VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

RYCHLÁ REKONSTRUKCE OBRAZU TKÁNÍ S VYUŽÍTÍM GRAFICKÉ KARTY

FAST TISSUE IMAGE RECONSTRUCTION USING A GRAPHICS CARD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE KRISTIÁN KADLUBIAK
AUTHOR

VEDOUCÍ PRÁCE ING. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Fotoakustická spektroskopia je jedna z najmodernejších zobrazovacích metód a nachádza uplatnenie vo vedných odboroch ako medicína, biochémia, materiálová technológia a mnoho ďalších. Vďaka svojim vlastnostiam je fotoakustická spektroskopia veľmi vhodná špecificky pre medicínske účely. Táto metóda je neinvazívna a zároveň zaručuje vysokú presnosť zobrazenia. Za vysokú presnosť metóda vďačí pokročilým, časovo náročným výpočtom, medzi ktoré patria operácie ako FFT a trilineárna interpolácia. Táto bakalárska práca sa zaoberá akceleráciou daných metód na grafickej karte. Naša implementácia naplno využíva rozličné vlastnosti moderných grafických kariet ako napríklad zdielaná pamäť alebo textúrový hardware. Implementáciu sme testovali na jednej z najvýkonnejších grafických kariet určených na high performance computing. Jednalo sa o kartu NVIDIA K20m. V tomto prostredí sa našej implementácií podarilo zrýchlit niektoré časti rekonštrukcie viac než 400-násobne. V jednorazovom móde rekonštrukcia trvala o niečo dlhšie než samotná MATLAB verzia. Je to spôsobené nutnosťou prevodu dát medzi prostredím MATLAB a CUDA kódom, i keď sa podarilo znížiť veľkosť prenášaných dát o 37%. Spracovanie väčších dávok fotoakustických snímkov by ukázalo skutočný potenciál implementácie.

Abstract

The photoacoustic spectroscopy is a recently developed imaging method that finds applications in many scientific fields such as medicine, biochemistry, materials engineering and many others. The photoacoustic spectroscopy finds particularly nice applications in medicine due to its properties such as non-invasiveness, non-aggressiveness and great accuracy. The source of this accuracy lies in advanced time-consuming calculations including operations like FFT and trilinear interpolation. This thesis is dedicated to the acceleration of this technique on a graphics card. In our implementation, we have taken a full advantage of various features provided in modern GPUs such as shared memory and texture hardware. Our implementation has been tested on one of the most powerful GPU designed for high performance computing, namely NVIDIA K20m. In this environment, our application speeds up certain parts of reconstruction by a factor above 400. In a single run mode, the whole reconstruction runs a bit longer than the pure MATLAB version due to the necessity of transferring data between MATLAB and the CUDA code, although the developed approach reduced the data transfers between MATLAB and GPU by 37%. The real potential of the implementation reveals while processing large batches of photoacoustic images.

Klíčová slova

Fotoakustická spektroskopia, rekonštrukcia obrazu, GPGPU, CUDA.

Keywords

Photoacoustics spectroscopy, image reconstruction, GPGPU, CUDA.

Citace

Kadlubiak Kristián: Fast Tissue Image Reconstruction Using a Graphics Card, bakalářská práce, Brno, FIT VUT v Brně, 2015

Fast Tissue Image Reconstruction Using a Graphics Card

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiří Jaroš, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Kristián Kadlubiak
17.05.2015

Acknowledgement

This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).
I would like to thank my supervisor Ing. Jiří Jaroš, Ph.D. for all his support and kind approach.

© Kristián Kadlubiak, 2015
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Contents 1
1 Introduction 3
1.1 Photoacoustic spectroscopy 3
1.2 The k-Wave toolbox 3
2 Graphic processing unit 4
2.1 Main differences between GPU and CPU 4
2.2 GPGPU 5
2.2.1 GPGPU frameworks 5
2.3 CUDA-capable GPU architecture 6
2.3.1 Host interface 7
2.3.2 Copy engine 7
2.3.3 DRAM adapter 7
2.3.4 Device memory 7
2.3.5 Streaming multiprocessor 7
2.4 CUDA thread execution model 8
2.4.1 Kernel 9
2.4.2 Grid 9
2.4.3 Block 9
2.4.4 Thread 9
2.4.5 Warp and lane 10
2.5 CUDA memory model 10
2.5.1 Global memory 10
2.5.2 Local memory 11
2.5.3 Registers 11
2.5.4 Shared memory 11
2.5.5 Texture memory 11
2.5.6 Constant memory 11
2.6 GPGPU capabilities 12
3 MATLAB implementation 13
3.1 Identifying acceleration opportunities 13
3.2 Trilinear interpolation 14
3.3 Fftshift and ifftshift 16
3.4 Fast Fourier transformation 16
4 Implementation 17

4.1 Common features 17
4.2 The first version 18
4.2.1 Fast Fourier transformation 18
4.2.2 Fftshift and ifftshift 20
4.2.3 Trilinear interpolation 22
4.3 The second version 25
4.3.1 Fftshif and ifftshift 25
4.3.2 Trilinear interpolation 28
4.4 The third version 31
4.4.1 Trilinear interpolation 31
4.5 Discussion 34
5 Conclusion 36
6 References 38

1 Introduction

In this bachelors thesis we focus on possibilities of accelerating scientific computations by graphics processing unit. The term scientific computation in our case means reconstruction of tissue image obtained via photoacoustic spectroscopy. We use an open-source MATLAB toolbox k-Wave specially designed for purposes of tissue image reconstruction as a reference to validate our implementation. Our main goal is to design and implement efficient solution, which will lead to maximal acceleration of image reconstruction. To be able to perform such a task, it is essential to understand photoacoustic spectroscopy and study principles used in the k-Wave toolbox and MATLAB to reconstruct the image. The task also requires gathering knowledge about architecture of graphics processing unit in general and familiarization with possible means of programing of graphics processing unit. Afterwards, we have to fully understand principles and details of programing platform and design and implement solution according to gained knowledge.

1.1 Photoacoustic spectroscopy

Photoacoustic spectroscopy is based on photoacoustic effect discovered by Alexander Graham Bell in 1880. Photoacoustic effect appears when electromagnetic energy is absorbed by a sample of matter, which results in heating and expansion.[2] This process can be measured as the ultrasonic waves are produced by rapid expanding matter. As a source of electromagnetic radiation is often used a laser and intensity has to vary, either periodically modify or pulse modify.[19] Photoacoustic effect can be used to determine certain features of examined sample. In biomedicine, Photoacoustic effect is used as a non-invasive imaging method.

For example, photoacoustic imaging is very useful in studies of a vascular system development of embryo *in vivo*, as it is a non-invasive, non-aggressive method. The vascular system imaging is possible by significant difference in light absorption of haemoglobin circulating in vessels and surrounding tissue.[19] This same principle is used in tumour angiogenesis monitoring because cancer tissue is largely supplied with blood vessels, which provides sufficient contrast to differentiate tumour and healthy tissue. The photoacoustic imaging can be used in many others diagnostic procedures, such as blood oxidation mapping, functional brain imaging and skin melanoma detection.

1.2 The k-Wave toolbox

The k-Wave is open source third party toolbox for MATLAB developed for simulation of photoacoustic wave fields in either homogeneous or heterogeneous material in one, two or three dimensions and reconstruction of wave fields obtained via photoacoustic spectroscopy.[16] To make photoacoustic spectroscopy efficient tool for research, medicine and industry it is essential to make k-Wave a fast solution. Therefore, there is lot of focus on speed of simulation and reconstruction. In this matter, several significant measures were taken.

Photoacoustic wave equations are partial differential equations and they are used in simulation of wave fields in k-Wave. The most common numerical methods for solving partial differential equations are finite-difference, finite-element and boundary-element method.[16] Common methods achieved unsatisfying results in the simulation, as they were time-consuming. Major disadvantages of traditional methods are many grid points per wavelength and small time-step

size to minimize numerical error. Therefore, pseudo-spectral and k-space methods were implemented. These methods have their own disadvantages, but disadvantages were suppressed by special techniques. The pseudo-spectral method is based on Fourier series, which can be efficiently calculated by fast Fourier transformation. Only two grid points per wavelength are needed when the pseudo-spectral method is used. The pseudo-spectral method brought improvement in a spatial domain. The k-space method is used to achieve improvement in time domain, because it allows greater time-steps while preserving accuracy.[16]

Quality of simulation is very important because same principles can be used in reconstruction of photoacoustic image, so quality of reconstruction is dependent on quality of simulation techniques. The k-Wave allows two methods Time Reversal Image Reconstruction for arbitrary sensor shape and One-Step Image Reconstruction for a planar measurement surface.[16]

Besides special methods implemented to improve application's performance, there is also another way of increasing speed of algorithms. Parallelism and optimization techniques could be used to improve the performance. Acceleration on GPU is supposed to have a large impact on performance, as k-Wave is working with large amount of data.

2 Graphic processing unit

At beginning of computer graphics all necessary calculations were done by central processing unit (CPU). As computer graphics became more complex CPU got overloaded with graphical computation and performance of CPU declined rapidly. This trend resulted in development of certain dedicated hardware for accelerating graphical computation. This kind of specific hardware is today commonly known as a graphical processor unit (GPU).

The GPU is an electronic circuit specially designed to accelerate creation of images in the display buffer which is then displayed on a video device. Modern GPU possess highly parallel architecture very efficient in calculations with a block of data up to 5 GB. This feature is widely used not only in computer graphics, but also in physical calculation, simulations and generally in a high-performance computing. The first generation of GPU was designed as fixed-function accelerators with a limited set of instructions. This architecture proved itself insufficient as computer graphics being evolved over time. The need for programmable GPU resulted in the development of programmable GPUs.

2.1 Main differences between GPU and CPU

The main difference between CPU and GPU is in their architecture. Nowadays, CPU is composed of several cores, therefore a few different processes can run on CPU at a same time. CPU also contains great hierarchy of caches which makes them optimized for context switching and complex calculations.

On the other hand GPU provide much greater level of parallelism and therefore much greater throughput. For example, GeForce GTX TITAN is equipped with 2688 cores capable of floating-point operations compared to Intel Haswell architecture containing eight cores each of which is equipped with AVX2 capable of producing 32 floating-point operations per cycle.[8][14] We can see that there is significant difference in maximum count of operations per cycle for each architecture. However, we have to take in consideration that clock rate of GPU is about one third of CPU, depending on specific models. Despite this fact GPU can easily outperform CPU on specific type of problems. In fact theoretical single-precision performance of GPU GeForce GTX TITAN is about 5x greater than theoretical performance of Intel Haswell architecture. It is important to mention that GPU lacks optimizations like long pipelines and out of order execution important for general-purpose performance. Thus, not all problems are suitable to be accelerated on the GPU.

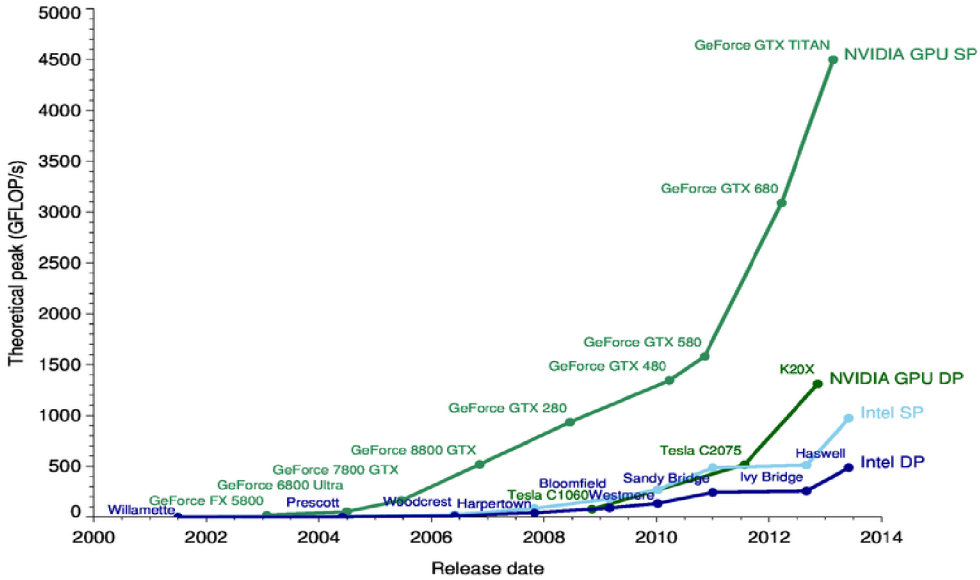


Figure 1 GPUs and CPUs performance benchmark [5]

2.2 GPGPU

The term GPGPU stands for general purpose computing on graphics processing unit. The GPGPU is name given to a concept in which GPU features are exploited to accelerate computations usually handled by the CPU. This concept is vastly used for an acceleration of calculation involved in fields like bioinformatics, molecular biology, image processing, particle physics and many others.

2.2.1 GPGPU frameworks

The GPGPU framework is platform which contains mechanisms that allows transferring computation on GPU. Two main platforms are open-source OpenCL framework and CUDA framework.

OpenCL is open-source standard for cross-platform parallel programing developed and maintained by Khronos group. Its main purpose is to enable writing applications that can be executed across heterogeneous devices such as CPU, GPU, digital signal processor, FPGA and many others. OpenCL can be used standard languages as C or C++ for programing purposes. OpenCL defines API to be able to control and execute code on various devices. It implies that key feature of OpenCL standard is compatibility with various devices created by various vendors.

CUDA stands for compute unified device architecture. It is proprietary platform for parallel computing and programing model developed by graphics card vendor NVIDIA. It provides mechanisms to be able to write and execute applications, which exploit GPU to accelerate computations. Main advantage of CUDA results from fact that it is a proprietary platform and therefore CUDA is optimized for a use with NVIDIA GPUs.

All things considered, for purposes of this thesis we chose CUDA platform, because, as aforementioned, CUDA provide better results than OpenCL when used with NVIDIA graphic card present in our testing environment.

2.3 CUDA-capable GPU architecture

CUDA is supported by four different microarchitectures:

- Tesla microarchitecture firstly presented in 2006, with GeForce 8800 GTX
- Fermi microarchitecture firstly presented in 2010, with GeForce GTX 480
- Kepler microarchitecture firstly presented in 2012, with GeForce GTX 680
- Maxwell microarchitecture firstly presented in 2014, with GeForce GTX 750

Although, there is difference between each architecture, all architectures possess common hardware features:

- Host interface that connects GPU with CPU via PCI express bus
- Copy engines
- DRAM adapter, which interconnect GPU and its device memory
- Device memory and caches
- Certain number execution units organized in so called streaming multiprocessors

Figure 2 GPU architecture of Kepler class [11]

2.3.1 Host interface

Host interface is responsible for all communication between CPU and GPU. It includes reading of Command buffer which is special CPU memory area used to submit commands. Host interface is in charge of reading commands from this special memory area. Host interface also decodes and delegates commands further to GPU.

2.3.2 Copy engine

Copy engine is hardware capable of preforming memory transfers between CPU and GPU, while computation is being done on GPU. First microarchitectures do not feature a copy engines. Later on copy engines were only capable of transferring linear device memory. Today, GPUs are equipped with up to two copy engines, which can convert between CUDA arrays and linear memory. Two copy engines provide full-duplex memory transfers.

2.3.3 DRAM adapter

Memory operations bandwidth and latency have a great impact on GPU performance, therefore GPUs possess powerful DRAM interface, which provides bandwidth high above 100GB/s and includes hardware support for merging multiple memory operations. Earliest hardware required contiguous memory addresses and memory alignment. With introduction of SM 1.2 requirement for memory alignment was removed. However there is still a performance penalty.

2.3.4 Device memory

The device memory is an equivalent of RAM memory of CPU. In this memory all data transferred from CPU are stored. For example NVIDIA Tesla K20 is equipped with GDDR5 memory with capacity 5 GB and throughput of 208 GB/s.[9] The global memory is cumbersome and slow, therefore L2 cache is present in modern GPUs to enhance main memory performance.

2.3.5 Streaming multiprocessor

The main component of GPU is streaming multiprocessor (SM), which is in charge of all computations. Number of SMs on card is model-specific, but architecture of SM remains in the main the same. Each multiprocessor consists of:

- Execution units capable of 32-bit integer, single-precision and double-precision floating-point arithmetic.
- Special function units for computing single-precision approximations of mathematical functions (log, exp, sqrt, sin, cos, etc.)
- Instruction cache, warp scheduler and dispatch unit for scheduling and dispatching instruction execution by execution units
- Load/Store units
- Register field for storing local variables
- Shared memory with L1 cache for communication between threads and storing temporary result
- Constant cache for broadcasting constant variable to each thread

- Cache texture hardware with various functions (1D, 2D, 3D prefetching, interpolation etc.)

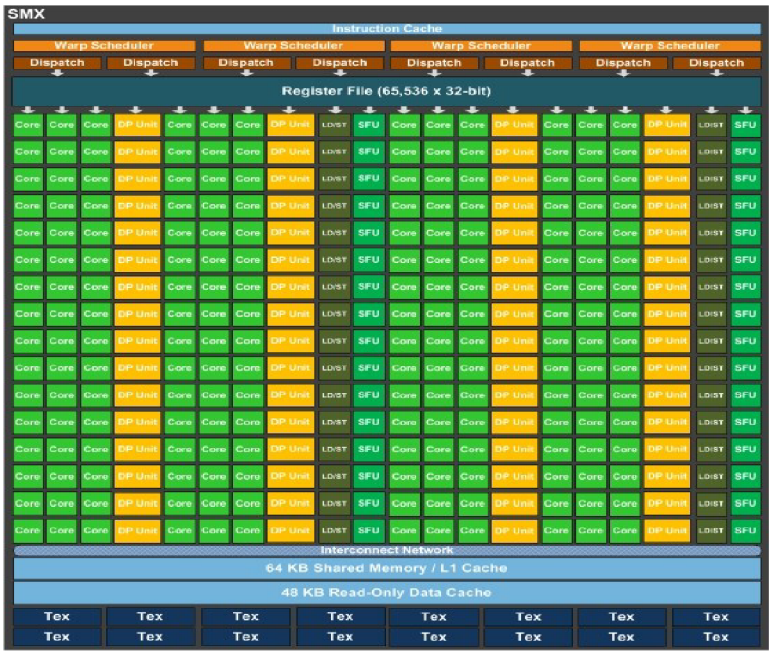


Figure 3 streaming multiprocessor architecture of Kepler class GPUs [10]

2.4 CUDA thread execution model

Thread arrangement has a significant influence on execution time of application using CUDA. Arrangement of threads is specified by grid size and block size. The important aspect of thread execution model is the concept of warp.

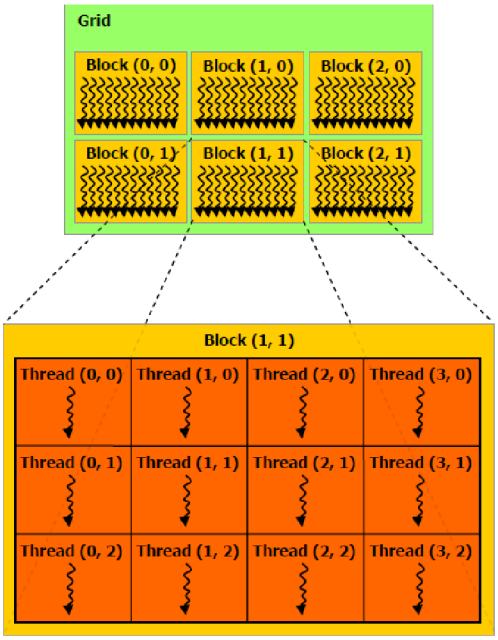


Figure 4 CUDA thread execution model [12]

2.4.1 Kernel

The kernel is equivalent of procedure found in common programing languages. Kernels are part of application that are computed on GPU. Kernels is declared by keyword `__global__` . Launch of kernel is similar to traditional function or procedure call, only difference is presence of special triple angel bracket construction in which grid size and block size is specified.

2.4.2 Grid

The size of grid specifies number of blocks in three dimensions. The maximum size of the grid can be up to 65535 x 65535 blocks for 1.x computation capable hardware and 65535 x 65535 x 65535 blocks for 2.x computation capable hardware. The blocks within the grid tend to be assigned on different SM to maximize performance, although few different blocks can reside on the same SM.

2.4.3 Block

The block is abstraction of independent execution unit. It is a group of threads which are executed on the same SM. Block size can be also specified in three dimensions and the block can contain up to 512 threads in total for 1.x computation capable hardware and 1024 threads in total for 2.x and above computation capable hardware. The CUDA provide mechanisms for inter-block communication and synchronization.

2.4.4 Thread

The thread is elementary part of execution. Each thread has its own unique Id within the block. Resolving global Id of the thread is essential, as it is only mechanism to assign correct portion of the work to the thread. To help resolving global Id of thread built-in variables of type `dim3` are available from each thread. The `dim3` type is composed with 3 integer variables each for one dimension:

- gridDim specifies dimensions of the grid in the blocks
- blockDim specifies dimensions of the block in threads
- blockIdx specifies index of certain block within the grid
- threadIdx specifies index of certain thread within the block

Then statement for computing global index, supposing the grid and the block are defined only in one dimension, should look as follows:

```
globalIdx = (blockIdx.x*blockDim.x) + threadIdx.x;
```

If the grid and the block are both define in all three dimensions, indices have to be calculated separately for each dimension:

```
idx_x = (blockIdx.x*blockDim.x) + threadIdx.x;
idx_y = (blockIdx.y*blockDim.y) + threadIdx.y;
idx_z = (blockIdx.z*blockDim.z) + threadIdx.z;
```

Afterwards the global index is calculated using these indices.
If there is a need to calculate flatten 3D index within block especially when accessing shared memory, it can be calculated as follows:

```
localIdx = thredIdx.z*blockDim.y*blockDim.x +threadIdy*blockDim.x +
threadIdx.x;
```

Sometimes there is more work for the kernel that can all thread in grid process in one run then the global index have to be recalculated calculated as follows:

```
globalIdx += gridDim.x*blockDim.x;
```

2.4.5 Warp and lane

Threads are executed simultaneously SIMD-like in 32-thread packs called wraps. All of 32 threads execute the same instruction and there is only one active warp per SM. The warp id is called the lane. Both values warp id and lane id can be computed from the local id of the thread.

```
warpId = localId / 32
laneId = localId & 31
```

Warps are the part of the mechanism of covering memory latencies. When one warp reaches an instruction resulting in, for example, global memory access, which can last for hundreds of clocks cycles, warp scheduler activate another warp until data transfer is over.

2.5 CUDA memory model

The CUDA platform offers developer various types of the memory whether it is physical or logical memory. Each of these memories serves different purpose and has its own advantages and disadvantages.

2.5.1 Global memory

It is a physical memory, which creates the main memory pool for GPU. It means that it is accessible from the each thread of the GPU. All data transferred from the CPU onto the GPU resides in this memory and therefore each application running on the GPU need to access global memory at some point. This memory has the greatest capacity and the lowest bandwidth compared to all other physical memories present in the GPU. For example NVIDIA Tesla K20 is equipped with the GDDR5 memory with capacity 5 GB and throughput of 208 GB/s, as we mentioned in the section 4.1.

To reduce impact of the low bandwidth, global memory is accessed via the L2 cache and each SM is equipped with the L1 cache. By turning L1 caching on and off we can influence global memory load granularity. When L1 caching is enabled the size of memory transaction is 128B, when otherwise the size of transaction is 32B. These transactions are aligned to 128B or 32B respectively. As we mentioned in the section 2.4.5 SM has only one active warp at the time. When warp executed operation results in the global memory access, memory sub-system tries to merge memory transfers to as least transfers as possible. For example, the warp is requesting 32 consecutive 4B floats aligned to128B. Supposing L1 caching is turned on this 128 byte memory request will be satisfied in one global memory transfer. However, if memory request is not aligned to 128B, two memory transfers are needed. In the worst case, if each of 32 float are situated in different 128B blocks, transaction is satisfied with 32 global memory transfers. Therefore it is crucial to ensure sensible access pattern in the application in order to achieve maximum performance. Setting specific caching options can minimalize penalty for unpredictable access patterns.

2.5.2 Local memory

The local memory is logical memory and it is used for the local variables when block request more space for its local variables than SM offers. This memory is accessible only by the thread and it is situated in global memory pool and therefore it has the same properties.

2.5.3 Registers

Each SM has its own field of registers. These registers are used to store local variables of each thread which resides on the SM and are accessible only by that thread. Register is the fastest memory type.

2.5.4 Shared memory

The shared memory is present on each SM and shares same memory pool with the L1 cache. Shared memory is accessible for each thread within the same block and it is often used as a mean of inter-block communication. The shared memory is often used for reduction of the penalty caused by chaotic access pattern into the global memory. In order to reduce the penalty, block has to firstly load values from the global memory with coalesced access pattern, store them in the shared memory and then access the shared instead. However, shared memory bandwidth is also affected by unpredictable access pattern. For devices with the compute capability 2.x and above shared memory is divided into 32 banks. In the shared memory, 32 consecutive 4-byte values are assigned to 32 banks. If each thread within the block request a value assigned to the different bank, transfer is done in one transaction. On the other hand, if we access shared memory with the stride equal to 2, it leads to 2-way bank conflict. The two-way bank conflict is situation, where two different threads request values assigned to the same bank and therefore memory transfer is carried out in two transactions. In the worst case, 16-way bank conflict can occur and it results in 16 memory transactions. If all 32 threads of active warp request the value from the same bank, this request is satisfied in one transaction in the broadcast fashion.

2.5.5 Texture memory

This type of memory is located in the global memory, but it is cached and accessed via dedicated hardware present in each SM. This memory is accessible from each thread. The texture hardware has some interesting properties. It is able to perform interpolation in 1D, 2D or 3D and whenever some value is requested texture hardware also prefetches surrounding values based on their position in 1D, 2D or 3D array. Due to prefetching, texture memory can be exploited to reduce impact of chaotic access pattern for specific applications.

2.5.6 Constant memory

The constant memory is special 64KB read-only memory. This memory is mot modifiable by kernel and therefore has to be initialized prior to the kernel launch. This memory is cached with dedicated cache and it is capable broadcast.

2.6 GPGPU capabilities

This section presents benefits of using GPGPU concept to boosts up processing of data blocks. The GPU performance is compared with the CPU performance accelerated using MPI. The benchmark was performed on simple code multiplying two complex vectors.
Benchmark specification
CPU:[7]

- dual eight-core 2.4 GHz Intel Sandy Bridge E5-2665 Processor
- peak performance 38.4 GFLOPS per core
- 256 KB L2 cache per core
- 20 MB L3 cache per processor
- processor memory bandwidth 51.2 GB/s

GPU:[13]

- NVIDIA GeForce GTX 580
- 512 CUDA cores
- Fermi class computation capability 2.0
- peak performance 1581.1 GFLOPS
- 768KB L2 cache
- 1.5 GB GDDR5 memory
- memory bandwidth 192.4 GB/s

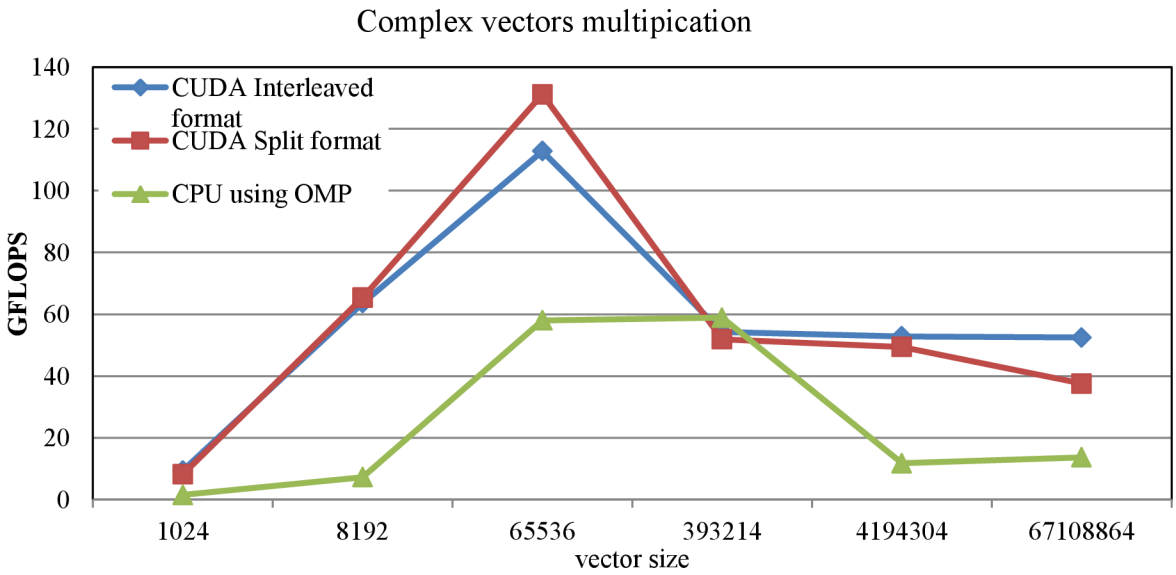


Figure 5 performance comparison of complex vector multiplication

In Figure 5 red and blue lines represent performance of the GPU. We can see that GPU reaches peak performance for vectors consisting of 65536 elements. The size of input of this vector is 512KB and therefore, whole input is present in the L2 cache of GPU. For vector size 393214, we can see dramatic decrease in the performance. It is due to fact that for this vector size, input is no longer present in the L2 and it has to be accessed through the global memory. We can deduce that the performance is bottlenecked by global memory bandwidth. Green line represents CPU code accelerated using OMP.

The CPU code reaches its peak performance at vector size 393214. Input size for this vector size is about 3MB and it can reside entirely in L2 cache of the CPU. The CPU performance exhibits the same behaviour as the GPU. This behaviour is known as a memory bond problem. The peak performance of GPU is more than 2x higher than the peak performance of CPU.

3 MATLAB implementation

3.1 Identifying acceleration opportunities

It is essential to identify proper candidates for the GPU acceleration in order to effectively accelerate a reconstruction script. Figure 6 is profile of reconstruction script. From knowledge gained via profiling, we should be able to identify acceleration opportunities which will lead to best results. Besides accelerating operations themselves, we also have to design our implementation is such way that it will require as few data transfers as possible.

We also have to take into consideration the number of operation which will be accelerated, as a process of proper implementation and tuning of GPU codes can be very time-consuming. We have to consider whether effort needed to implement operation does not exceed the possible benefits.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
261	`p = interp3(kgrid.ky, w, kgrid...`	1	1.677 s	31.1%	
285	`p = real(ifftshift(ifftn(iffts...`	1	1.075 s	19.9%	
219	`p = sf.*fftshift(fftn(fftshift...`	1	0.898 s	16.6%	
189	`sf = c^2*sqrt( (w/c).^2 - kgri...`	1	0.581 s	10.8%	
228	`p(abs(w) < (c*sqrt(kgrid.ky...`	1	0.369 s	6.8%	
All other lines			0.793 s	14.7%	
Totals			5.392 s	100%	

Figure 6 MATLAB profile

The Figure 6 shows that time spent in the different parts of the script is not evenly distributed, which is positive, because tasks where time spent in the different sections is evenly distributed are more difficult to accelerate and requires complex knowledge about a problem.

Lines 261, 285, 219 from figure consume 67.6 % of overall computation time and include just several operations. Therefore, they are suitable candidates and their acceleration on the GPU can lead to significant speedup. It in our case speed-up cannot exceed 3x.

The line 261from Figure 6 consists of operations:

- `interp3` – trilinear interpolation

The line 285 from Figure 6 consists of operations:

- `real` – extraction of real part from complex array
- `ifftshift` – circular shift of array in each dimension
- `ifftn` – inverse fast Fourier transformation

The line 219 from Figure 6 consists of operations:

- `sf.*` - element-wise matrix multiplication
- `fftshift` – circular shift of array in each dimension (invers to `ifftshift`)
- `fftn` – fast Fourier transformation

We will focus primarily on the function `interp3`, as it takes up 31% of the overall time itself and has the greatest influence on the overall performance.

In next section operations `interp3`, `fftshif`, `ifftshift`, `fftn`, `ifftn` will be closely discussed. Operations `real` and `sf.*` are omitted since being trivial.

3.2 Trilinear interpolation

The interpolation is method to estimate a value of a point if the function value is known only for surrounding points and not for the desired point itself. There are many types of interpolations and the most common and basic interpolation is linear. If interpolated function is linear then the value obtained via linear interpolation is exact, otherwise it provides only estimation. Linear interpolation is widely used in many fields to estimate values because of its simplicity and a low computation cost.

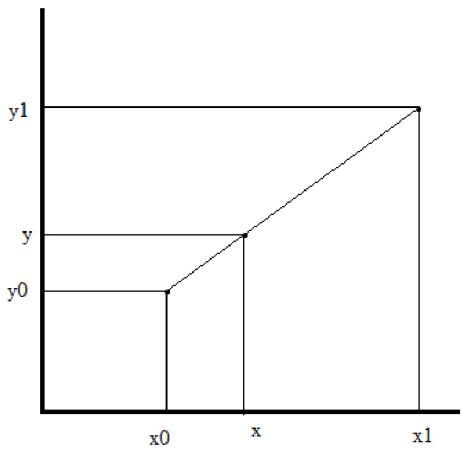


Figure 7 linear interpolation

Linear interpolation can be calculated as follows:[17]

suppose linear function

$$y = f(x) \tag{1}$$

and two given points $x0$ and $x1$, then value in point x equal to

$$f(x) = y0(1 - m) + y1m \tag{2}$$

where

$$m = \frac{x - x0}{x1 - x0} \tag{3}$$

In general, interpolation of high dimensional function can be performed as a set of normal linear interpolations in each dimension respectively. In case of image reconstruction the function is 3D. The principles of linear interpolation applied on a volume are called trilinear interpolation.

Suppose we have given a cube of points: P_{000}, P_{100}, P_{010}, P_{110}, P_{001}, P_{101}, P_{011}, P_{111}, and values in these points C_{000} ... C_{111} according to Figure 8

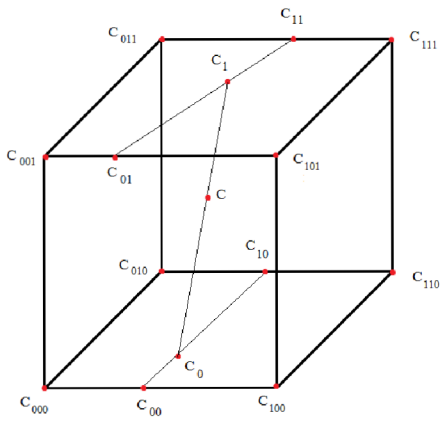


Figure 8 decomposed trilinear interpolation

then interpolated value C of point P which is situated inside of cube can be calculated as follows:

$$\begin{aligned} C_{00} &= C_{000} * (1 - dx) + C_{100} * dx \\ C_{10} &= C_{010} * (1 - dx) + C_{110} * dx \\ C_{01} &= C_{001} * (1 - dx) + C_{101} * dx \\ C_{11} &= C_{011} * (1 - dx) + C_{111} * dx \end{aligned} \tag{4}$$

where

$$dx = \frac{Px - P_{000}x}{P_{100}x - P_{000}x} \tag{5}$$

$$\begin{aligned} C_0 &= C_{00} * (1 - dy) + C_{10} * dy \\ C_1 &= C_{01} * (1 - dy) + C_{11} * dy \end{aligned} \tag{6}$$

where

$$dy = \frac{Py - P_{000}y}{P_{010}y - P_{000}y} \tag{7}$$

$$C = C_0 * (1 - dz) + C_1 * dz \tag{8}$$

where

$$dz = \frac{Pz - P_{000}z}{P_{001}z - P_{000}z} \tag{9}$$

Very similar solution consisting of seven separated linear interpolations is used in the MATLAB function `interp3`. It is very straightforward approach which does not requires complicated calculations and therefore, it is suitable to be implemented on the GPU.

3.3 Fftshift and ifftshift

Opereations `fftshift` and `ifftshift` are MATLAB functions closely associated with fast Fourier transformation. They ensure correct run of functions like `fftn` and `ifftn`. These functions are used both prior to and after fast Fourier transformation. Both shifts swap first half with second half for 1D array, swap quadrants as shown on Figure 9 for 2D array and swap octets for 3D array. Operation `fftshift` is inverse to itself when all dimensions are even, otherwise, `ifftshift` is invers function to `fftshift` function. If `x` is an 1D array and the size of array is odd `fftshift(fftshift(x))` ≠ `x`. It is also possible to think of both shifts for higher dimensions as composition of 1D shifts. For example 2D shift can be obtain as shown on Figure 9 by applying 1D shift on each row along y-axis and then applying 1D shift on each column along x-axis.

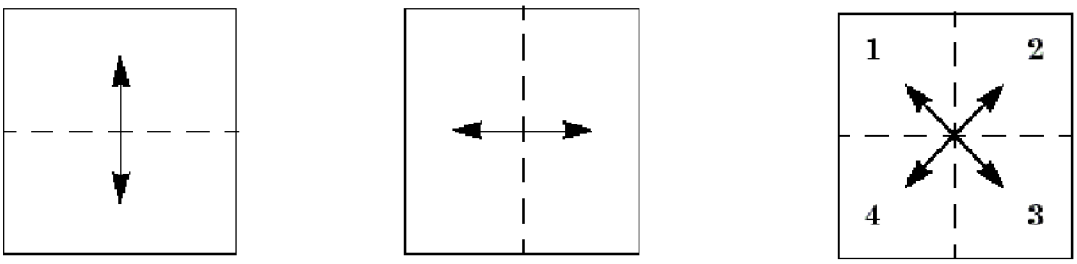


Figure 9 fftshift decomposition

Main difference between `fftshif` and `ifftshift`, as aforementioned, is how shifts behave when the number of elements in dimension is odd. The effect of these functions can be described as follows:
Suppose x is vector of numbers

```
fftshift(x) = vect_shl (x,floor(x.len/2));
ifftshift(x) = vect_shr (x,floor(x.len/2));
```

3.4 Fast Fourier transformation

The concept of DFT (discrete Fourier transformation) is largely used in many signal and image processing application. It is a process of transformation of a signal from a time domain to a frequency domain. It means that an infinite periodical signal has only few coefficients in the discrete Fourier series. Therefore, it is an efficient way to store, manipulate and reconstruct signals in the computer science.

DFT is calculated as follows: [4]

For series of N complex numbers $x_0, x_1 \dots x_{N-1}$

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi \frac{ikn}{N}} \tag{10}$$

From equation 10, it is clear that for each of N elements there has to be N addition and N evaluation of inside of a sum. It implies that computational complexity of DFT is $O(n^2)$.

Modern day application however requires computation of the DFT on large amount of data in the order of millions and for this purpose, traditional DFT is unacceptable. FFT (fast Fourier transformation) is an ideal solution. Most common algorithm of FFT, Cooley-Tukey algorithm, exhibits computational complexity $O(n \log n)$.[3]

4 Implementation

In this section we will discuss a design, an implementation and achievements. The development of application is divided into versions and so is this section. For every version, we separately discuss each operation and provide partial benchmarks and results.

4.1 Common features

These common features apply to all further algorithms and results, if not stated otherwise.

In application, axes are oriented as can be seen on Figure 10. Axes x, y and z refer to depth, width and height of volume respectively. All data are stored in flatten 3D array in row-major order.

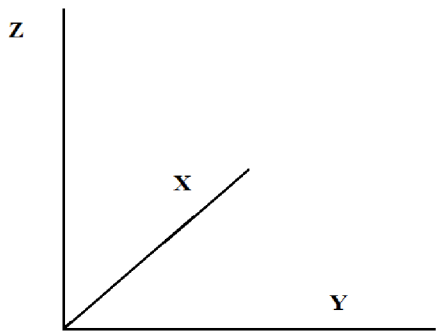


Figure 10 axis orientation

All results presented in the section 4 are measured without any overheads, that means that time does not include memory transfers, memory allocation time and so on. Only exception is the MATLAB embedded GPU acceleration, where the MATLAB provides no way to exclude overheads.

RMSD and RMS values are used to calculate an error of interpolation and they can be calculated as follows:[6]

suppose x is a signal obtained by implemented interpolation and ref is a referential signal

$$RMSD = \sqrt{\frac{\sum_{i=1}^{n}(x_i - ref_i)^2}{n}} \tag{11}$$

$$RMS = \sqrt{\frac{\sum_{i=1}^{n} ref_i^2}{n}} \tag{12}$$

The MATLAB version:

- R2014a-EDU

The CUDA version:

- 6.5

The benchmarks specification:

The CPU: (7)

- dual eight-core 2.3 GHz Intel Sandy Bridge E5-2665 Processor
- peak performance 32.4 GFLOPS per core
- 256 KB L2 cache per core
- 20 MB L3 cache per processor
- processor memory bandwidth 38.4 GB/s

The GPU: (6)

- NVIDIA Tesla K20
- 2496 CUDA cores
- Kepler class computation capability 3.5
- peak performance 3950 GFLOPS
- 5 GB GDDR5 memory

4.2 The first version

In first version, we applied a naïve approach to the design and implementation of desired operations. We used the knowledge obtained by studying principles and the MATLAB implementation of functions.

4.2.1 Fast Fourier transformation

The fast Fourier transformation is a complex problem including complex mathematics and recursion. We decided not to try "reinvent a wheel", as it may result in either failure of whole application or unsatisfying performance of application.

We came to conclusion that best solution is to use cuFFT library. We have chosen this library because of its simple interface and optimized performance. This library is capable 1D, 2D and 3D complex-to-complex, complex-to-real and real-to-complex transformations and the size of input data

is limited only by the memory of graphics card, as cuFFT is based on divide-and-conquer principle presented in Cooley-Tukey algorithm. (11)

On the other hand, there are several drawbacks of the library. For example, cuFFT algorithm is not designed in way that it can take all advantages of advanced modern day FFT algorithms. Another disadvantage is that cuFFT shows peak performance on problems of size the power of 2.

Unfortunately, real data obtained by photoacoustic spectroscopy are not of size the power of 2 in real cases.

The benchmark specification:

The CPU:

- 2.67 GHz Intel Xeon quad-core 5550

GPUs:

- NVIDIA Tesla C2050
- NVIDIA Tesla C1060

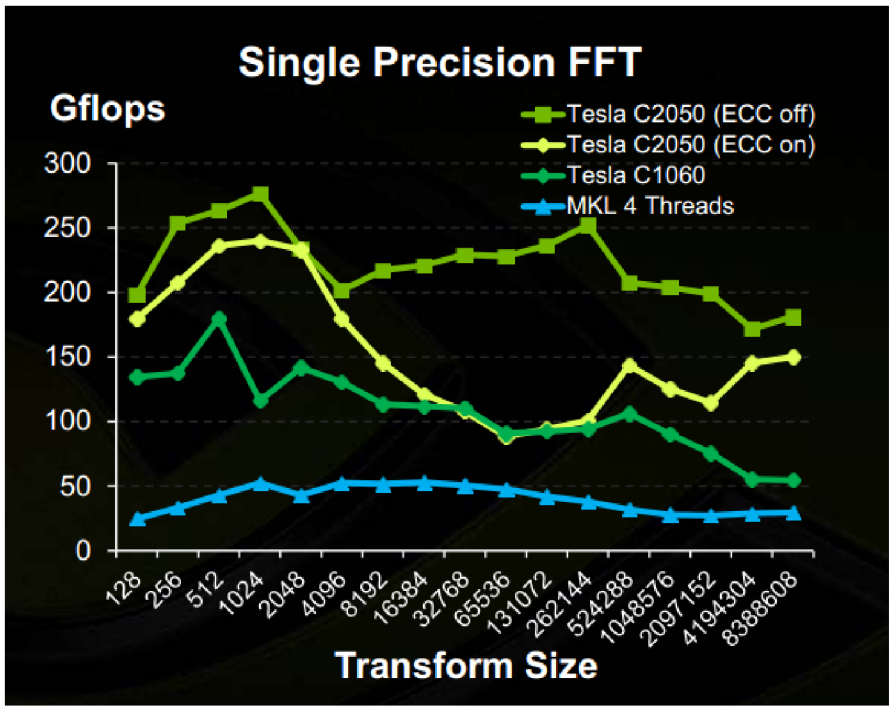


Figure 11 performance comparison of cuFFT [15]

In Figure 11, we can see a blue line which represents performance of standard implementation using Intel math kernel library (MKL). We can see no significant change of the performance for presented transform sizes. Peak performance is about 50 GFLOPS. Green and yellow lines represent the performance of cuFFT library on two GPUs and effect of ECC (error correction code). The performance varies with the transform size. It is due to fact that cuFFT recursively divides transform into factors of the initial size. The performance of cuFFT is dependent on these factors. However, peak performance of cuFFT on Tesla C2050 is around 275 GFLOPS which is 5.5x higher than the peak the performance of CPU.

In conclusion, the benchmark proved that use of cuFFT library can provide solid performance boost to the calculation and therefore it was chosen as a final version of operations `fftn` and `ifftn` used in our application. Fast Fourier transformation will be no longer mentioned in further sections.

4.2.2 Fftshift and ifftshift

As mentioned in section 3.3, effect of operation `fftshift` and `ifftshift` on 3D space can be achieved by applying 1D shifts step-by-step in each dimension.

Therefore, we can assume that simple 1D shift CPU code would be similar to a pseudocode as follows:

suppose x is array of integers

```
for(i=0;i<x.len/2;i++)
{
      Swap(x[i],x[i+x.len/2]);
}
```

Inplementation

This 1D code can be easily parallelized and transformed onto the GPU. Each thread calculates it's `index` and `offset_index` by adding `x.len/2` to `index` and then swap an element `x[index]` with an element `x[offset index]`. For this operation, the kernel has to run with only `x.len/2` of threads.

To turn 1D principle into 3D, we have to add few mechanisms to be able to perform 1D shift along desire axis. Solution is straightforward, we replace array length with size of dimension and because we are working with flatten 3D array, we have to change calculation of both `index` and `offset_index`. The `index` is calculated as follows:

```
index = z*volume.width*volume.depth + y*volume.depth + x;
```

where x, y, z are coordinates of the requested point. In real GPU kernel, x, y and z are replaced with thread indices calculated as mentioned in section 2.4.4. Then if we want to compute offset index properly, we have to add `dim.size/2` multiplied by a stride in particular dimension. The stride is distance in flatten 3D array between two neighbouring points along dimension. For example, the stride in z-dimension is `volume.width*volume.depth`. Again, number of kernel threads needed to perform 3D shift along one dimension is `matrix.size/2`. To obtain overall effect we had to implement three CUDA kernels working in each dimension, as 3D shift can be composed with 1D shifts in each dimension according to section 3.3.

The greatest complication became reaching shifting effect instead of swapping effect when seize of dimension was odd.

Suppose we have vector of number:

0 1 2 3 4 5 6

If we apply `fftshift` without odd dimension issue properly treated output will be:

0 1 2 3 4 5 6

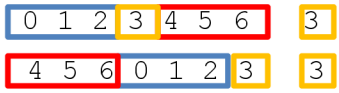
3 4 5 0 1 2 6

To obtain proper result in case of odd dimension each thread of kernel has to load the value `x[offset index + stride]` into temporary variable, store the value `x[index]` to `x[offset index]` and afterwards stores the value in temporary variable to `x[index]`. Outcome of this operation is as follows:

0 1 2 3 4 5 6

4 5 6 0 1 2 6

We can see that modified `fftshift` still leaves wrong value at the end of the vector. To eliminate this effect, we had to take some measures before and after operation itself. Due to lack of global synchronization, two other kernels were implemented for each dimension. These kernels are launched in case of odd dimension size. First kernel creates a backup of middle values, in 3D space it is a plane perpendicular to respective axis with respective coordinate equal to `floor(dim.size/2)`. Second kernel stores values to end of the vector, in 3D at the perpendicular plane with respective coordinate equal to `dim.size -1`. After implementation of all modification we were able to obtain correct result:



To implement `ifftshift` we used the identical approach with few differences. The main kernel loads value `x[offset index]` into temporary variable an then stores value `x[index]` to `x[offset index + stride]`. The kernel prior to the main kernel creates backup of values situated at end of vector, in 3D the perpendicular plane with respective coordinate equal to `dim.size -1`. And then the kernel launched after the main kernel stores backed up values in the middle of vector, in 3D space it is plane perpendicular to axis with respective coordinate equal to `floor(dim.size/2)`.

To be able to handle complex output of FFT two other sets of kernels were created to preform `fftshift` and `ifftshift` on complex data.

Results

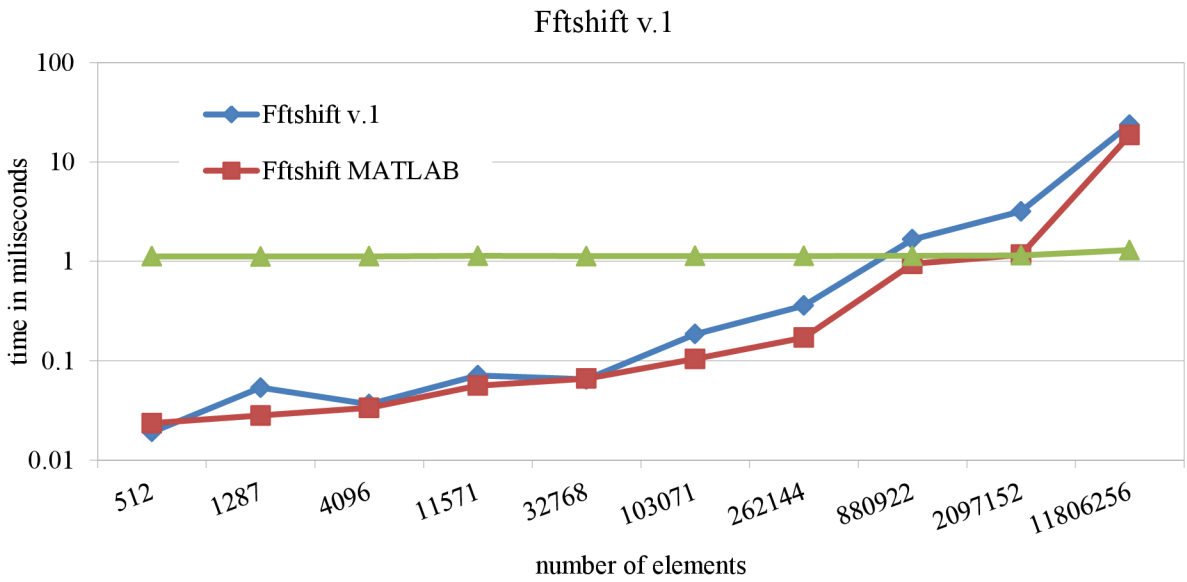


Figure 12 performance comparison of fftshift v.1

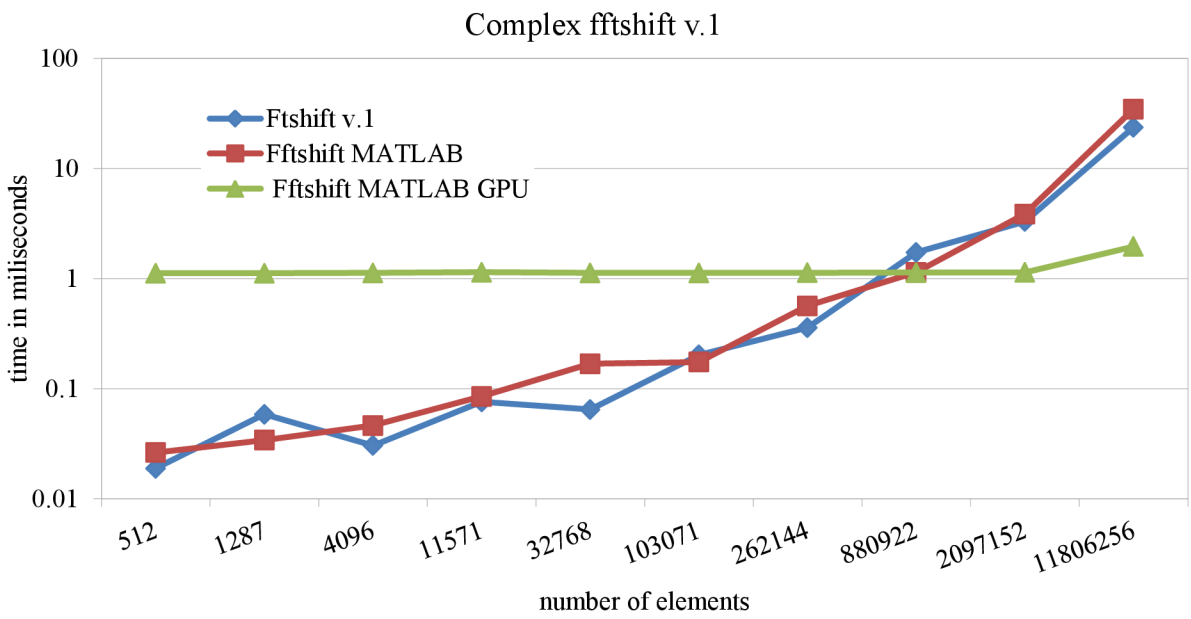


Figure 13 performance comparison of complex fftshift v.1

Discusion [1]

In Figure 12 and Figure 13 the blue line represents our implementation. We can see that the blue line has almost same behaviour as red line representing the MATLAB native implementation. We can notice that the performance of our implementation drops in cases where the size of input is not equal to the power of 2. This is due to not coalesced memory access. Green line represents the MATLAB embedded GPU acceleration and we can notice that for input sizes up to 2097152 elements, performance is almost identical. It is most probably caused by the internal overhead in the MATLAB. For the real data of size 11806256 elements, MATLAB embedded GPU acceleration provides best results and our implementation falls behind. In fact, the difference of performance between our implementation and MATLAB embedded GPU acceleration is in order of magnitude.

Figure 12 and Figure 13 also shows that there is almost no difference between performance of float `fftshif` and complex `fftshift` in our implementation.

Main advantages:

- Simple implementation, no need for special mathematical functions
- *In situ* solution (if we omit temporary arrays)

Main disadvantages:

- Many kernels have to be launched to perform single operation

All things considered, the concept has to be changed from the scratch in further versions.

4.2.3 Trilinear interpolation

Outcome of trilinar interpolation can be obtained by composing seven linear interpolations and can be decomposed into several equations from section 14. This straightforward approach, as aforementioned, is ideal to be implemented on the GPU, because this approach only uses simple mathematical operations as multiplication, division and addition.

[1] We only provide benchmarks and discussion to `fftshift`. All discussed features are also valid for `ifftshif`.

Implementation

Suppose seven matrices with same size:
`smp_x, smp_y, smp_z, val, intrp_x, intrp_y, intrp_z;`
Matrices `smp_x, smp_y` and `smp_z` contains x-, y- and z- coordinates of sample points respectively. Coordinates has a constant spacing.
Matrix `val` contains sampled (function) values in sample points. This matrix correlates with matrices smp_x, smp_y and smp_z.
Matrices `intrp_x, intrp_y` and `intrp_z` contains x-, y- and z- coordinates of points in which trilinear interpolation has to be carried out.
We divide the sample points into sub-cubes of size 2 x 2 x 2. Each thread operates with one sub-cube. It means that each thread has to load eight values from `val` and coordinates in each of eight sample point. From the fact that all coordinates has the constants step in between we can deduce that the sub-cube creates a real geometrical cube. Therefore, there is no need to load all eight coordinates in each dimension, as cube with size 2x2x2 contains only two different values in one dimension. Each thread calculates the index into matrices `smp_x, smp_y, smp_z` and `val`. No special calculation of index including stride or offset has to be implemented, because sub-cubes of this size overlap perfectly. Each thread has to load two coordinates of the sample points in each dimension and eight values from the flatten 3D array:

```
x0 = smp_x[index];
x1 = smp_x[index + stride_in_x];
```

etc.

```
c000 = val[index];
c100 = val[index + stride_in_x];
c010 = val[index + stride_in_y];
c110 = val[index + stride_in_y + stride_in_x];
```

etc.
After this step, each thread properly sets up all variables of regarding its sub-cube. Every thread then has to process each interpolation point stored in arrays `intrp_x, intrp_y` and `intrp_z`, because more than one point can appear inside of the sub-cube.

To speed up this process of iteration trough each interpolation point, each block has three arrays of shared memory buff_x, buff_y and buff_z with the size equal to count of threads per block. Firstly, all threads read different values from global memory according to thread's local index. Local index can be calculated in a way shown in section 2.4.4. Then each thread stores value loaded from global memory into shared memory using the same local index. This way with only few load and store operations kernel has as many points ready to be process, as there are threads per block. After all load and store operation, kernel has to call a block-scope barrier by `__syncthreads` to prevent reading of shared memory before all validate data are properly stored. The barrier call is must because as mentioned in section 2.4.5 there is no guarantee on sequential order of warps execution.

After shared memory is filled with valid data, each thread iterates through all points in shared memory and calculates interpolated value of point. The kernel takes advantages of shared memory broadcast capability. When all threads within the warp request the value assigned to the same bank of shared memory, this memory request is satisfied in one go in broadcast fashion. The calculation of the interpolation is based on equations 4-9. The condition determining whether the point lies within the sub-cube operated by the thread or not is present at the very end of the calculation. The location of the condition prevents a massive divergence within warp even at a cost of some unnecessary calculations. The condition itself is based on equations 5, 7 and 9, where all coefficients dx, dy and dz has to be

from interval ⟨0,1⟩ in case where the point lies within the sub-cube. If the condition holds true, the thread write back the interpolated value to the global memory. After this section, kernel has to use barrier to ensure that all work on current data in the shared memory has been done and the shared memory can be overwritten by new data. This principle is repeated until all interpolation points are processed.

Results

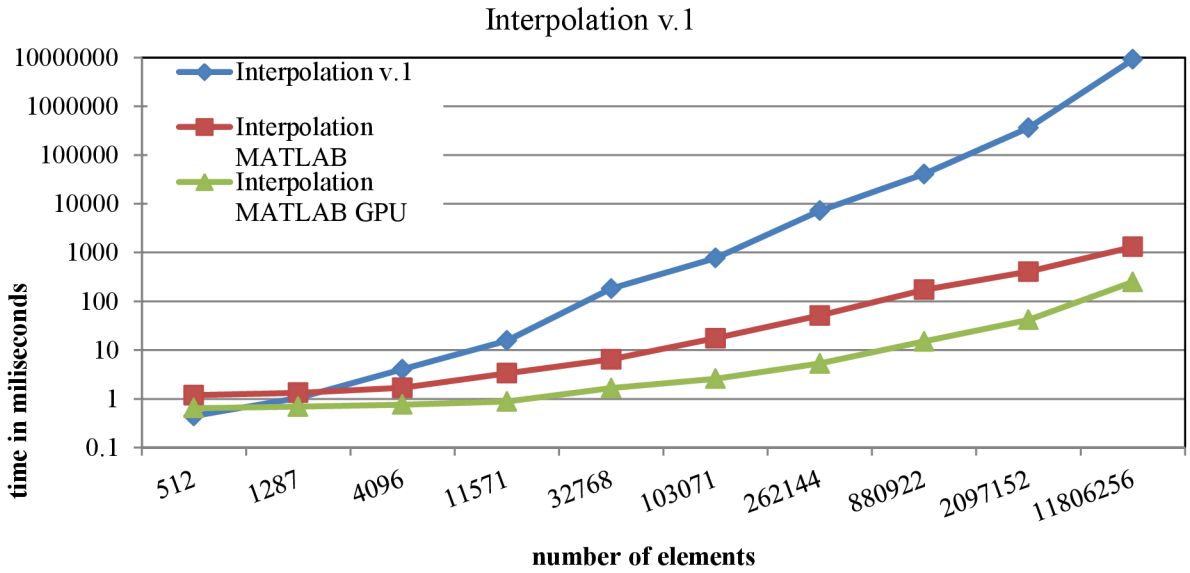


Figure 14 performance comparison of interpolation v.1

Efficiency	
Global Load Efficiency	99.3%
Global Store Efficiency	21.2%
Shared Efficiency	50%
Warp Execution Efficiency	99.7%
Non-Predicated Warp Execut	99.7%
Occupancy	
Achieved	49.9%
Theoretical	50%
Limiter	Registers

Figure 16 GPU efficiency and occupancy

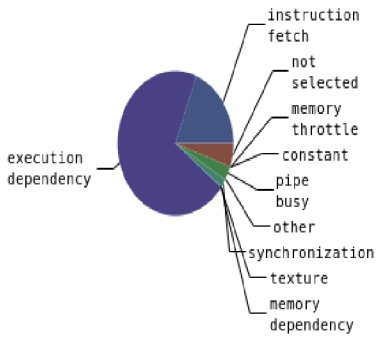


Figure 15 stall reasons

Discussion

In Figure 14, the blue line represents implemented GPU kernel. The kernel performance fall way behind both native MATLAB implementation represent by red line and MATLAB embedded GPU accelerated implementation represented by green line. The implementation exhibits a quadratic computational complexity. The elapsed time to compute a real data input was about 2.5 hours and achieved performance was only 1,3 Kpoints interpolated per second. Compared to 47.2 Mpoints interpolated per second achieved by the MALTAB embedded GPU acceleration on the real data sample, our implementation reaches only 0,003% of its performance. From Figure 15, we can deduce

that the main reason of slow run of the application is an execution dependency. Execution dependences are inevitable part of the kernel. They create a large portion of the stall reasons mainly due to the number of the operations. In further versions it is necessary to reduce the amount of the mathematical operation to achieve a better performance. Figure 16 shows that the kernel has almost optimal global memory access pattern. Poor global store efficiency is caused by irregular stores. In Figure 16 we can see that there is a space for an improvement in accessing shared memory, which can result in a speedup. However, improvement of the shared memory access will not bring a radical change to the performance. We can also see that the implementation does not allow a full utilization of GPU due to a number of registers per kernel. This condition is also inevitable.
Advantages:

- Relatively simple and straightforward implementation

Disadvantages:

- Unacceptable run time for the real data even with some optimizations
- Numerous memory accesses and calculations
- Solution does not allow full utilization of GPU

From discussed results it is clear that in order to improve the performance the number of memory accesses and calculations has to be dramatically decreased.

4.3 The second version

The discussion of results of first version apparently shows that design of the implementation has to be radically changed in order to provide satisfying results.

4.3.1 Fftshif and ifftshift

The main disadvantage of previous version of `fftshif` and `ifftshift` was the number of kernels needed to successfully preform the transformation. Each operation required at least three kernels. It means that all memory accesses and calculations had to be preform three times. And if we take in consideration that a launch of kernel takes some time on itself, it is clear that concept of multiple kernels is definitely not suitable.

Implementation

Completely different approach was adopted for this version. It is approach that combines all required steps from previous version in one kernel. To this purpose a new method of calculating `offset_index` was developed which puts all steps needed in the previous version the into one kernel. The offset index is in current version calculated as flows:
Suppose flatten 3D array `mtx`

```
shift_in_x = (idx.x + ceil(dim_x.size/2)) mod dim_x.size;
shift_in_y = (idx.y + ceil(dim_y.size/2)) mod dim_y.size;
shift_in_z = (idx.z + ceil(dim_z.size/2)) mod dim_z.size;
offset_index = shift_in_z*mtx.width*mtx.depth;
offset_index += shift_in_y*mtx.depth;
offset_index += shift_in_x;
```

If `offset_index` is calculated this way, it tells the thread which value is supposed to appear on the position defined by the index after `fftshift`. However lack of the global synchronization prevents the kernel from working *in situ*. Therefore, kernel has to work with two disjoined spaces.

To make this principle works as `ifftshif` instead of `fftshift` all what has to be done is to replace the function `ceil` with the function `floor`.

Results

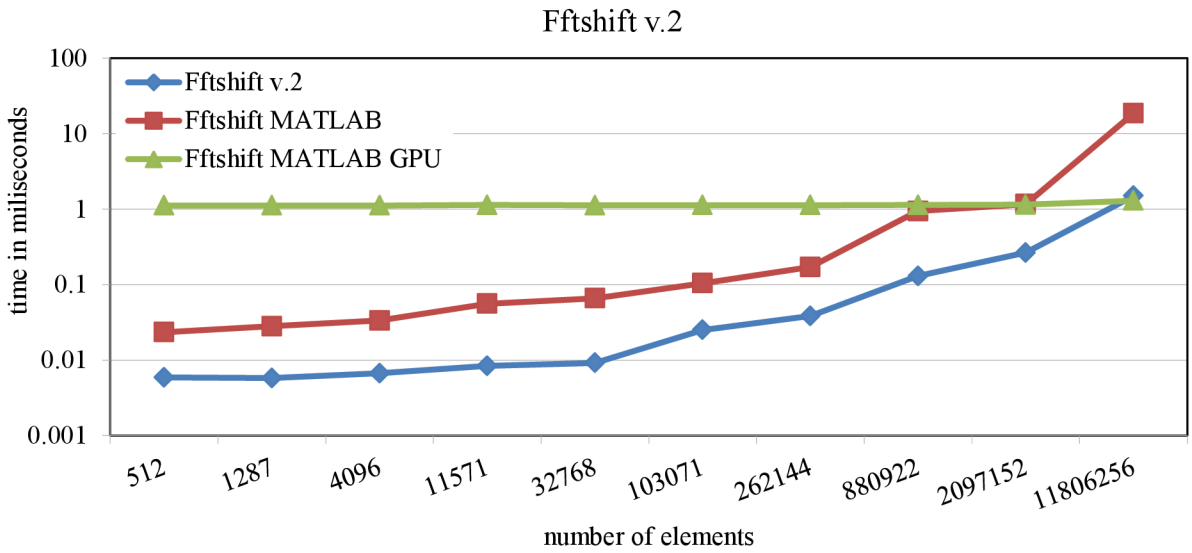


Figure 17 performance comparison of fftshift v.2

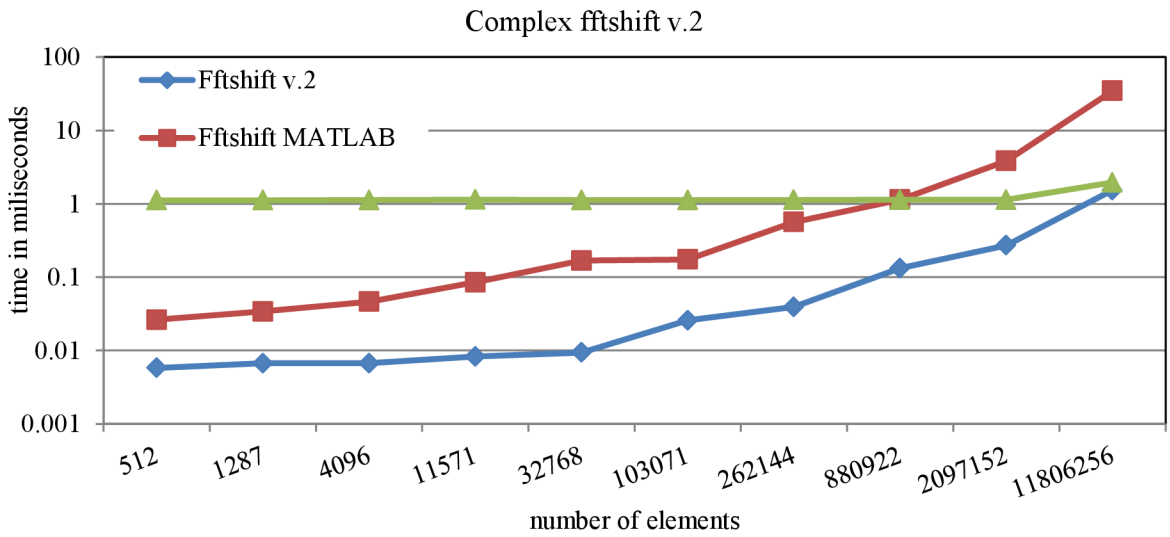


Figure 18 performance comparison of complex fftshift v.2

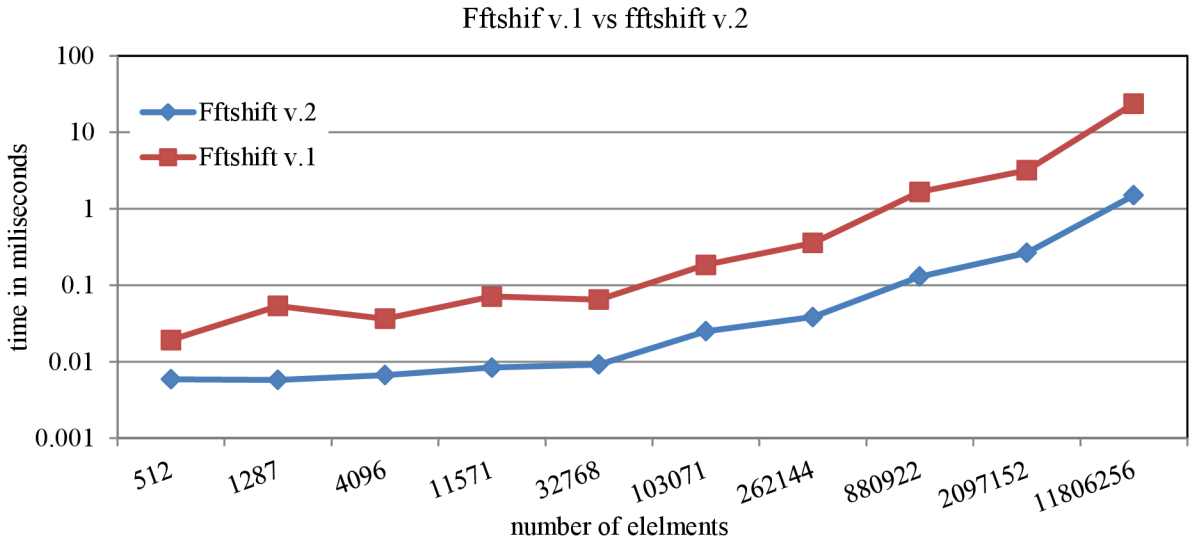


Figure 19 performance comparison of different versions

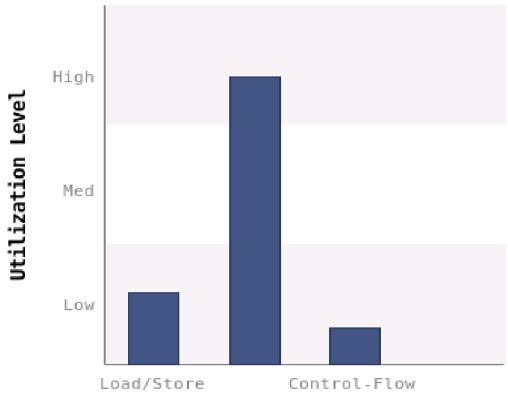


Figure 21 hardware utilization level

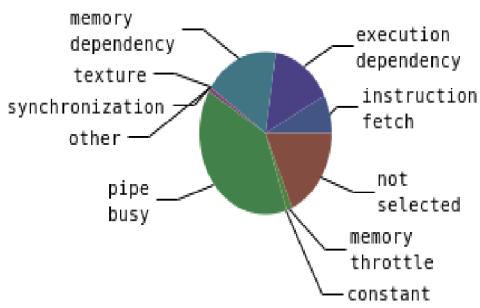


Figure 20 stall reasons

Results

Rank	Description
100	[4 kernel instances] void radixM_kernel<unsigned int=37, radixM_float>(fftDirection_t, unsigned
25	[2 kernel instances] void radixM_kernel<unsigned int=11, radixM_float>(fftDirection_t, unsigned
17	[1 kernel instances] interp(float const *, float const *, float const *, float2*, dimDesc_t, flo
11	[2 kernel instances] void spRadix0004A::kernel1Tex<fftDirection_t=-1>(Complex<float>*, unsigned
11	[2 kernel instances] void spRadix0004A::kernel1Tex<fftDirection_t=1>(Complex<float>*, unsigned i
7	[1 kernel instances] full_fftshift(float2*, float2*, float*, dimDesc_t, dimDesc_t, dimDesc_t)
6	[1 kernel instances] fftshift(float*, float2*, dimDesc_t, dimDesc_t, dimDesc_t)
6	[1 kernel instances] full_ifftshift(float2*, float2*, dimDesc_t, dimDesc_t)
6	[1 kernel instances] ifftshift(float2*, float*, dimDesc_t, dimDesc_t)
5	[1 kernel instances] void spRadix0049B::kernel3Tex<fftDirection_t=-1>(Complex<float>*, unsigned
5	[1 kernel instances] void spRadix0049B::kernel3Tex<fftDirection_t=1>(Complex<float>*, unsigned i

Figure 22 impact of kernel acceleration on overall performance

Discussion[2]

It can be sad that this implementation achieves a satisfying performance. In Figure 17 and Figure 18 there is the blue line representing computation time of our implementation, the red line representing

[2] We only provide benchmarks and discussion to `fftshift`. All discussed features are also valid for `ifftshif`.

native MATLAB implementation and the green line representing MATLAB embedded GPU acceleration. The difference between the MATLAB performance and implemented GPU kernel performance is in the order of magnitude for real data sample. We can also state that for the real data kernel provide performance comparable to MATLAB embedded GPU acceleration. Performance of the MATLAB embedded GPU acceleration is dragged down by the internal overhead for input sizes below 118062256 elements. In Figure 21, we can see that utilization level of the hardware by arithmetic operations is high. Moreover, the main stall reason is busyness of pipe according to Figure 20. From this information we can deduce that implementation is bottlenecked by the high utilization of the execution unit by the operation modulo. It would be possible to adopt advanced techniques to replace the modulo operation.

Figure 22 shows importance of the kernel acceleration based upon their potential benefit to the overall performance. We can see that shift operations themselves have no significant effect on overall performance according to Figure 22.

Advantages:

- Only one kernel is needed to perform the operation

Disadvantages:

- Usage of time consuming operation (modulo)
- Operation is not performed *in situ*

After considering an impact which has the performance of `fftshift` and `ifftshift` on the overall performance of application, decision was made not to continue in the development of `fftshift` and `ifftshift` GPU acceleration. This is final version of `fftshift` and `ifftshift` implementation used in the application and therefore `fftshift` and `ifftshift` will be no longer mentioned in further sectionss.

4.3.2 Trilinear interpolation

As mentioned in the section 4.2.3, the previous version of trilinear interpolation has the unacceptable performance. It was result of its quadratic computational complexity, which was caused by fact that for N sample points and M interpolation points each of N threads has to iterate trough M points. This afterward led to numerous global memory accesses. Therefore this approach is inapplicable.

Implementation

For this version the concept was completely reversed. In this version we take full advantage of texture hardware present in each SM. As mentioned in section 2.3.5 texture hardware is capable of trilinear interpolation and it is also equipped with the cache. This type of memory is the best solution for unpredictable access pattern. Unlike previous version, each thread now operates with only one interpolation point. Values in original sample points are stored in a 3D cudaArray and accessed via texture hardware.

Suppose same matrices from section 4.2.3. From experiences with sample points we know that matrices smp_x, smp_y and smp_z contain only as much different values as size of respective dimension. For example, each plane of smp_y, which is perpendicular to y-axis, contains same values. Distance between two different values in matrices smp_x, smp_y and smp_z is equal to stride in particular dimension. What is more, all values have same spacing and are sorted in ascending order.

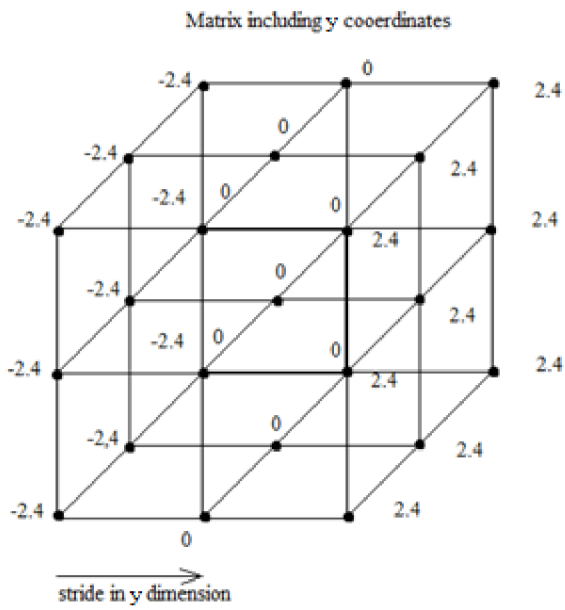


Figure 23 MATLAB matrix data layout

The attempt to reduce memory accesses cerates complication because values in texture memory are indexed in x-, y- and z-axis in intervals `<0,val.depth-1>`,`<0,val.width-1>` and `<0,val.hight-1>` respectively and do not correspond to its sample points coordinates. Therefore each kernel has to pre-calculate texture indices according to its interpolation point as follows:

```
step_y = (smp_y[1*stride_in_y] - smp_y[0]);
texture_index_y = (interp_y - smp_y[0])/step_y;
```

According to Figure 23, suppose `interp_y` is equal to 1. If we remap `interp_y` into the interval ⟨0,2⟩ using this method, remapped value is equal to 1.41.

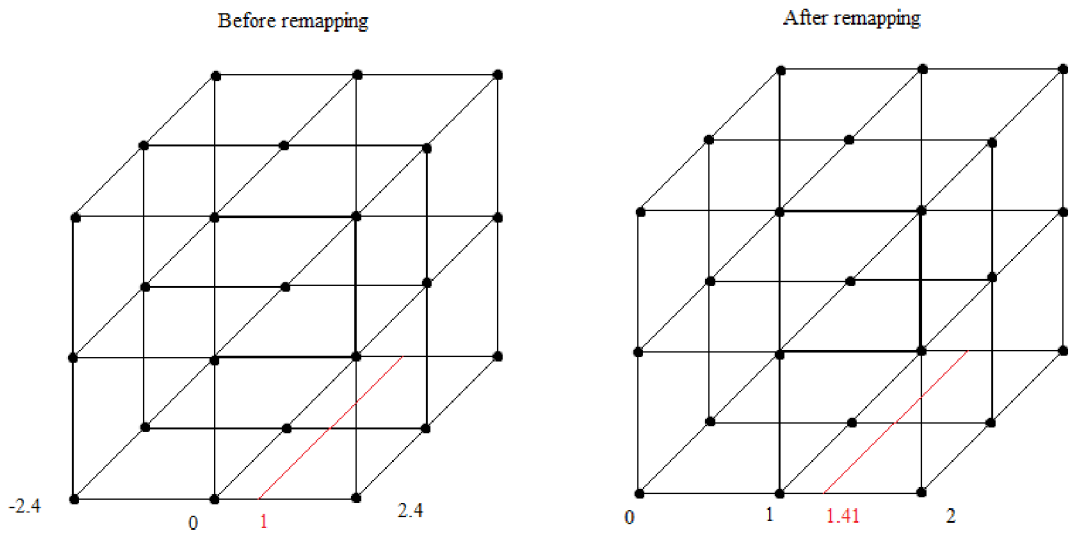


Figure 24 remapping of indices

To be able to calculate texture indices kernel only need initial value of sample points coordinates and the respective spacing, this way we are able to reduce input data size by 37%. When indices are calculated for each dimension, each thread uses these indices to obtain interpolated value from the

texture. To this purpose `tex3D` function is used. Providing indices passed to function are not integers, it returns hardware interpolated value. Afterwards each thread stores this value to respective position in the global memory.

Results

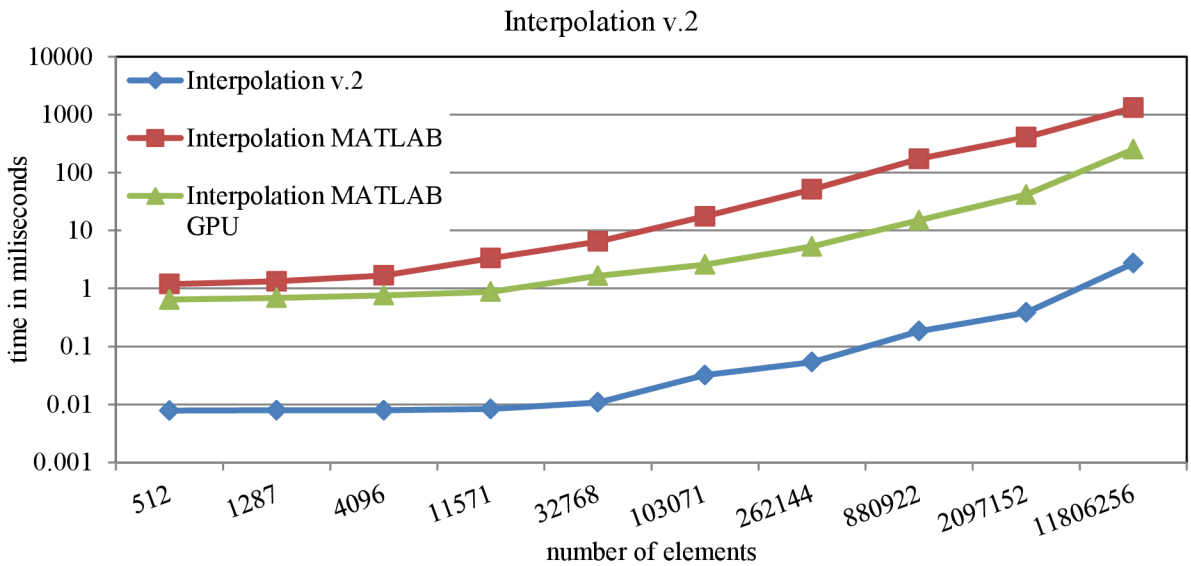


Figure 25 performance comparison of interpolation v.2

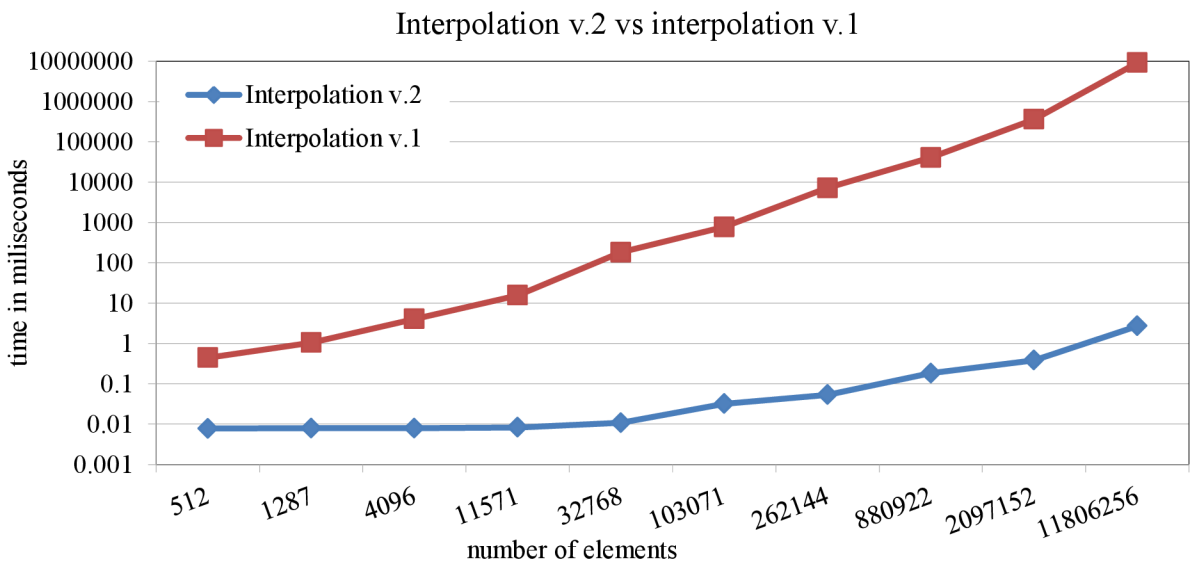


Figure 26 performance comparison of different versions

Start	863.895 ms (863,89
End	874.174 ms (874,17
Duration	10.279 ms (10,279,
Size	101.632 MB
Throughput	9.887 GB/s
Memory Type	
Source	Device
Destination	Array

Figure 27 memory transfer details

Discussion

In Figure 25, the blue line represents our implementation, the red line represents native MATLAB version and the green line represents MATLAB embedded GPU acceleration. Based on the presented results, this version of interpolation provides outstanding performance, which is incomparable to both MATLAB implementation and MATLAB GPU acceleration. According to results achieved for the real data sample, show in Figure 25, it can be estimated that the kernel is capable of processing 4.3 Gpoints per second and MATLAB embedded GPU acceleration achieves throughput of 47.2 Mpoints per second. Our implementation has 91x greater performances than MATLAB using GPU. Error of this implementation is 8.64 using the RMSD. The RMS of referential signal produced by MATLAB is 874.3. The error of this method is 0.98% compared to referential signal.

In Figure 26, the blue line represents current version of implementation and the red line previous version. The figure demonstrates performance growth in current version. For the real data sample previous implementation only reaches the throughput of 1,3 Kpoints per second, where this implementation reaches the throughput of 4,3 Gpoints per second. The performance of current version is 3,307,692x greater compared to previous version. The main reason of this growth is dramatic reduction of number of operations and memory accesses. This version also reduces the amount of data needed to be transferred onto GPU, which lower overall run length.

On the other hand, as we can see in Figure 27, time to initialize the cudaArray with real data sample drags down overall speed-up of this version.

Advantages:

- Good precision
- Great speed
- Reduction of input by 37%

Disadvantages:

- Need to initialize cudaArray

4.4 The third version

In previous version some interesting results were achieved. It presented few techniques and principles which were proven usable. It also sets the trend which should be followed in future design.

4.4.1 Trilinear interpolation

Design of this implementation takes fully advantage of achievements and discoveries gained in previous development. The design combines computation method of first version with logic of second version to produce implementation with great performance.

Implementation

Only difference between current and previous implementation is that in this version the interpolation is calculated using equations 4 - 9. Therefore this method does not need to initialize the cudaArray. As shown in section 4.3.2, each matrix of sample points coordinates contains only a fragment of different values compared to all elements in matrix. Function values defined in sample points are stored in 1D texture to reduce impact of unpredictable global memory access pattern.

After kernel is launched, each thread loads coordinates of respective interpolation point. Afterwards coordinates are remapped to obtain texture indices for each dimension respectively using method described in section 4.3.2. In most cases these indices are not integers. If the thread takes whole number part of indices and uses them to obtain the signal value, thread is guaranteed to obtain the value of point P_{000} from Figure 8. What is more decimal parts of indices serve as coefficients otherwise obtained with equations 5, 7 and 9. After the thread calculates indices of point P_{000}, all values needed to compute trilinear interpolation are loaded from the texture memory. Then the interpolation is calculated in the same way as in section 4.2.3.

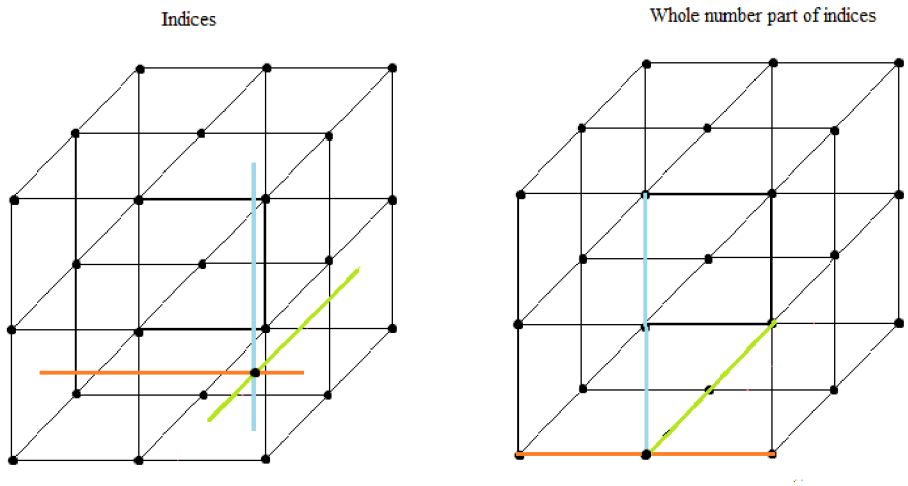


Figure 28 effect of whole number part of indices (4)

Results

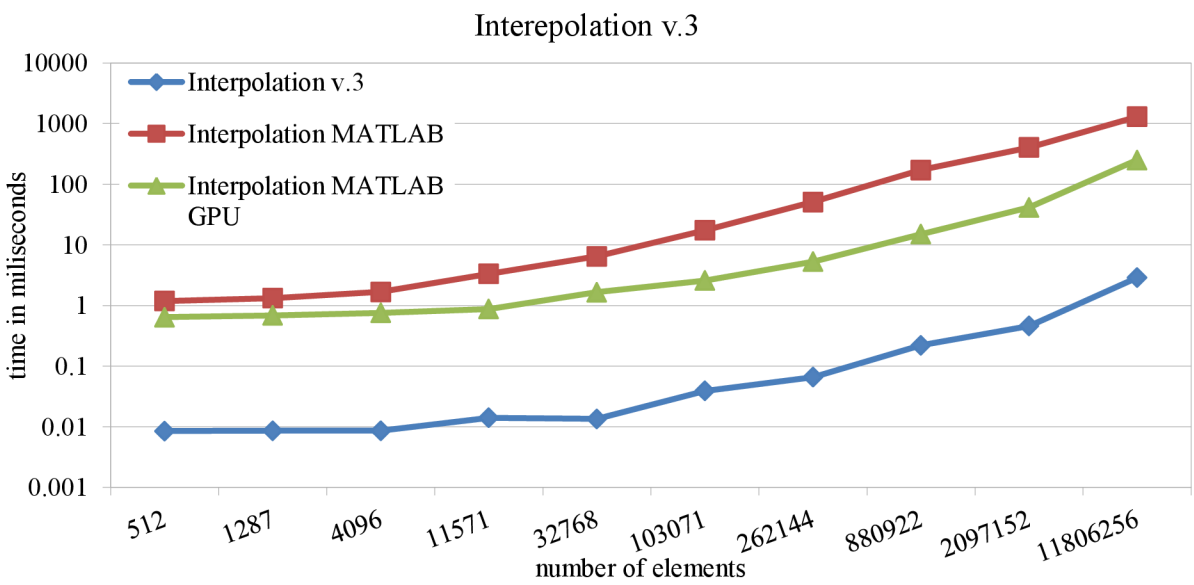


Figure 29 performance comparison of inpterpolation v.3

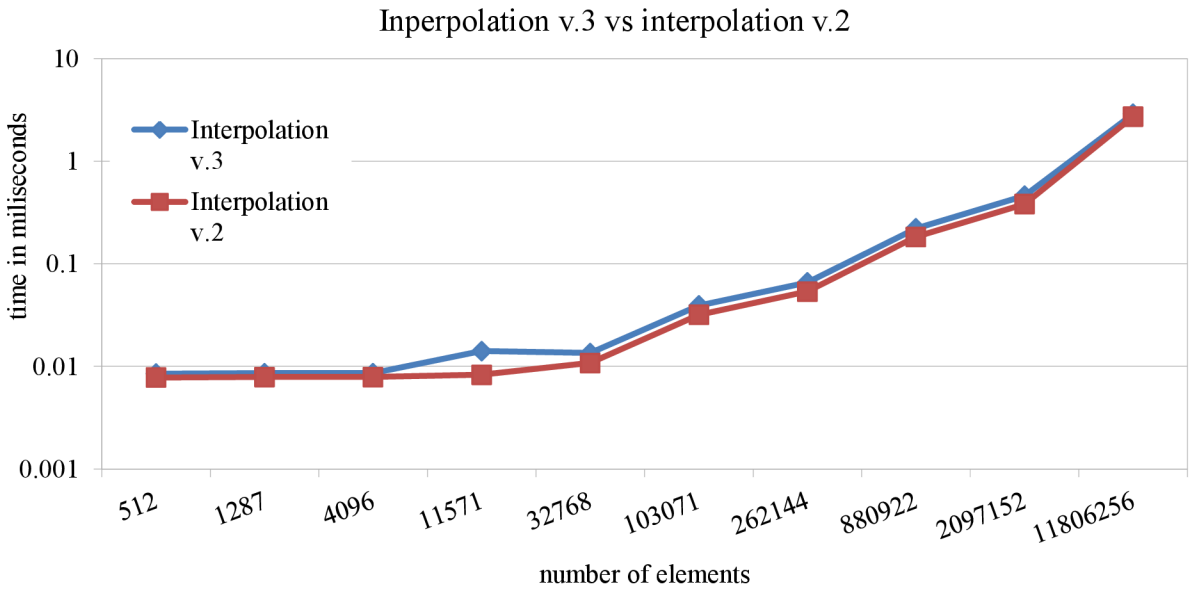


Figure 30 performance comparison of different versions

Efficiency	
Global Load Efficiency	81.9%
Global Store Efficiency	88.6%
Shared Efficiency	n/a
Warp Execution Efficiency	97.8%
Non-Predicated Warp Execut	93.2%
Occupancy	
Achieved	69.4%
Theoretical	75%

Figure 32 GPU efficiency and occupancy

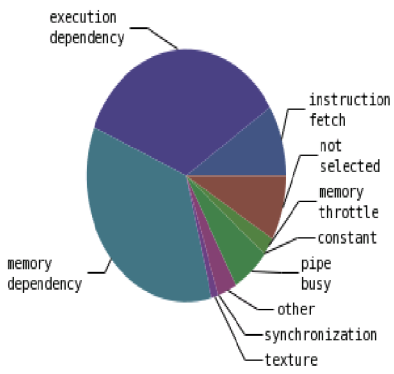


Figure 31 stall reasons

Discussion

This implementation meets all requirements previously set. It reaches considerable performance without using texture hardware for interpolation. In Figure 29 our implementation is represented with the blue line, the MATLAB implementation is represented with the red line and the MATLAB embedded GPU implementation is represented with the green line. We can see that our implementation has greatest performance compared to MATLAB implementations. It can be estimated that for the real data sample the kernel is able to interpolate 4.1 Gpoints per second. Compared to the MATLAB GPU implementation 47.2 Mpoints per second throughput, our implementation achieves 86.8x greater performance. The error of this implementation is 12.9 using the RMSD. The RMS of referential signal produced by MATLAB is 874.3. Therefore, error of this method is 1.47% compared to referential signal.

In Figure 30, current implementation is represented by the blue line and previous implementation is represented by the red line. There is almost no difference between performances of both

implementations. For the real data sample current version achieve 94.3% of performance of previous version.
The kernel exhibits good occupancy and global memory efficiency, as it may be seen in Figure 32. Most significant stall reasons are execution and memory dependences, which are inevitable.
Advantages:

- Relatively great speedup achieved

Disadvantages:

- Grater error compared to previous version

In conclusion, there is not many possible ways to significantly speed up this particular implementation. Since kernel provides significant acceleration it is used in the application.

4.5 Discussion

This section is dedicated to presentation of overall results and achievements.

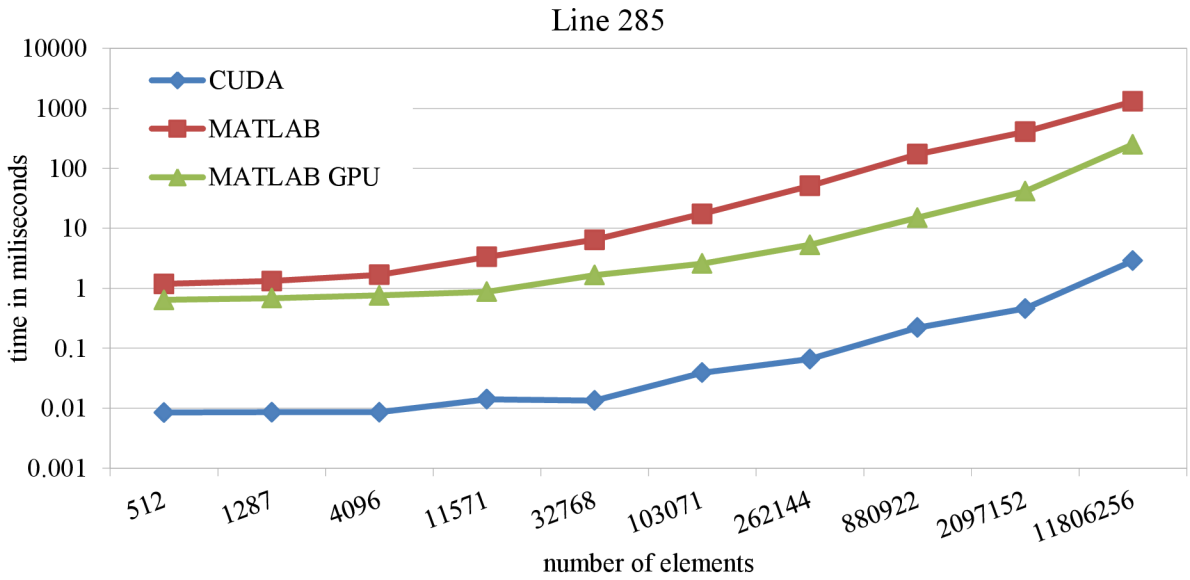


Figure 33 performance of implementations of line 285

Discussion to Figure 33 can be found in section 4.4.1.

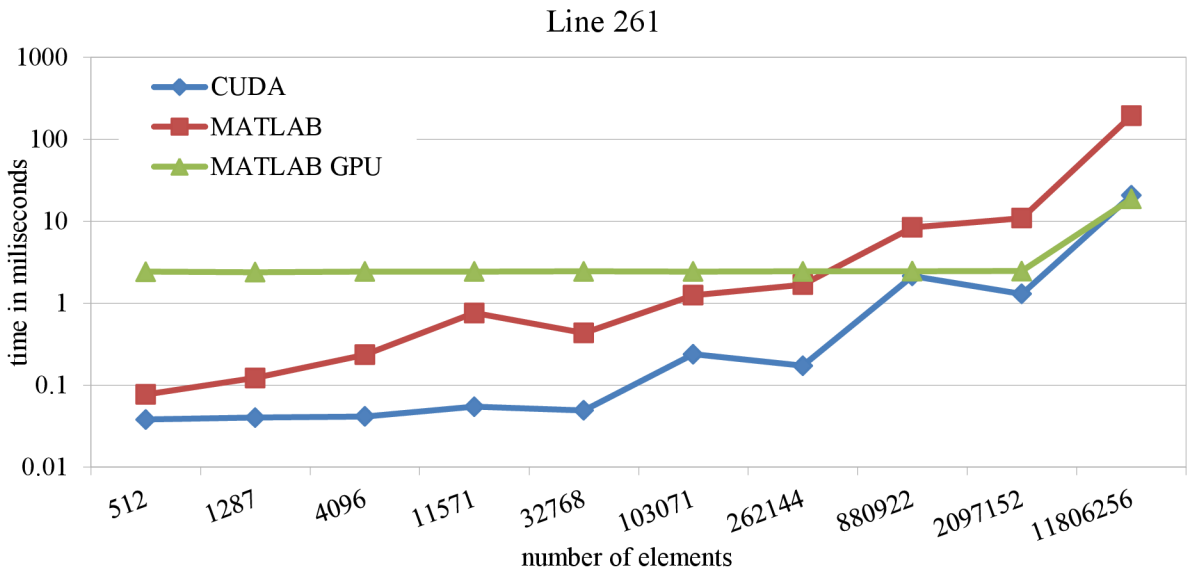


Figure 34 performance of implementations of line 261

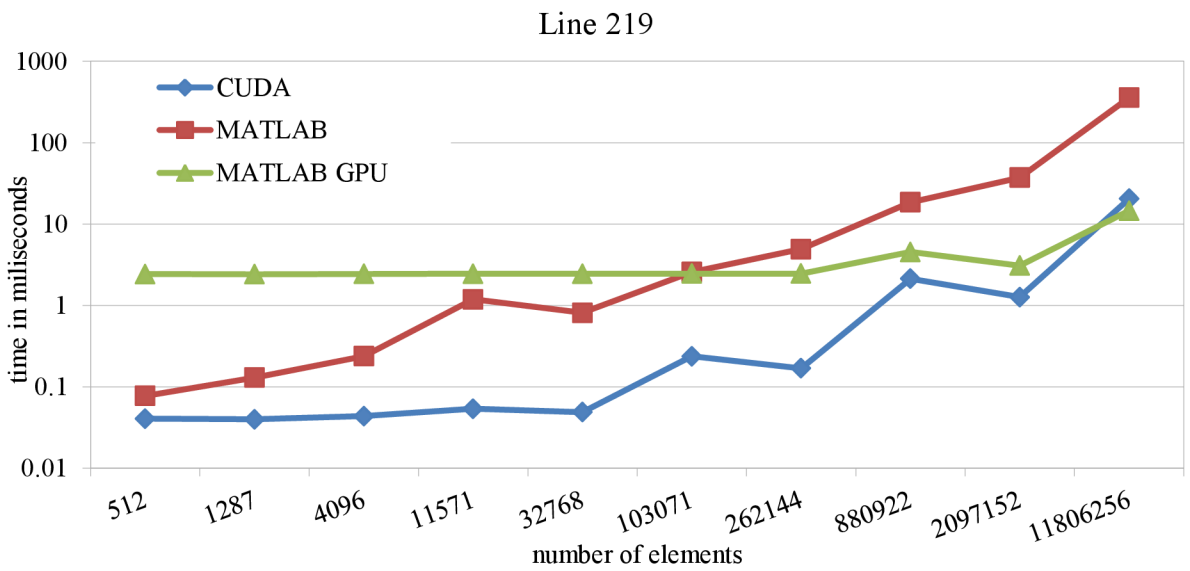


Figure 345 performance of implementations of line 219

In figure 34 and Figure 34 our implementation is represented by the blue line, the MATLAB implementation is represented by the red line and the MATLAB GPU implementation is represented by the green line. We can see that our implementation provides the best performance for input sizes that are power of 2. It is due to fact that that cuFFT library achieves best results for such sizes. Using data presented earlier in section 4.3.1, we can estimate that cuFFT takes up to 82% of run-time of these lines and therefore performance is strongly dependant on cuFFT library performance. Based on f and Figure 34, we can say that the performance of our implementation is at least comparable with MATLAB embedded GPU acceleration.

Considering results for real data sample, application provide 9.4x speed-up compared to the MATLAB implementation for line 261 and 17.5x speed-up for line 219.

The most notable improvement in terms of performance was achieved for line 285. This is positive aspect because line 285 was most time consuming line in reconstruction script. It took up to

31% of overall time to compute line. Therefore these achievements should result in notable speed-up of whole reconstruction.

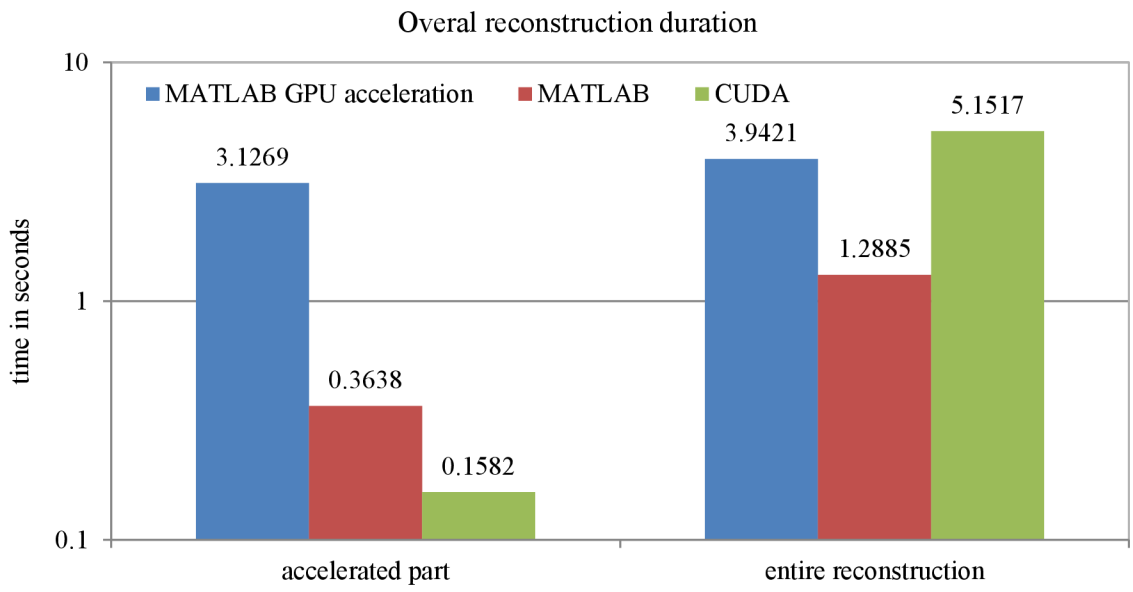


Figure 356 overall results

From Figure 35, we can say that our implementation has outstanding computation performance. Our implementation is about 20x faster than native MATLAB implementation and about 2.3x faster than the MATLAB embedded GPU acceleration in computing accelerated part of script.

On the other hand, Figure 356 also shows that for real purposes the implementation requires great amount of overhead and data transfers to run correctly and therefore does not provide overall acceleration of implementation. The main reason of poor overall performance is data transfers to and from application, despite fact that since second version of trilinear interpolation application need only 63% of former input size.

In conclusion, we discover that our implementation is not suitable to accelerate a single image reconstruction. It would be much more efficient to use the implementation in batch fashion where all images have same dimensions and properties. This would mean that five out of six data matrices needed to reconstruct image would be constant for all of images. Another way of harnessing full potential of the implementation is to create standalone application independent of the MATLAB.

5 Conclusion

If we sum up achieved results, we can state that our implementation fulfilled every goal set. It managed to accelerate selected calculations, needed in image reconstruction, approximately 20 times. The greatest acceleration was achieved for trilinear interpolation, the most time-consuming operation in the reconstruction. In fact, our implementation managed to accelerate trilinear interpolation on real data by factor 452. It can be considered outstanding performance boost, if we take in consideration that theoretical performance of used GPU is only 6.5 times greater than theoretical performance of the CPU. The approach used in our implementation also reduced amount of the input data by 37%. Each stage of development revealed different aspect of the GPU programing. Therefore, we were able to better understand certain details of the GPU programing and come up with solutions that improved the performance. Our implementation can find application in many scientific researches using

photoacoustic spectroscopy as a tool. Supposing it is used in batch mode, our implementation is able to turn image reconstruction, which was formerlly matter of days, into image reconstruction which is matter of hours. The main drawback of our application is a dependency on the MATLAB. This dependency implies that large amount of data have to be at some point transferred between the MATLAB and our application. In fact, these data transfers are so time-consuming that application used in a single mode does not provide any acceleration. For further development, we suggest disposing of dependency on MATLAB.

6 References

1. **AMD.** OpenCL™: The Future of Accelerated Application Performance Is Now. *Amd.com.* [Online] 2011. [Cited: 3 18, 2015.] https://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf.
2. **BELL, Alexander G.** On the production and reproduction of sound by light. *American Journal of Science.* 1880,118. ISSN: 0002-9599
3. **COOLEY, James W. and John W.TUKEY.** An algorithm for the machine calculation of complex Fourier series. *Mathics of Computation.* 1965, 19. ISSN: 0025-5718
4. **FOURIER, Joseph.** *Théorie analytique de la chaleur.* Paris : Firmin Didot Père et Fils, 1822. OCLC 2688081.
5. **GALLOY, Michael.** CPU vs GPU performance. *Michaelgalloy.com.* [Online] 2012. [Cited: 5 5, 2015.] http://michaelgalloy.com/wp-content/uploads/2013/06/cpu-vs-gpu.png.
6. **HYNDMAN, Rob J. and Anne B. KOHLER.** Another look at measures of forecast accuracy. *International Journal of Forecasting.* Vol. 22. ISSN: 0169-2070
7. **IT4I.** Anselm cluster documentation. *It4i.cz.* [Online] 2015. [Cited: 3 17, 2015.] https://docs.it4i.cz/anselm-cluster-documentation.
8. **KENNEDY and MIKE.** Intel Haswell. *Research.engineering.wustl.edu.* [Online] [Cited: 3 11, 2015.] http://research.engineering.wustl.edu/~songtian/pdf/intel-haswell.pdf.
9. **LEVITES, Julia and Stephen JONES.** Iside Kepler: world's fastest and most efficient accelerator. *On-demand.gputechconf.com.* [Online] [Cited: 4 20, 2015.] http://on-demand.gputechconf.com/gtc-express/2012/presentations/inside-tesla-kepler-k20-family.pdf.
10. **NVIDA.** Kepler SMX architecture. *Custompcreview.com.* [Online] 2012. [Cited: 3 30, 2015.] http://www.custompcreview.com/wp-content/uploads/2014/02/nvidia-smx-architecture-block-diagram-kepler.jpg.
11. **NVIDIA.** GeForce GTX 680 block diagram. *Pcmag.com.* [Online] 2012. [Cited: 3 14, 2015.] http://www5.pcmag.com/media/images/285620-nvidia-geforce-gtx-680-block-diagram.jpg.
12. **NVIDIA.** CUDA thread execution model. 2015. *3dgep.com.* [Online] [Cited: 4 5, 2015.] http://3dgep.com/wp-content/uploads/2011/11/grid-of-thread-blocks.png.
13. **NVIDIA.** GeForce GTX 580 specifications. *Geforce.co.uk.* [Online] 2011. [Cited: 4 20, 2015.] http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-580/specifications.
14. **NVIDIA.** GeForce GTX TITAN specifications. *Geforce.co.uk.* [Online] 2015. [Cited: 4 20, 2015.] http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-titan/specifications.

15. **NVIDIA.** Tesla C2050 performance benchmarks. *siliconmechanics.com.* [Online] [Cited: 5 10, 2015.] http://www.siliconmechanics.com/files/C2050Benchmarks.pdf.
16. **TREEBY, Bradley and B. T. COX.** k-Wave: MATLAB toolbox for the simulationand reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics.* 15, 2010. ISSN: 1083-3668
17. **WAGNR, Rick.** Multi-linear interpolation. *bmia.bmt.tue.nl.* [Online] [Cited: 1 13, 2015.] http://bmia.bmt.tue.nl/people/BRomeny/Courses/8C080/Interpolation.pdf.
18. **WILT, N.** *The CUDA handbook: comprehensive guide to GPU programing.* Boston : Addison-Wesley, 2011. ISBN-13: 978-0321809469.
19. **ZHANG, H. F. et al.** Functional photoacoustic microscopy for high-resolution and noninvasive in vivo imaging. *Nature Biotechnology.* 24, 2006. ISSN: 1087-0156