/news/camera-megapixels-why-more-isnt-always-better-smartphones-u , May 4, 2013.

[7] Khronos Group. Khronos Releases OpenCL 2.0. https://www.khronos.org/news/press/khronos-releases-opencl-2.0 , July 22, 2013.

[8] JEDEC. DDR3 SDRAM standard JESD79-3F. http://www.jedec.org/standards-documents/docs/jesd-79-3d , July 2012.

[9] Mark Noel Gamadia Nasser Kehtarnavaz. *Real-time Image and Video Processing: From Research to Reality*. Morgan & Claypool Publishers, Jan 1, 2006.

[10] nVidia. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/index.html , March 5, 2015.

[11] openacc standard.org. OpenACC API FAQ. http://www.cray.com/sites/default/files/resources/OpenACC_213462.11_OpenACC_FAQ_FNL. , 2013.

[12] openacc standard.org. The OpenACC API. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf , August 2013.

[13] Jose Pagliery. MasterCard will approve purchases by scanning your face. http://money.cnn.com/2015/07/01/technology/mastercard-facial-scan/index.html , July 1, 2015.

[14] SkyBiometry. https://skybiometry.com/ .

[15] Pradheeba. P SruthiMol. P. Spoofing recognition for face with masks: Ananalysis. June 2015.

[16] Tiago de Freitas Pereira, André Anjos, José Mario De Martino1, Sébastien Marcel. LBP - TOP based countermeasure against face spoofing attacks. http://publications.idiap.ch/downloads/papers/2012/deFreitasPereira_LBP_2012.pdf , 2012.

Appendix A

CD content

- *bob* This directory contains specially edited versinon of BOB framework. To use it to compute LBP-TOP, first install all dependencies from:

  ```
  https://github.com/idiap/bob/wiki/Dependencies
  ```

  then use 'zc.buildout' method to install it:

  ```
  $ python bootstrap.py
  $ ./bin/buildout
  ```

  After installing BOB, insert testing database into 'database' directory inside 'bob' directory. this database can be obtained through

  ```
  https://idiap.github.io/bob/
  ```

  or on request. Recomended database set is 'replay' (https://www.idiap.ch/dataset/replayattack) Due to big size of this database set (3.3GiB) it was not possible to pack it on this CD.

  When database is on it's place, follow

  ```
  https://pypi.python.org/pypi/antispoofing.lbptop/
  ```

  for usage instructions.

- *data* Directory contains one sample video sequence and script converting video data into 8bit grayscale format to be processed by program. Script requires libavconv to be present on system, but can be easily rewritten to use ffmpeg instead.

- *doc* This directory contains all necessary files to build thesis.

  Research work is written using latex and can be build by running

  ```
  $ make
  ```

- *example* Contains executable file for testing performance boost. It also cantains two OpenCL files and data file. Executable is built for Linux AMD64 platform.

- *src* All source files needed to build program are placed in this directory. Before building it is necessary to change MAKEFILE to reflect directory of OpenCL includes and OpenCL linkable library.

- *README.TXT* Document describing content of CD with some useful instructions how to use individual files.

Appendix B

LBP-TOP matlab implementation

```
function Histogram = LBPTOP(VolData, FxRadius, FyRadius,
                            TInterval, NeighbourPoints,
                            TimeLength, BorderLength,
                            bBilinearInterpolation,
                            Bincount, Code)

% This function is to compute LBP_TOP features for a video sequence
% Copyright 2009 by Guoying Zhao & Matti Petikainen
% Matlab version was created by Xiaohua Huang
% Edited by Ondrej Benus

[height width Length] = size(VolData);

XYnPoints = NeighbourPoints(1);
XTnPoints = NeighbourPoints(2);
YTnPoints = NeighbourPoints(3);

if (Bincount == 0)
        % normal code
        nDim = 2^(YTNeighborPoints);
        Histogram = zeros(3, nDim);
else
        % uniform code
        Histogram = zeros(3, Bincount);
end

if (bBilinearInterpolation == 0)
  for t = TimeLength + 1 : Length - TimeLength
    for yc = BorderLength + 1 : height - BorderLength
      for xc = BorderLength + 1 : width - BorderLength
        CenterVal = VolData(yc, xc, i);

        %XY plane
        BasicLBP = 0;
```

```
FeaBin = 0;

for p = 0 : XYnPoints - 1
  X = floor(xc + FxRadius * cos((2 * pi * p) / XYnPoints) + 0.5)
  Y = floor(yc - FyRadius * sin((2 * pi * p) / XYnPoints) + 0.5)

  CurrentVal = VolData(Y, X, t);

  if CurrentVal >= CenterVal
    BasicLBP = BasicLBP + 2 ^ FeaBin;
  end
  FeaBin = FeaBin + 1;
end

% When Bincount is 0, then basic LBP-TOP is performed without
% performing changing distribution using Code lookup table

if Bincount == 0
  Histogram(1, BasicLBP + 1) = Histogram(1, BasicLBP + 1) + 1;
else
  Histogram(1, Code(BasicLBP + 1, 2) + 1) =
                    Histogram(1, Code(BasicLBP + 1, 2) + 1) + 1;
end

%XT plane
BasicLBP = 0;
FeaBin = 0;

for p = 0 : XTnPoints - 1
  X = floor(xc + FxRadius * cos((2 * pi * p) / XTnPoints) + 0.5)
  Z = floor(t + TInterval * sin((2 * pi * p) / XTnPoints) + 0.5)

  CurrentVal = VolData(yc, X, Z);

  if CurrentVal >= CenterVal
    BasicLBP = BasicLBP + 2 ^ FeaBin;
  end
  FeaBin = FeaBin + 1;
end

% When Bincount is 0, then basic LBP-TOP is performed without
% performing changing distribution using Code lookup table

if Bincount == 0
  Histogram(2, BasicLBP + 1) = Histogram(2, BasicLBP + 1) + 1;
else
  Histogram(2, Code(BasicLBP + 1, 2) + 1) =
                    Histogram(2, Code(BasicLBP + 1, 2) + 1) + 1;
```

```
        end

        %YT plane
        BasicLBP = 0;
        FeaBin = 0;

        for p = 0 : YTnPoints - 1
          Y = floor(yc - FyRadius * sin((2 * pi * p) / YTnPoints) + 0.5)
          Z = floor(t + TInterval * cos((2 * pi * p) / YTnPoints) + 0.5)

          CurrentVal = VolData(Y, X, t);

          if CurrentVal >= CenterVal
            BasicLBP = BasicLBP + 2 ^ FeaBin;
          end
          FeaBin = FeaBin + 1;
        end

        % When Bincount is 0, then basic LBP-TOP is performed without
        % performing changing distribution using Code lookup table

        if Bincount == 0
          Histogram(3, BasicLBP + 1) = Histogram(3, BasicLBP + 1) + 1;
        else
          Histogram(3, Code(BasicLBP + 1, 2) + 1) =
                                Histogram(3, Code(BasicLBP + 1, 2) + 1) + 1;
        end

      end
    end
  end
else % billinear interpolation
  for t = TimeLength + 1 : Length - TimeLength
    for yc = BorderLength + 1 : height - BorderLength
      for xc = BorderLength + 1 : width - BorderLength
        CenterVal = VolData(yc, xc, i);

        %XY plane
        BasicLBP = 0;
        FeaBin = 0;
        for p = 0 : XYnPoints - 1
          %bilinear interpolation
          %float = single -- in matlab
          x1 = single(xc + FxRadius * cos((2 * pi * p) / XYnPoints));
          y1 = single(yc - FyRadius * sin((2 * pi * p) / XYnPoints));

          u = x1 - floor(x1);
          v = y1 - floor(y1);
```

```
  ltx = floor(x1);
  lty = floor(y1);
  lbx = floor(x1);
  lby = ceil(y1);
  rtx = ceil(x1);
  rty = floor(y1);
  rbx = ceil(x1);
  rby = ceil(y1);

  CurrentVal = floor(
                    VolData(lty, ltx, t) * (1 - u) * (1 - v) +
                    VolData(lby, lbx, t) * (1 - u) * v +
                    VolData(rty, rtx, t) * u * (1 - v) +
                    VolData(rby, rbx, t) * u * v
                    );

  if CurrentVal >= CenterVal
    BasicLBP = BasicLBP + 2 ^ FeaBin;
  end

  FeaBin = FeaBin + 1;
end

% When Bincount is 0, then basic LBP-TOP is performed without
% performing changing distribution using Code lookup table

if Bincount == 0
  Histogram(3, BasicLBP + 1) = Histogram(3, BasicLBP + 1) + 1;
else
  Histogram(3, Code(BasicLBP + 1, 2) + 1) =
                    Histogram(3, Code(BasicLBP + 1, 2) + 1) + 1;
end

%XT plane
BasicLBP = 0;
FeaBin = 0;
for p = 0 : XTnPoints - 1
  %bilinear interpolation
  %float = single -- in matlab
  x1 = single(xc + FxRadius * cos((2 * pi * p) / XTnPoints));
  z1 = single(t + TInterval * sin((2 * pi * p) / XTnPoints));

  u = x1 - floor(x1);
  v = z1 - floor(z1);
  ltx = floor(x1);
  lty = floor(z1);
  lbx = floor(x1);
  lby = ceil(z1);
```

```
  rtx = ceil(x1);
  rty = floor(z1);
  rbx = ceil(x1);
  rby = ceil(z1);

  CurrentVal = floor(
                    VolData(yc, ltx, lty) * (1 - u) * (1 - v) +
                    VolData(yc, lbx, lby) * (1 - u) * v +
                    VolData(yc, rtx, rty) * u * (1 - v) +
                    VolData(yc, rbx, rby) * u * v
                    );

  if CurrentVal >= CenterVal
    BasicLBP = BasicLBP + 2 ^ FeaBin;
  end

  FeaBin = FeaBin + 1;
end

% When Bincount is 0, then basic LBP-TOP is performed without
% performing changing distribution using Code lookup table

if Bincount == 0
  Histogram(3, BasicLBP + 1) = Histogram(3, BasicLBP + 1) + 1;
else
  Histogram(3, Code(BasicLBP + 1, 2) + 1) =
                    Histogram(3, Code(BasicLBP + 1, 2) + 1) + 1;
end

%YT plane
BasicLBP = 0;
FeaBin = 0;
for p = 0 : YTnPoints - 1
  %bilinear interpolation
  %float = single -- in matlab
  y1 = single(yc - FyRadius * sin((2 * pi * p) / YTnPoints));
  z1 = single(t + TInterval * cos((2 * pi * p) / YTnPoints));

  u = y1 - floor(y1);
  v = z1 - floor(z1);
  ltx = floor(y1);
  lty = floor(z1);
  lbx = floor(y1);
  lby = ceil(z1);
  rtx = ceil(y1);
  rty = floor(z1);
  rbx = ceil(y1);
  rby = ceil(z1);
```

```
            CurrentVal = floor(
                                VolData(ltx, xc, lty) * (1 - u) * (1 - v) +
                                VolData(lbx, xc, lby) * (1 - u) * v +
                                VolData(rtx, xc, rty) * u * (1 - v) +
                                VolData(rbx, xc, rby) * u * v
                                );

            if CurrentVal >= CenterVal
              BasicLBP = BasicLBP + 2 ^ FeaBin;
            end

            FeaBin = FeaBin + 1;
          end

          % When Bincount is 0, then basic LBP-TOP is performed without
          % performing changing distribution using Code lookup table

          if Bincount == 0
            Histogram(3, BasicLBP + 1) = Histogram(3, BasicLBP + 1) + 1;
          else
            Histogram(3, Code(BasicLBP + 1, 2) + 1) =
                                Histogram(3, Code(BasicLBP + 1, 2) + 1) + 1;
          end

        end
      end
    end
end

%% normalization
for j = 1 : 3
  Histogram(j, :) = Histogram(j, :)./sum(Histogram(j, :));
end
```

Appendix C

Experiment record

```
Getting device information.....
Default device is:
        name: Loveland
        type(4): GPU
        3D images:
                max width: 2048
                max height: 2048
                max depth: 2048
        max WG size: 256
                x: 256
                y: 256
                z: 256
Checking all requirements.
        Device available........yes
        Device compiler available........yes
        Support for images........yes
        3D image width........yes
        3D image height........yes
        3D image depth........yes
        Local atomics........yes
Device is able to run kernel: yes

Loading video into memory.
Data(26419200) loaded.
Testing....ok
```

	kernel	naive	native	speedup	
i	[ms]	[ms]	[ms]	kernel	naive
0	1138	1637	38005	33.3963	23.2162
1	726	1580	37966	52.2948	24.0291
2	762	1560	38007	49.878	24.3635
3	718	1545	37981	52.8983	24.5832
4	730	1552	37976	52.0219	24.4691

```
|    5|      721|     1550|    38005|52.7115|24.5194|
|    6|      720|     1533|    37976|52.7444|24.7723|
|    7|      728|     1543|    37946|52.1236|24.5924|
|    8|      725|     1537|    37974|52.3779|24.7066|
+----+---------+---------+---------+-------+-------+
|SUM|     6968|    14037|   341836|       |       |
|AVG|         |         |         |  49.058|24.3525|
+----+---------+---------+---------+-------+-------+
```

Table C.1: Time of execution with -DEARLY_EXIT compiler option

i	*real*	*user*	*sys*
0	17.156	0.756	0.156
1	0.753	0.608	0.052
2	0.761	0.604	0.048
3	0.746	0.584	0.068
4	0.757	0.580	0.068
5	0.748	0.612	0.044
6	0.928	0.588	0.072
7	0.758	0.608	0.052
8	0.778	0.576	0.072
9	0.764	0.600	0.060