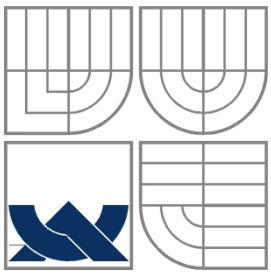
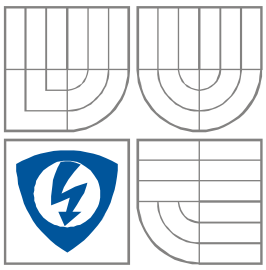


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV JAZYKŮ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF LANGUAGES

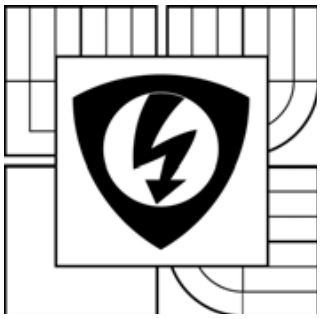
FAST SIGNAL PROCESSING
RYCHLÉ ZPRACOVÁNÍ SIGNÁLŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — Ivo Rychlý
AUTOR PRÁCE

SUPERVISOR — Ing. Jan Mikulka, Ph.D.
VEDOUCÍ PRÁCE

BRNO, 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav jazyků

Bakalářská práce

bakalářský studijní obor
Angličtina v elektrotechnice a informatice

Student: Ivo Rychlý ***ID:*** 136109

Ročník: 3 ***Akademický rok:*** 2014/15

NÁZEV TÉMATU:

Fast Signal Processing

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je seznámení se s jednoduchými metodami zpracování signálů/obrazů a jejich implementace v prostředí Matlab. Součástí práce bude literární rešerše v oblasti využití rychlých metod pro zpracování signálů pomocí grafického procesoru s platformou CUDA/OpenCL.

DOPORUČENÁ LITERATURA:

[1] MOURA, Jose. What is signal processing? [President's Message. *IEEE Signal Processing Magazine* [online]. 2009, vol. 26, issue 6.

[2] PERROT, Gilles, Stéphane DOMAS a Raphaël COUTURIER. Fine-tuned High-speed Implementation of a GPU-based Median Filter. *Journal of Signal Processing Systems* [online]. 2014, vol. 75, issue 3.

Termín zadání: *9.2.2015* ***Termín odevzdání:*** *22.*5.2015

Vedoucí práce: Ing. Jan Mikulka, Ph.D.
Konzultanti bakalářské práce: *Kenneth Froehling, M.A.*

doc. PhDr. Milena Krhutová, Ph.D.
předseda oborové rady

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

ABSTRACT

An increasing amount of data in modern image processing requires a new approach to algorithms. Parallelization of an algorithm is one way how to speed up the processing speed. Architectures like CUDA and OpenCL facilitate the parallelization of the algorithms for graphics processing units and they bring parallel computing to a broader audience. This thesis is about basics of image processing, parallelization and algorithms and how parallelization effects the speed of computing.

KEYWORDS

algorithms, parallelization, OpenCL, CUDA, image processing

ABSTRAKT

Zvětšující se množství dat v moderním zpracování obrazu vyžaduje nové postupy ve vytváření algoritmů. Jedním z postupů jak zrychlit algoritmus je paralelizace. Architektury jako CUDA a OpenCL ulehčují paralelizaci algoritmů pro grafické karty a otevírají paralelní zpracování širší veřejnosti. Tato práce se zabývá základy zpracování obrazu, paralelizace a algoritmů a jak paralelizace ovlivňuje rychlost zpracování.

KLÍČOVÁ SLOVA

algoritmy, paralelizace, OpenCL, CUDA, zpracování obrazu

PROHLÁŠENÍ

Prohlašuji, že svoji bakalářskou práci na téma „Fast Signal Processing“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne ....

(podpis autora)

Acknowledgment

I would like to thank my supervisor Ing. Jan Mikulka, Ph.D. for his help and guidance with this thesis and also my language supervisor Kenneth Froehling, M.A.

Bibliografická citace díla:

RYCHLÝ, Ivo. *Fast Signal Processing: Bakalářská práce.* Brno, 2015. Vedoucí práce: Ing. Jan Mikulka, Ph.D., FEKT VUT v Brně.

Content

Introduction

Signal processing is a broad field of study as it is happening in almost everything as the definition can be applied to many things. Our brains are, for example, signal processing machines. The image processing is part of signal processing and can be conducted on both analog and digital images. The digital image processing was for a long time held back by huge computer requirements that were necessary for real time computation of some complex image processing techniques. The introduction of parallel computing opened new possibilities for image processing as the power of personal computers was rising exponentially with time. Today the customer grade hardware is able to calculate huge amounts of data and programming interfaces like CUDA and OpenCL allowed a broader audience to utilize this power by simplifying the parallelization process of algorithms for graphics processing units (GPUs). Thanks to that the parallelization of a sequential code for multiple cores is now easier. This work focuses on the image processing algorithms and their parallelization for GPUs with the use of CUDA and OpenCL and on comparison of the parallel algorithms to their sequential counterparts.

Firstly, the signal processing is described with basic examples and processing scheme. Also the image processing itself is more closely introduced with some techniques that fall into this category, mainly the median filtering which is a technique used in the experimental part. Then the parallel computing is introduced with basic GPU core architecture of the Geforce GTX 770. There are also CUDA and OpenCL in this chapter as they are the interfaces used to create prallel algorithms for the devices. The algorithms are then brought to more detail in the next chapter along with research about the use of the parallel algorithms in scientific papers. Finally an experimental part is conducted which compares the processing speeds of the parallel algorithms to the sequential ones on two separate devices. Also a comparison of the used hardware is done as both devices ran the same algorithms. The main goal of the experimental part is to see a considerable speed up of the parallel algorithms.

1 Signal processing

Signal processing is a term used generally for anything that generates, transforms, extracts or interprets information from a signal. The signal, however, is not only some physical form of an information that changes with time and/or space. Signals can be dealing with symbolic or abstract information like slow and fast. Signals, therefore, include almost anything you can think of, from the physical world, for example, audio, video, communication, radar, sonar, geophysical, biological, chemical, musical and other data. Signal processing is then the use of mathematical, statistical, computational, heuristic, and/or linguistic representations to generate, transform, transmit and learn from the signals. [1]

Signal is represented by 2D or 3D graph. Typical 2D signal is a sound recording, which can be easily put into graph which displays amplitude (the volume of the sound) over a period (time).

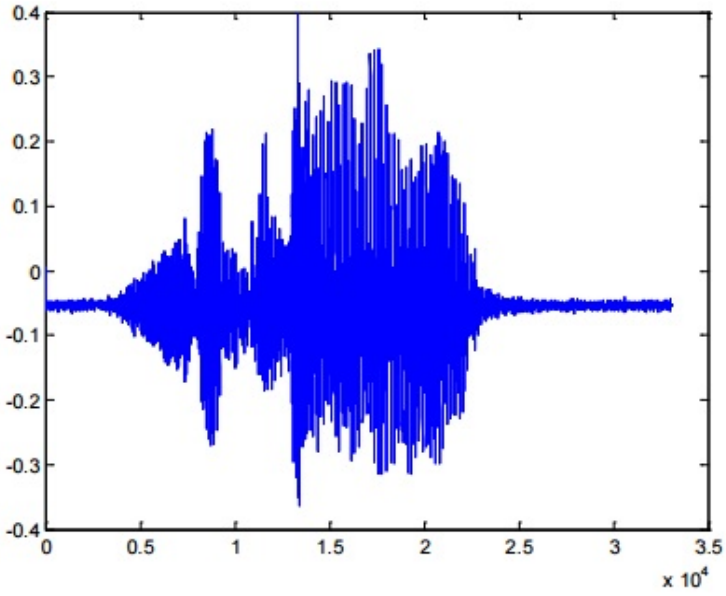


Fig. 1. Signal of a word “signál” [2]

Fig. 1 is a typical image assosciated with signals as this 2D representation is used in many fields like seismology, where the amplitude spikes show the earthquake. When dealing with an image a 3D representation can be used.

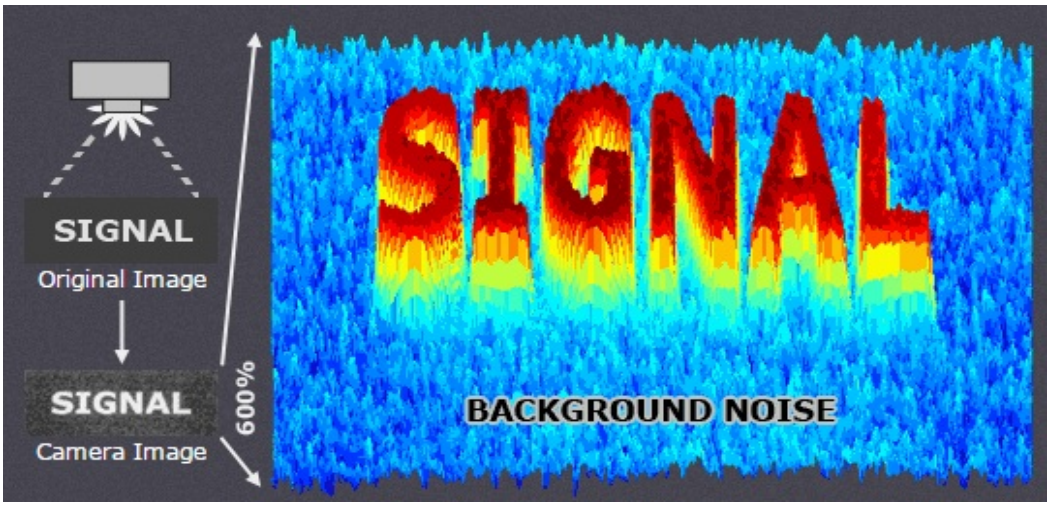


Fig. 2. 3D representation of digital camera image [3]

Fig. 2 shows an image taken by digital camera which automaticly enhances the image from the captured signals.

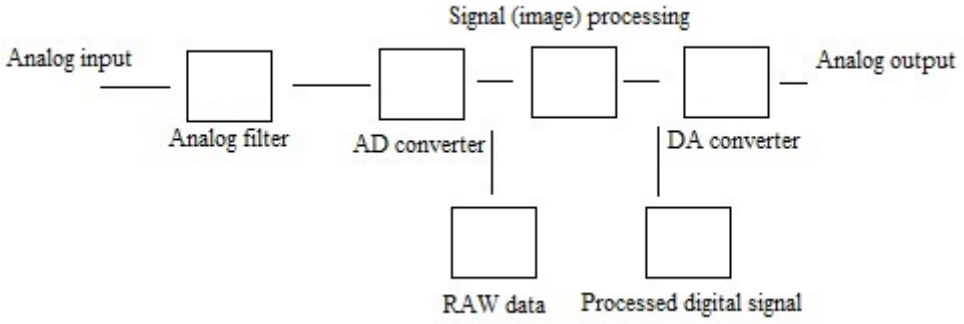


Fig. 3. Typical signal processing scheme [2]

Fig. 3 shows a scheme of signal processing chain. There is an analog input (an image) which is taken by the analog filter which can be any device that captures the analog signal. In digital cameras it can be, for example, a charge-coupled device (CCD) which is the image sensor that captures the light. This device also usually contains the AD converter which is an analog to digital signal converter. After these steps a raw data (also known as primary data) file is obtained which is the unprocessed digital data of the signal. The signal processing (image processing) is then applied on this raw data and the processed digital signal is obtained. In some cases an analog otput is required and therefore the digital to analog converter is used to obtain an analog signal.

1.1 Image processing

Image processing is a form of signal processing where after the image is processed, the final product is an image or set of data about the image. Into this category falls every image operation, for example, rotating the image, changing color schemes and applying filters, or some more advance techniques like noise removal, edge detection, etc.. It is an important topic since it is a mean of understanding the data for a large number of applications. This wide variety leads to the evolution of specific analysis and processing techniques that differ with the type of the image and/or with the task that needs to be performed.

The first step in image processing is the acquisition of the image using some kind of sensor (camera). This data is then processed and digital image is created. Typical cameras create images that are two-dimensional representations of the three-dimensional visual world.

However, not all images are without faults and problems. Images, among other things, can have an unwanted signal called noise. Some examples of this would be blurred image from wrongly focused camera or the blur that happens if the objects moves (or the camera). Signals also usually have a background noise that is mainly introduced to the signal by the mechanism for gathering the data (like the CCD) or from damaged source (for example a film grain would produce noise). When dealing with noise a signal to noise ratio (SNR) is measured. This ratio shows how much stronger the actual signal is compared to the noise. The background noise is usually neglected as it has low value compared to the actual signal. There are several methods of image processing to enhance the quality of the image and to reduce or eliminate the noise completely. Some image processing techniques are:

1) histogram equalization
2) highpass and lowpass filters
3) morphological gradient.

1) Histogram equalization is in image processing used to adjust the image intensity to enhance the contrast of the image. This method is often used in science imaging as it operates best in gray scale color depth (black and white) and it can highlight images

with low contrast. This technique can also be used on color images (on each layer of RGB) but it usually creates an unrealistic picture. Modified histogram equalizers can be used in other fields such as audio signal processing to compensate for noise distortions.

Fig. 4. Application of histogram equalization on the left images.
Top one represents a bad equalization as the image was overexposed
The bottom is then a good example of equalization [4]

2) Highpass and lowpass filters are used to attenuate low and high frequencies respectively. In image processing a lowpass filter can be used to remove noise and to smoothen the details of the image. The highpass filter is good for edge detection as edges have higher frequencies. The mix of these to filters can be used and it is called bandpass filter. This filter can have the cut off frequencies set on both sides of the frequency range.

3) Morphological gradients are used in image processing to enhance the intensity variations in images. The intensity of a pixel and its neighbors is enhanced and the variations are located. These variations are assumed to be edges of the objects and that is why this method is used mainly in edge detectors

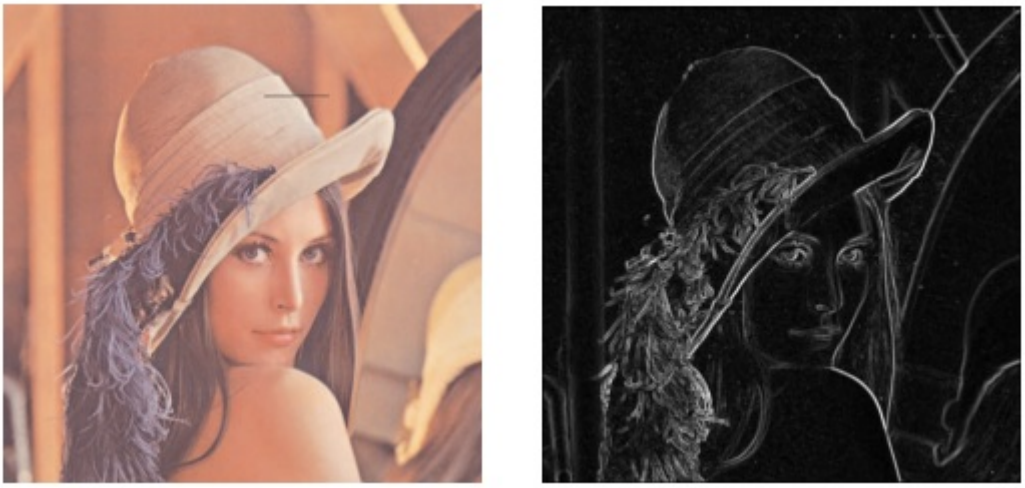


Fig. 5. Example of colour morphological gradient operator for edge detection [5]

The computing requirements of these algorithms increase as these techniques get more and more complicated. Today our technology allows for large data acquisition and, therefore, high computational power is required to process these data. To process larger, more detailed images in the same time frame a new approach has to be taken. One of these approaches is parallel computing.

1.2 Median filtering

In the experimental part a median filter, which is a nonlinear digital filtering technique often used to remove noise (salt-and-pepper denoising) from the image, is going to be used. It is a great filter to reduce the power of noise while minimizing the edge blurring. The median filtering takes a pixel and replaces its gray-level value with a median value of its neighbors [6]. The number of neighboring pixels, that is the median counted from, is set by the user and depends on what wants to be achieved and how the image looks. The main drawbacks of this filter in the past were its computing complexity and non linearity which was a problem on sequential hardware (longer processing times). This filter can be parallelized in terms of burst calculating the input pixel. The algorithm will allow us to calculate all the neighboring pixels at the same time. For example, in the 5×5 filter the sequential algorithm would have to go pixel by pixel to calculate the value so it would need 24 operations to get each neighboring pixel. This can be done with parallelization on CUDA accelerated GPU in one go on 24 cores at the same time,

decreasing the processing time. The more neighboring pixels is taken the higher impact will parallelization have until a place is reached, where the number of pixels is higher than the number of processors.

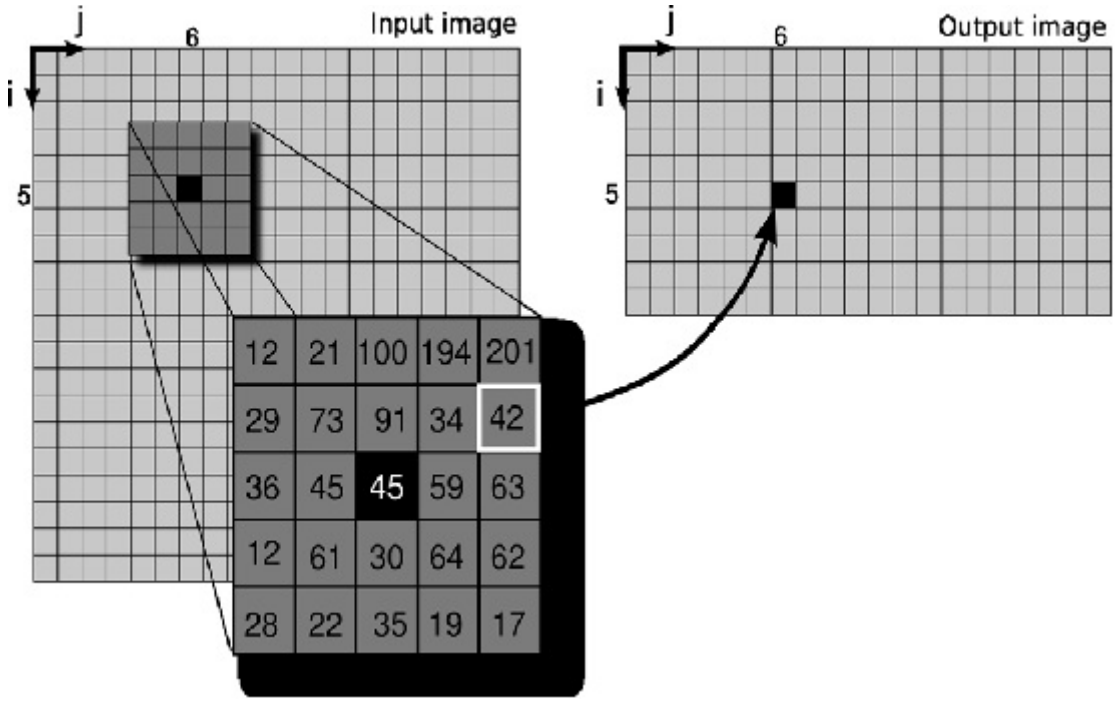


Fig. 6 Illustration of 5 × 5 median filtering, applied on pixel coordinates (5,6). [6]

2 Parallel computing

Parallel computing is presently used in almost every device as the modern central processing unit (CPU) and the graphics processing unit both use parallel computing. In the beginning parallel computing was only used in highly specialized devices and only by a few enthusiasts. As its development progressed CPUs started to reach its limits with a clock rate around 4 GHz, where heat generation and power consumption became increasingly inefficient, CPU manufacturers began to introduce multiple cores to overcome this limit. This meant that CPUs needed programs that could utilize these additional cores. However, writing parallel computer programs is more difficult than writing sequential ones, because concurrency introduces several new classes of potential software bugs. In addition the synchronization and the following communication between the different subtasks are the greatest obstacles for getting good parallel program performance.

That is why, even with multi-core processors, not many programs actually used them as creating the parallel program was more difficult. With this the general purpose graphic processing units (GPGPU), which can be used for other tasks (i.e. not exclusively graphics) with their distinguishing feature of bidirectional information transfer, started to become more practical. As they became more powerful and the number of cores increased, the complexity of programs was getting larger and a good knowledge of OpenGL or DirectX was required. This requirement was keeping away the general public from utilizing the power of GPUs and that is why universal architectures were introduced to help programmers with creating programs on such devices. Two of the most used architectures today are OpenCL and CUDA. [7]

2.1 GPU Architecture

There are many types of GPU architectures and GPU cores which are usually changed with every new series the GPU manufacturers releases. Because the experimental part is done on Nvidia GPUs, namely GTX 660 Ti and GTX 770, the focus will be on Kepler GK 104 architecture, which both of these cards with slight variations have. The GTX

660 Ti has slightly modified GK 104 that features less cores and lower clock rate than the full GK 104 that is in, for example, GTX 680 and GTX 770. The number of CUDA cores in GTX 660 Ti is 1344 with 915 MHz base clock rate compared to 1536 CUDA cores with 1046 MHz base clock rate in GTX 770.

The Kepler micro-architecture is a successor to the Fermi micro-architecture and focuses mainly on energy efficiency by increasing the performance per watt of energy consumed. The Kepler is made by 28 nm process and is in some GTX 6 series and 7 series GPUs with varying configurations (e.g. GK104 and GK110). The Kepler introduces a new generation of Streaming multiprocessors (SMX) where most of the advancement has been made.

The GPU cores of Nvidia GPUs are divided into several blocks. Each CUDA core is a Streaming processor (SP) inside of a Streaming multiprocessors (SM or SMX) which are part of Graphics Processing Clusters (GPCs) which are the big units that form the whole GPU core. The GTX 770 GK 104 has eight streaming multiprocessors each containing two arrays of 96 streaming processors in four Graphics Processing Clusters (therefore, 192 SP for each SMX, two SMX for each GPC and a total of 1536 SP in eight SMX and four GPC in GTX770)

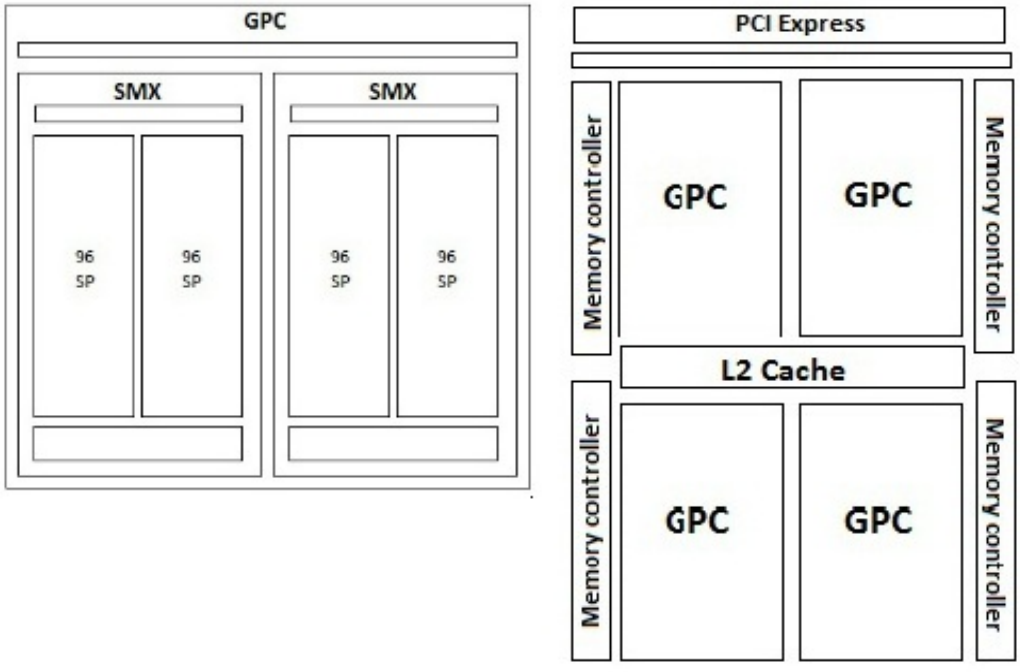


Fig. 7. Simplified view of GPC (on the left) and GPU core (on the right)

2.2 CUDA

CUDA is parallel computing architecture developed by NVIDIA for their own hardware. CUDA is used for parallel processing and, of course, only on NVIDIA GPUs, for which it is highly optimized. CUDA has two APIs (application programming interface): Runtime API and Driver API. These APIs are there to make the programming process much easier and more open to general public with C/C++, Fortan and Python as the supported programming languages with other third party language extensions. CUDA also contains large number of libraries, performance analysis tools, debugging solutions, etc. This means that you can accomplish more complicated functions and achieve better performance without the need of knowledge associated with programming for GPUs. [8]

CUDA is uasually classified as SIMD (single instruction, multiple data) architecture. This architecture means that one operation (kernel) is executed on many cores simultaneously. However, most of the hardware is also MIMD (multiple instruction, multiple data) which means that each processor can process different data at the same time and be independent of one and another. That is why the sub category SPMD (single program multiple data) was created which is basicaly the same as SIMD with a bit different approach to cores, etc. The SIMP and SPMD are prefered techniques as they are easier and more straight foward to parallelize compared to MIMD where for each core a different code needs to be written.

Kernels are used in CUDA, which are programs that manage the software requests for CPU and GPU and execute the program X times by X threads. In language C the kernel is defined using the __global__ key word.

A simple example of a part of a code for vector adding on CPU and on GPU.

An example of C code for vector adding [9]:

```
	#define SIZE		1024

void	 VectorAdd (int*a, int*b, int*c, int n )
{
	int i;

	for( i = 0; i < n: ++i)
		c[i] = a[i] + b[i];
}
```

```
int main()
{
	int *a, *b, *c;

	a = (int  *)malloc(SIZE*sizeof(int));
	b = (int  *)malloc(SIZE*sizeof(int));
	c = (int  *)malloc(SIZE*sizeof(int));

	for( int i = 0; i < SIZE; ++i )
	{
		a[i] =  i;
		b[i] =  i;
		c[i] =  0;
	}

	VectorAdd(a, b, c, SIZE);

	for( int i = 0; i < 10; ++i)
		printf("c[%d] = %d\n", i, c[i]);

	free(a);
	free(b);
	free(c);

	return 0;
}
```

An example of CUDA C code for vector adding on GPU (modified previous C code) [9]:

```
	#define SIZE	1024

__global__  void    VectorAdd (int*a, int*b, int*c, int n )
{
	int i = threadIdx.x;

	if (i < n)
		c[i] = a[i] + b[i];
}

int main()
{
	int *a, *b, *c;
	int *d_a, *d_b, *d_c;

	a = (int  *)malloc(SIZE*sizeof(int));
	b = (int  *)malloc(SIZE*sizeof(int));
	c = (int  *)malloc(SIZE*sizeof(int));

	cudaMalloc( &d_a, SIZE*sizeof(int));
	cudaMalloc( &d_b, SIZE*sizeof(int));
	cudaMalloc( &d_c, SIZE*sizeof(int));

	for( int i = 0; i < SIZE; ++i )
	{
		a[i] =  i;
		b[i] =  i;
		c[i] =  0;
	}

	cudaMemcpy( d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice );
	cudaMemcpy( d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice );
	cudaMemcpy( d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice );
```

```
        VectorAdd<<< 1,  SIZE >>>  (d_a, d_b, d_c, SIZE);

        cudaMemcpy( d_c, c, SIZE*sizeof(int), cudaMemcpyDeviceToHost );

        for( int i = 0; i < 10; ++i)
              printf("c[%d] = %d\n", i, c[i]);

        free(a);
        free(b);
        free(c);

        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_c);

        return 0;
}
```

The key word `__global__` is used to indicate that the program is going to be executed on the GPU. The code `threadIdx.x` is there to indicate on which thread the code is going to be executed on. Each kernel gets a unique ID which is called upon by the `threadIdx` key word. The thread ID can be identified by using up to three-dimensional coordinates. The code changes slightly by adding appropriate coordinates (`threadIdx.y.z`) in x,y and z axis. This threads then can be arranged in to a block with up to 1024 threads (this number varries by the GPU used etc.). This one thread block is then executed on the SPs. The number of threads is specified in the `<<< 1,  SIZE >>>` code word, where in our case the SIZE was defined at the start of the code and shows the number of threads. The number one in the code word is the number of blocks used. In order to identify the blocks a similar code word as with the threads (`blockIdx.y`) is used. The code `MatAdd<<<numBlocks, threadsPerBlock>>>` is then used to handle multiple blocks with variable number of threads. Other differences compared to the standard code is the memory handling. Two memories are in play here (CPU and GPU) and that is why the memory needs to be allocated also on the GPU by the `cudaMalloc` code. This also enables the GPU to send error data back to the CPU if there is a problem with the code. Then the code `cudaMemcpy( d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice )` is used for copying the data on to the GPU memory. The last part of the code is the one that tells the CPU that the data needs to be copied from CPU (host) to the GPU (device). After the code is processed on the GPU another `cudaMemcpy` code is used, however in a reversed order (device to host), which copies the results from the GPU memory and allows the CPU to access the results of the program. There are of

course many other ways how to approach each code, however, this small example shows that parallelization of basic programs with CUDA is relativly simple and is not much different from basic C language.

2.3 OpenCL

OpenCL (Open Computing Language) is an open standard for cross-platform (heterogeneous) parallel programing language managed by Khronos Group, a non-profit consortium. Heterogeneous computing refers to a system that uses several types of processors at the same time, e.g. CPU and GPU. OpenCL uses modified C99 programming language and also includes APIs, libraries and a run-time systems. Programmers can then use OpenCL to write portable code where the hardware related details are being dealt with automatically by the OpenCL run-time environment.
OpenCL provides two ways of parallel processing: Task-parallel and Data-parallel. For the use of GPGPUs and image processing the Data-parallel processing is the most used one as it can work with hundreds of cores.[7]
The OpenCL language is similiar to the CUDA language, where different key words are used to perform the same tasks. For example, instead of the CUDAs `cudaMalloc()`, a code `clCreateBuffer()` is used. A different code word is also used to identify that it is going to be an OpenCL kernel. Instead of CUDAs `__global__` a `__kernel` is used. The differences come from using a different programming languages, however, the basic idea is the same as they both use C language as the mother language. There are some other differences mainly coming from the fact that OpenCL is not only GPU focused. When starting with OpenCL the code `clGetPlatformIDs` shows all available platforms for OpenCL. Which is just to name a few differences. The focus of OpenCL is that the code written, for example, for CPU, should with simple device change in the code (to tell the kernel to use different device) and small change in memory handling, run on any other supported device like GPU.

3 Algorithms

Computer programs for these architectures are composed out of algorithms. Depending on the complexity of the task (in this case filters) an appropriate algorithm needs to be used. An algorithm is a process for solving defined problems in a finite number of steps. From the first step (state, input) the operation goes step by step through the instructions (rules) of the algorithm to produce an output.

Algorithms must have the properties as mentioned above: finite number of operations, have all steps defined and have at least one output. Depending on the type of the algorithm there can be, for example, an algorithm that is repeating itself (recursive); one that is repeating only one part (block) of itself (iteration); algorithms that depend on previous values and performs them the same way every time (deterministic); or those that can have multiple outcomes (non-deterministic). A good algorithm is the one that can get to the output with the least amount of steps in the fastest way possible.

Making a good algorithm requires optimization for a certain type of hardware. An algorithm can perform faster on a slower machine for which it is optimized than on much powerful machine. So, when designing an algorithm, the type of hardware the algorithm is going to be running on, like processing units (CPU, GPU), the amount of memory the machine has etc. needs to be taken into account (i.e. where OpenCL and mainly CUDA help with the optimization). In computer science there is an algorithm analysis for which the sole purpose is to find out the best way to approach the designing of an algorithm for certain task on certain hardware. In algorithm analysis, for example, cost models (generally uniform and logarithmic models), assign cost to every operation. With that the best way how to approach the problem and design an appropriate algorithm can be determined. In order to use an algorithm in parallel computing, the algorithm itself needs to be parallelized (CUDA, OpenCL platforms). Parallelization offers in most of the cases a decrease in computational time, however, some algorithms are harder to parallelize than others.

4 Research

The research chapter is divided in to five subchapters. The chapter 4.1 is brief history of parallelization. The chapter 4.2 shows how memory handling can speed up the algorithms and how it is a important part of the parallelization process. The chapter 4.3 deals with accuracy of the computations and how important it is to consider accuracy when dealing with precise computations. The chapter 4.4 deals with efficiency of parallel computing. And the chapter 4.5 shows an algorithm that can automaticaly parallelize other algorithms.

4.1 Parallelization

Parallelization was at first done mainly by scientists and reaserchers as the parallelization process was difficult and time consuming. A lot of debugging and coding on a trial and error basis had to be done as there were no basic guidelines on how to parallelize and no softwares like CUDA or OpenCL to help. The basic auto parallelization softwares which were present during the times were inefficient and impossible to apply on more complex algorithms. At first the parallelization was done on one core CPUs connected togeather to create a multiprocesors. The first attempts at computing on ordinary GPUs started around year 2001 when the GPU manufacturers made their GPUs programmable. This allowed people to more easily translate the code to GPU mainly for the computing of matrices and vectors. However, the programming for GPUs was still hard and slow as many basic things were missing such as debuggers. Currently most of the parallelized algorithms are run on GPGPUs and multi-core CPUs as more and more programs utilize the parallel architectures.

4.2 Memory handling

There are many studies proving that parallelization has generally a positive effect on the speed of algorithms. However, to process something fast means not only to parallelize

the program, optimize it and run it through powerful machine. It is also about finding an appropriate procedure or approach, where different types of algorithms doing the same thing can achieve different results on a different set of data. Small changes in, for example, how the data is handled within the GPU or CPU can also increase the processing speeds. The next three studies show that the way the memory is handled has a considerable impact on the speed of the algorithms.

In a study [10] from year 1995 the team tried parallelization of the Monte Carlo algorithm which is a random deterministic algorithm. They focused on two MIMD machines: a shared-memory architecture (SMA) multi-processors, which is a architecture where all of the processors access the same global memory, and a distributed-memory architecture (DMA) multi-computers, where each processor has its own memory. The SMA machine was an Alliant FX/2800 with 24 Intel i860 processors. The Intel i860 had a maximal clock rate of 40 Mhz. The DMA machine was an Intel Paragon parallel computer with 96 compute nodes consisting of the same i860 processors each with 32 MB of memory. During their reaserch they found out that programing for the SMA was easier as it was similiar to the sequential programming, with only one memory to work with. They were also working with two programs: MPMD (multiple program, multiple data) and SPMD. The SPMD algorithms were faster than the MPMD algorithms, even though they predicted the contrary, on both of the tested computers. Their speed ups were nowhere close to the modern parallel algorithms, but even with their basic hardware and software they managed to speed up the processing compared to one core solutions approximately two to five times. However, the decrease in processing time was not always high enough to justify the extra effort put in the parallelization. The test also showed that the SMA machine had higher efficiency (better speed up) for some algorithms compared to the DMA machine (around 50% for SMA and 35% for DMA).

Another study concerning memory handling is study [11]. This study does not focus on different memory handling between machines like the previous one (SMA vs DMA), but on how algorithms doing the same operation (data mining) work with memory. The team created five techniques for data mining algorithms to parallelize them on shared memory parallel machines. This study from 2005 works with single core processors connected to form a multi processor. The data are then saved in cache memory of the

processors and each processor is reffered to as thread. The techniques were developed to avoiding race conditions in the parallel data mining algorithm as multiple threads may want to update the same element. The easiest technique is the full replication technique, which avoids race conditions by giving each thread its own data and they update only the data assigned to them. After the computation is done the data from all of the threads are merged. The next four techniques are all locking techniques, which work on one set of data. The full locks technique avoids race conditions by allowing each thread to lock the part of the data (candidate) that it is going to change. So if a thread works one candidate and it needs to change the value of the candidate, the candidate is locked by the thread. If another thread wants to work with the same candidate it needs to wait for the first thread to release the lock. The threads usually did not have to wait for the other threads as the number of candidates is high compared to the number of threads. Main problems with this technique come from memory handling. The memory needed to support the locks is quite large and also a cache miss (memory location is not found) could happen when the data is trying to get updated. The cache miss can also happen when two threads want to access different elements on the same cache memory and this problem is called false sharing. The next three techniques are then each dealing with one of the presented problems. The optimized full locks technique is trying to overcome the cache misses from wrong memory location. This technique allocates the candidate and the lock in the same cache block. Because of this the number of cache misses and also the possibility of false sharing is reduced. However, this scheme does not reduce the memory usage. The next technique, fixed locking, is dealing with the memory size problem. This technique limits the number of available locks which reduces the amount of required memory. Main problem of this technique is that the posibility of the thread having to wait for available lock rises. This technqiue also does not lower the possibility of cache misses. The last one is the cache-sensitive locking technique. This technique combines the ideas from optimized full locking and fixed locking. A single lock is used for all the operations done in the cache block, therefore, one lock contains multiple candidates. This results in lower memory requirements and reduced cache misses and false sharing possibility. All of the techniques were then compared between each other in six criteria (Table 1).

	Full Replication	Optimized Full Locks	Cache Sensitive Locks	Full Locks	Fixed Locks
Memory Requirement	very high	high	low	high	low
Parallelism	very high	high	medium	high	low
Locking Overhead	none	medium	high	medium	high
Cache Misses	low	medium	low	high	medium
False Sharing	none	yes	none	yes	yes
Merging Costs	yes	none	none	none	none

Table 1 . Comparing all five techniques. [16]

All of the techniques were then run on the computer to find out the performance differences between them (Fig. 8).

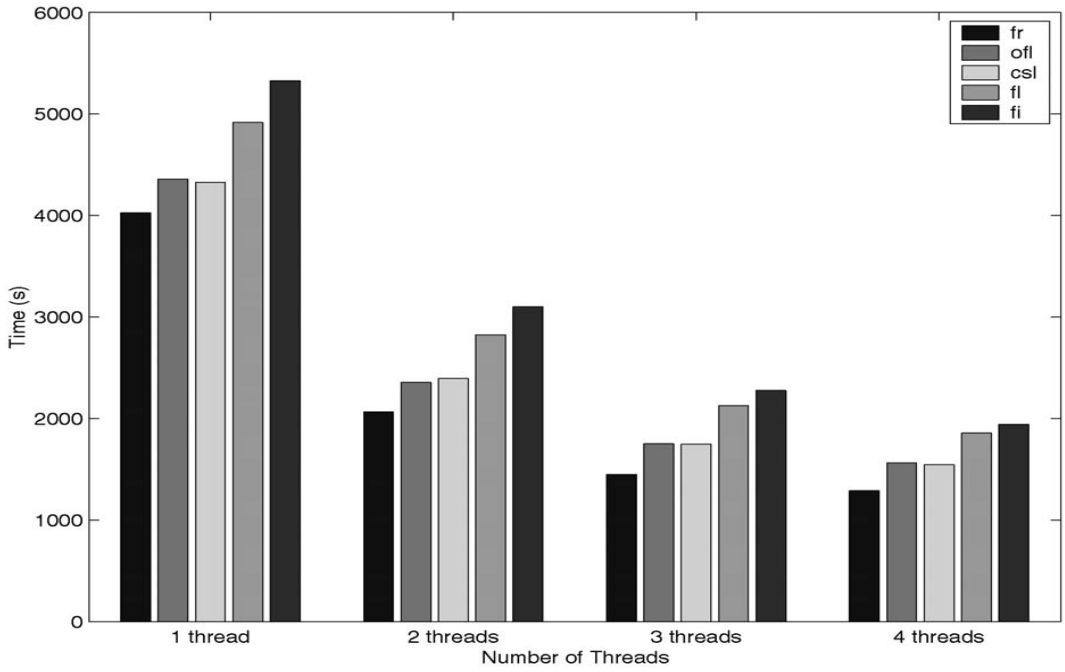


Fig. 8 . Comparing all five techniques for a priori (computation time).

fr - full replication; ofl - optimized full locks; csl – cache sensitive locks; fl – full locks; fi – fixed locks [16]

The team then decided to use three of the techniques techniques (fr, ofl, csl) and apply them on the data mining algorithms. The reason why the fl and fi techniques were not used is mainly for the memory requirements, as if these method would have been used on larger data set the memory requirement would slow them down considerably. This study shows that how memory is handled is important as it can have profound impact on the speed of the algotihm.

The study [12] also compares algorithms performing the same task, but each with different approach to memory. This study was conducted in 2014 and used CUDA to

calculate the weight with genetic algorithms (GA) in convolution filters. Weight in convolution filters determines frequency response characteristics, and therefore, the effect of the filter operation on the image. Weight is calculated in a mask which is an area set by the user (for example a grid of 3x3) around a pixel that is going to be processed. The genetic algorithm is then used instead, to find the global minimum in search domain, and calculate the weight of the convolution mask. Because these operations are computationally heavy with arithmetic operations the parallelization on GPU provides a considerable speed up. The tests were conducted on different sizes of the images (resolution) with changing population. Population is used in GA and it is a set of numbers that determine the probable number of solutions that represent the filter weight. The filter weight is calculated for each member of the population by a fitness function. The GA adjust the filter weight according to the values of the fiteness function of the population members and is trying to minimize the mean absolute error (MAE) between the original and filtered images. The used hardware was a GeForce GTX 660 as the GPU and a AMD FX 8320 running at 3,5 GHz as the CPU.

Image dimensions	Population size	Sequential	DM	PBM	BBM	SBM
256 x 256	128	44.02	14.08	4.09	2.52	0.76
	256	88.35	27.35	9.41	4.52	1.22
	512	176.7	55.89	18.53	9.57	2.26
512 x 512	128	176.69	48.46	17.66	9.61	2.16
	256	359.75	96.28	34.89	17.37	4.09
	512	720.33	192.24	69.35	37.44	7.94
1024 x 1024	128	714.88	186.15	69.15	38.58	8.32
	256	1443.02	373.44	137.4	68.68	15.52
	512	2893.06	746.72	274.15	148.37	30.7

Table 2. Measured times of the methods in seconds (mask size: 3x3, results: average from 100 iterations)
Sequential - sequential code; DM – Direct method; PBM – Population based method; BBM – Block based method; SBM – Sub-images based method; [14]

As shown in Table 2 multiple methods on how to approach the GPU based acceleration were used. The Sequential is implementation on the CPU with sequential code. The direct method is the slowest of all of the GPU accelerated methods, but its implementation on GPU is the simplest. The DM computes fitness values for each

member of the population individualy and each pixel is determined by the `threadIdx.y` and `blockIdx.y` function. Each individual then has its MAE determined and all the values are synchronized in the blocks. Then the total block shared MAE value is determined and with that the values for the image. This method is the slowest as it does not take advantage of the thread local memory and also requires calling of the kernel for each population number which slows down the computing. Next one is the population based method which is slightly modified DM. In the PBM the image is loaded to the global memory which allows the thread to access all of the pixels and calculate the MAE for the entire population and not only for one pixel. So according to the mask size the thread can access the input pixel and its neighbors, unlike the DM where each neighboring pixel is accessed on by one, and copy them on to the local thread memory. This method therefore eliminates the repetitive kernel calls and significantly reduces the number of reads of the global memory. Another method used is the block based method that utilizes the block shared memory which is faster than the global memory. In this method all the data is saved in the global memory and then according to the mask size a block of threads is created and all the data needed to calculate this block is transfered to the block shared memory. The threads access this memory instead of the global one and because this memory has faster access speeds the performance is increased. The last method is the sub-images based method which is the fastest method developed by the team. The difference between this method and the BBM is that an image sub-block is moved to the shared block memory instead of the mask size. The image sub-blocks are determined beforehand and their size depends on the picture size, size of the mask, GPU architecture etc. The image sub-block needs to have a size that fits with the size of the mask (for 3x3 mask it is a block of, for example, 18x18 pixels). This means that the threads in the block calculate the values for all of the pixels in the image sub-block which significantly reduces the number of global memory reads as a larger area is located in the block memory. MAE is calculated for each block and the MAE for the whole picture is calculated from all of the blocks. This method provided approximately 55 to 90 times more acceleration over a sequential code. The speed up was achieved mainly by utilizing the block shared and local thread memory more efficiently than the previous methods. The computational times raised almost linearly with the increasing picture size and population showing that the algorithms are scalable.

4.3 Accuracy

Another thing that has to be considered, mainly in scientific reaserch, is the accuracy of the floating points. The accuracy of the floating point describes how accurate the results of the computations are. Mainly when working with inexact results the programming it self effects the accuracy, for example, from rounding of the calculated values. The inaccuracies come from both hardware and software.[13] The two following studies deal with the floating point accuracy.

In a work [14] from 2003 the team tried to develop a numerical algorithm for GPU. The difficulty of algebraic equations challenged the accuracy and memory of the systems. Active reaserch was conducted with regards to parallelization of numerical solvers on multi-processor architectures, but the new fully programmable floating point pipelines on GPUs drove some to try parallelize them on GPUs. Because no APIs for programming on GPUs were present during the time, the team decided to use Pixel Shader 2.0 API, which is a program used in computer graphics to calculate effects of each pixel. This meant that the results of the computation were actually a 1D or 2D texture maps that represented the vectors or matrices. The main problem with this approach was the precission and accuracy. The tested GPU was ATI 9800, which was not supporting every technology that was available at the time, mainly the fully programmable floating point pipelines. However that was true for most of the GPUs in that time as most offered only partial programming. The system also used a Intel Pentium 4 2.8 GHz processor. The test was then conducted with vectors and matrices of sizes 512^2 to 2048^2 so that the memory limitation are not met. The measured times on the GPU were also timed without the necessary initial memory transfer on to the GPU memory. They got a considerable speed up with a standard arithmetic operations (adding, multiplication), which was about 12 to 15 times faster on the GPU compared to an optimized software implementation on the same system. The least efficient operation, compared to the sofware solution, was the reduce operation which was only two or three times faster on the GPU. It was mainly due to more complex pixel shader program which was needed for this operation. The speed ups then droped by a factor of approximatly two, when the higher precision textures where used to increase the accuracy of the computations. In the end, they achieved good speed up which could

have been even higher if a fully programmable GPU was used. But even with not the best GPU the speed up was worth the parallelization time.

Paper that focused on image processing [15] used the GPU to power digital image correlation (DIC) in non-contact and full field optical measurements. Algorithms used in DIC are uasualy not parallelized as the codes are relativly complex. One of the few that was already parallelized is the integer-pixel DIC algorithm, however, the most useful one, iterative sub-pixel DIC algorithm, was still not implemented on GPUs. This algorithm is superior to non-iterativ sub-pixel DICs in higher accuracy and precission. The main barrier for parallelization of DIC algorithms is that they create an initial guess of the calculation, which is based on the assumption of continuous deformation. Then they measure the deformation at the point of interest (POI) and the displacement of its neighboring POIs is estimated and used by the subsequent iterative procedure. So the reaserchers in the paper choose to use the path-independent DIC method which estimates the initial guess at each POI independently. Another problem they have encountered with GPUs is that they excel in single-precision floating points which in many algorithms means that the accuracy of the measurments can be distorted. And when using the double-precision floating point, the speed of the GPU goes down significantly (usually it is halved). However the method used in this paper did not suffer from the lowered accuracy and the difference between the single and double precision in accuracy was only 0.01%. Because of that the single-precision approach was used without a lost of processing speed. Their algorithm was coded in C++ language based on CUDA and was run on GeForce GTX 760 and Intel i5-3570 (3.4 GHz) CPU. They achieved up to a 76 times speed up compared to the sequential algorithm on CPU and at the time of the study (27 November 2014) it was probably the fastest computation speed of a iterative sub-pixel DIC method ever reported with $1.66 \cdot 10^5$ POI/s.

4.4 Efficiency

However, parallelization and optimization is not only about speeding up the system, but also about how efficiently the processing power is used. This concern comes in to play more and more as power consumption of the modern computers is increasing. Currently the newly developed hardware is not only fast, but also efficient.

In a study [16] the parallelization is viewed from the point of savings. Synthetic Aperture Radar (SAR) is a radar which is used to create images and can be, for example, used on spacecrafts (satellites). The satellite takes raw data from its antennas and sends them to the Earth, where the data is then processed to get the image. This approach is not ideal as the bandwidth of the satellite is taken by sending the raw data and then still some image processing is needed on Earth to obtain the final image. This binds the procedure to take a longer time and the images can not be received immediately and, for example, if the owner of the satellite is not the customer, the image then needs to be sent from the processing center to the customer, prolonging the process even more and with additional costs to the process. Therefore, an idea was introduced to process the images on the satellite in real time. This is where the efficiency part comes to play as energy is the main concern with satellites, but the speed is also needed. The team tried to parallelize and optimize the 2-Dimensional Fourier Matched Filtering and Interpolation (2DFMFI) algorithm to achieve the best performance and the highest efficiency to lower the power consumption too. The 2DFMI was coded in C and in the end they achieved over 50% efficiency with a speedup of 66.4 times out of the theoretical linear speedup of 128 times (128 cores).

4.5 Automatization

To make parallelization process easier and available even for unskilled programmers, programs (algorithms) for an automatic parallelization are also created.

The study [17] is focused on automatic creation of parallel algorithms for GPU from sequential algorithms made for CPU. The method they used for the automatic parallelization is skeleton implementation. Skeleton based parallelization creates flexible code environment and prior unsupported algorithms can be defind for the skeleotn parallelization. The target language of the program is C++ CUDA programming language. The code generation is performed by manualy selecting skeleton implementation from a pool. Then the skeleton passes the code thru a selection of rewrite rules. This approach is diffrent from the template code generation as the final code can be further optimize if necessary. Because this program is focused on image processing algorithms there are four classes to choose from: pixel to pixel algorithms,

neighboring pixels, reduction to scalar and reduction to vector. Most of the basic image processing algorithms can fit in to one of these categories or the more complex ones can be broken down to a combination of these four categories. Depending on the chosen category the rewrite rules are applied and the sequential code is rewritten by these rules. The team then run the skeleton automatization on multiple sequential codes for CPU. The native CPU codes were then compared to the automaticaly parallelized codes run on GPU. The Table 2 shows the times it took each algorithm to do the same task on CPU and GPU. From the results (Table 3) it is evident, that even codes that were parallelized by a program which followed basic predefined rules, the parallelization still offers a speed up compared to CPU. Even though this technique offers great results it is not widely used one. Mainly because the algorithms that are processed need to be in one of the pre defined category, and if they do not belong to one a new one needs to be created for them to be able to use this method.

algorithm	CPU	auto-GPU	algorithm	CPU	auto-GPU
Binarize	106	0.34	Dilate	445	1.67
Copy	99	0.34	Gradient	432	1.45
Transpose	88	0.5	Sum	37	0.28
Blur	208	0.74	Max	41	0.57
Sobel	230	1.21	Min	41	0.56
Erode	151	0.65	Histogram	213	0.47

Table 3. Time comparison between CPU code and auto parallelized GPU code in ms [17]

5 Experimental part

In the experimental part two algorithms were compared on CPU and GPU. The two algorithms in question are median filtering and vector adding. The median filtering is highly parallelizable algorithm and is used to showcase the difference between running such program on CPU and GPU. On the other hand, the vector adding is simple sequential algorithm that should not benefit GPU computing. All of the tests were done in Matlab and on two separate devices.

The first computer (next just as Computer A) had an Intel Core 2 Quad Q9400 processor and GTX 770 graphics card. The Intel Core 2 Quad Q9400 CPU has four processor cores with a base frequency of 2.7 GHz. This processor averages 3437 points in CPU benchmark [18], the GTX 770 graphics card has an average of 6148 points in GPU benchmark [19]. The second computer (next just as Computer B) had an Intel Core i7-3770K processor and GTX 660Ti graphics card. The Intel Core i7-3770K CPU also has four processor cores with base frequency of 3.5 GHz, however it has two logical cores for each physical one. This means that this processor is rated as eight core CPU as it can run two operations on one physical core at the same time. This solution of adding logical and physical cores, is to promote parallelization on CPUs and to increase the processing speeds without the need of increasing the core frequency. This processor then averages at 9610 points in CPU benchmarks [18]. As can be seen, this processor scores almost three times more points mainly because the benchamark tool used in these test can utilize all of the eight cores of the CPU. The GTX 660Ti is however a bit slower, scoring on average 4689 points in GPU benchmark [19].

	Computer A	Computer B
CPU	3437	9610
GPU	6148	4689

Table 4. Benchmark performance of tested computers

The vector adding algorithm had a size of 1024 elements and was run in three different ways. First by simple sequential algorithm on CPU, then by build in function of Matlab (adding function of Matlab: C = A+B) and finally by the kernel on GPU. The median filtering algorithm had a mask size of nine and was run in four different ways on a picture with resolution of 512x512. At first sequentially on CPU, then again on CPU,

but with the use of Matlab function `medfilt2`, thirdly by GPU with memory transfer (the measured time contains the transfer of data from device to host and vice versa) and lastly by the same algorithm but this time with all the data already allocated on the GPU memory.

These two algorithms were then run on the two computers through Matlab and the time it took them to complete each of the operation was measured. The table shows average measured times with accuracy of +/- 10%.

	Computer A	Computer B
On CPU sequentially	4.5551 s	4.6713 s
On CPU with Matlab functions	61.6069 ms	31.1494 ms
On GPU with memory transfer	1.5917 ms	1.3107 ms
On GPU with allocated data	0.37209 ms	0.23165 ms

Table 5. Measured times of median filtering algorithm

	Computer A	Computer B
On CPU sequentially	0.4877 ms	0.4006 ms
On CPU by Matlab function	0.2243 ms	0.1554 ms
On GPU	0.3360 ms	0.2126 ms

Table 6. Measured times of vector adding algorithm

The results obtained from the measurments agrees with what was predicted. The highly parallelizable operation such as median filtering excels on GPUs. Also when the data is alocated on the GPU memory before hand the computation time decreases. The memory handling is uasualy the biggest issue when dealing with GPU computations as the memory on the GPU is small. The vector adding then shows that the difference between sequential operations are minimal and not that high speed up is achieved.

The computations done by the build in functions of Matlab are generaly faster than ordinary codes that are not optimized. The main difference was in median filtering using the build in function of Matlab `medfilt2` which utilizes the Matlab matrix operations for faster calculation.

Interestingly the results between each computer differed in a way that does not correspond with the speed of the computers. Computer B should have been faster on all of the CPU related measurments as it has faster CPU, but should have been behind with

the GPU computing as the GTX660 Ti is slower GPU.

Looking firstly at the CPU computing. The median filter that was run sequentially on CPU was not able to average faster results on the faster CPU, but in the computing with the Matlab functions the faster CPU performed better. This is a case of better optimatization of the Matlab function that can utilize the faster CPU. The sequential code (on CPU sequentially) did not use the full potential of the CPU (only around 40%) and only four of the eight available cores on the Intel i7.

This means that the processing times on CPUs may have been lowered by better optimatization of the algorithms and that there is still room for improvements.

In the GPU computing it was interesting that the slower GPU was able to outperform the newer GTX 770 in all of the measurments. This could have been caused by several variables. Either the code could not take the advanteage of the faster GPU because it was not optimised enough, or the tasks were not demanding enough. Or, which is more probable, it was due to an external influence acting on the system. The GPU could have been holded back by the slower CPU or other components in the computer that effect the speed of the whole system, things like motherboards, RAM memories etc., which can all create a bottleneck for the GPU. The simple rule would apply here stating that the system is only as fast as its slowest component. Among other things to consider are different versions of Matlab, Windows etc., which all effect the performance of the computer (mainly programms running in background that can slow down the computer). These all effects and variables could have been eliminated if the measurment was conducted in laboratory conditions. However these results show that there are many things influencing the speed of the algorithms and to achieve the best results more things need to be considered than just the code.

The median filtration was applied on the image (Fig. 9) with mask size of nine. It is apparent that the image got blurred a little bit. This shows that the mask size was too high, however the noise in the image was eliminated. The noise is apperent mainly around the hat on the background (the white dots) which are gone from the image after the median filtration. The bigger the mask size is the more blurred the image will look. To minimize the blurring of the image the mask size needs to be lowered. When a mask with a size of four (Fig. 10) is used on the image the blurring effect is almost not evident, but the noise is reduced significantly.

Fig. 9. Left image is before and right image is after the median filtration with mask size nine

Fig. 10. Application of the median filter of mask size four (not ideal size) on the original image

The median filter needs to fine tuned for each individual picture. Mainly the resolution effects the blurring of the image. If the image has low resolution (like the image used here) the effects of the median filter will be lowered as the larger mask can not be used because of excessive blurring (like in Fig. 9 with mask size of nine). The higher the resolution of the image is, the bigger mask size can be used and the effect of blurring can be reduced.

Conclusion

This work focused on hardware and software of parallelized image processing algorithms. The technology of parallel computing is relatively modern and spreads to all computer devices in a way of multi-core processors or new multi-core graphics cards. The programming architectures (CUDA, OpenCL, etc.) were created to overcome a lot of challenges mainly on the code side of things. These architectures make the parallel computing not only more accessible, but also offer an additional speed up with their refined code and optimatization.

The research confirmed that the parallelization overall has a positive effect on the speed up of the algorithms. However, to achieve the best performance a lot of steps have to be taken. From finding an appropriate algorithm that is the most suitable one for the task to optimatization of such algorithm on the used hardware. One of the things to look for when optimizing the code is how the memory is handled. The memory handling has huge effect on the speed of the algorithm, where algorithm with efficient memory handling can be up to five times faster than the inefficient one [14]. Among other things to consider is also the accuracy of the computations. This is relevant mainly in scientific experiments where accuracy is critical. To achieve the needed accuracy an appropriate approach has to be taken when writing the algorithm and proper hardware has to be used. The parallelization can also be used to make more efficient systems where speed is not the most important part, but things like power consumption are. Parallelization then allows the use of less powerful hardware that can perform at the same speeds but with higher efficiency.

The experimental part confirmed that even a simple parallel algorithm can speed up a calculation heavy operation. The parallel solution of the median filter had a speed up of approximately seven times compared to the sequential algorithm. It was also proven that not so computationaly demanding operation like vector adding, which can not be even parallelized properly, does not benefit greatly from the parallelization. The results also shown that optimatization is as important as anything else when considering the parallelization. The optimized Matlab functions were in median filtering considerably faster then the ordinary sequential code and in vector adding they were the fastest. Also when comparing the results between the two devices the optimatization is highlighted

even more, where even the more powerfull hardware was not able to outperform the weaker one due to number of reasons.

When creating an algorithm for a device that supports CUDA the CUDA architecture should be used instead of OpenCL as it should provide better speeds. However, when another hardware is used or hybrid computing (both CPU and GPU at the same time) is used the OpenCL is probably the best possible solution as it supports much wider range of hardware. When deciding if to parallelize the algorithm, firstly the difficulty of the process should be considered, to find out if the parallelization time is worth the speed up it could bring.

Used literature

[1] MOURA, Jose. What is signal processing? [President's Message. *IEEE Signal Processing Magazine* [online]. 2009, vol. 26, issue 6, pp. 6-6 [cit. 2014-12-20]. DOI: 10.1109/MSP.2009.934636. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5230869

[2] KREJSA, Jiří. *Základy zpracování signálu* [online]. [cit. 2015-05-11]. Available at: http://www.umt.fme.vutbr.cz/~ruja/vyuka/ZZS/DSP01.pdf

[3] Digital camera image noise. [online]. [cit. 2015-05-04]. Available at: http://www.cambridgeincolour.com/tutorials/image-noise.htm

[4] BERTALMÍO, Marcelo. 2014. From image processing to computational neuroscience: a neural model based on histogram equalization. *Frontiers in Computational Neuroscience* [online]. **8**: - [cit. 2015-05-11]. DOI: 10.3389/fncom.2014.00071. ISSN 1662-5188. Available at: http://journal.frontiersin.org/article/10.3389/fncom.2014.00071/abstract

[5] EVANS, Adrian N. 2004. Morphological gradient operators for colour images. *2004 International Conference on Image Processing, 2004. ICIP '04* [online]. 2004 [cit. 2015-05-04]. DOI: 10.1109/icip.2004.1421766. Available at: http://people.bath.ac.uk/eesane/Colour/ICIPOct2004.pdf

[6] PERROT, Gilles, Stéphane DOMAS and Raphaël COUTURIER. Fine-tuned High-speed Implementation of a GPU-based Median Filter. *Journal of Signal Processing Systems* [online]. 2014, vol. 75, issue 3, pp. 185-190 [cit. 2014-12-20]. DOI: 10.1007/s11265-013-0799-2. Available at: http://link.springer.com/10.1007/s11265-013-0799-2

[7] TIEN, Tsan-Rong and Yi-Ping YOU. Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine. *Software: Practice and Experience* [online]. 2014, vol. 44, issue 5, pp. 483-510 [cit. 2014-12-20]. DOI: 10.1002/spe.2166. Available at: http://doi.wiley.com/10.1002/spe.2166

[8] COOK, Shane. *CUDA Programming a Developer's Guide to Parallel Computing with GPUs*. Online-Ausg. Burlington: Elsevier Science, 2012. ISBN 01-241-5988-5.

[9] *NVIDIA CUDA Zone* [online]. [cit. 2015-05-11]. Avalaible at: http://devblogs.nvidia.com/parallelforall/cudacasts-episode-2-your-first-cuda-c-program/

[10] WIDMANN, Albert H. and Ulrich W. SUTER. Parallelization of a Monte Carlo algorithm for the simulation of polymer melts. *Computer Physics Communications*. 1995, vol. 92, 2-3, pp. 229-251. DOI: 10.1016/0010-4655(95)00092-0.

[11] RUOMING JIN, GE YANG and G. AGRAWAL. Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. In: *IEEE Transactions on Knowledge and Data Engineering* [online]. 2005, pp. 71-89 [cit. 2015-05-20]. DOI: 10.1109/TKDE.2005.18. ISSN 1041-4347. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1363766

[12] AKGÜN, Devrim and Pakize ERDOĞMUŞ. GPU accelerated training of image convolution filter weights using genetic algorithms. *Applied Soft Computing*. 2015, vol. 30, pp. 585-594. DOI: 10.1016/j.asoc.2015.02.010.

[13] WHITEHEAD, Nathan and Alex FIT-FLOREA. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs* [online]. [cit. 2015-05-20]. Available at: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

[14] KRÜGER, Jens and Rüdiger WESTERMANN. Linear algebra operators for GPU implementation of numerical algorithms. *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*. 2003. DOI: 10.1145/1201775.882363.

[15] ZHANG, Lingqi, Tianyi WANG. High accuracy digital image correlation powered by GPU-based parallel computing. *Optics and Lasers in Engineering*. 2015, vol. 69, pp. 7-12. DOI: 10.1016/j.optlaseng.2015.01.012.

[16] KRAJA, F., G. ACHER and A. BODE. Parallelization techniques for the 2D Fourier Matched Filtering and Interpolation SAR algorithm. *2012 IEEE Aerospace Conference* [online]. IEEE, 2012 [cit. 2015-01-14]. DOI: 10.1109/AERO.2012.6187225. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6187225

[17] NUGTEREN, Cedric, Henk CORPORAAL and Bart MESMAN. 2011. Skeleton-based automatic parallelization of image processing algorithms for GPUs. *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation* [online]. [cit. 2015-05-11]. DOI: 10.1109/samos.2011.6045441.

[18] CPU Benchmarks. [online]. [cit. 2015-05-05]. Avalaible at: http://www.cpubenchmark.net/

[19] Videocard Benchmarks. [online]. [cit. 2015-05-05]. Available at: http://www.videocardbenchmark.net/