VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH BINÁRNÍCH AMPLITUDOVÝCH HOLOGRAMŮ PRO OPTICKÉ GENEROVÁNÍ ULTRAZVUKU AKCELEROVANÝ POMOCÍ GPU

GPU-ACCELERATED DESIGN OF OPTICALLY GENERATED ULTRASOUND USING BINARY AMPLITUDE HOLOGRAMS

BAKALÁŘSKÁ PRÁCE

BACHELOR‘S THESIS

AUTOR PRÁCE AUTHOR	Martin Knotek
VEDOUCÍ PRÁCE SUPERVISOR	Ing. Jiří Jaroš, Ph.D.

Brno, 2016

Abstrakt

V této práci se zabýváme možnostmi urychlení vědeckých výpočtů s použitím grafických výpočetních jednotek. Termínem vědecký výpočet v tomto kontextu rozumíme specifický algoritmus, který počítá povrch binárních hologramů, jež se používají při generování ultrazvuku. Zaměříme se na návrh hologramu, zvláště pak na rychlost, se kterou můžeme vypočítat povrch takového hologramu. Za tímto účelem použijeme dvě populární platformy pro paralelní zpracování dat – CUDA a OpenMP. Výsledný povrch hologramu je důležitý, protože ovlivňuje specifické fyzikální vlastnosti hologramu.

Abstract

In this thesis, we deal with the possibilities of the acceleration of scientific computations using the graphical processing unit. The term scientific computation in this context means an algorithm, which computes binary holograms that are used to generate ultrasound. We will concentrate specifically on the design of the hologram, focusing at the speed we can achieve when computing the surface of the hologram. For this purpose, we will use two popular parallel data processing platforms – CUDA and OpenMP. The surface design pattern of the hologram is important due to the fact, that it determines the hologram's specific physical characteristics.

Klíčová slova

GPGPU, CUDA, OpenMP, vysoce náročné výpočty, paralelní výpočty, ultrazuk

Keywords

GPGPU, CUDA, OpenMP, high performance computing, parallel computing, ultrasound

KNOTEK, Martin. Návrh binárních amplitudových hologramů pro optické generování ultrazvuku akcelerovaný pomocí GPU. Brno, 2016. 40 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Jaroš Jiří.

Statement

I claim I have elaborated this thesis on my own under the supervisor Ing. Jiří Jaroš, Ph.D.

I have specified all literal sources and publications I have sourced from.

……………………
Martin Knotek
1.2.2016

Acknowledgement

I would like to thank primarily to the supervisor of my work, Ing. Jiří Jaroš, Ph.D. He was always nice, always had an idea of how to improve the work. Both his team-leading and programming skills are much appreciated.

Furthermore, a big thank you belongs to my family, relatives and closest friends. They were a huge support at all times.

Another appreciation of gratitude is addressed to the whole set of computer equipment that lasted long enough for me to be able to finish this thesis.

Thank you.

Contents

1 Introduction

In computer science, there are many different areas and types of work, algorithms, ideas, data types, etc. Most computer users are pure consumers; they use the computer to get along their day and to make their lives easier. There are also people who create programmes for computers - web pages, databases, games. At last, but not least, there are programmers and applications that are oriented in speed and overall performance. These applications execute and implement very difficult scientific or simulation algorithms and are extremely demanding on hardware/software performance.

In general, there are two ways you can achieve better computing speed – getting a faster, more powerful hardware or creating an optimised software. Creating a new set of hardware equipment is expensive and takes a long time, whether it is processor units, graphical units, or other specialised cards. On the other hand, when one is given a certain equipment and is supposed to implement a very slow and performance-demanding algorithm on "*what he's got*", it is a challenge. And all these little challenges change the world.

One of many areas of high performance computing is laser-generated ultrasound. The photoacoustic effect occurs when a time varying optical source is incident on an optically absorbing material. The incident photons are absorbed and converted to heat, which causes a small temperature rise resulting in local pressure increase which generates an acoustic pulse. Within the past decade there has been increasing interest in optically generated ultrasound (OGUS) for biomedical applications due to a steady increase in the acoustic pressures that can be generated.

For example, nano-composites composed of separate elastomeric and absorbing components possessing high optical absorption, efficient heat transduction, and high thermal expansion have been used to generate focused pressures of 50 MPa. OGUS has several clear advantages compared to piezoelectrics. These include very wide bandwidths of 100 s of MHz, non-contact excitation, and flexible element size. In addition, by controlling both the optical pulse shape and spatial illumination pattern to a two-dimensional absorber, it is possible to achieve a high degree of control over the resulting acoustic field in three dimensions.

A binary amplitude hologram is a 2-D binary pattern designed to control the distribution of light or sound of a particular wavelength in three dimensions. The pattern of a particular hologram has to be designed and computed according to a set of target points. The hologram's size influences the complexity of its design, meaning larger dimensions of a hologram mean complex and slow design procedure, which takes a long time.

The design procedure can obviously be accelerated. As it is mentioned above, evolving a new hardware is expensive, that is why it is a good idea to use the equipment we already have and exploit its performance. There are many kinds of hardware we can use, though for our purpose multi-cored processors and graphical processing units are the most interesting.

Those chips are not very easy to program, but they possess a huge performance potential, which would be rather shame not to use in our advantage. The fundamental question is: can we transform a piece of code using high performance computing paradigms and create a faster solution, which will produce the same, if not better results?

2 Massively parallel computation

In the computing field, the term *massively parallel* refers to the use of a large number of processors (or cores, computers) to perform a set of coordinated computations in parallel (simultaneously).

There are several ways we can perform such a parallel computation. For example, we can connect multiple computers and create a computational grid, or we can use a specialized chip, that was designed to perform a simultaneous computation.

One kind of those chips is, among others, a graphical processing unit (GPU), which we will use to run our code.

There are several ways we can approach the parallel concept, in our thesis we will use the OpenMP library running designed for CPU and the CUDA platform for controlling the GPU.

2.1 Graphical processing unit

The GPU is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. The first GPUs appeared in the 1970's to accelerate the drawing of graphics for various arcade games.

Newest GPUs have many different functions (programmable shaders, techniques to reduce aliasing, etc.) and can be used not only to produce a graphic output, but to perform a general algorithm as well. In our work we will profit from this ability and we will use the GPU as an instrument for our parallel computation.

2.2 Differences between GPU and CPU

CPU is latency based, while GPU is throughput based. Thus, CPU is ideal for sequential codes, while GPUs are perfect for massively parallel codes.

The CPU contains a low number of cores, today it varies between 4 (desktops) – 16 (servers) cores. Each of these cores is able to process a small amount of threads at the same time, typically one or two. Another typical characteristic of a CPU is a large cache memory, hiding the latency of memory system and the logic of sequential applications, or the out of order instruction execution. An advantage of the CPU may be a less strict restriction in the memory size, which in today's computers reaches from 8GB up to 32GB.

On the other hand, a GPU is composed of a large amount of cores, where each core can process hundreds of threads, if not thousands at the same time.

This huge number of threads then allows a large acceleration of a specific kind of applications, depending on the type of algorithm and its optimization. There are codes, which are very difficult to parallelize, for example traversing through a list or a binary tree, hash functions, user-interaction functions and others.

The built-in memory size on the GPU reaches from low units of GB, most often 2 – 8 GB. There are obviously devices that possess a larger amount of memory – 12 or even 16GB (NVIDIA Pascal card).

The differences between CPU and GPU are summarised in the following table:

Table I: Description of a development platform [2], [3].

	CPU - Intel Core i7-920	GPU - NVIDIA GTX 580
Chip frequency [MHz]	2660	1544
Number of cores	4	16 (SM units)
Number of threads	8	Max. 16 x 1536
Memory size [GB]	12GB	1.5
L2 cache size [KB]	256	786
SIMD Widht	4	32

2.3 Streaming multiprocessor

Streaming multiprocessor (SM) is a basic execution unit, which controls all the computations. The number of SMs is dependent on the type of the card; however, the SM architecture is very similar across all card models. SM among others contains load/store units, floating-point and integer units, registers, shared memory and many others. A GPU is then composed of several SMs that allow us to run a large amount of threads.

2.4 OpenMP

OpenMP (**Open** **M**ulti-**P**rocessing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems.

The strong advantage of OpenMP is its simple usage and high productivity. We used this API on a CPU, thus we could only run smaller amount of threads compared to CUDA.

2.5 CUDA platform

The Compute Unified Device Architecture (CUDA) is an environment that allows us to program GPUs that are CUDA-compatible. A part of this package is also a compiler necessary to compile and run our applications on the GPU.

For the proper run of the application, it is necessary to correctly separate the logic of the programme among threads and blocks. It is also crucial to manage the shared memory in the correct way, correctly use the indexes that identify threads in blocks etc.

In the upcoming chapter, we will describe a few basic terms from the CUDA context, which are important to understand when working with CUDA platform.

2.5.1 Grid

Grid identifies the number and organization of blocks. The maximum number of runnable blocks may vary depending on the device.

Blocks from the grid are assigned to particular SMs, no order of execution is provided. Once one of the SMs finishes its work on one block, another block is automatically assigned and the SM can start the computation again.

2.5.2 Block

Block is an abstract unit of threads decomposition. Its purpose is a more simple orientation in threads and their less difficult understanding when programming. In addition, it allows thread cooperation (shared memory to hide memory latency). Threads in a block can be organized into a 1D, 2D or 3D structure. The maximum amount of threads in one block varies depending on the device; usually you can run up to 512 or 1024 threads.

2.5.3 Threads

Once a kernel is launched, it is executed as a grid of parallel (simultaneously running) threads. One kernel launch can spawn even thousands of threads.

A thread is the most basic element of execution on the GPU. Threads and blocks can be organized into a 1D, 2D or 3D structure. The order in which the threads are executed is not guaranteed, so it is a programmer's task to create kernels that are independent on the order of execution.

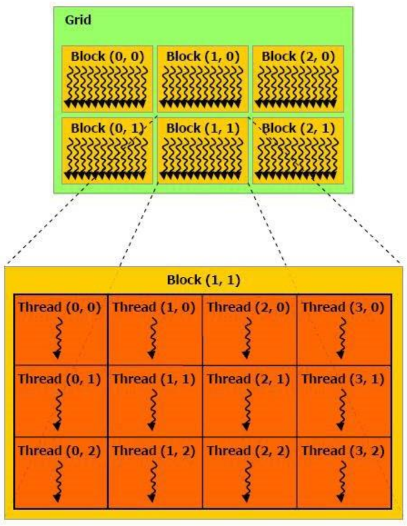

Figure 1. Threads and block organization.

Source: http://3dgep.com/wp-content/uploads/2011/11/grid-of-thread-blocks.png

In Fig. 1 we can see a kernel launch (a grid), that has the following configuration:

Six blocks in a grid, each block held 12 threads.

2.5.4 Kernel

This term means a piece of code of the programme that is supposed to be run on the GPU. It is labelled with the *__global__* keyword that identifies a function that can be launched from the CPU and is executed on the GPU. When running a kernel, we have to specify a launch configuration in which the computation is meant to be executed.

2.5.5 Global memory

This type of memory is the largest, but accesses to it are relatively expensive (take a long time). It is a place where threads can share data. The data stored in this memory are persistent between kernel launches.

That is why it is suitable to use the shared memory, in which we can store frequently used data and save the time, it takes to repetitively load data from one specific area of the global memory.

All threads can access the global memory.

2.5.6 Shared memory

Shared memory's size only counts in tens of KB, but access to it is very fast. This memory is assigned and reserved for one block only. This means each thread in one particular block can load and store data from the shared memory that has been assigned to that block. Sharing data between two or more blocks using the shared memory is not possible.

2.5.7 Registers

Each SM has its own set of registers, into which threads store their own local variables. Accesses to the registers are the fastest compared to other types of GPU memory. Each thread can access only the registers that have been assigned to the thread.

2.6 Performance comparison

To test the GPU abilities and to compare them with CPU, we have implemented a few microbenchmarks based on matrix-matrix multiplication and measured the time it takes to compute the product as well as the number of floating-point operations per second (*GFLOPS*).

Each GPU kernel represents one concept of CUDA code optimization to see which factors actually make an impact on the performance.

CPU implementation was accelerated using the OpenMP platform to employ multiple treads and exploit SIMD vector units.

In the following chapters, we will show graphs with GFLOPS values we achieved while running different types of kernels. These kernels will now be briefly introduced.

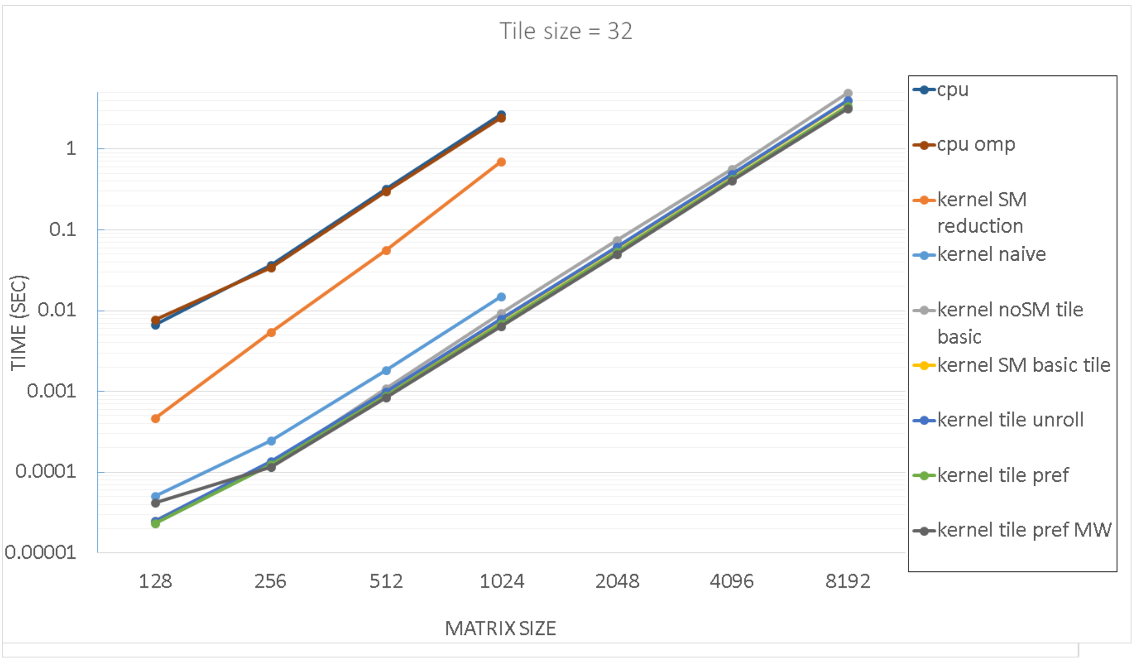

Figure 2. Kernel times comparison.

In Fig. 2, we can see the kernels execution time comparison. The CPU code and some CUDA kernels were not launched for sizes larger than 1024x1024 elements due to an extensive execution time, or because of the impossibility to run a particular kernel (restricted by maximum number of threads we can run in a block).

The horizontal axis *MATRIX SIZE* identifies the matrix size of a square matrix that means each matrix contained *MATRIX SIZE*^{2} elements.

Kernel description:

Kernel naive – the simplest possible implementation of matrix multiplication. No shared memory is used, the grid contains as many blocks as there are rows in the result matrix. That means one row of the matrix is assigned to one block. Each block contains MATRIX SIZE threads. Each thread then computes one element of the result matrix by multiplying all elements in the specific row and column.

Kernel SM reduction – shared memory is used, the grid contains the same amount of blocks as the number of elements in the matrix, number of threads per block corresponds to the matrix size. Threads cooperatively load the product of one particular element from the row and one element from the column of the source matrices into the shared memory. Once the products are stored in the shared memory, threads cooperatively perform a summing reduction and the result of this reduction is the final value of one element of the result matrix.

Kernel noSM tile basic – this kernel does not use the shared memory. It uses the tile principle instead, each block processes one tile (a part of the result matrix).

Kernel SM basic tile – shared memory is used in this kernel. This kernel runs in phases, in each phase the threads cooperate on loading two tiles from two source matrices into the shared memory.

This shared memory data is then used to compute the result – each thread in the tile computes one element of the tile. In each phase threads keep their own progressive value of the product, which represents the summed product of the elements that were processed in the past phases.

Once all phases are done, each thread in one tile then stores the final product in the global memory.

Kernel tile unroll – same idea as the *SM basic tile* with the difference in unrolled loops – the loop calculating the dot product is hand unrolled.

Kernel tile pref – same idea as *SM basic tile*, this time with the use of the prefetch technique, when the kernel is accessing some data from the global memory while is computing the result with some other data at the same time.

With this technique, we can utilize both arithmetic logic and load/store units, which means less time is spent when threads are waiting to get data from the global memory.

Kernel tile pref MW – same principle as *tile pref.* Now however, not only one tile is processed by one block, but two (MW = More Workload).

CPU and **CPU OMP** then stand for a single threaded CPU code and the multithreaded OpenMP code. The OpenMP CPU code is performing the standard naive matrix multiplication loop with a parallel reduction.

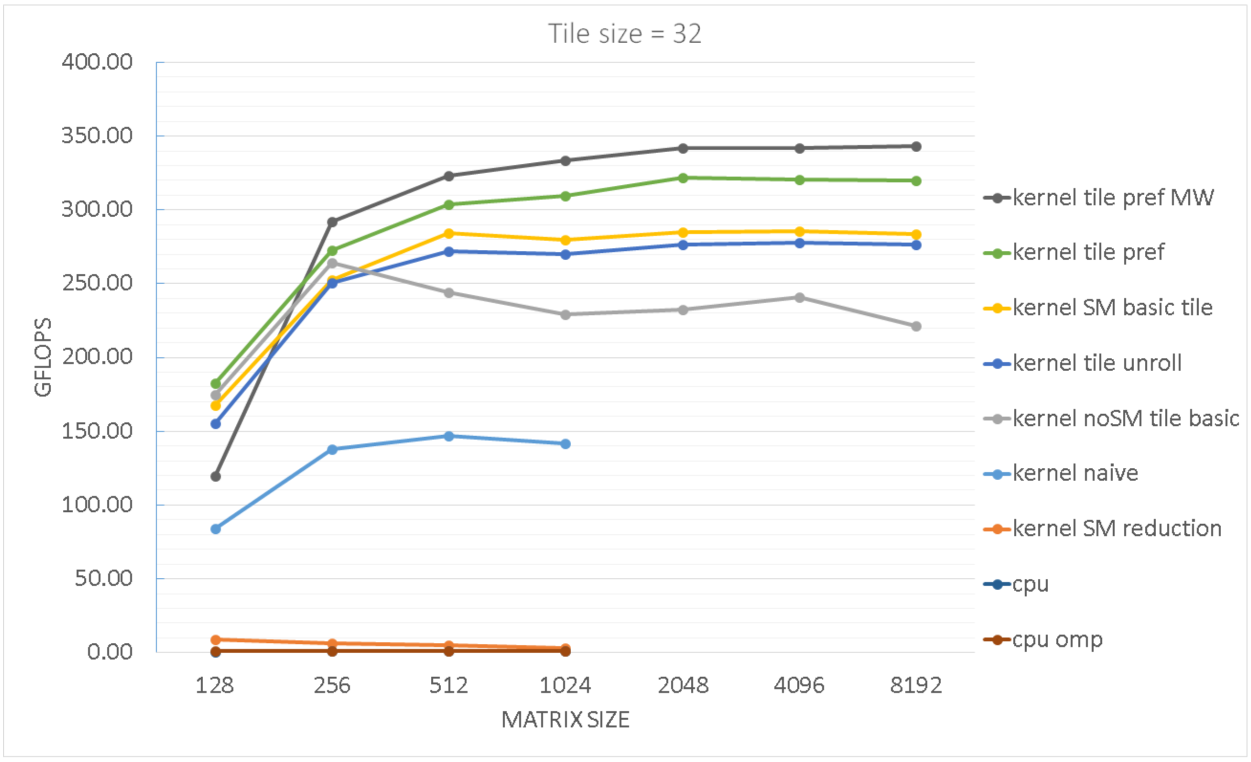

Figure 3. Absolute values of kernel performance.

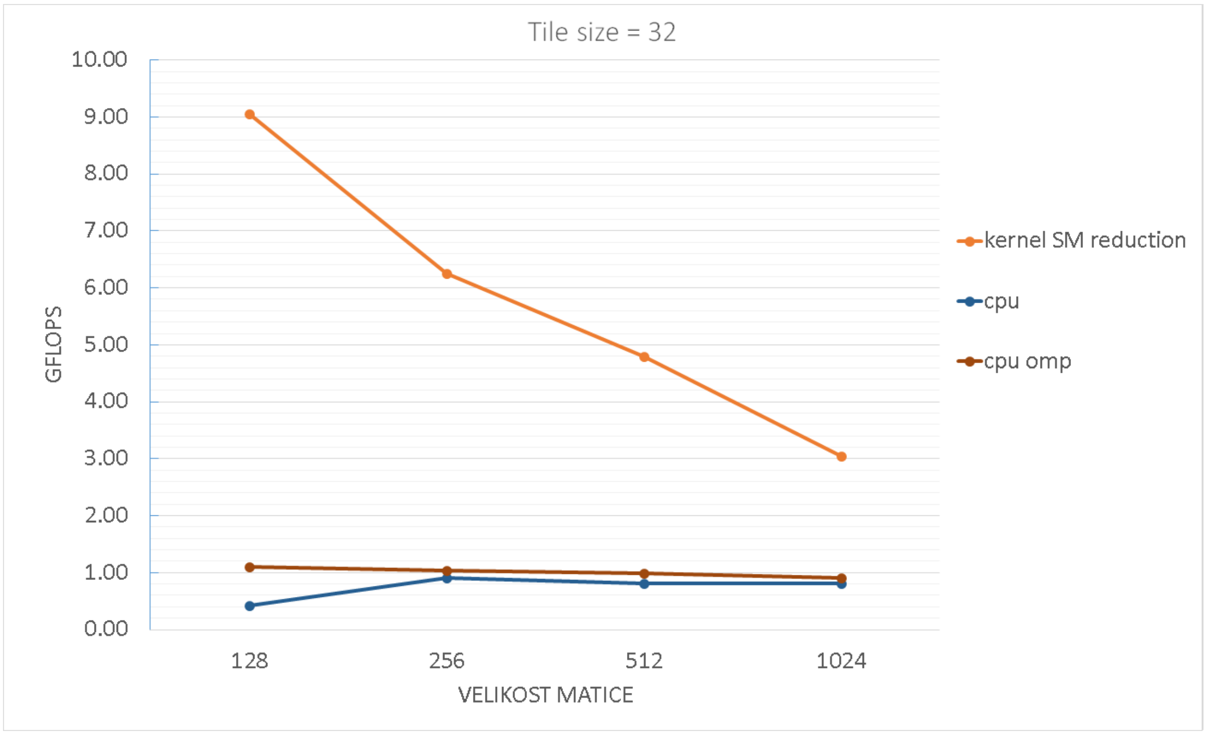

Figure 4. Detailed graph showing CPU and reduction kernel performance.

In the Figure 3 we can see the peak performance values of our kernels. The best results were achieved with the *tile pref MW* kernel, with the value of 343 GFLOPS. We can also see the performance differences in the particular implementations using different approaches and optimizations.

The theoretical peak performance of the CPU we used for the test is 47 GFLOPS [2]. We achieved only around 1 GFLOPS because we did not use any particular optimization.

As of the GPU, the peak performance of the GTX580 is 1581 GFLOPS. We managed to achieve 343 GFLOPS with our best kernel, which is 21% of the possible performance.

2.7 Summary

We were able to find and implement several versions of the matrix-matrix multiplication kernels, each of which with a slightly different approach and optimisation level (prefetch, more workload for one block). We were also able to determine techniques that are better to avoid in some cases (reduction).

Unfortunately, we were not able to achieve the maximum peak performance of the processor and the GPU, as explained above. Nevertheless, we have proofed that some codes are very easy to parallelize and with these codes, we can clearly see that the parallel approach is beneficial.

3 MATLAB implementation

3.1 Binary hologram

A binary amplitude hologram is a 2-D binary pattern designed to control the distribution of light or sound of a particular wavelength in three dimensions. Pixels (elements) that are "on" in the pattern transmit waves which constructively interfere at the design points. Pixels that are "off" do not transmit [1].

This hologram is then printed on an actual piece of material and used to generate ultrasound of specific characteristic, e.g. high acoustic pressure at certain points, very high frequency, specific shape of the focus, etc.

There are two ways we can generate the binary hologram.

First, one may use the ray tracing method, which is designed to generate a single acoustic focus. For each hologram, a target point, aperture size (i.e., hologram size), pixel size (i.e., the size of a hologram element), and design wavelength are defined. The pressure on the surface of the 2-D hologram is then calculated by approximating the focal point as a monochromatic point source oscillating at the design frequency. The calculated 2-D pressure on the hologram surface is then thresholded with positive values of pressure set to 1 and negative values set to 0 to produce a binary hologram. We can see an example of a plotted hologram on Fig. 5.

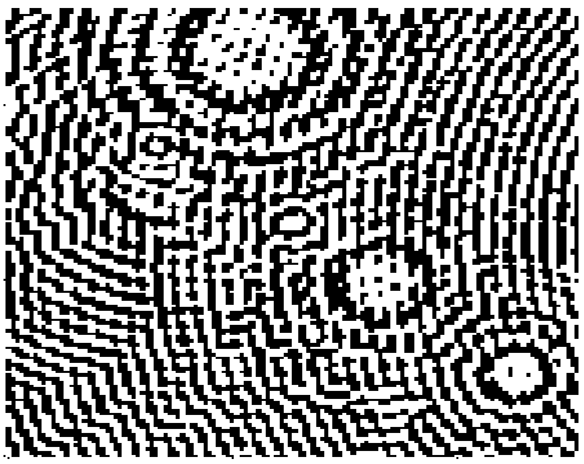

Figure 5. A binary hologram example.

It turned out however, that a hologram generated by the ray-tracing method shows large variations in pressure generated at the target points.

As a result, an optimization approach was developed for the calculation of holograms with multiple foci – a binary search algorithm. We will describe this algorithm in the following chapter.

In addition, we have to establish a mechanism that will allow us to compare one hologram to another and which will measure the total 'quality' of the created hologram. This mechanism will in the upcoming text be denoted as "the cost".

The cost associated with a state of a hologram is evaluated using a cost function first used by Clark in the design of binary optical phase holograms [8]. This is given by

$$C = -|\bar{p}| + \alpha\sigma,$$

where $|\bar{p}|$ is the average magnitude of the complex pressure at the target points, σ is the standard deviation of the pressure over the target points, and α is a factor weighting the two terms. Empirically a value between 1 and 2 for α was found to provide a good balance between maximizing the pressure at each target point and minimizing the variation.

3.2 Binary search algorithm

The hologram is initialized in a randomized binary state and the cost associated with this state is computed using a cost function. The idea of the optimization is the following:

The states of single pixels are flipped, new cost is evaluated and the new state is kept if it decreased. Pixels are chosen randomly, with each pixel on the hologram being explored once before repeat tests. This continues for as long as the number of changes in the iteration is less than *0.01%* of the total number of pixels.

The random-exhaustive approach to pixel selection was found to converge more rapidly than ordered or non-exhaustive selection. The random initialization and ordering of the pixel selection means the algorithm converges to different holograms between model runs and from different initial states.

The algorithm is summarised in Fig. 6.

The initial implementation in MATLAB takes a very long time when bigger holograms are created (e.g. size of 64x64 elements and bigger). This means the time complexity is one of the factors that we will try to improve, as well as keeping the quality of generated holograms on par with MATLAB.

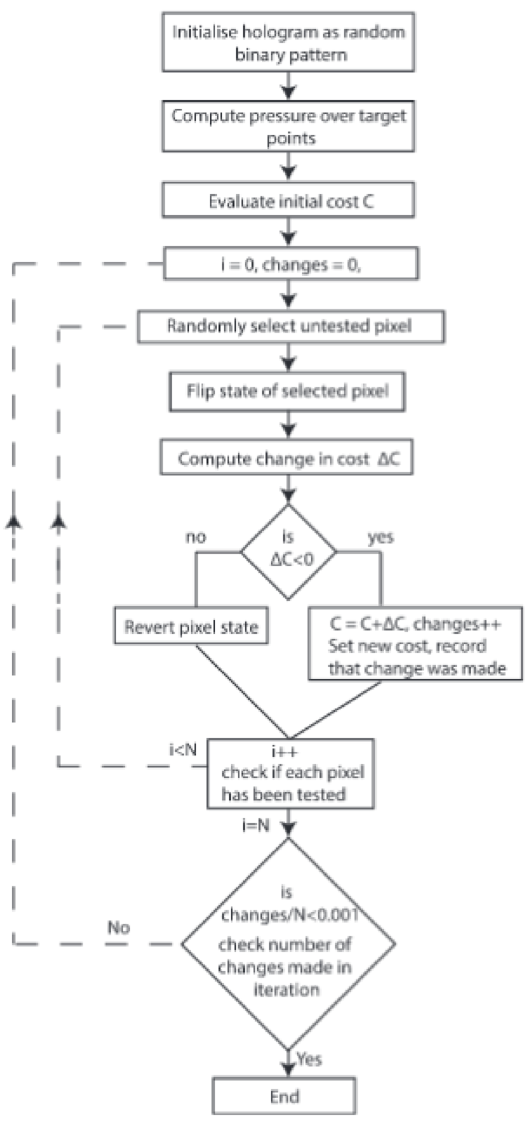
Initialise hologram as random binary pattern
Compute pressure over target points
Evaluate initial cost C
i = 0, changes = 0,
Randomly select untested pixel
Flip state of selected pixel
Compute change in cost ΔC
is ΔC<0
no
yes
Revert pixel state
C = C+ΔC, changes++ Set new cost, record that change was made
i<N
i++ check if each pixel has been tested
i=N
is changes/N<0.001 check number of changes made in iteration
No
Yes
End

Figure 6. Binary search algorithm flowchart [1].

3.3 Code analysis

The starting point of our work was a piece of code written in MATLAB based on an article by Michael Brown at al [1]. We analysed this code, determined the bottlenecks and tried to eliminate them.

The actual code then consists of two loops. The first, outer loop, determines the precision in which the computation is. There is very little we can do about this loop since it is a control loop.

The second loop, however, is the one we can improve. This inner loop traverses through all the elements of the hologram. The whole hologram can be stored in the memory as an array of numbers (integers, doubles). What the serial (MATLAB, C) implementation does, is it goes through all the elements sequentially one after another. Here is the point where the possible optimization comes to play – we can process and evaluate several elements of such an array at the same time. This can be achieved using threads, when each thread evaluates a smaller part of the entire hologram resulting in faster calculation.

The elements for evaluation are chosen randomly, based on a randomized array of indexes. This array is randomly permutated after each iteration of the outer loop.

When evaluating one element, its value is flipped, the new cost of this state is computed and if it is lower than the previously lowest cost, the new state is accepted.

The problems identified in the MATLAB implementation are following:

(1) Matlab uses Just In Time (JIT) compiler to translate a script to the machine code (use another programming language – C/C++ in our case, could improve performance)

(2) Memory in Matlab is dynamically allocated and freed (use one chunk of memory throughout the whole computation)

(3) Work only on one hologram at a time – might not find the best possible solution because the techniques gets stuck in a local optima

(4) The whole hologram is processed sequentially; we will use threads to simultaneously perform the computation

There are a couple of special MATLAB functions in the code, that we will have to rewrite, e.g. *randperm(), ind2sub()* [4, 5] etc. However, these are only auxiliary functions and the C language code will be very similar to the MATLAB code.

3.4 Performance

We measured the MATLAB code performance to set a reference point, which we could compare our work to when improving the algorithm.

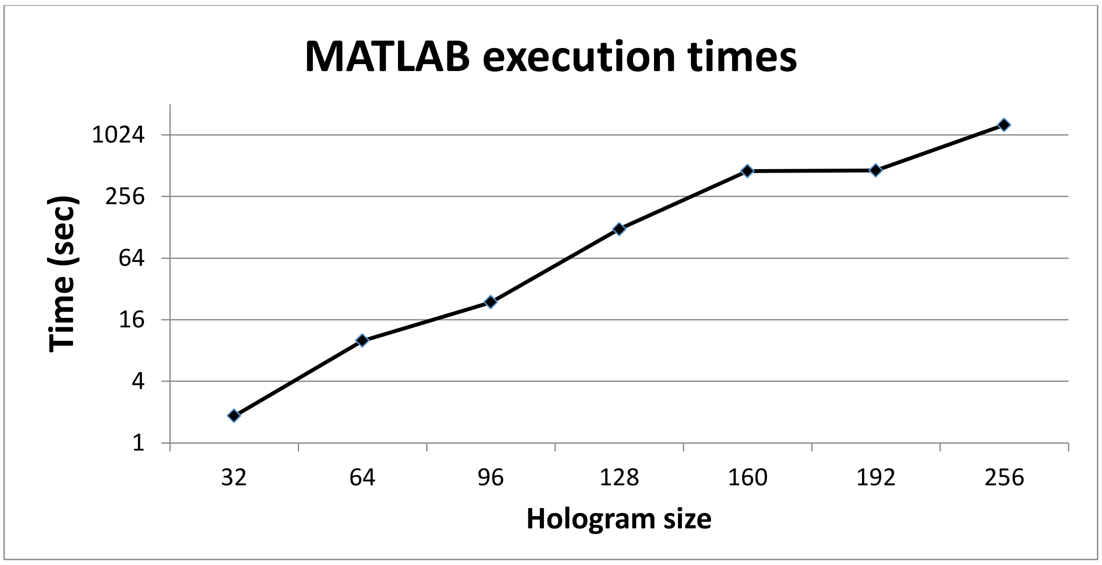

Figure 7. MATLAB execution time.

As you can see from Fig. 7, the results are quite poor; hologram size of 128x128 takes over 200 seconds (!) to compute.

However, the raw time value is a little bit out of context, 200 seconds might be a reasonable value if the time it takes to print the hologram takes two days.

On the other hand, if we wanted to change the focus of the target points in real time, this particular latency value is unbearable.

3.5 Summary

The background, use and importance of our project were explained.

We have also determined the bottlenecks of the MATLAB implementation and suggested a solution to each of them.

The algorithm we presented is intrinsically parallelizable; we want to focus on creating and evaluating multiple holograms at the same time to find the best possible solution. Furthermore, we want each hologram to be evaluated and processed by multiple threads simultaneously.

We approach this algorithm as a scientific computation and partially as a simulation. We want to minimize the time it takes to compute the hologram.

The performance of MATLAB implementation was measured and we will use this as a comparison to our future work on the project. We need some kind of feedback to see whether we are headed in the right direction.

4 Serial C implementation

4.1 Code analysis

There were a couple of specific MATLAB functions in the script that we had to rewrite in C. The purpose of these functions is among others: compute the standard deviation, compute magnitude, compute cost, create random permutation, etc.

As explained in chapter 3.3, there are two loops performing the whole computation. We are maintaining this concept in our serial C implementation; one loop evaluates the precision while the other goes through all elements of the hologram.

We can see the basic idea of the algorithm in the following pseudo-code:

```
while ( precision not reached ){
        create random index array()

        for ( all elements in the hologram ){
                take one index from the index array
                take one element at the index from the hologram
                compute the new state depending on this element

                if ( cost of the new state is lower than the previous cost ){
                        accept the change at the element
                        accept the new state and cost
                }
        }
}
```

We want to investigate the difference between the JIT compiled MATLAB script and the compiled C code.

The serial C implementation solved two of our initial issues:

- C is a compiled language
- We are using one piece of memory during the whole computation.

After rewriting the original MATLAB script to C language, it has shown that the performance difference between the two solutions is huge (see Fig. 8).

4.2 Performance

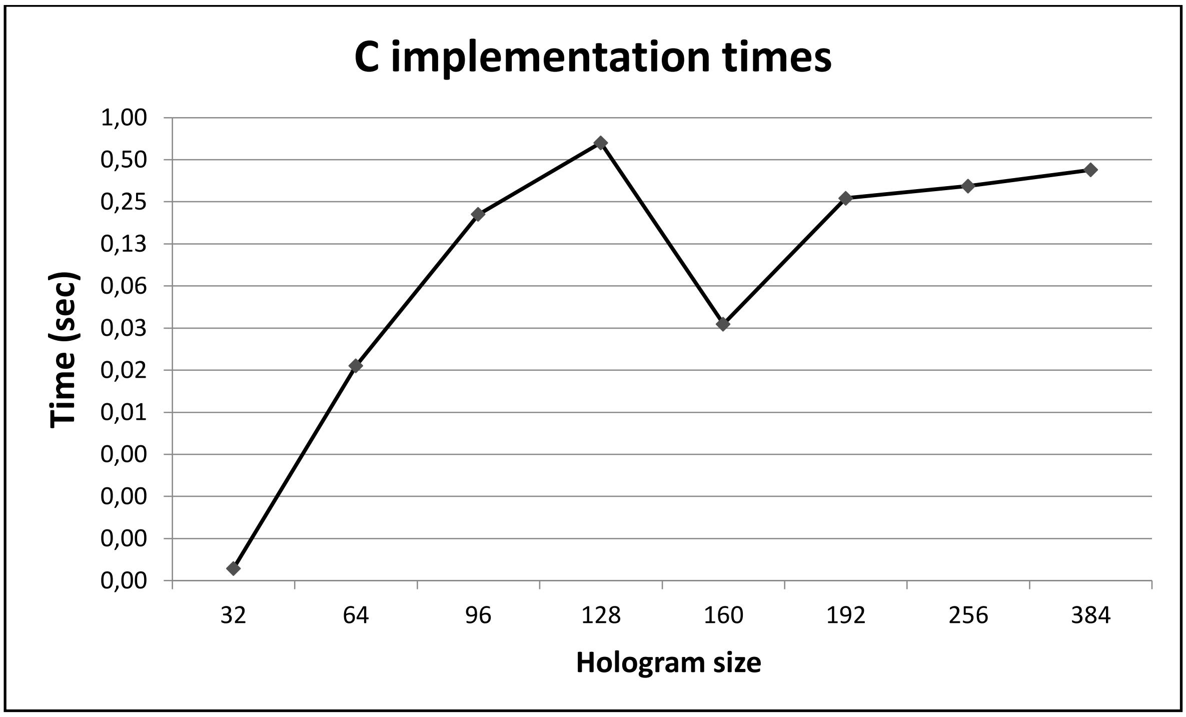

Figure 8. Execution of the sequential code written in C.

From Fig. 8, we can clearly see that the time necessary to create a hologram is significantly lower compared to the MATLAB version, no matter of the size.

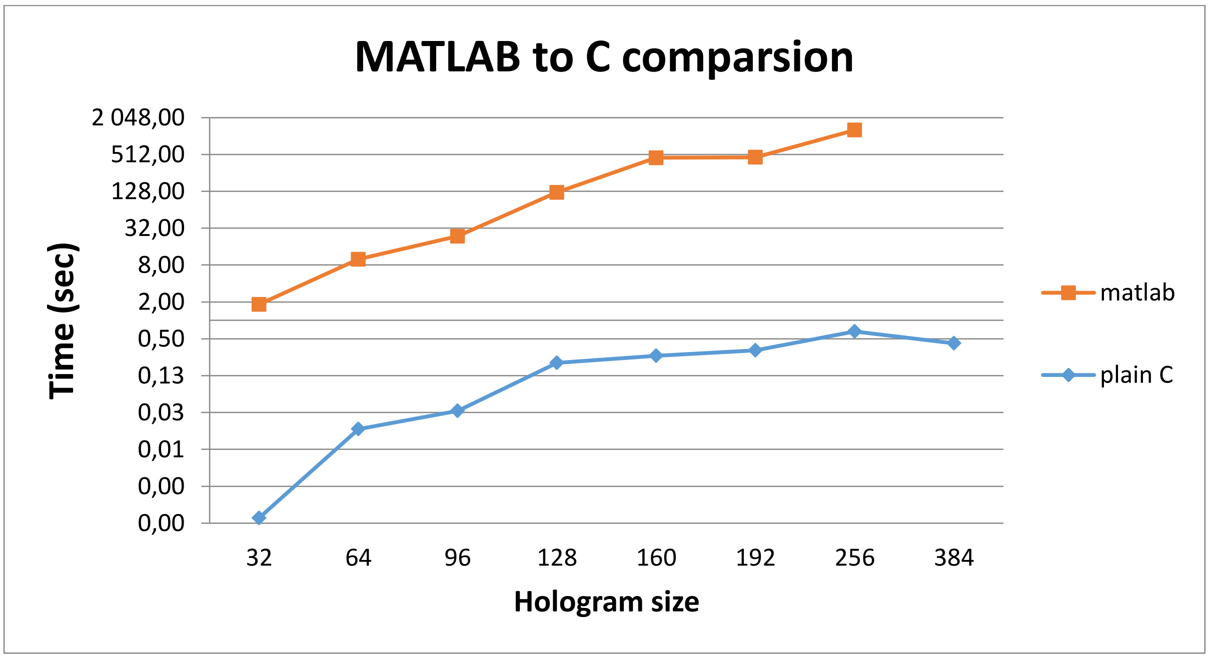

Figure 9: MATLAB to C implementation comparison

Figure 7 shows the difference between MATLAB and C implementations in execution times. Both vertical and horizontal axises are in logarithmic scale because the differences are enormous, reaching three orders of magnitude

4.3 Summary

Rewriting the original MATLAB version to the C language proved to make a huge performance difference in terms of execution time. By using C, we have solved issues 1 and 2 mentioned in chapter 3.3.

The C code also uses a random number generator, which does not make a big difference in performance and increases the quality of the final hologram.

However, this version of our code is still only working on one particular hologram at a time. By using multithreaded approach, we can run several threads or blocks and start the binary search from different initial states, which might end up with creating better cost evaluated holograms. This is the area we want to focus on and put our CUDA knowledge in practice.

5 OpenMP implementation

5.1 Code analysis

In our OpenMP implementation, we will focus on working simultaneously on multiple holograms at once, as mentioned in chapter 3.3. OpenMP works with threads, just like CUDA. These threads run on CPU, so there is no need to move data between CPU and GPU memory.

We chose to run one thread per one hologram we want to create. For example, we want the CPU to work on eight holograms at once. Then the number of threads we will run for the computation will be eight. One thread processes one hologram in its own separated memory space. This is the idea of how to start the binary search from different initial states, which should lead to creating better-evaluated holograms.

The algorithm each thread is performing is the exact same algorithm as the serial C implementation and was described in chapter 4.1.

5.2 Performance

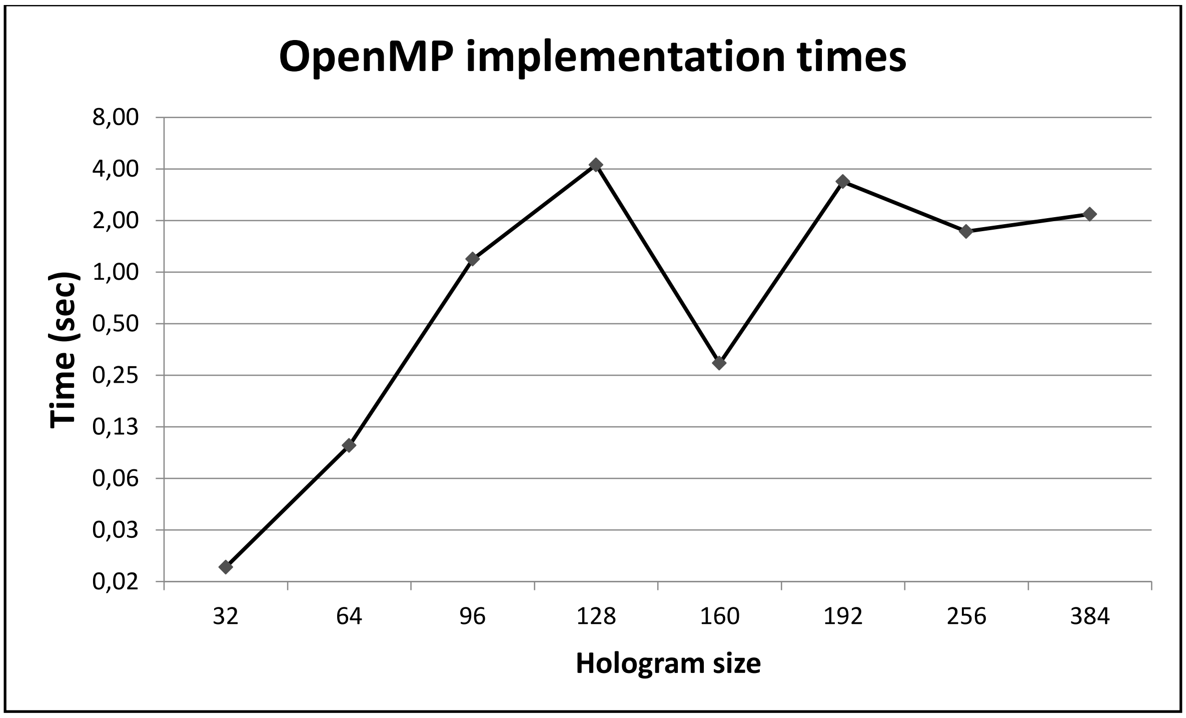

Figure 9. OpenMP execution times.

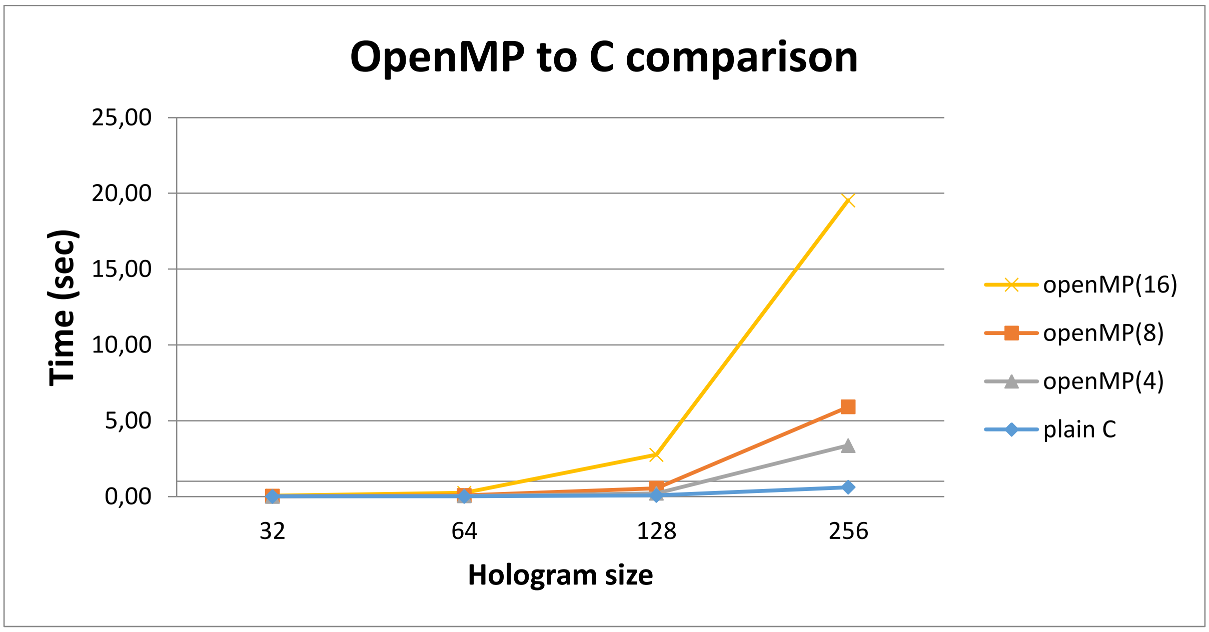

Figure 10. OpenMP compared to C version.

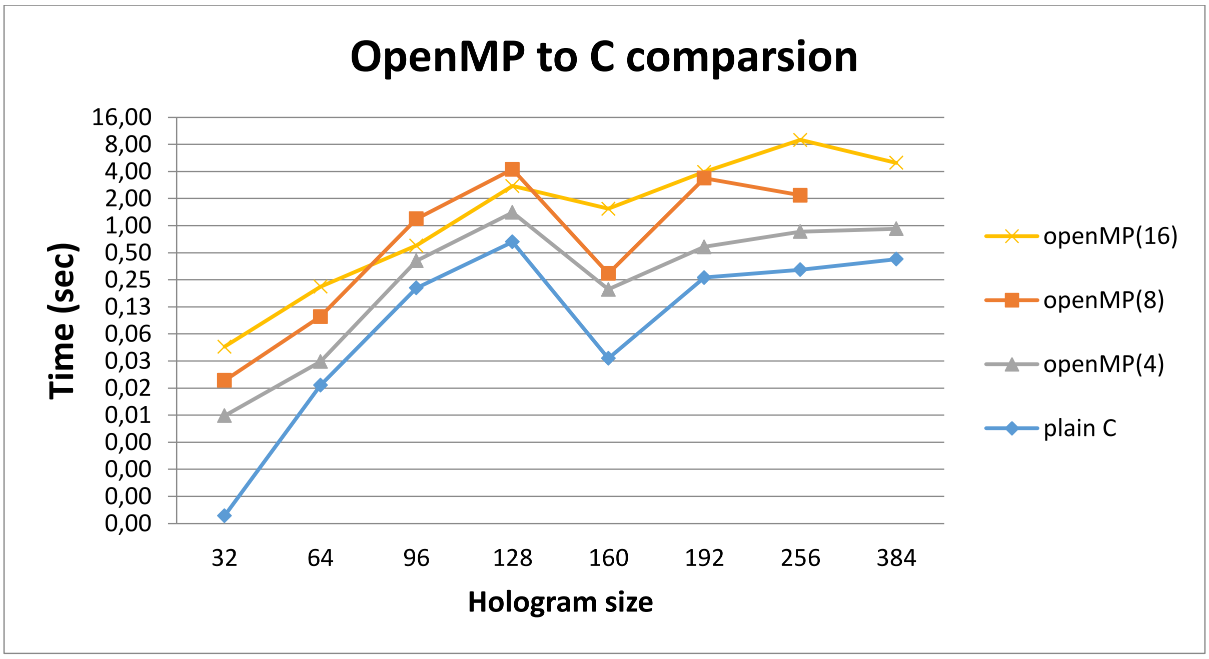

Figure 11. MATLAB, C and OpenMP version comaprison, logarithmic scale.

The number in brackets *()* means the number of holograms that were processed simultaneously.

From Figures 9 – 11, we can see the execution time of the OpenMP version sits right between the C and MATLAB versions, being just slightly slower than C, but still many times faster than MATLAB.

However, this is not an entirely fair comparison; the MATLAB and the serial C implementation are only processing one hologram, while the OpenMP version is performing computation on multiple holograms at the same time. A more detailed description will be presented in the chapter 8.

5.3 Summary

The OpenMP brought us into parallel hologram processing. We are still benefiting from C as a compiled language, we are still using one and only piece of memory for the whole calculation. The difference is, with the OpenMP version we are working on multiple holograms at the same time.

We can observe performance drop when increasing the amount of holograms that are processed at one time. We will try to eliminate this effect in the following kernels running od CUDA compatible GPUs.

6 CUDA implementation

As I have already mentioned, there are several ways how to implement the algorithm. We have created three kernels, each with a little different approach and performance.

I will describe all of our kernels in the following text, as well as the differences between particular versions and their performance.

6.1 Naive kernel

6.1.1 Code analysis

This is the first kernel we have implemented and is based entirely on the OpenMP version. That is one thread per one hologram. This solution is very straight forward, and easy to implement.

The downsides of this solution are:

- Memory requirements
- Not processing the hologram itself in a parallel manner

The two issues mentioned above are connected in a way – in this version, we use one thread to calculate one hologram, which means each thread has to have its own memory space, where the whole hologram and a few other data is stored. This implies huge memory requirements if we were to calculate a very large hologram, when it might not fit in the memory. For example, calculating a hologram of size 256x256 elements, using 1024 threads in 128 blocks would require the following memory space:

Hologram size		*integer(4B)*		*num. of threads*		*num. of blocks*	
256x256	x	4	x	1024	x	128	> 30GB

This is over 30GB of memory, which will not fit in the GPU. A simple solution to this problem is reducing the number of blocks to 4, which will fit in the GPU with total size of *1024* MB. The total number of holograms (256x256 elements) we can process using this kernel is 1024*4 = 4096 holograms (aka. the number of initial states from which we start the binary search). The number of holograms that can fit in the GPU memory is dependent on the hologram size.

The number of blocks and threads that perform the computation can be altered, resulting in different values and their combinations, but the idea of the restriction remains the same.

We can exploit the GPU memory and chip performance, because we can run more blocks and threads, increasing the chance of finding a better evaluated hologram. This will be described in the chapter 8.

In the Naive kernel, each thread is performing a sequential calculation, which is resulting in low performance and will be improved in the future kernels described in the following chapters.

6.1.2 Performance

This kernel was launched in the following configuration:

- 64 blocks,
- 32 threads in one block,

That is 64x32 = 2048 holograms processed at the same time.

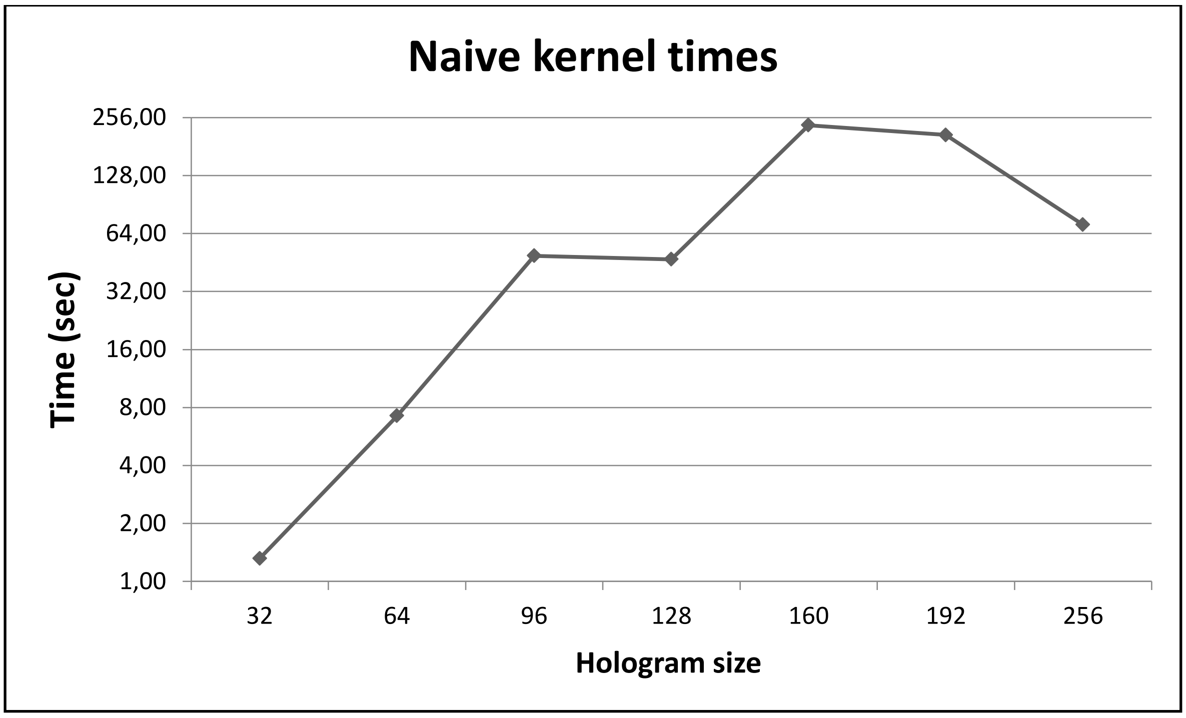

Figure 12. Execution times of the Naïve kernel.

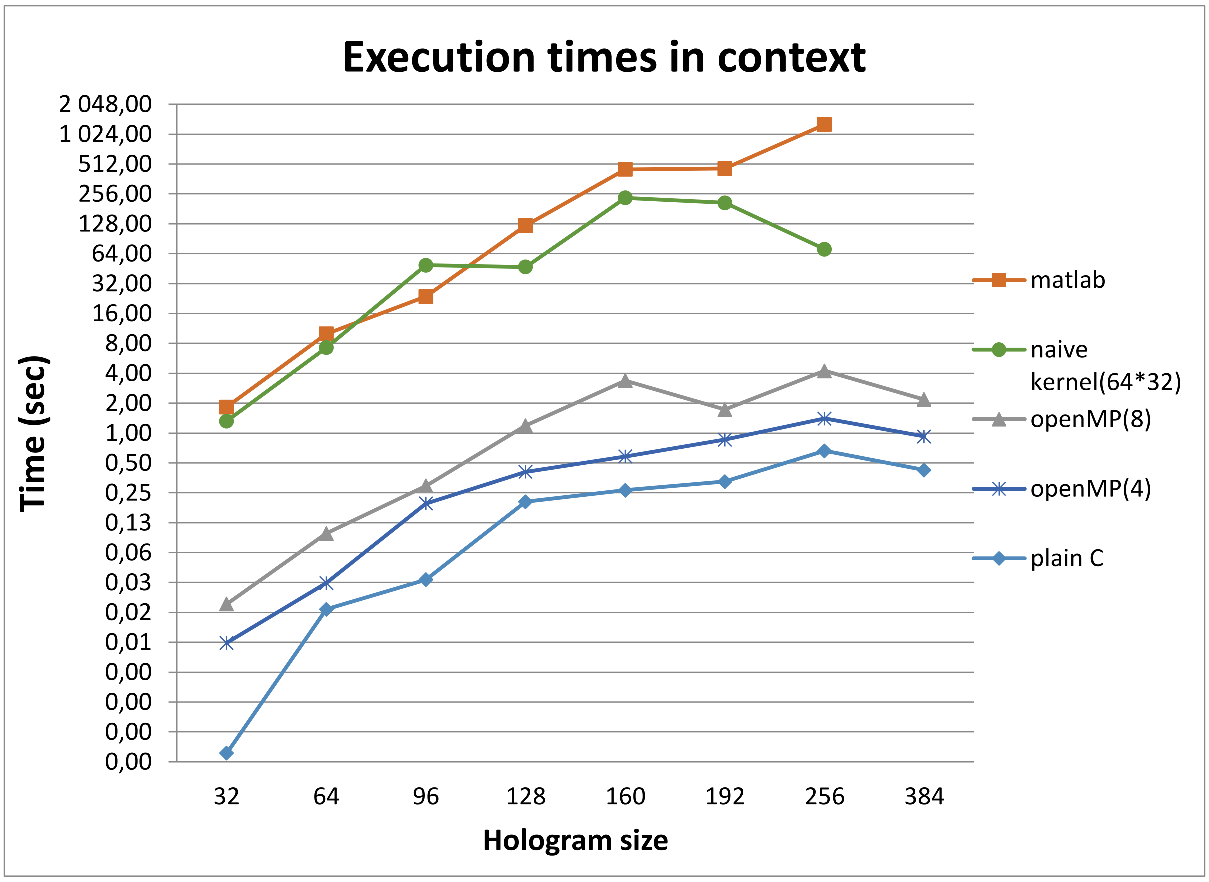

Figure 13. Naive kernel compared to other versions.

From the Fig. 13, we can see the Naive kernel performance compared to other versions of the algorithm. We can tell that the Naive kernel is much slower than the OpenMP (C) versions; actually, it is very close to MATLABS's performance.

As it is already mentioned in the text, the comparison on the graph is slightly out of context because MATLAB and serial C implementation only work on one hologram. Other versions compute several holograms at the same time (OpenMP processes up to 16 holograms while the Naive kernel can process 2048+ holograms, depending on their size).

6.1.3 Summary

For the sake of simplicity, we have 'ported' the OpenMP version of our algorithm to the GPU - CUDA and tried it out to see what sort of results we can achieve.

It turned out the results were poor – much slower than C versions, and just a tiny bit faster than the MATLAB version.

This kernel showed us the importance of organizing threads and blocks properly and using them in the right way to achieve the best results.

In the following kernels, we will try to evolve the Naive kernel, using slightly different approaches and techniques to demonstrate a proper usage of the GPUs. We will concentrate on parallel processing of several holograms at once with the use of blocks and threads.

6.2 Hologram in shared memory

6.2.1 Code analysis

As it is obvious from the name of this kernel, we have moved the whole hologram into the shared memory. Since the shared memory is accessible to all threads in a block, we now have one hologram per CUDA block.

The major advantage of this distribution is a much efficient use of CUDA threads. Threads now cooperatively load the hologram into the shared memory and then the whole hologram is divided equally among all threads and each thread does its own computation on its own data.

Because the shared memory is in use in this kernel, we have to ensure the threads will safely load/store data from/to the shared memory, we also have to use barriers to prevent threads using invalid data.

In addition, we do not use the random generator. We do not need to, because in this implementation we are starting the binary search at a large number of different, randomized holograms. The serial CPU version only randomly searches in a small surrounding of the possible solution; while on the GPU we can start searching from many (even thousands) initial holograms, increasing the chance of finding a better hologram. All those different holograms are then calculated in a slightly different way; their surfaces are slightly different. These nuances in the surfaces then make the difference between the final costs. They also make the difference in the hologram's ability to fulfil the required physical characteristics. From all the holograms we created, we can then choose the one that has the best cost.

All threads in one block are working on one hologram at the same time. Each thread is computing the cost of one element after another and once one element's cost is computed, all threads perform a reduction to determine, which thread computed the best fitting cost. Once the reduction is finished, the corresponding change in the hologram is made and the best cost is stored into the shared memory. The change and store is only made by one thread, the one that computed the best cost. The rest of threads wait for the one to perform the changes and then all threads copy the new best cost in order to continue their computation with valid data.

Let us demonstrate the algorithm on the following pseudo code:

```
Cooperatively load hologram into shared memory

while ( precision not reached ){

    for ( all elements belonging into thread’s compute space ){
        take one element from the hologram (each thread takes a different element)
        compute the new state and cost depending on this element
        save the new cost into an array of costs (shared memory)
        perform a parallel reduction on this array to find the best cost

        if ( thread’s local cost equals the best cost from the array){
            if ( cost of the new state is lower than the previous cost ){
                accept the change at the element
                accept the new state and cost
            }
        }
    }
}

Cooperatively store computed hologram into the global memory
```

After the whole loop has finished and the hologram has been calculated, the threads again cooperate on storing the hologram into the global memory, so that we can later retrieve the best hologram.

The major disadvantage of this kernel is the memory requirement. More specifically, the shared memory limitations. Modern GPUs have 64kB of shared memory, which is enough to store a hologram of the dimensions 96x96 elements. One element is an integer that takes 4 bytes of memory, 96x96x4 = 36864. A hologram of the dimension 128x128x4 would also fit in the shared memory all by itself, but we are using the shared memory for some other variables as well, so this is not possible.

6.2.2 Performance

This kernel was launched in the following configuration:

- 64 blocks,
- 32 threads in one block.

That is 64 holograms processed at the same time.

The influence of the number of holograms on the cost of the final state that is computed by the algorithm will be investigated in the chapter 8.

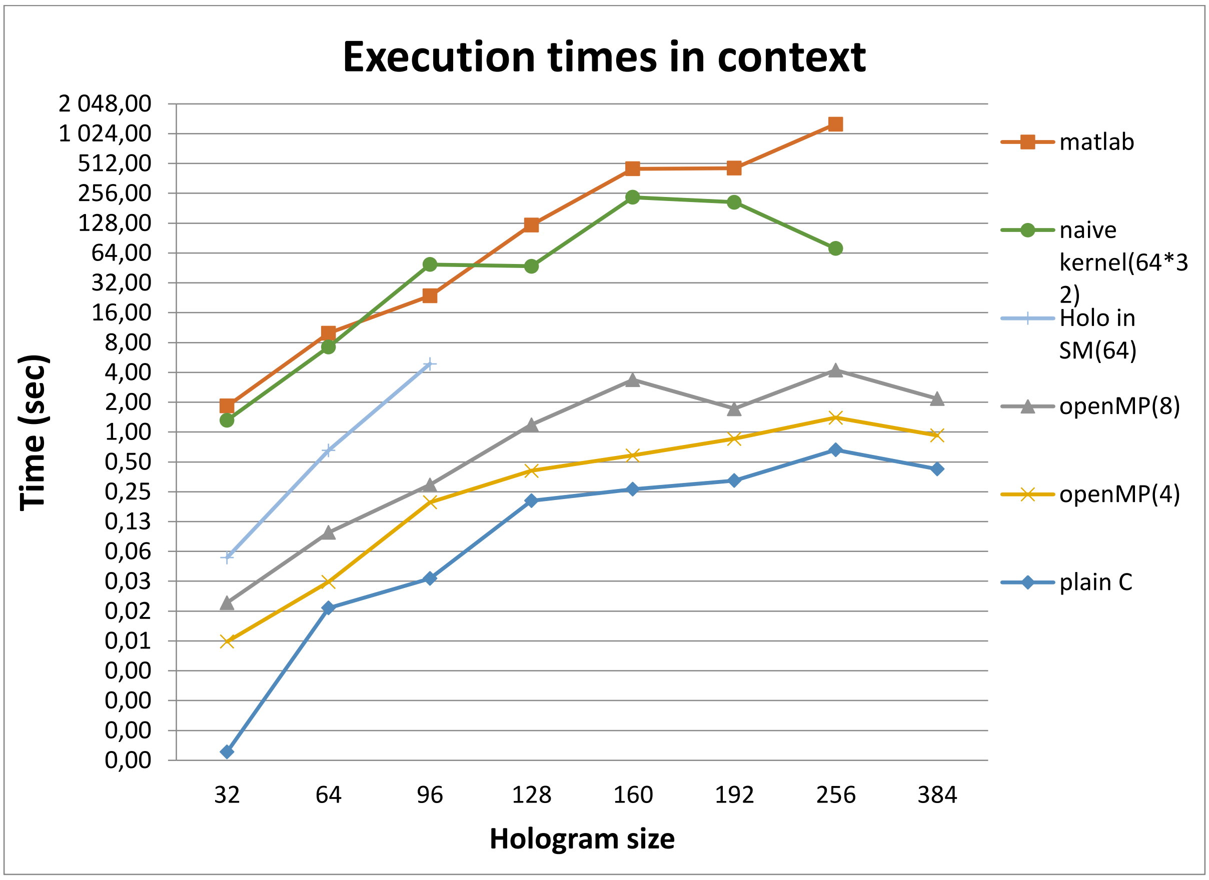

Figure 14. Hologram in SM kernel in context.

Unfortunately, due to the limitations mentioned in the previous chapter, we cannot compare this kernel performance on larger holograms. However, it is right between the naive kernel and the OpenMP implementation, which is exactly the spot one would expect it to take due to the facts mentioned in the previous text.

6.2.3 Summary

The Hologram in SM kernel is a good starting point to better parallel hologram calculation, thanks to its more efficient use of threads and blocks. By implementing this kernel we have also learned how to correctly load and store data from/to shared memory cooperatively and how to separate the work load among all the threads.

The work load separation is also the reason we can only use this kernel when the hologram size is not too large – with sizes that are square and are a power of 2 we can very easily manage thread’s workload.

If we take a look back to chapter 3.3 we can see we have solved all the problems mentioned there – we are using a compiled language, we are using pieces of memory that have only been allocated once, we are working on multiple holograms at once and these holograms are processed in a parallel manner.

The goal now is to come up with a kernel that is more efficient and more practical; we will describe this kernel in the following chapter.

6.3 Reduced integers kernel

6.3.1 Code analysis

The main issue with the kernel described in the previous chapter was its memory demand. The new version of that kernel, called Reduced integers kernel, has lowered memory requirements by reducing 32 integer elements into one element.

We have used the following idea:

An integer data type has 32 bits. The holograms we are creating are binary, which means their element's values can be either ones or zeros. To store a one or a zero you need exactly one bit. Knowing those facts, we can store 32 of the original elements (integers) into just one reduced element (integer), while the actual data semantics and hologram semantics is observed. See the picture below:

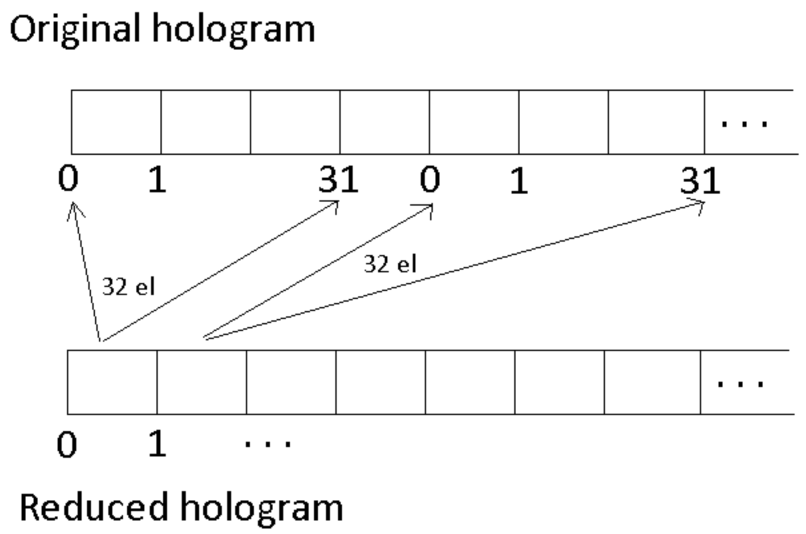

Figure 15. Reduced hologram kernel visualization.

The original hologram is the one that is created first in the function. This hologram is created in a usual way using an array of integers.

After the original hologram is created, we then reduce it following the principle mentioned above – we compact 32 original integer elements into one reduced element (a bit array).

Once the reduction is complete, the reduced holograms are then copied to the GPU global memory and from this memory, the threads cooperate on loading the reduced hologram into the shared memory.

Using this approach, we can store a hologram up the size of 384x384 original elements, which is a massive improvement to the previous kernel (only 96x96). We can of course store larger holograms, but the size 384x384 is the largest that was proofed to be calculated correctly.

The principle used to compute the hologram is the same as in the *Hologram in shared memory* kernel – each hologram is distributed over one block, threads in one block then cooperate on computing this hologram.

6.3.2 Performance

This kernel was launched in the following configuration:

- 64 blocks, that is 64 holograms processed at the same time.
- Number of threads in one block was chosen as high as possible to reach the best possible performance. The best possible configuration is summarised in Table II:

Table II: threads per block in the Reduced integers kernel

Hologram size	32	64	96	128	160	192	256	384
Threads in block	32	128	32	128	32	128	128	128

The value of 32 threads per block is then a universal configuration that works with all hologram sizes.

The number of blocks depends on how many holograms we want to process at the same time; the best compromise is 64 blocks. This number of blocks ensures great performance while keeping the final cost of the hologram higher then any serial version's cost (see chapter 7.3 for more details).

Furthermore, the Nvidia Tesla K20 GPU [9], that our binary search algorithm was executed on, can run 13 (SMs) x 2048 (threads per SM) = 26624 threads simultaneously. This implies that we can run up to 128 blocks at the same time, each with 128 threads when computing hologram of size 256x256 without losing computing performance (128 threads in 128 blocks makes a total of 128*128 = 16384 threads, using 128 threads in 256 blocks would make 128*256 = 32768 threads, which would not "fit" in the GPU, resulting in performance loss).

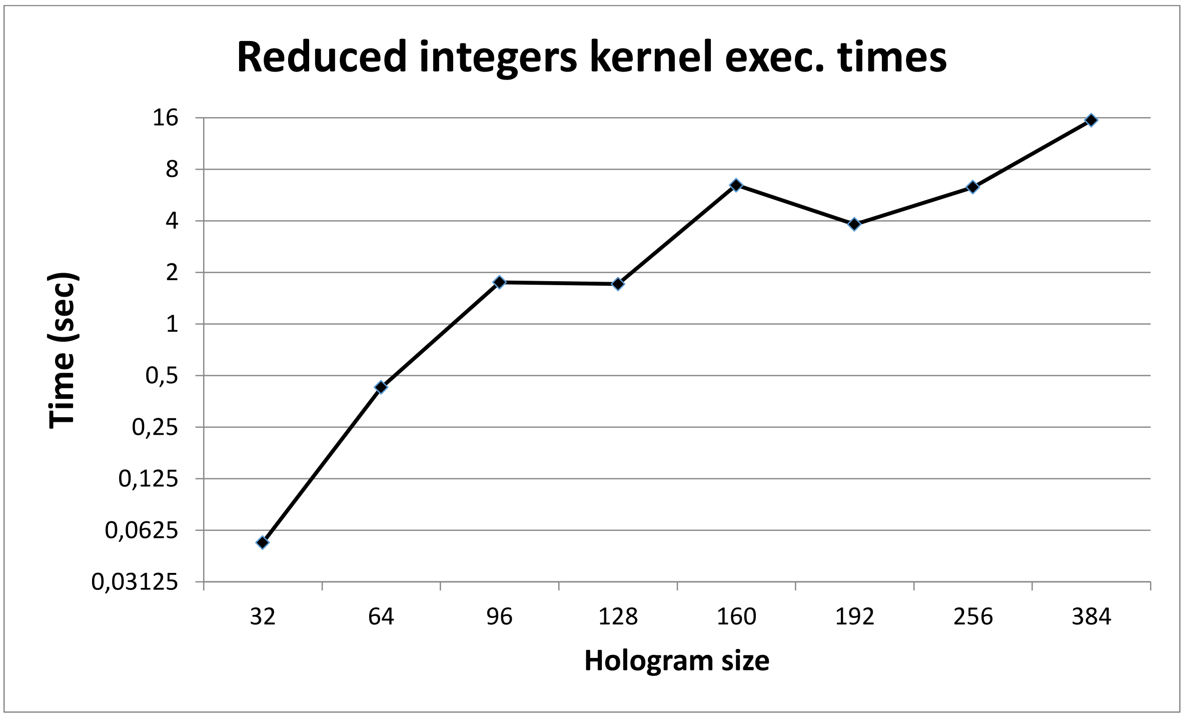

Figure. 16. Reduced integer kernel execution times.

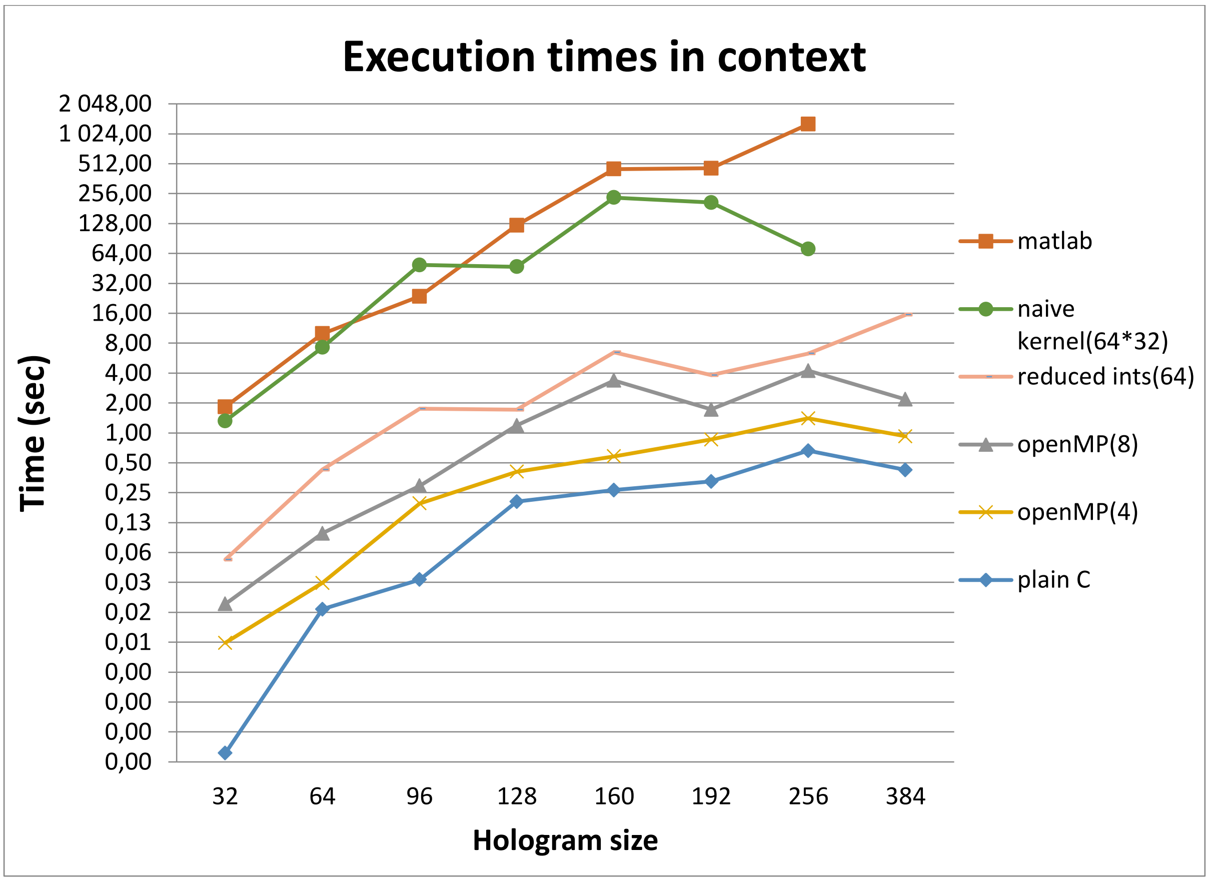

Figure 17. Reduced integer kernel execution times in context.

From the Fig. 17 we can see that our *Reduced integers kernel* is comparable to the OpenMP 16 threaded version. However, in the OpenMP version, the amount of holograms being processed at the same time (the number of threads stated in the brackets) is many times lower than the number of hologram we can process using the *Reduced integer kernel***.**

6.3.3 Summary

The Reduced integer kernel finally brought us into the CPU/C/OpenMP territory. While being slightly slower than the OpenMP version, we can compute multiple holograms at once without a reason to worry about the execution time.

On the other hand, there is still a minor disadvantage – memory requirements. Even with this advanced kernel, we are still limited by the size of the shared memory per block. To overcome this limitation we would have to use a different approach. For example, not using the shared memory at all and loading/storing data in a different way, dividing the hologram calculation into several phases, randomly choosing a part of the hologram from the global memory, etc.

7 Overall summary

Looking back to chapter 3.3, we managed to solve all of our issues with the original MATLAB script:

- We used the C programming language to avoid JIT compiling
- Allocating the required memory area once, before the computation started, has also been accomplished
- In our best implementations, we are working on multiple (even thousands) of holograms at the same time, which leads to creating better holograms (see chapter 8)
- In the *Reduced integer kernel*, each hologram is also processed by several threads simultaneously

All those points led to creating an application, whose performance was significantly faster than our starting point – the MATLAB script.

We came up with a few versions of the binary search algorithm, each presenting a slight performance improvement. The absolute fastest implementation is the *serial C*. This for example, can compute a hologram of the size 256x256 in *0.661153* seconds, compared to MATLAB's *1279* (!) seconds.

The serial versions however, are not processing the hologram in a parallel manner and they are not working on multiple holograms at once. That is why we used the CUDA platform, to explore the possibilities of parallel computation and apply them to our problem.

Using CUDA, we were able to compute the hologram of size 256x256 in 6.29 seconds, which is slower than *serial C*, but still many times faster than the original MATLAB script. On the other hand, our CUDA versions compute hundreds, even thousands of holograms at once, which leads to superior final hologram. If we were to compute even 64 holograms and reach a comparable cost of the hologram with the *serial C* version, it would take approximately *0.66 x 64 = 42 seconds.* The kernel that processes the same amount of holograms on CUDA takes *6.29 seconds.*

Since the *Reduced integer kernel*, as we called it, can compute many holograms simultaneously and those holograms are also randomly initialized, it does also achieve a better cost of the final hologram (described in chapter 8). It is up to the user's choice to decide between the quality and speed.

8 Conclusions

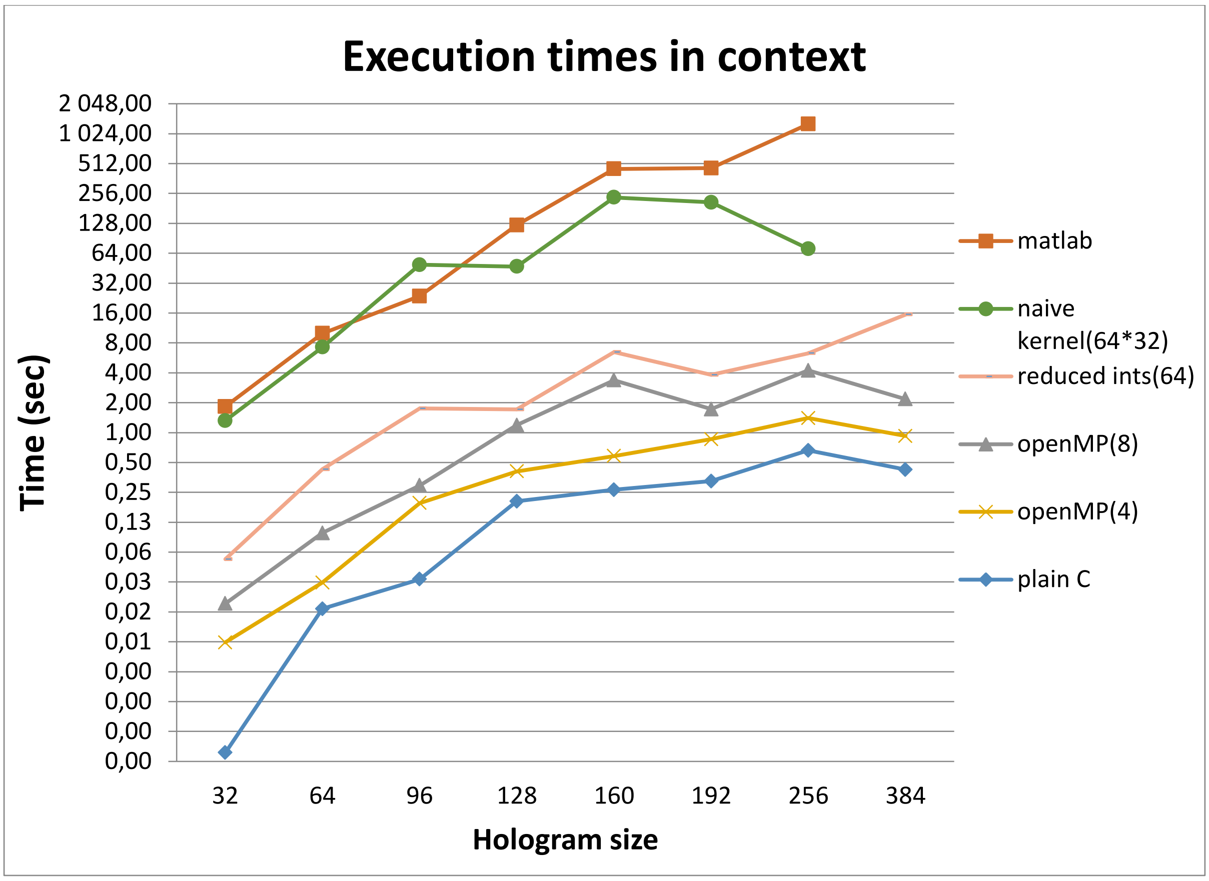

Figure 18. Particular solutions in comparison.

In this chapter we will discuss one particularly important aspect of our work – the cost (quality) of the final holograms. Since our primary task was to run the algorithm on a GPU, we will mainly discuss differences connected with the best kernel compared to the MATLAB or the serial C version.

If we take a look at the Fig. 20, we can clearly see that the serial C implementation was the fastest. However, there is an aspect, which might make you use some other solution but the serial C – it is the hologram's final cost.

The final costs depend on multiple factors:

- Hologram's target points
- Random initial hologram state
- Number of holograms computed at the same time

When measuring the solutions, we set the target points to be the same for all versions. In each experiment the hologram was initialized randomly, that is why we performed several experiments and then averaged the results. The last factor, the influence of the number of blocks computed at the same time will be discussed later in this chapter.

The cost itself is an important metric that defines how well the hologram fits the target points. The better the cost, the better the physical characteristics of the hologram will be.

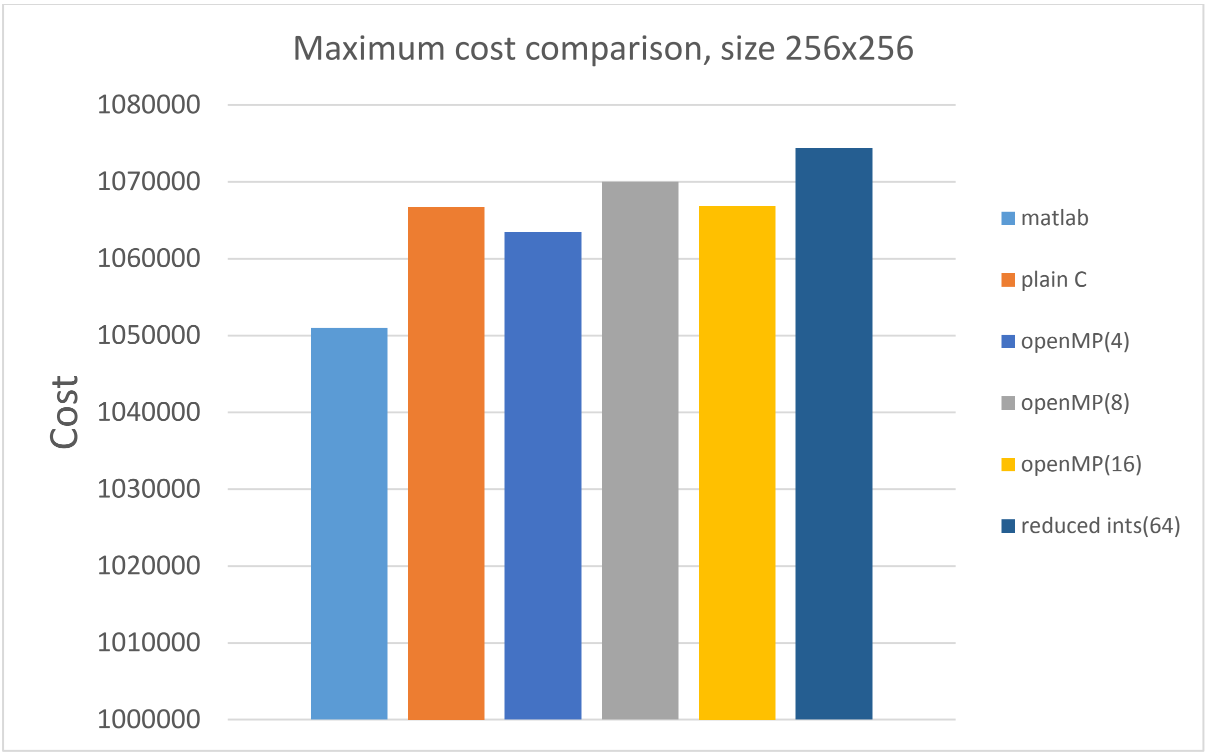

Figure 19. Hologram's final costs comparison.

In the Fig. 21 we can see the final costs (maximum cost from 10 experiments) comparison of the best implementations. Even though the serial C was the fastest approach, it did not produce quite the best hologram.

Our versions make a total difference about 1% - 2%, which does not seem significant, but in real world applications even a slight improvement in hologram quality can have a huge impact on the focus quality, especially when the hologram is used in an elastic medium (bones for example).

In general, if we are interested in the fastest approach, we shall use the *serial C* version. If on the other hand, we want to create a very accurate hologram and we are not in the need of saving as much time as we possibly can, we shall use the *Reduced integer kernel*, which will produce a superior hologram.

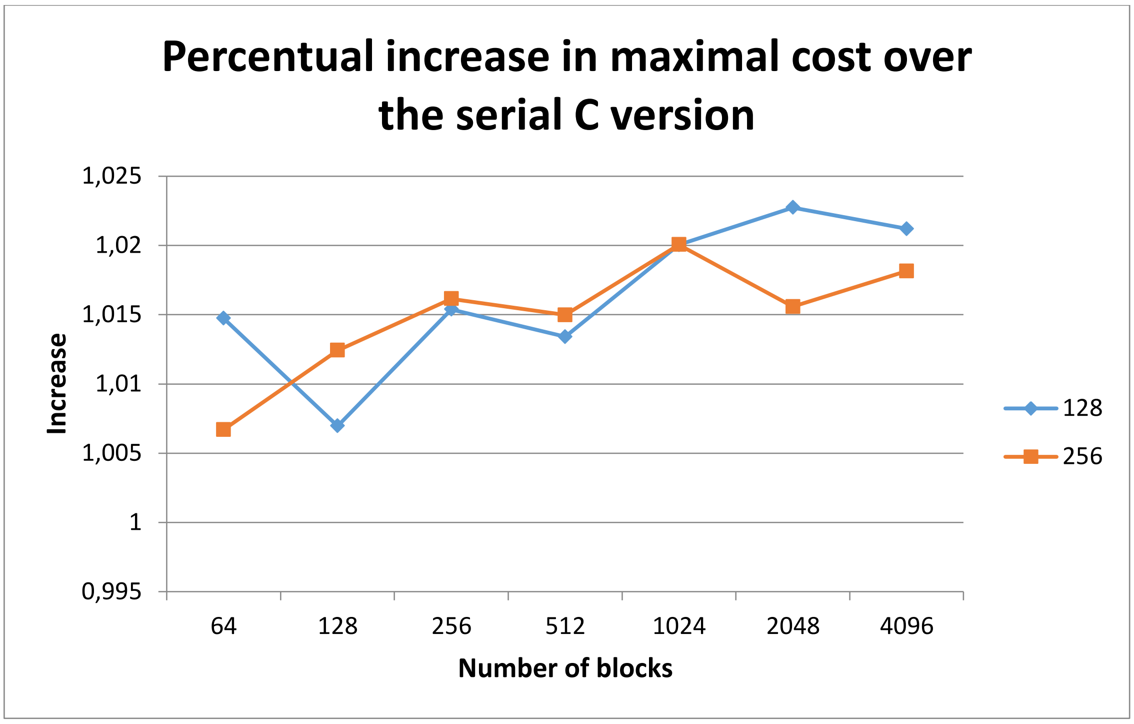

Figure 20. Reduced integer kernel best costs compared to serial C.

In the Fig. 21 we can see the influence of using more blocks to compute more holograms at the same time. It is a comparison of the *Reduced integer kernel* to the *serial C*. We can see the number of blocks launched (the number of holograms computed) on the horizontal axis. On the vertical axis, there is the value of

$$\frac{\text{Reduced integer kernel's cost}}{\text{serial C cost}}.$$

This value defines the multiple of *kernel's* cost over the *serial C* cost. Higher is better. It is also an average value of 10 performed experiments, both for the *serial C* and the *Reduced kernel* implementations. There is a clear patter in the graph – the more blocks we run, the higher chance there is we will compute a better cost-evaluated hologram.

9 Hologram examples

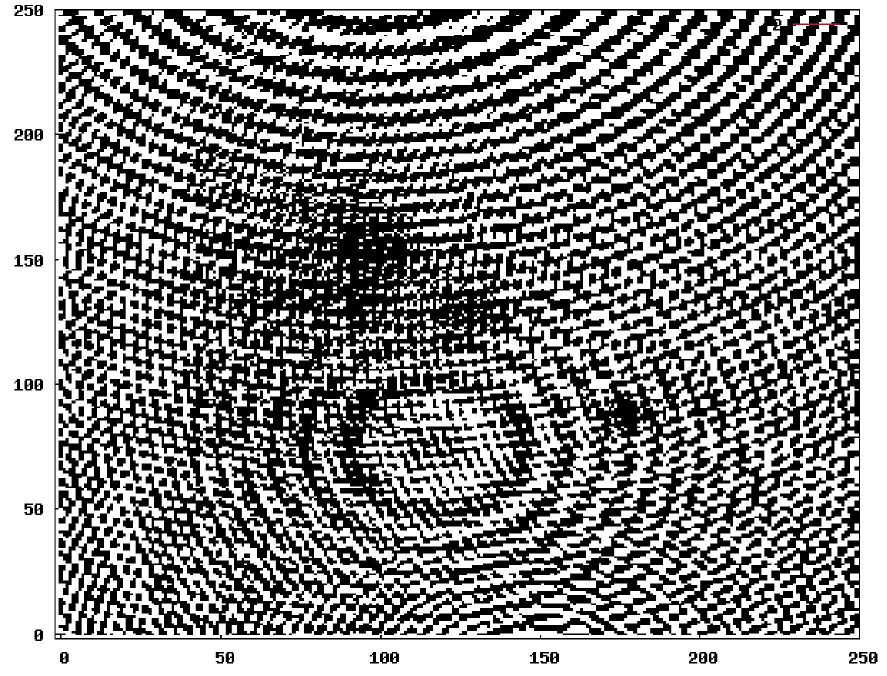

Figure 21. Naïve kernel generated 256x256 hologram example. Defects visible.

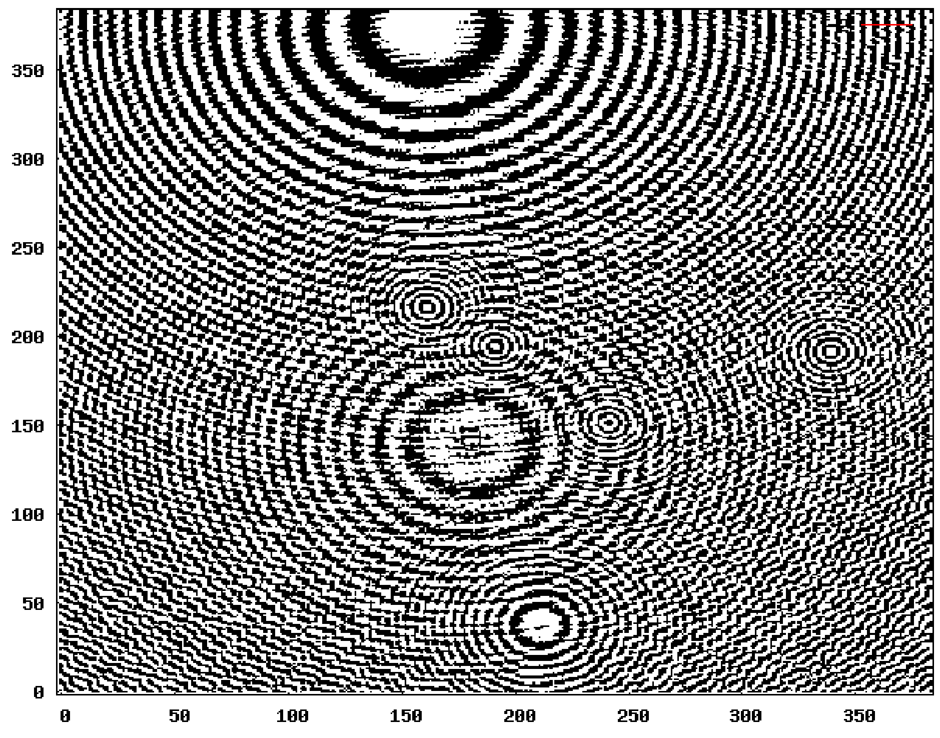

Figure 22. Reduced integers kernel generated 384x384 hologram. No defects.

10 References

[1] BROWN Michael D., JAROS Jiri, COX Ben T. and TREEBY Bradley E. Control of Broadband Optically Generated Ultrasound Pulses Using Binary Amplitude Holograms. *The Journal of the Acoustical Society of America*. 2016, vol. 139, no. 4, pp. 1637-1647. ISSN 1520-8524.

[2] CUDA Device Query example, run on pcjaros-gpu.fit.vutbr.cz

[3] Program 'lscpu' run on pcjaros-gpu.fit.vutbr.cz

[4] Randperm function, http://www.mathworks.com/help/matlab/ref/randperm.html

[5] Ind2Sub function, http://www.mathworks.com/help/matlab/ref/ind2sub.html

[6] OpenMP documentation, http://openmp.org/openmp-faq.html#OMPAPI

[7] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-purpose GPU Programming. Upper Saddle River, NJ: Addison-Wesley, 2011. Print.

[8] M. Clark, "A direct search method for the computer design of holograms," Ph.D. thesis, Imperial College, London, UK, 1997.

[9] Nvidia Tesla K20 specifications, https://www.microway.com/hpc-tech-tips/nvidia-tesla-k20-gpu-accelerator-kepler-gk110-up-close/