

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

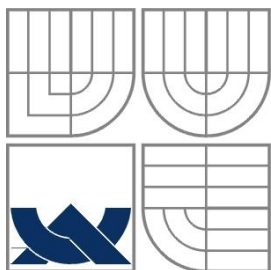
**FAST RECONSTRUCTION OF PHOTOACOUSTIC
IMAGES**

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

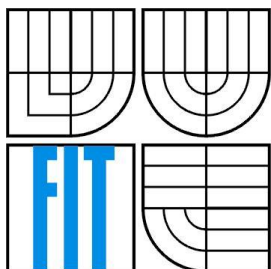
AUTOR PRÁCE
AUTHOR

Filip Kukliš

BRNO 2015



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 FOTOAKUSTICKÝCH OBRAZŮ

FAST RECONSTRUCTION OF PHOTOACOUSTIC IMAGES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

FILIP KUKLIŠ

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. JIŘÍ JAROŠ, PhD.

BRNO 2015

Abstrakt

Schopnost rekonstrukce fotoakustických obrazů je důležitým předpokladem pro studium měkkých tkání, nebo vaskulárního a lymfatického systému v malém prostoru a ve vysokém rozlišení. V současné době řešení vyžaduje enormní výpočetní výkon a je znatelně časově náročný. V této studii by jsme rádi představili nové řešení, které by bylo mnohem rychlejší a jednodušší na použití. Moje řešení je až 20x rychlejší a potřebuje o čtyřicet procent méně paměti. Toto řešení může být lepší alternativou pro vědce, kteří studují měkké tkáně pomocí fotoakustického zobrazování.

Abstract

The ability of reconstruction of photoacoustic images is important requirement to study soft tissues or vascular and lymphatic systems in high resolution but in small space. Today solution needs extensive computing power and it is noticeably time-consuming. In this study we would like to introduce a new solution which would be a way much faster and easy to use. My solution is up to 20x faster and needs forty percent less memory. This solution may be a better alternative for scientist who study soft tissues by photoacoustic imaging.

Klíčová slova

Fotoakustické zobrazování, Ultrazvuk, Vysoce-výkonné výpočty, Paralelné výpočty, Vektorizace

Keywords

Photoacoustic imaging, Ultrasonic, HighPerformanceComputing, Parallel computing, Vectorisation

Citace

Kukliš Filip: Fast Reconstruction of Photoacoustic Imaging, bakalářská práce, Brno, FIT VUT v Brně, 2015

Fast Reconstruction of Photoacoustic images

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, PhD.

Další informace mi poskytli Bradley Treeby, PhD. a Ben Cox, PhD.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Kukliš
16.05.2015

Poděkování

I would like to acknowledge the support of Jiri Jaros.

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).

© Filip Kukliš, 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..

Obsah

Obsah.....	1
1 Introduction.....	3
2 Applycation and theorethical background	4
2.1 Photoacoustic tomography.....	4
2.2 Photoacoustic microscopy	5
2.3 Theory of photoacoustic imaging	6
3 Current implementation of Photoacoustic Imaging in Matlab.....	8
3.1 Detailed description and analysis	8
4 Analysis of the current implementation in Matlab.....	11
5 Implementation of a C++ solution	14
5.1 Program design and used libraries	17
5.1.1 Implementation and verification.....	17
5.1.2 HDF5	17
5.1.3 OpenMP pragmas	17
5.2 Fourier transform – FFTW3	18
5.2.1 Matlab implementation.....	18
5.2.2 C++ implementation	18
5.3 Shifts.....	19
5.3.1 Matlab implementation.....	19
5.3.2 C++ implementation	19
5.4 Trilinear interpolation.....	20
5.4.1 Matlab implementation.....	20
5.4.2 C++ implementation	20
6 Analysis.....	22
6.1 High performance fascilities.....	22
6.2 Software used for analysis	22
6.2.1 PAPI.....	22
6.2.2 INTEL Vtune	23
6.3 Analysis of each function	24
6.3.1 Function Sf.....	24
6.3.2 Multiplication of complex numbers.....	25
6.3.3 Shifts.....	26
6.3.4 FFT	27
6.3.5 Interp.....	29

6.4	Analysis of whole computing	31
6.5	Analysis of whole program.....	33
7	Conclusion	35

1 Introduction

The main goal of this study is to rewrite algorithm of reconstruction of photoacoustic imaging implemented in Matlab as a part of international project K-Wave into C++ code as a hardware friendly and parallel code. Our effort is a way more faster code with less demand to main computer memory. We want to achieve this by simplification, parallelisation, vectorisation and code written for the given hardware.

By Matlab you can compute almost every type of mathematical or physical problem. Its main advantage is complexity and portability to many platforms. On the other hand the main disadvantage is mainly its price. Even if it uses very well optimized algorithms, its complexity may be an disadvantage for specific problems, because it does not optimise code to actual hardware.

Our effort is to create hardware friendly C++ solution which is independent on Matlab. The main advantage is that the code is implemented to concrete hardware, it know its architecture as a structure of cache levels or main memory. Another advantage is that this algorithm can be specific to one concrete computation, not every part have to be as complex as Matlab one. Often many images are computed to create three-dimensional image or video from reconstructed images, therefore is convenient that C++ solution can be also run on many compute nodes at once.

In the Figure 1 is shown output from photoacoustic imaging, three dimensional image of melanoma in vivo. It requires huge computing power to create this image. We want to use as much power as hardware provides to decrease computing time of photoacoustic imaging.

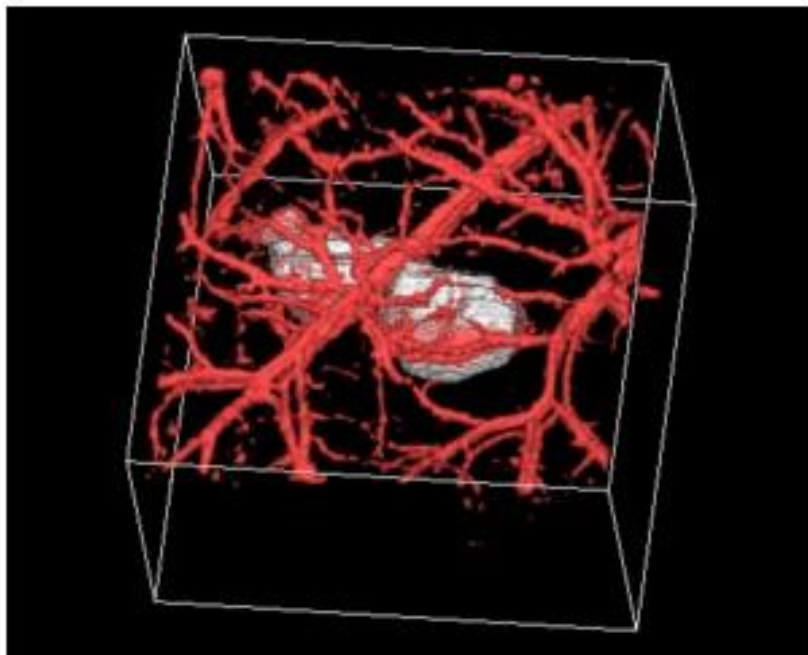


Figure 1: 3D photoacoustic imaging of melanoma in vivo.[7]

2 Application and theoretical background

A photoacoustic imaging is an emerging technique which can provide label-free non-invasive three-dimensional image of the vasculature to the depths of several cm with a spatial resolution ranging from tens to hundreds of microns (depending on the depth). It is based upon the generation of ultrasound waves through the absorption of nanosecond laser pulses by light absorbing tissue chromophores. The acoustic waves travel to the tissue surface where they are detected by an ultrasound receiver array (Figure 2). From the detected signals, the three-dimensional (3-D) images (which are proportional to the absorbed optical energy distribution) can be reconstructed. An image reconstruction is based on the acoustic time reversal algorithm.[1]

2.1 Photoacoustic tomography

Photoacoustic tomography (PAT), also known as thermoacoustic or optoacoustic tomography, is a rapidly emerging imaging technique that holds great promise for biomedical applications. PAT is a hybrid technique that exploits the high optical contrast of tissue with the high spatial resolution of ultrasonic methods. The goal of PAT is to determine an estimate of an object's spatially variant absorbed optical energy density from measurements of pressure wave fields that are induced via the thermoacoustic effect. Because the optical absorption characteristics of tissue vary strongly with hemoglobin content, knowledge of the absorbed optical energy distribution can yield both structural and functional information.[2]

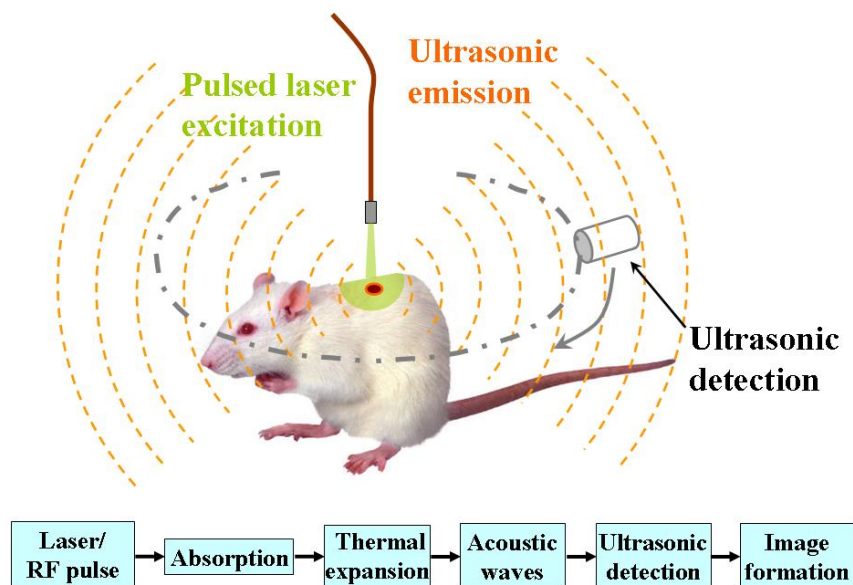


Figure 2: Schematic illustration of photoacoustic imaging.

2.2 Photoacoustic microscopy

Photoacoustic microscopy (PAM) is a hybrid *in vivo* imaging technique that acoustically detects optical contrast via the photoacoustic effect. Unlike pure optical microscopic techniques, PAM takes advantage of the weak acoustic scattering in tissue and thus breaks through the optical diffusion limit (~ 1 mm in soft tissue). With its excellent scalability, PAM can provide high-resolution images at desired maximum imaging depths up to a few millimeters. Compared with backscattering-based confocal microscopy and optical coherence tomography, PAM provides absorption contrast instead of scattering contrast. Furthermore, PAM can image more molecules, endogenous or exogenous, at their absorbing wavelengths than fluorescence-based methods, such as wide-field, confocal, and multi-photon microscopy. Most importantly, PAM can simultaneously image anatomical, functional, molecular, flow dynamic and metabolic contrasts *in vivo*. Focusing on state-of-the-art developments in PAM, this Review discusses the key features of PAM implementations and their applications in biomedical studies.[3] The scheme is in Figure 3.

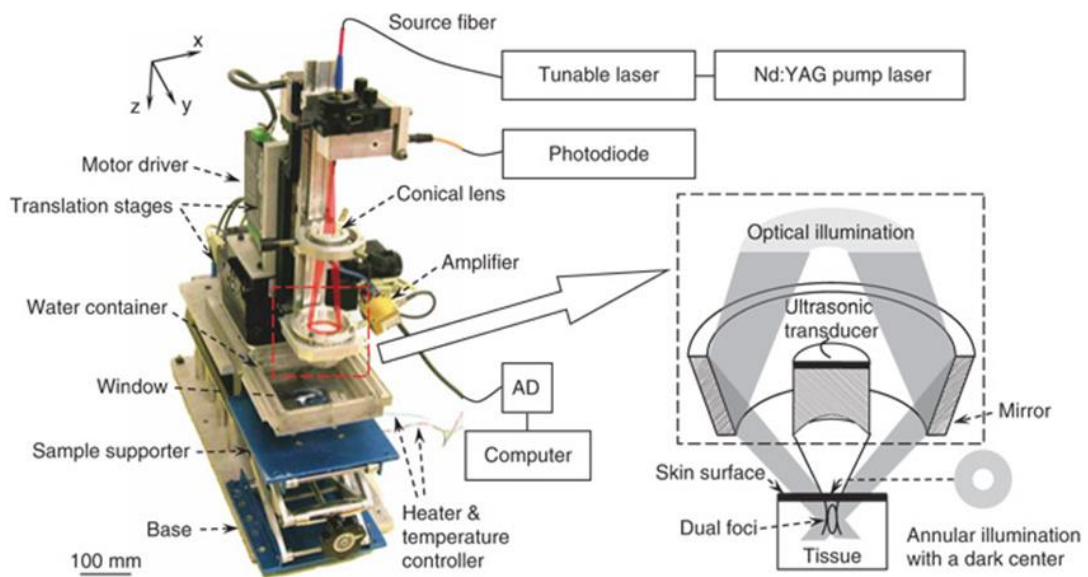


Figure 3: Experimental set-up of dark field reflection mode PAM system.[7]

2.3 Theory of photoacoustic imaging

The standard imaging model for photoacoustic imaging is derived from the acoustic wave equation in either the space–time or space–frequency domain. The space–frequency domain representation $\tilde{\mathbf{p}}(\mathbf{r}, \omega)$ of the acoustic field is related to the space–time representation $p(\mathbf{r}, t)$ by a Fourier transform, viz. Equation (1).

$$\tilde{p}(r, \omega) = \int_{-\infty}^{\infty} dt p(r, t) e^{i\omega t}, \quad (1)$$

where ω is the temporal frequency coordinate and $\mathbf{r} = (x, y, z)$.

In PAT applications, the acoustic field obeys an inhomogeneous Helmholtz equation:

$$[\nabla^2 + k^2(\mathbf{r})]\tilde{p}(\mathbf{r}, \omega) = \frac{i\omega\beta}{c_P} A(\mathbf{r})H(\omega), \quad (2)$$

where $k(\mathbf{r}) = \omega/c(\mathbf{r})$ is the spatially varying wavenumber, $c(\mathbf{r})$ is the local speed of sound, β is the thermal expansion coefficient, C_P is the specific heat capacity (at constant pressure), $A(\mathbf{r})$ is the absorbed optical energy density, and $H(\omega)$ describes the spectral content of the exciting optical or microwave pulse. We will let $A(x, y, z)$ denote $A(\mathbf{r})$ expressed explicitly in Cartesian coordinates. The pressure field away from the acoustic source can be expressed as

$$\tilde{p}(\mathbf{r}, \omega) = \frac{i\omega\beta H(\omega)}{c_P} \iiint_V d^3r' G(\mathbf{r}, \mathbf{r}', \omega) A(\mathbf{r}'), \quad (3)$$

where $G(\mathbf{r}, \mathbf{r}', \omega)$ is an appropriate Green function and V denotes the support volume of $A(\mathbf{r})$. Equation (3) represents an imaging model for PAT expressed in the temporal frequency domain. For homogenous acoustic media, $G(\mathbf{r}, \mathbf{r}', \omega) = e^{ik|\mathbf{r}-\mathbf{r}'|} / (4\pi|\mathbf{r}-\mathbf{r}'|)$. In general, the Green function can only be found analytically when the speed-of-sound map possesses certain symmetries. Otherwise, numerical methods must be employed to approximate the Green function needed to specify the imaging model in Eq. (3).

The solution to the inverse problem for PAT, i.e., the estimation of $A(\mathbf{r})$ based on knowledge of $\tilde{\mathbf{p}}(\mathbf{r}, \omega)$ and $H(\omega)$, is based on Eq. (3) but also incorporates information about the measurement geometry. In the case where the medium is acoustically homogeneous with the speed of sound c and the measurement aperture corresponds to a plane, taken to be $z = 0$ without loss of generality, a Fourier-transform-based solution to the inverse problem has been established. Let $\bar{\mathbf{p}}(k_x, k_y, \omega)$ denote the two-dimensional (2D) spatial Fourier transform of the pressure data $\tilde{\mathbf{p}}(x, y, z, \omega)$ evaluated on the measurement plane $z = 0$:

$$\bar{\mathbf{p}}(k_x, k_y, \omega) = \iint_{-\infty}^{\infty} dx dy \tilde{p}(x, y, z = 0, \omega) e^{-i(k_x x + k_y y)}. \quad (4)$$

Similarly, let $\mathcal{A}(k_x, k_y, k_z)$ denote the 3D Fourier transform of $A(x, y, z)$:

$$A(k_x, k_y, k_z) = \iiint_{\infty} dx dy dz A(x, y, z) e^{-i(k_x x + k_y y + k_z z)}. \quad (5)$$

It has been demonstrated that the values of $A(k_x, k_y, k_z)$ that reside within a sphere of radius ωc centered at the origin of 3D Fourier space can be determined from the measured pressure data as

$$A\left(k_x, k_y, \sqrt{\frac{\omega^2}{c^2} - k_x^2 - k_y^2}\right) = \frac{-2C_P \bar{p}(k_x, k_y, \omega)}{\omega \beta H(\omega)} X \sqrt{\frac{\omega^2}{c^2} - k_x^2 - k_y^2}, \quad (6)$$

where effects related to the transducer response have been suppressed. One notes that the k_z coordinate of $A(k_x, k_y, k_z)$ is found via a nonlinear mapping of k_x , k_y , and ω . From knowledge of the estimated Fourier components, a low-pass-filtered estimate of $A(\mathbf{r})$ can be determined by use of the 3D inverse Fourier transform. In Section 3 a generalization of Eq. (6) is established for the case where the optical absorber described by $A(\mathbf{r})$ is embedded in a stratified planar acoustic medium. The effects of the finite size of the detector and the finite length of the excitation pulse can be readily included in the reconstruction algorithm as described in.[2]

3 Current implementation of Photoacoustic Imaging in Matlab

The code uses a k-space algorithm which performs (1) a Fourier transform on the data p_{tyz} along both t , y , and z dimensions (into wavenumber-frequency space), (2) a mapping, based on the dispersion relation for a plane wave in an acoustically homogeneous medium, from wavenumber-frequency space to wavenumber-wavenumber space, and finally (3) an inverse Fourier transform back from the wavenumber domain to the spatial domain. The result is an estimate of the initial acoustic pressure distribution from which the acoustic waves originated.

Steps (1) and (3) can be performed efficiently using the fast Fourier transform (FFT); they are therefore fastest when the number of samples and number of detector points are both powers of 2. The mapping in step (2) requires an interpolation of the data from an evenly spaced grid of points in the wavenumber-frequency domain to an evenly-spaced grid of points in the wavenumber-wavenumber domain. The option 'Interp' may be used to choose the interpolation method.

The physics of photoacoustics requires that the acoustic pressure is initially non-negative everywhere. The estimate of the initial pressure distribution generated by this code may have negative regions due to artefacts arising from differences between the assumed model and the real situation, e.g., homogeneous medium vs. real, somewhat heterogeneous, medium; infinite measurement surface vs. finite-sized region-of-detection, etc. A positivity (or non-negativity) condition can be enforced by setting the optional 'PosCond' to true which simply sets any negative parts of the final image to zero.

3.1 Detailed description and analysis

```
1     sf=c^2*sqrt((w/c).^2-kgrid.ky.^2-
kgrid.kz.^2)./(2*w);

2     sf(w == 0 & kgrid.ky == 0 & kgrid.kz == 0) = c/2;

3     p = sf.*fftshift(fftn(fftshift(p)));

4     p(abs(w) < (c*sqrt(kgrid.ky.^2 + kgrid.kz.^2))) = 0;

5     p = interp3(kgrid.ky, w, kgrid.kz, p, kgrid.ky,
w_new, kgrid.kz, interp_method);

6     p(isnan(p)) = 0;

7     p = real(ifftshift(ifftn(ifftshift(p))));

8     p = p( (Nt + 1)/2:Nt, :, :);

9     p = 2*2*p./c;
```

Listing 1: Simplified Matlab code

Line 1 and 2 : calculate the scaling factor using the value of k_x , where $k_x = \sqrt{(w/c)^2 - k_{grid}.k_y.^2 - k_{grid}.k_z.^2}$ and then manually replacing the DC value with its limit (otherwise NaN results).

Line 3 : compute the FFT of the input data $p(t, y, z)$ to yield $p(w, k_y, k_z)$ and scale.

Line 4 : exclude the inhomogeneous part of the wave.

Line 5: compute the interpolation from $p(w, k_y, k_z)$ to $p(k_x, k_y, k_z)$; for a matrix indexed as $[M, N, P]$, the axis variables must be given in the order N, M, P

Line 6: set values outside the interpolation range to zero

Line 7: compute the inverse FFT of $p(k_x, k_y, k_z)$ to yield $p(x, y, z)$

Line 8: remove the left part of the mirrored data which corresponds to the negative part of the mirrored time data

Line 9: correct the scaling - the forward FFT is computed with a spacing of Δt and the reverse requires a spacing of $\Delta z = \Delta t * c$, the reconstruction assumes that p_0 is symmetrical about z , and only half the plane collects data (first approximation to correcting the limited view problem)
($p_{z \times y}$)

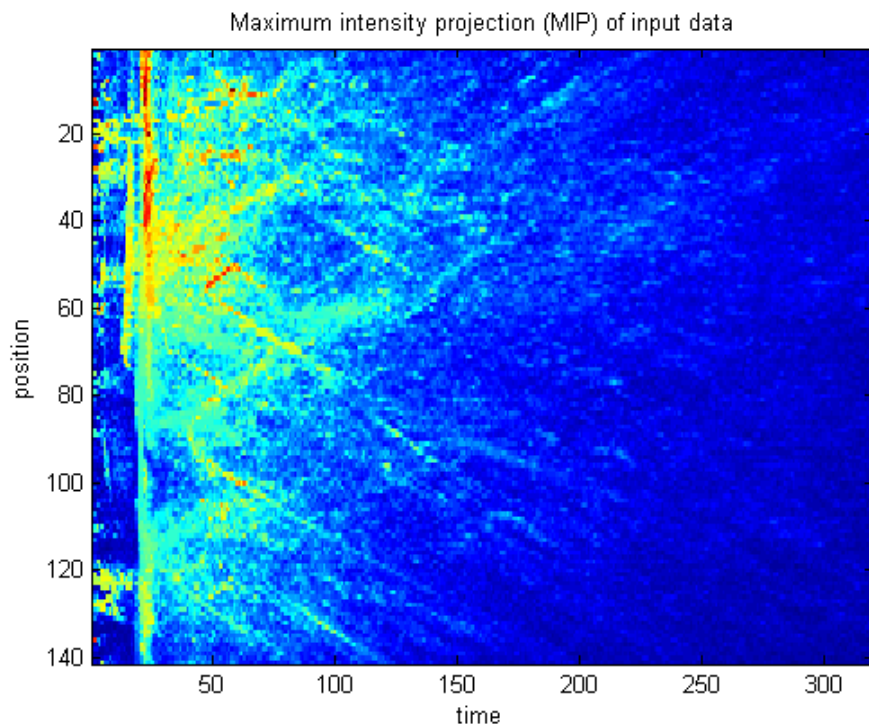


Figure 4: Input signal data recorded outside the tissue.

The input signal a is three-dimensional matrix and can be seen in the Figure 4. Two dimensions of the matrix are the data from the sensor and the third is time. Resolution can be upsampled by the nearest-neighbour algorithm. It is shown that the quality of reconstructed images depends on the input resolution in the Figure 5, Fig. 6 and Fig. 7. The difference in quality of the pictures can be seen by eye. From the resulting quality we want to aim on the 64x upsampled input data and its reconstruction.

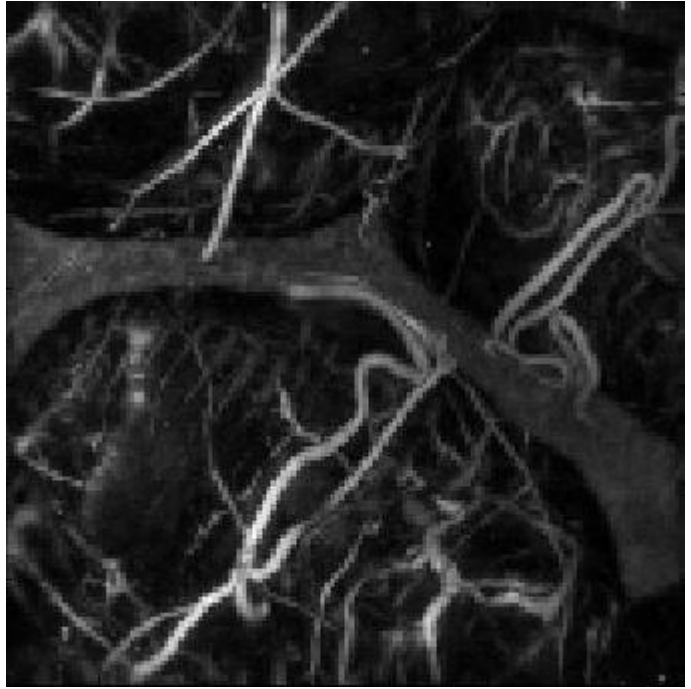


Figure 5: Result, reconstructed image of a mouse embryos in vivo. Original input data 320x141x141.

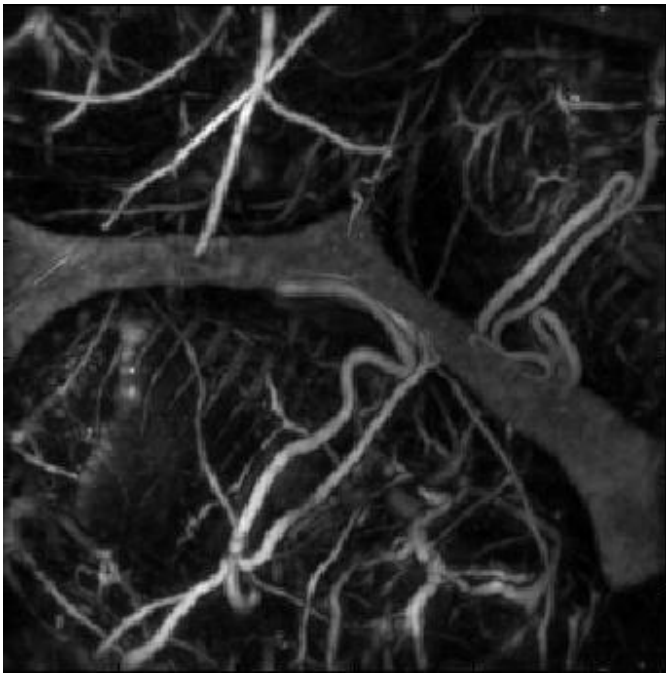


Figure 6: Result, reconstructed image of a mouse embryos in vivo. 8x upsampled input data 639x281x281.

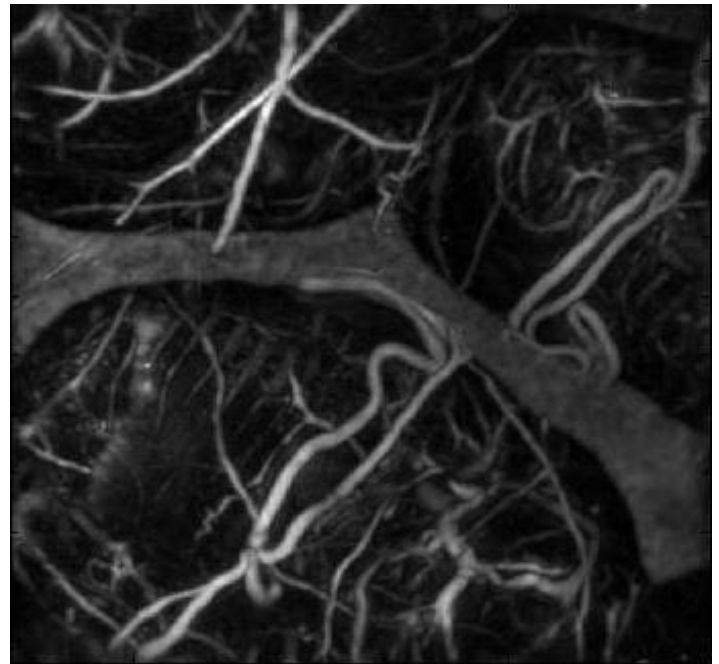


Figure 7: Result, reconstructed image of a mouse embryos in vivo. 64x upsampled input data 1277x561x561.

4 Analysis of the current implementation in Matlab

The Matlab Profiler was used for the code analysis. The assumption was that with higher resolution of the input data the computing time was longer and time of each function rised constantly. In this part we want to find out functions where the most time was spent and the relation between computing time and resolution used.

In the Figure 8 we can see that most time was spent on trilinear interpolation, more than thirty percent for a non upsampled image. But the total time of computing is around six seconds, which is so little that it is not worth to optimise the code at this resolution.







Line Number	Code	Calls	Total Time	% Time	Time Plot
186	<code>p = interp3(kgrid.ky, w, kgrid...</code>	1	1.994 s	31.8%	
195	<code>p = real(iffshift(iffn(iffst...</code>	1	1.260 s	20.1%	
175	<code>p = sf.*fftshift(ffn(fftshift...</code>	1	1.055 s	16.8%	
170	<code>sf = c^2*sqrt((w/c).^2 - kgri...</code>	1	0.682 s	10.9%	
181	<code>p(abs(w) < (c*sqrt(kgrid.ky...</code>	1	0.433 s	6.9%	
All other lines			0.839 s	13.4%	
Totals			6.264 s	100%	

Figure 8: Output from Matlab profiler input resolution: 320x141x141

In the Figure 9 we can see that with higher resolution order of functions is completely different. The most time was spent on forward and inverse Fourier transforms.







Line Number	Code	Calls	Total Time	% Time	Time Plot
175	<code>p = sf.*fftshift(ffn(fftshift...</code>	1	28.933 s	42.2%	
195	<code>p = real(iffshift(iffn(iffst...</code>	1	28.849 s	42.0%	
186	<code>p = interp3(kgrid.ky, w, kgrid...</code>	1	5.495 s	8.0%	
170	<code>sf = c^2*sqrt((w/c).^2 - kgri...</code>	1	1.928 s	2.8%	
181	<code>p(abs(w) < (c*sqrt(kgrid.ky...</code>	1	1.162 s	1.7%	
All other lines			2.269 s	3.3%	
Totals			68.636 s	100%	

Figure 9: Output from Matlab profiler input resolution: 639x281x281

In the [Figure 10](#) we can see that most time was spent on trilinear interpolation again. Whole computation lasts 885 seconds which is around fifteen minutes. Therefore we want to optimise the code at this resolution. In this resolution reconstructed images look pretty good but computing time is long.







Line Number	Code	Calls	Total Time	% Time	Time Plot
186	<code>p = interp3(kgrid.ky, w, kgrid...</code>	1	329.652 s	37.2%	
195	<code>p = real(ifftshift(ifftn(iffts...</code>	1	167.038 s	18.9%	
170	<code>sf = c^2*sqrt((w/c).^2 - kgri...</code>	1	135.031 s	15.2%	
175	<code>p = sf.*fftshift(fftn(fftshift...</code>	1	109.808 s	12.4%	
181	<code>p(abs(w) < (c*sqrt(kgrid.ky...</code>	1	54.192 s	6.1%	
All other lines			89.830 s	10.1%	
Totals			885.550 s	100%	

Figure 10: Output from Matlab profiler input resolution: 1277x561x561

Table 1: Dependence of computing time and memory demand at resolution

Resolution of input data	Multiple of resolution	Computing time	Multiple of time	Memory demand	Multiple of memory demand
320x141x141	1 x	6 s	1 x	2,7 GB	1 x
639x281x281	8 x	68 s	11.3 x	20 GB	7.4 x
1277x561x561	64 x	885 s	147.5 x	125 GB	46.3

In the Table 1 we can see the dependance of the computing time and memory needs on resolution used. It is expected that the computing time will rise with resolution. With the higher resolution increase of computing time is huge, it rises more than resolution of input data, which is expected. On the other hand memory demand does not rise as fast as resolution, but is high enough to attempt to reduce it.

Two outcomes resulting from this analysis are important. The first finding is that the computing time of functions does not rise linearly due to the fact that complexity of FFT is $O(n \log n)$ and there are higher requirements for the memory using higher resolution of input data. The second crucial point is that four lines were selected which takes more than eighty percent of whole computing. These lines can be seen in the Figure 10 as lines 186 195 170 and 175. Therefore, it is important to optimise these lines for a better performance.

5 Implementation of a C++ solution

A lot of tests have been performed to understand the behaviour of parallelized and vectorised code in the different situations. For example working with data which fits in L1, L2 or L3 cache, working with constant data but in the different number of iterations, loops with or without vectorization or testing differences between `#pragma omp for` and manually written parallel loop with OpenMP. These and many other tests have been done to gain expertise. Some of microbenchmark results are shown in the following graphs. The axis Y represents the computing time and axis X represents a number of cores. Both of them are in logarithmic scale. The presumption was that curve in these graphs should fall linearly without waves.

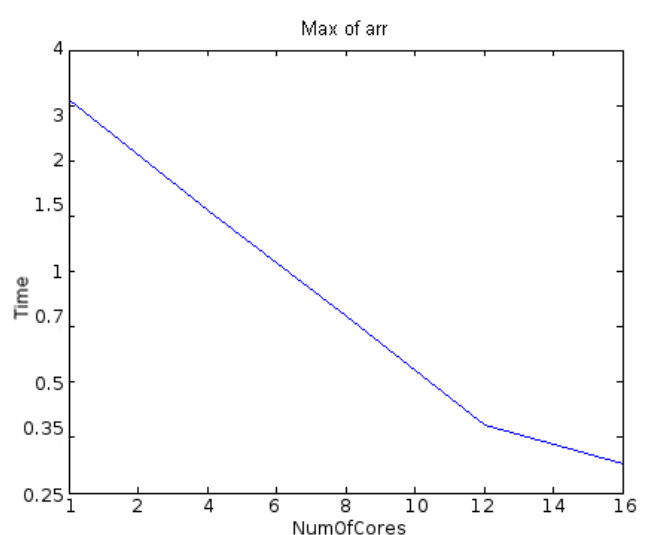
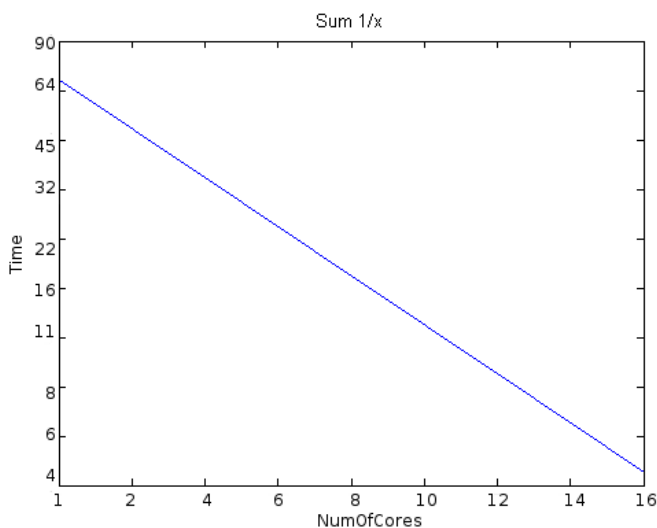
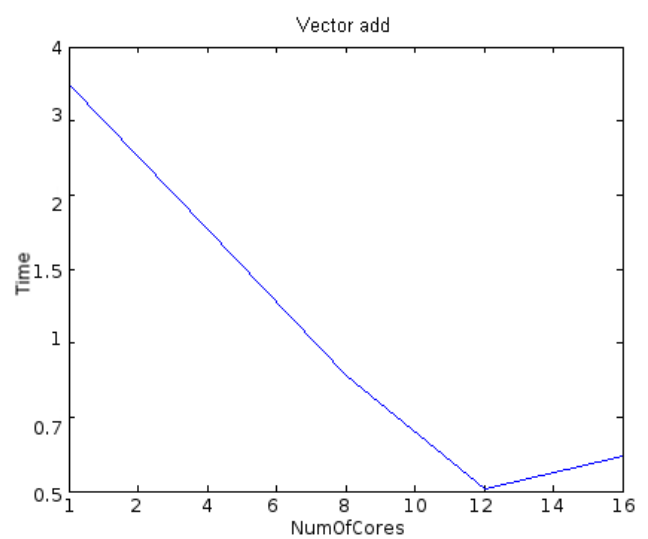
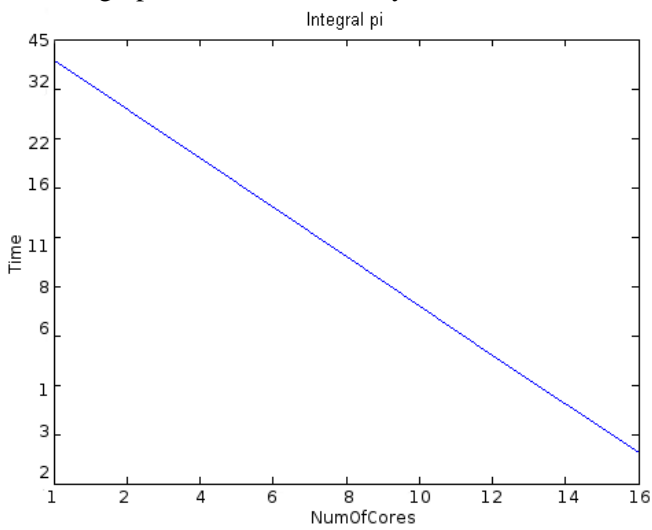


Figure 12: Ideal scaling at unused operations

Figure 11: Test of used operations

In the Figure 11 it is shown what the ideal scaling should look like. This test proves that ideal scaling can be achieved. However, in this case, there are data shared and only the computation is distributed over cores. This type of operations we will not use. On the other hand in the Figure 12 it is shown non-ideal graphs (with higher number of cores it takes more time to compute or a curve is not linear), which should be caused by maximal memory bandwidth. There are operations which we will use such a vector addition or searching in the array. There it is shown that real solutions never look like an ideal graph in the following graph.

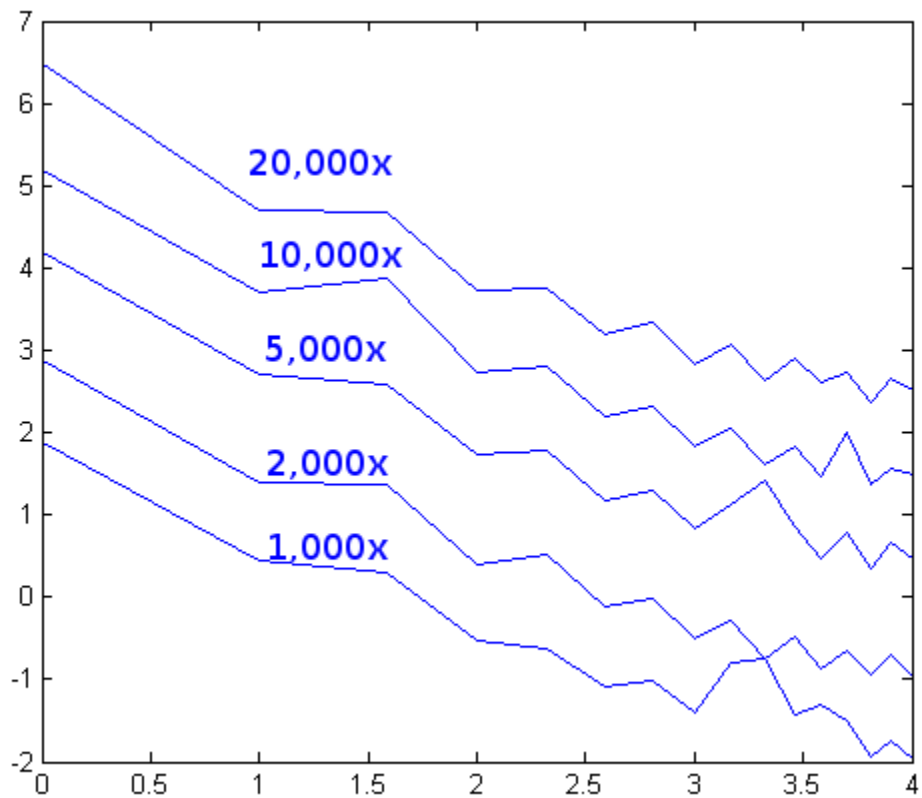


Figure 13: Repeated iterations of vector addition

In the Figure 13, time is measured for all from 1 to 16 cores. In this graph computing time of vector addition is shown computed several times with more or less iterations. This test was done to understand influence of thread creation on computing time. The divergence between the cores and a curve, which is definitely not straight, can be seen. This appearance is caused by divisibility of data by core numbers.

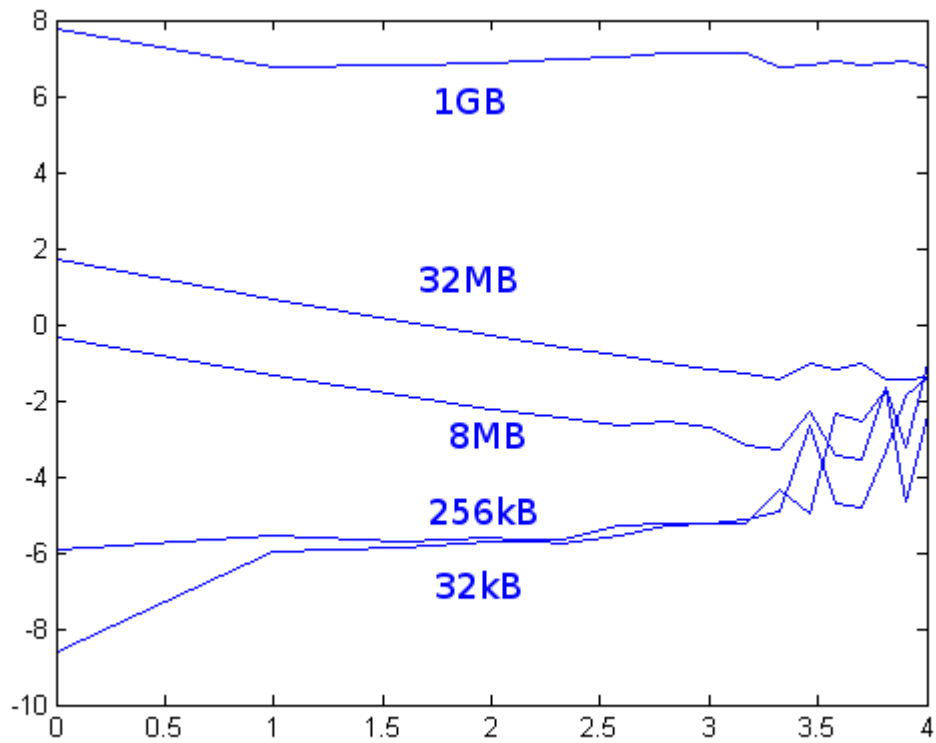


Figure 14: Vectorised vector addition

Figure 14 is very interesting. There is shown that the computing time of vector addition with vectorization and vectors were selected to fit in L1, L2 or L3 cache. But the result looks like that there is an error in the code. It was found that it is not an error. The vector addition is really fast by using the vectorization, therefore the computer memory and threads creation decelerate it, insofar on one core it can be faster than on sixteen cores. The first outcome of the tests is that sometimes it is not necessary to parallel the code and the second one is that there are a lot of mistakes which can be made in the parallelize code. However, many of them were detected in tests such as a first touch problem or data antialiasing.

5.1 Program design and used libraries

The C++ solution is implemented as a stand-alone program. An alternative was a mex function in Matlab (built-in function). Its main advantage is passing arguments without saving them to hard drive. A standalone program was chosen, because we wanted independence from Matlab. The problem with passing arguments by hard disk can be eliminated by using a SSD (solid state drive) or RAMDISK.

5.1.1 Implementation and verification

Method of implementation was to make a set of unit tests and compare them to Matlab routines. At first input arguments of the function were saved in Matlab to hard drive. After implementation in C++ saved arguments were used as input arguments of C++ solution. Output was also saved to hard drive and then loaded in Matlab after function. By doing this the C++ solution was verified. At the end every function was optimized for hardware to accelerate computation. After implementation, verification and optimization functions become part of whole program in C++.

The computation uses real and imaginary parts of complex numbers. These numbers are stored in three-dimensional matrices. But they are saved as two-dimensional matrices. First dimension is used to store three-dimensional matrix of real numbers and the second to store three-dimensional matrix of imaginary part of complex numbers.

5.1.2 HDF5

HDF5 was used for passing arguments from Matlab to C++ code. HDF5 is a file format designed to store and organize large amounts of numerical data. It is used for passing arguments from Matlab to a stand-alone C++ program. It is supported by Matlab and there is a library for C++.

5.1.3 OpenMP pragmas

To parallelize and vectorize code these pragmas were mostly used:

```
#pragma omp parallel for (1)
```

```
#pragma omp parallel for simd (2)
```

Pragma(1) commands to parallelize for loop and pragma(2) commands even to vectorize the code.

5.2 Fourier transform – FFTW3

5.2.1 Matlab implementation

`Y = fftn(X)` returns the discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = ifftn(X)` returns the `n`-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

5.2.2 C++ implementation

The FFTW3 library was used for compute Fourier transforms in the new solution. It was chosen because of its speed and simplicity.

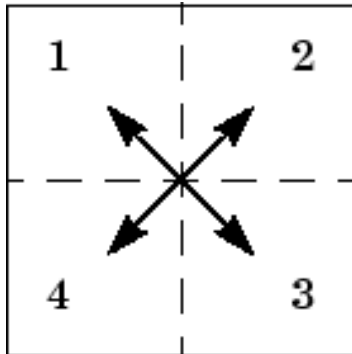
The **Fastest Fourier Transform in the West (FFTW)** is a software library for computing discrete Fourier transforms (DFTs) developed by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology.

FFTW is known as the fastest free software implementation of the Fast Fourier transform (FFT) algorithm (upheld by regular benchmarks). It can compute transforms of real and complex-valued arrays of arbitrary size and dimension in $O(n \log n)$ time.[5]

At the begining FFTW has to find an optimized plan by actually computing several FFTs and measuring their execution time. This can take some time, with highest resolution it can be more than twenty minutes. But whenever you create a plan, the FFTW planner accumulates wisdom, which is information sufficient to reconstruct the plan. After planning, you can save this information to disk. This wisdom have to be computed only once for given resolution and other times can be loaded from disk.

5.3 Shifts

5.3.1 Matlab implementation



$Y = \text{fftshift}(X)$ rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

For vectors, `fftshift(X)` swaps the left and right halves of X . For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.

For higher-dimensional arrays, `fftshift(X)` swaps "half-spaces" of X along each dimension.

$Y = \text{ifftshift}(X)$ swaps the left and right halves of the vector X . For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If X is a multidimensional array, `ifftshift(X)` swaps "half-spaces" of X along each dimension.

5.3.2 C++ implementation

The first implementation was that the matrix was passed by *pragma for* and for each element was computed position of another element. Between these two cells data was swapped. This whole function was parallelized.

The implementation was ten time faster, but I wanted to try another implementation. In the second implementation the data was not swapped between two elements of the matrix but whole matrix lines were divided in half and this whole half of the line, whole vectors was swapped. For that reason the function `memcpy()` was used, copying the whole vector from one part of the memory to another. However this implementation was as fast as the first one. This is due to the fact that this function is memory bound. With higher number of cores or with less computations the time do not have to necessarily decrease.

At the end, the first implementation was chosen since it has a simpler code and comparable performance. Two instances of the routine swapping two matrices(real and imaginary part) at the same time and two instances swap only one matrix at the time.

5.4 Trilinear interpolation

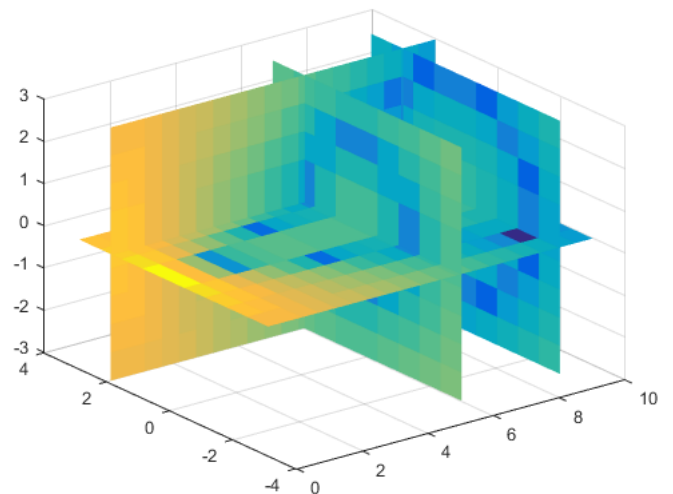
Trilinear interpolation is a method of multivariate interpolation on a 3-dimensional regular grid.

It approximates the value of an intermediate point (x, y, z) within the local axial rectangular prism linearly, using data on the lattice points. For an arbitrary, unstructured mesh (as used in finite element analysis), other methods of interpolation must be used; if all the mesh elements are tetrahedra (3D simplices), then barycentric coordinates provide a straightforward procedure.[6]

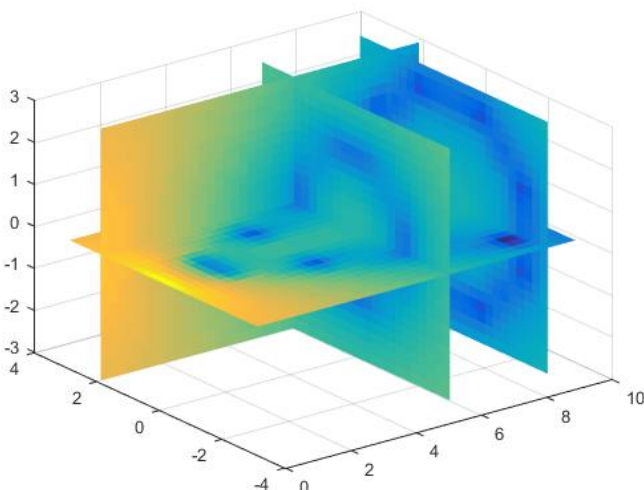
5.4.1 Matlab implementation

```
Vq = interp(X, Y, Z, V, Xq, Yq, Zq)
```

returns interpolated values of a function of three variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. X , Y , and Z contain the coordinates of the sample points. V contains the corresponding function values at each sample point. Xq , Yq , and Zq contain the coordinates of the query points.



5.4.2 C++ implementation



After understanding that the data is interpolated only in one direction (in one coordinate axis) the decision was not to implement a complex trilinear interpolation or to use existing algorithm or existing C++ library, but the custom solution. The not

complex solution was proposed. This solution can not be used as fully working trilinear interpolation and it will be used only for this project.

For every point in the three-dimensional matrix the new interpolated value has to be computed. The matrix is passed point by point by parallelized `for loop` and for every point there is a search of new dimension of the point. Linear search algorithm was used for searching. If the search was succesfull, then the calculation of the point interpolation is done. If the search was not succesfull the point does not have an interpolated value and the result should be NaN(not a number). But instead of NaN, zero was the result in my implementation, because of the fact that in Matlab is NaN overwritten by zero after interpolation.

This implementation was only two and half times faster than the Matlab implementation. Therefore the linear search was replaced by the Binary search algorithm. After some tests and analyzes the dimension was searched from the end of the vector by linear searching algorithm. This time the code was more than twelve times faster.

This implemenation can not be used as fully working interpolation, but does what exectly has to do. The solution is parallelized but was not vectorized, because of condition in search.

6 Analysis

6.1 High performance facilities

All work was done on Anselm, a supercomputer cluster in Ostrava, Czech Republic. The Anselm cluster consists of 209 compute nodes, totaling 3 344 compute cores with 15TB RAM and giving over 94 Tflop/s theoretical peak performance. Each node is a power-ful x86-64 computer, equipped with 16 cores, at least 64GB RAM and 500GB hard drive.[4]

The compute node used for testing consists of two eight-core Intel Sandy Bridge E5-2740 processor and 96GB memory. Processors support Advanced Vector Extensions (AVX) 256-bit instruction set.

Intel Sandy Bridge E5-2470 Processor

- eight-core
- speed: 2.4 GHz, up to 3.1 GHz using Turbo Boost Technology
- peak performance: 18.4 Gflop/s per core
- caches:
 - L2: 256 KB per core
 - L3: 20 MB per processor
- memory bandwidth at the level of the processor: 38.4 GB/s

6.2 Software used for analysis

6.2.1 PAPI

PAPI (Performance Application Programming Interface) provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

PAPI was used to measure various performance indicators such as time, FLOPS(FLoating-point OperationsPerSecond) or distribution of load on CPU cores.

Table 2: List of used PAPI events

PAPI Event	Description
PAPI_FP_OPS	<i>Floating point operations</i>
PAPI_DP_OPS	<i>Double precision operations</i>
PAPI_TOT_CYC	<i>Total cycles</i>
PAPI_TOT_INS	<i>Instructions completed</i>
PAPI_L3_TCA	<i>Level 3 total cache accesses</i>
PAPI_L3_TCM	<i>Level 3 total cache misses</i>
PAPI_L3_TMR	<i>Level 3 total miss ratio</i>
MEM_BANDW	<i>Memory bandwidth</i>
MFLOPS	<i>Million floating point operations per second</i>
VEC_MFLOPS	<i>Vectorised MFLOPS</i>

6.2.2 INTEL Vtune

VTune Amplifier assists in various kinds of code profiling including stack sampling, thread profiling and hardware event sampling. The profiler result consists of details such as time spent in each sub routine which can be drilled down to the instruction level. The time taken by the instructions are indicative of any stalls in the pipeline during instruction execution. The tool can be also used to analyze thread performance. The new GUI can filter data based on a selection in the timeline.

6.3 Analysis of each function

6.3.1 Function Sf

Sf function is rewritten line 1 from Matlab:

$$sf = c^2 * \sqrt{(w/c).^2 - kgrid.ky.^2 - kgrid.kz.^2} ./ (2*w); \quad (1)$$

The whole computing was parallelized and but not vectorised because of condition in the computing. In the Table 2 is shown output from PAPI. MFLOPS are not very high, although the computing time of C++ solution is more than seventy-five times faster than Matlab solution, 135 seconds in Matlab versus 1.8 second in C++.

Table 3: PAPI output, function: Sf

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_DP_OPS	813M	811M	809M	808M	807	806
PAPI_TOT_CYC	2102M	2179M	2175M	2247M	2781M	2792M
PAPI_L3_TMR	14.3%	12%	12%	13%	14%	12.8%
MFLOPS	0.000100556	8.88892e-05	3.66668e-05	9.88892e-05	3.33334e-05	0.000106667
VEC_MFLOPS	437	436	435	435	434	433
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_DP_OPS	805M	805M	804M	805M	806M	807M
PAPI_TOT_CYC	2823M	2773M	2809M	2787M	2272M	2192M
PAPI_L3_TMR	14%	12.4%	14.8%	11.9%	12.5%	13.5%
MFLOPS	3.44e-05	9.16e-05	6.00e-05	1.26e-04	8.61e-05	5.44e-05
VEC_MFLOPS	433	433	433	433	433	434
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_DP_OPS	808M	809M	811M	812M	12934M	
PAPI_TOT_CYC	2824M	2790M	2279M	2196M	40029M	
PAPI_L3_TMR	14.1%	13.8%	13.4%	12.1%	13.1625%	
MFLOPS	17.11e-04	1.27e-04	8.94e-05	5.50e-05	0.0013	
VEC_MFLOPS	434	435	436	437	6950	

```
-----
sf :: wall time 1.79999 s
-----
```

6.3.2 Multiplication of complex numbers

In function `Complex`, the multiplication of complex numbers is implemented. The matrix `sf` computed in function `Sf` and output of `fftshift` and `fft` are multiplied.

```
p = sf.*fftshift...;
```

In the Table 3 we can see that FLOPS are not very high again. It is due to fact that there is too many accesses to main memory to only one operation. But C++ solution is many time faster than Matlab solution again.

Table 4: PAPI output, function: `Complex`

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_DP_OPS	326M	336M	337M	339M	328M	337M
PAPI_TOT_CYC	4437M	4580M	4564M	4583M	4593M	4586M
PAPI_L3_TMR	18.3%	18.5%	18.5%	18.6%	18.7%	18.4%
MFLOPS	2.48e-05	4.86e-05	2.37e-05	3.92e-05	2.76e-05	4.75e-05
VEC_MFLOPS	198.371	204.41	204.802	206.193	199.272	204.723
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_DP_OPS	336M	339M	329M	327M	339M	329M
PAPI_TOT_CYC	4582M	4582M	4597M	4601M	4583M	4599M
PAPI_L3_TMR	18.4%	18.5%	18.6%	18.5%	18.6%	18.7%
MFLOPS	3.70e-05	5.19e-05	4.75e-05	5.02e-05	3.42e-05	2.48e-05
VEC_MFLOPS	204.202	206.014	199.711	199.013	205.99	199.875
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_DP_OPS	327M	339M	327M	328M	5331M	
PAPI_TOT_CYC	4596M	4573M	4594M	4574M	73231M	
PAPI_L3_TMR	18.5%	18.6%	18.5%	18.7%	18.5368%	
MFLOPS	4.97e-05	5.08e-05	3.42e-05	1.87e-05	0.000610968	
VEC_MFLOPS	198.624	206.148	198.712	199.454	3235.51	

```
-----  
complex :: wall time 1.64792 s  
-----
```

6.3.3 Shifts

The Matlab functions `fftshift()` and `ifftshift()` were implemented as four functions, because of the fact that each of these functions occurs in Matlab code twice. The first time the function shifts whole complex numbers, real and imaginary parts and the second time only real part of complex number. The wall time of the first function is higher than others because of clearing imaginary part of matrix. In the Table 4 is shown that shifts are limited by memory bandwidth. Actually memory bandwidth is slightly higher than maximal bandwidth by processor, it is due to CPU caches.

Table 5: PAPI output, function: `fftshift`

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_L3_TCA	17.16M	16.60M	16.61M	167.42M	14.77M	16.62M
PAPI_L3_TCM	5.78M	5.63M	5.61M	5.68M	5.22M	5.66M
PAPI_TOT_CYC	3061M	3038M	2990M	3025M	3536M	3037M
PAPI_L3_TMR	33.7%	33.9%	33.8%	34%	35.3%	34%
MEM_BANDW	5300MB/s	5300MB/s	5300MB/s	5300MB/s	5300MB/s	5300MB/s
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_L3_TCA	14.81M	14.72M	16.07M	15.98M	16.07M	14.22M
PAPI_L3_TCM	5.17M	5.22M	5.47M	5.44M	5.46M	5.11M
PAPI_TOT_CYC	3512M	3537M	3037M	3038M	3032M	3539M
PAPI_L3_TMR	34.9%	35.5%	34.1%	34.1%	34%	36%
MEM_BANDW	5300MB/s	5300MB/s	5300MB/s	5300MB/s	5300MB/s	5300MB/s
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_L3_TCA	14.42M	14.35	14.28M	14.26M	247.7M	
PAPI_L3_TCM	5.12M	5.09M	5.12M	5.13M	85.99M	
PAPI_TOT_CYC	3532M	3529M	3538M	3528M	52517M	
PAPI_L3_TMR	35.5%	35.5%	35.9%	36%	34.76%	
MEM_BANDW	5300MB/s	5300MB/s	5300MB/s	5300MB/s	84 800MB/s	

fftshifts :: wall time 3.80506 s, 1.26279 s, 1.25682 s, 1.28368 s

6.3.4 FFT

Forward and inverse Fourier transforms (`fftn()`, `ifftn()`) are implemented by FFTW3 library and reaches around 4500 MFLOPS on one core which can see in Table 5 and Table 6 is nearly maximum you can get by real application. It is well optimised and with this dimensions of matrix it is maximal performance.

Table 6: PAPI output, function: `fft`

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_FP_OPS	10816M	10815M	10815M	10816M	10815M	10815M
PAPI_DP_OPS	14780M	14773M	14786M	14784M	14791M	14784M
PAPI_TOT_CYC	14902M	13798M	12778M	13537M	13799M	13759M
MFLOPS	1900	1900	1900	1900	1900	1900
VEC_MFLOPS	2600	2600	2600	2600	2600	2600
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_FP_OPS	10815M	10816M	10814M	10815M	10815M	10814M
PAPI_DP_OPS	14779M	14772M	14783M	14777M	14786M	14781M
PAPI_TOT_CYC	13759M	13803M	14589M	13797M	13795M	13525M
MFLOPS	1900	1900	1900	1900	1900	1900
VEC_MFLOPS	2600	2600	2600	2600	2600	2600
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_FP_OPS	10815M	10812M	10815M	7166M	169397M	
PAPI_DP_OPS	14779M	13357M	14774M	10197M	231915M	
PAPI_TOT_CYC	13422M	13428M	13798M	9858M	216333M	
MFLOPS	1900	1900	1900	1260	29 800	
VEC_MFLOPS	2600	2600	2600	1790	40 800	

```
-----
fft :: wall time 5.68 s
-----
```

Table 7: PAPI output, function: *ifft*

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_FP_OPS	10593M	10593M	10593M	10595M	10595M	10595M
PAPI_DP_OPS	14534M	14539M	14540M	14538M	14536M	14537M
PAPI_TOT_CYC	12855M	12750M	12850M	12848M	12845M	12818M
MFLOPS	2300	2300	2300	2300	2300	2300
VEC_MFLOPS	3160	3160	3160	3160	3160	3160
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_FP_OPS	10596M	10596M	10595M	10595M	10591M	10596M
PAPI_DP_OPS	14536M	14536M	14535M	14535M	14535M	14536M
PAPI_TOT_CYC	12762M	12846M	12845M	12850M	12850M	12850M
MFLOPS	2300	2300	2300	2300	2300	2300
VEC_MFLOPS	3160	3160	3160	3160	3160	3160
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_FP_OPS	10596M	10595M	10596M	9712M	168652M	
PAPI_DP_OPS	14533M	14537M	14531M	12920M	230963M	
PAPI_TOT_CYC	12851M	12838M	12851M	12138M	204653M	
MFLOPS	2300	2300	2300	2300	36 700	
VEC_MFLOPS	3160	3160	3160	3160	50 200	

ifft :: wall time 4.6 s

6.3.5 Interp

Two prototypes were implemented. In the second one early six times less instructions were executed by the turning of the direction of search. It is shown in Table 7 and Table 8. Load distribution on CPU cores is not as good as by FFTW3 but it is still good enough. It is more than twelve times faster than interpolation in Matlab(`interp()`). The FLOPS performance is nearly maximum you can get without vectorisation.

Table 8: PAPI output, function: *interp*

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_FP_OPS	211973M	204085M	202294M	200712M	199370M	198302M
PAPI_DP_OPS	492M	485M	489M	492M	494M	495M
PAPI_TOT_CYC	345G	333G	330G	328G	326G	324G
PAPI_TOT_INS	846G	815G	808G	801G	796G	791G
MFLOPS	1705.08	1641.63	1627.23	1614.5	1603.71	1595.11
VEC_MFLOPS	3.9607	3.9019	3.9364	3.9585	3.977	3.9886
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_FP_OPS	197547M	197151M	197157M	197564M	198328M	199404M
PAPI_DP_OPS	496M	497M	497M	496M	496M	494M
PAPI_TOT_CYC	323G	322G	324G	323G	324G	326G
PAPI_TOT_INS	789G	787G	789G	789G	792G	796G
MFLOPS	1589.04	1585.86	1585.9	1589.17	1595.32	1603.98
VEC_MFLOPS	3.9972	4.0052	4.0048	3.9973	3.9899	3.9755
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_FP_OPS	200754M	202341M	204139M	206118M	3217247M	
PAPI_DP_OPS	491M	488M	485M	480M	7875M	
PAPI_TOT_CYC	330G	330G	333G	336G	5266G	
PAPI_TOT_INS	804G	808G	815G	823G	12857G	
MFLOPS	1657.93	1614.83	1627.61	1642.07	25879	
VEC_MFLOPS	3.9544	3.9303	3.903	3.8652	63.3467	

```
-----
interp :: wall time 124.319 s
-----
```

Table 9: PAPI output, function: interp

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_FP_OPS	38692M	36484M	35588M	34898M	34390M	34045M
PAPI_DP_OPS	492M	484M	488M	491M	493M	495M
PAPI_TOT_CYC	75328M	72495M	71305M	70464M	69966M	69385M
PAPI_TOT_INS	147G	140G	137G	135G	134G	133G
MFLOPS	1427.91	1346.36	1307.39	1313.32	1269.12	1256.42
VEC_MFLOPS	18.1805	17.8887	18.0135	18.1306	18.2068	18.287
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_FP_OPS	33840M	33751M	33752M	33844M	34053M	34400M
PAPI_DP_OPS	496M	497M	497M	496M	494M	493M
PAPI_TOT_CYC	69439M	69575M	70133M	69546M	69430M	70174M
PAPI_TOT_INS	132G	132G	133G	132G	133G	134G
MFLOPS	1248.82	1245.56	1245.61	1248.94	1256.67	1269.58
VEC_MFLOPS	18.3352	18.3522	18.3608	18.307	18.2601	18.2124
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_FP_OPS	34914M	35610M	36510M	37638M	562416M	
PAPI_DP_OPS	491M	488M	484M	480M	7866M	
PAPI_TOT_CYC	72241M	71650M	72498M	74429M	1138066M	
PAPI_TOT_INS	137G	138G	140G	144G	2188G	
MFLOPS	1288.51	1314.18	1347.43	1388.99	20755.3	
VEC_MFLOPS	18.1299	18.021	17.8836	17.7503	290.319	

 interp :: wall time 27.2208 s

6.4 Analysis of whole computing

In the Table 9 it is shown that the load distribution of the whole computing is good. Despite of the fact that a part of computing only works with memory the computer performance is reasonably high.

Table 10: PAPI output, whole computing

	THREAD0	THREAD1	THREAD2	THREAD3	THREAD4	THREAD5
PAPI_FP_OPS	60205M	58002M	57104M	56410M	55905M	55566M
PAPI_DP_OPS	30961M	30963M	30965M	30961M	30958M	30959M
PAPI_TOT_CYC	126G	125G	122G	122G	121G	123G
MFLOPS	1297.52	1250.05	1230.7	1215.73	1204.85	1197.54
VEC_MFLOPS	614.731	614.76	614.807	614.733	614.664	614.69
	THREAD6	THREAD7	THREAD8	THREAD9	THREAD10	THREAD11
PAPI_FP_OPS	55359M	55270M	55271M	55362M	55569M	55918M
PAPI_DP_OPS	30966M	30958M	30960M	30966M	30965M	30958M
PAPI_TOT_CYC	123G	123G	123G	123G	121G	121G
MFLOPS	1193.08	1191.17	1191.18	1193.15	1197.6	1205.13
VEC_MFLOPS	614.833	614.669	614.706	614.828	614.81	614.673
	THREAD12	THREAD13	THREAD14	THREAD15	TOTAL	
PAPI_FP_OPS	56433M	57126M	58032M	51904M	899445M	
PAPI_DP_OPS	30959M	30963M	30951M	24753M	489173M	
PAPI_TOT_CYC	126G	123G	125G	117G	1972G	
MFLOPS	1216.24	1231.17	1250.68	1118.63	19384.4	
VEC_MFLOPS	614.688	614.771	614.523	491.469	9712.35	

```
-----
kpr :: wall time 46.4004 s
-----
```

Table 10 shows comparison between the Matlab solution and the new C++ solution. The C++ solution is up to twenty times faster than Matlab solution with 64x upsampled input data. Also the memory consumption is less up to around forty percent.

Table 11: Comparison of Matlab and C++ solution

MATLAB solution		
Input data scale	Computing time	Memory allocation
1x	6 s	2.7 GB
4x	68 s	20 GB
64x	885 s	125 GB
C++ solution		
Input data scale	Computing time	Memory allocation
1x	0.45 s	1.1 GB
4x	4.7 s	8.5 GB
64x	46.4 s	67.5 GB

In the Figure 15 is shown concurrency and scalability of threads(1, 2, 4, 8, 16) in computing for 64x 8x and 1x upsampled data. Time decrease almost linearly with more threads. The computing on sixteen cores is around eleven time faster than on one core.

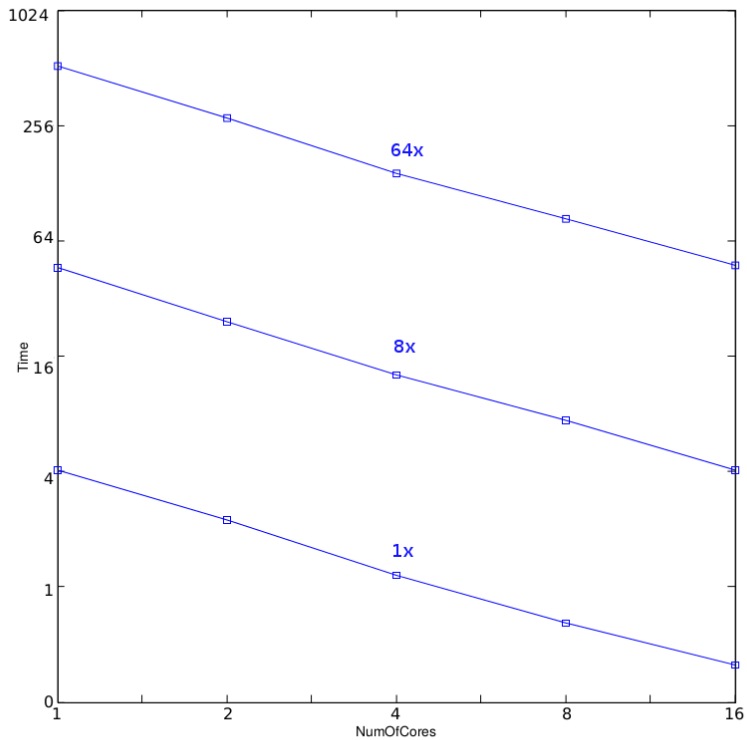


Figure 15 – Strong scaling

6.5 Analysis of whole program

This histogram in Figure 16 represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

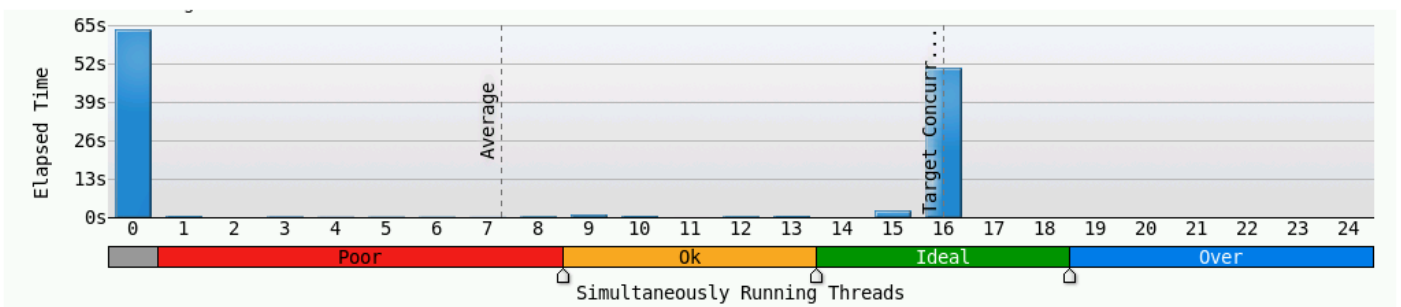


Figure 16: Thread concurrency histogram

This histogram in Figure 17 represents a breakdown of the Elapsed time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than thread concurrency if a thread is executing code on a CPU while it is logically waiting.

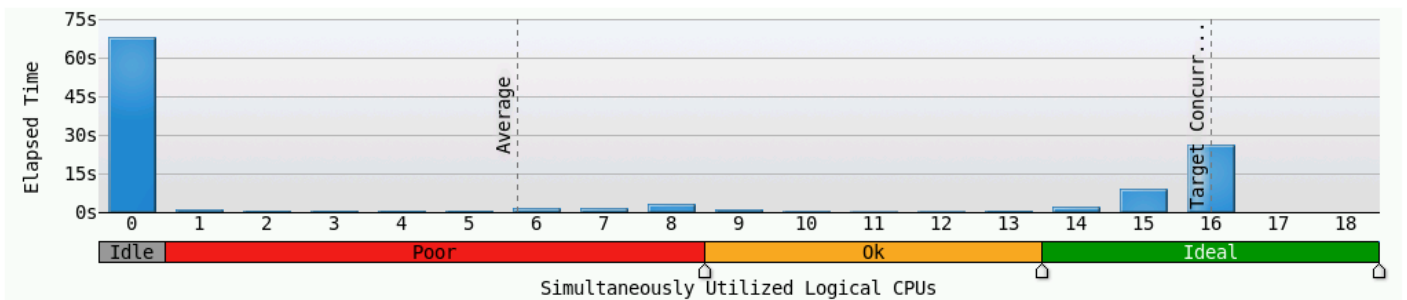


Figure 17: CPU usage histogram

The first column in Figure 16 and image Figure 17 is caused by the fact that this analysis is done to whole program not just computing. The `init()` function has to wait for I/O operations such as loading input arguments from hard drive. Program loading more than 30GB of input arguments which lasts around 60 seconds at HDD speed 500MB per second.

In the Figure 18 is shown histogram representing CPU time of each function and its thread concurrency. At the most time concurrency of threads is ideal.

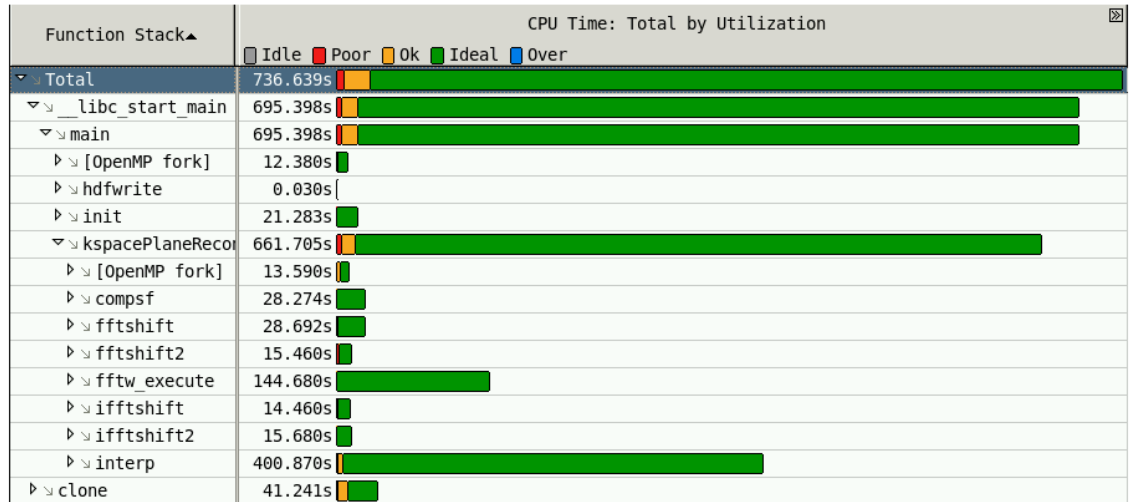


Figure 18: Thread concurrency top-down tree

In the Figure 19 is shown histogram representing wait time of each function. The functions waiting to I/O operations or to fork threads. The main computation does not wait which is ideal.

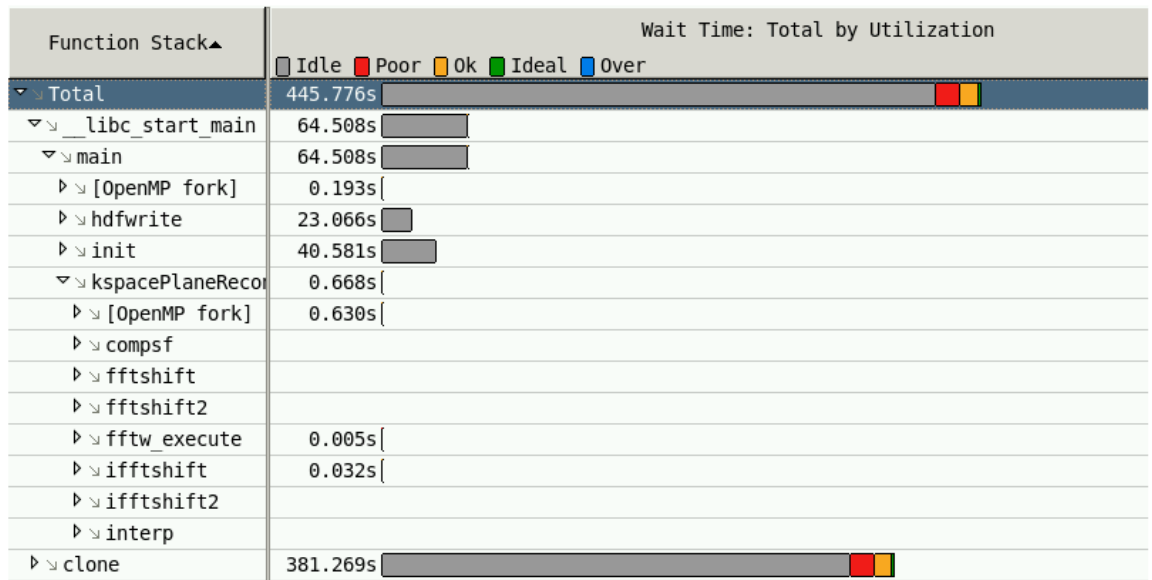


Figure 19: Waiting time of threads

7 Conclusion

This study has shown that photoacoustic imaging implemented in Matlab can be accelerated and optimised by using hardware-friendly code.

C++ solution can be up to twenty times faster than Matlab solution with forty percent less demand on main memory. The real acceleration depends on used input resolution, the acceleration is higher at higher resolution. It reaches seven to ten percent of theoretical performance of CPU, which can be improved by vectorisation other functions, but theoretical maximum of CPU performance can be achieved only by LINPACK not by the real program.

As mentioned the computing may be accelerated by vectorisation non vectorised functions, but reduce loading time of input arguments, which takes a lot of time will be more important. The loading time of input arguments from hard drive takes more time than whole reconstruction of photoacoustic images. Reduce the size of input arguments can be achieved by rewriting another part of Matlab solution and computing input arguments in C++ solution instead of loading them from hard drive.

The main contribution of this study is ability to obtain high resolution images of the vasculature or soft tissues many times faster than reference solution in Matlab. The work will be used as a part of international project k-Wave.

References

- [1] Laufer, J., Norris, F., Cleary, J., Zhang, E., Treeby, B., Cox, B., Johnson, P., Scambler, P., Lythgoe, M. and Beard, P.: *In vivo photoacoustic imaging of mouse embryos*. London, University College London, 2012.
- [2] Schoonover W. R., ANASTASIO A. M.: *Image reconstruction in photoacoustic tomography involving layered acoustic media*. National Center for Biotechnology Information, 2011.
- [3] Yao, J. and Wang, L. V.: *Photoacoustic microscopy*. *Laser & Photon. 2013. Rev.*, 7: 758–778. doi: 10.1002/lpor.201200060
- [4] Introduction. *Anselm cluster documentation* [online]. [ref. 2015-01-21]. Available from: <<https://docs.it4i.cz/anselm-cluster-documentation>>
- [5] Introduction. *FFTW* [online]. [ref. 2015-05-06]. Available from: <<http://www.fftw.org/>>
- [6] Muljadi, P. *Interpolation* [online]. [ref. 2015-05-06]. Available from: <<https://books.google.sk/books?id=PdT7PQPy83YC>>
- [7] Zhang, H. F.: *Functional photoacoustic microscopy for high-resolution and noninvasive in vivo imaging*. *Nature Biotechnology*, 2006.

Seznam příloh

Příloha 1. CD (Manál, zdrojové kódy, BP v elektronické podobě)