University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Optimization of Light Propagation Computation

Pilsen, 2013

Jan Kovanda

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan KOVANDA**
Osobní číslo:	**A11N0114P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Název tématu:	**Optimalizace výpočtu šíření světla**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování:

1. Prostudujte problematiku modelování šíření koherentního světla, zvláštní pozornost věnujte referenčnímu výpočtu popsanému v článku: P. Lobaz, „Reference calculation of light propagation between parallel planes of different sizes and sampling rates," Opt. Express 19, 32-39 (2011).
2. Navrhněte možnosti paralelizace výpočtu a odvoďte teoretická urychlení. Vybrané metody paralelizace implementujte a testujte.
3. Navrhněte metody pro hledání optimálního dělení výpočtu v prostředí s omezeným množstvím operační paměti, v potaz berte i paralelizaci výpočtu. Vybrané metody implementujte a testujte.

Statement

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, ... Jan Kovanda .....

Acknowledgements

I would like to thank to my thesis supervisor Ing. Petr Lobaz. I have him to thank for putting me into an interesting micro world of a digital and an optical holography. With a knowledge acquired this way I was able to make this work. Most importantly I would like to thank him for his calmness when dealing with me and a fact that whenever I had a problem I couldn't solve on my own, he always found time to help me.

I would like to thank my family that encouraged me when the work progress didn't go well.

Abstract

This document deals with an optimization process of a light propagation for use in a digital holography. At first a simple description of a light propagation is presented along with a problem caused by a lack of available computer memory needed for a propagation of a high quality hologram.

The search for fast Fourier transform library that meets our requirements follows. To support the choice of a right implementation, benchmarks aimed at computation speed and in parallel processing are provided. Afterwards the winning candidate is thoroughly tested in a way of transformation speed and time needed for planning of transformations.

With all the tests and the benchmarks done an optimization program is created. The program will try to find out the best settings for computation of the discrete Fourier transform on the basis of given input parameters and result of previous tests.

Keywords: optimization, light propagation, digital holography, computation speed, computer memory

Abstrakt

Tento dokument se zabývá optimalizací propagace světla pro použití v digitální holografii. Nejprve je jednoduše popsán proces propagace světla spolu s problémem, kterým je nedostatek volné paměti počítače potřebné pro propagaci hologramů vysoké kvality.

Následuje hledání takové knihovny rychlé Fourierovy transformace, která by vyhovovala našim požadavkům. Pro podporu volby správné implementace jsou poskytnuty srovnávací testy se zaměřením na rychlost výpočtu a paralelního zpracování dat. Poté je vítězný kandidát testován se zaměřením na rychlost transformací a času potřebného pro naplánování transformací.

Po ukončení všech srovnávacích testů je vytvořen optimalizační program, který se bude snažit zjistit na základě vstupních parametrů a výsledků předešlých testů nejlepší nastavení pro výpočet diskrétní Fourierovy transformace.

Klíčová slova: optimalizace, propagace světla, digitální holografie, rychlost výpočtu, počítačová paměť

Contents

1 Introduction

One of very common operations in computer generated holography is to calculate a propagation of a light between two parallel planes. A light propagation process has quite high requirements for a computer hardware. To create a digital hologram of an everyday object the data needed to recreate it in a form of a hologram can easily exceed units of terabytes. Those quantities of data cannot be processed in today's computer memory at once. Therefore there is a need for division of the data so they can be processed. To process such divided data we need to apply an algorithm that is able to process the divided data as mutually independent data sources.

As seen in figure 1.1 the light from every source point light is propagated in all directions. We are interested in the amplitude and the phase of the light hitting every particular sample of target (a sensor). Light changes its amplitude A and phase ϕ when travelling in space. In particular, a point light S of source emits light described by the amplitude and the phase. This light illuminates sample T of target – its amplitude changes due to propagation to $\alpha \times A$, its phase to $\phi + \Delta\phi$. Values of α and $\Delta\phi$ depend on the mutual position of point light S and target sample T.

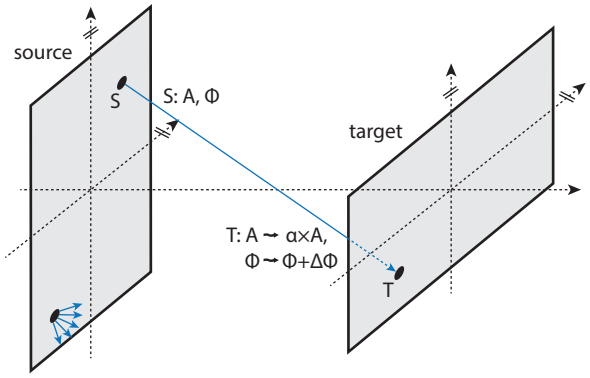


Figure 1.1: Propagation of a light from source to target

We will calculate a light propagation between two rectangular parallel areas, source (e.g. a spatial light modulator (SLM)) and target (e.g. a camera sensor). We use linear optics where it is assumed that light sources do not influence each other, share the same frequency (wavelength) and every single point $t(p,q)$ of target is affected by the light of all points $s(m,n)$ of source. This task can be solved with Rayleigh-Sommerfeld integral or other various approximations.

As already stated the light changes both amplitude A and phase ϕ by propagation. This change can by described by multiplication with a convolution kernel, i.e. 2-D array K_c. Light propagation is space invariant so only mutual position (and not their absolute positions) of points on source and target matters. We can discretize source and target by equidistant splitting into $M \times N$ and $P \times Q$ basic elements $s[m \times n]$ $(0 \leq m \leq M-1;\ 0 \leq n \leq N-1)$ and $t[p \times q]$ $(0 \leq p \leq P-1;\ 0 \leq q \leq Q-1)$ for a numerical calculation. By such splitting source s is discretely uniformly sampled into monochromatic point light sources.

Figure 1.2 describes such set. Source is divided into $M \times N$ basic elements, target into $P \times Q$ basic elements. Source plane $\xi\eta$ is parallel with target plane xy meaning that areas can only move in parallel with one another and in the z axis direction (change of mutual area distance).

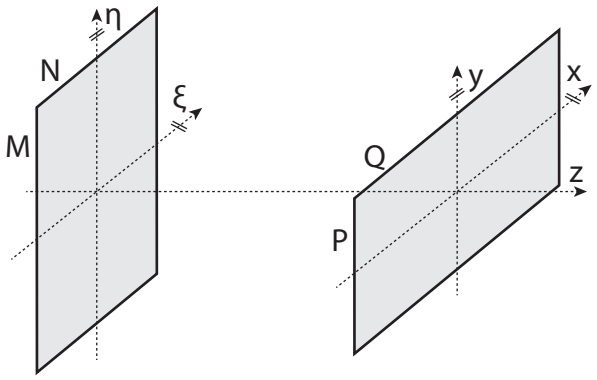


Figure 1.2: Position of source and target

The calculation of all elements $t[p,q]$ is most often done as a cyclic convolution and subsequent use of the discrete Fourier transform [Lob12]. The cyclic convolution has a form:

$$t[p,q] = -\frac{1}{2\pi} \sum_{m=0}^{2M-2} \sum_{n=0}^{2N-2} s[m,n] \times K_c[p-m(\text{mod } 2M-1), q-n(\text{mod } 2N-1)] \qquad (1)$$

where $s[m,n]$=0 for $M \leq m \leq M+P-1$ and $N \leq n \leq N+Q-1$ (i.e., the s is zero-padded to the size K_c). The important values of $t[m,n]$ are those for $0 \leq p \leq (M+P-1)-M$, $0 \leq q \leq (N+Q-1)-N$. The others are damaged by the cyclic behaviour of indices in arithmetic (mod $2N-1$ and mod $2M-1$). We can speed-up the computation significantly, because

$$t = -\frac{1}{2\pi}\mathbf{IDFT}(\mathbf{DFT}(s) \odot \mathbf{DFT}(K_c)) \qquad (2)$$

where t, s and K_c are two dimensional matrices of complex numbers, **DFT** is the discrete Fourier transform of a matrix, **IDFT** is the inverse discrete Fourier transform of a matrix and $\odot$ is the Hadamard product (element-by-element multiplication). If we would use a naive approach and computed the propagation by brute force, the algorithmic complexity would be O(N^4), if we assume both the source and the target to be discretized by $N \times N$ samples. The speed-up is expected due to the fact that the calculation of 2-D **DFT** or **IDFT** can be done in time $O(N^2 \log_2 N)$.

We will create a graphical representation of equation (2). In figure 1.3 we can see a discretized input data; source $M \times N$ and six times larger target $P \times Q$.

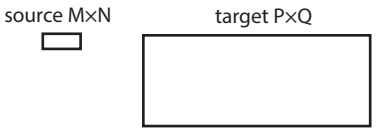
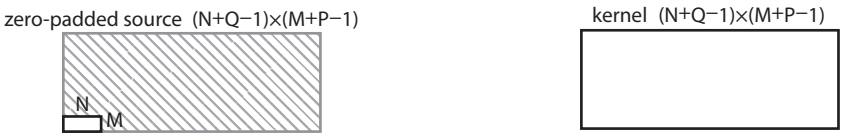


Figure 1.3: Graphical representation of input data source and required target

To compute a light propagation we need two memory spaces of sizes $(M+P-1)\times(N+Q-1)$ (see figure 1.4). Zero padded source is placed in the first space (fig. 1.4a). After the creation of the memory space the source data are copied into it. In our example the data are copied into left bottom corner. The rest of the memory space is filled with zeros (represented by grey hatches). The second memory space is for convolution kernel (fig. 1.4b).



(a) The source copied into first memory space

(b) The convolution kernel computed into second memory space

Figure 1.4: Two filled memory spaces $(M+P-1)\times(N+Q-1)$

After performing in-place **DFT**s of both memory spaces, results of element-by-element multiplication of both memory areas are saved into one of the areas, effectively replacing original content (fig. 1.5). In this case I chose the source data to be replaced.

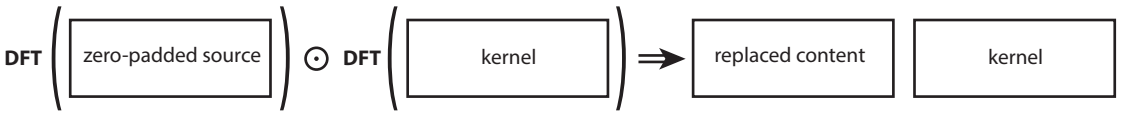


Figure 1.5: Transformation of the source and the kernel into new data that replace the source

Finally the **IDFT** is applied on the replaced content resulting in finished propagation (fig. 1.6). From whole replaced area we can only use area $P\times Q$ because those are the target data. Rest of the replaced area are unusable data damaged by cyclic behaviour (represented by grey hatches).

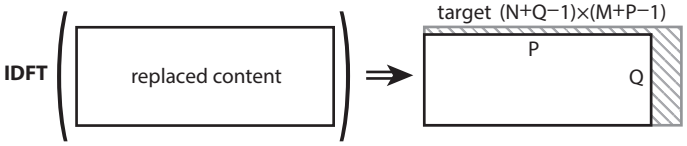


Figure 1.6: Application of **IDFT** and digestion of the target

We can see from equation (1) that the matrices s and t have to be padded to size K_c. Generally and as seen in examples 1.4a and 1.6 this padding means that for $M\times N$ and $P\times Q$, as much as 75% of elements are held in memory uselessly. Therefore, for big s and t we can expect a lack of usable memory shortly, especially when using a GPU for DFT calculation. For example a propagation of a microscopic source to extended target (detector) could require terabytes of data.

Two memory spaces of sizes $\geq (M+P-1)\times(N+Q-1)\times C_n$ where C_n is one complex number taking up 16 bytes of memory, are needed for the calculation of equation (2). Although in practice such two spaces may not be available. I will explain such situation by a real life example. Let us have a high quality hologram generated by a computer. The hologram is made of 50 000×50 000 samples (a source M×N) which corresponds to 50×50 mm. We are simulating its light propagation to a surface of an eye lens expressed in 5000×5000 samples (a target P×Q) which corresponds to 5×5 mm. To compute such propagation we need

$$(M+P-1)\times(N+Q-1)\times 2\times C_n = 54\,999\times 54\,999\times 2\times 16 \approx 90\,\text{GB}$$

of available memory. Allocation of such large quantity of memory is impractical. But we could split source into 10×10 parts (calling them "common tiles"), resulting into 5000×5000 samples (a source M'×N') and make 10×10 calculations (figure 1.7). For those calculations, we need only

$$(M+P-1)\times(N+Q-1)\times 2\times C_n = 9999\times 9999\times 2\times 16 \approx 3\,\text{GB}$$

of available memory. The same idea would apply with switched dimensions of source and target. By generalization we can accept that source can be split into S parts, target into T parts and then we have to calculate $S\times T$ propagations.

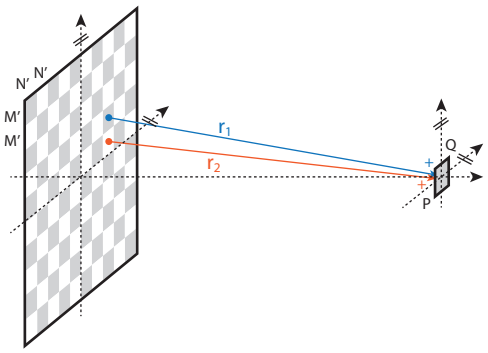


Figure 1.7: Division of source into 10×10 common tiles

The goal of this document is **finding** the right number and shape of **divisions** of source and target that will result in the fastest FFT computation times. The suggested technique focuses on a light propagation between two rectangular areas with various independent sizes. We need to find the minimum of a function with five parameters in the least practical time. We are given dimensions of a discretized source as two integers, a discretized target as two integers and a maximal allowed computer memory, that can be used in the process, as one integer.

Through this work I will test research results of Mr. Nedved [Ned12] who solved the same problem of finding the right division of source and target that will result in fast FFT computation times. He begins his work with a mathematical theory about optimal shape of source and target for the FFT. Outcome of the theory is that input of the FFT should have dimensions of $2^{\mathtt{M}}\times 2^{\mathtt{N}}$ where $\mathtt{M}, \mathtt{N} \in \mathbb{N}$. After that a process of optimizing a division of source and target into

smaller parts begins. In figure 1.8 we can se an area $M \times N$ divided into smaller parts $P_M \times P_N$. Lengths P_M and P_N are chosen so that a propagation of such part is the fastest and can be different for source and target.

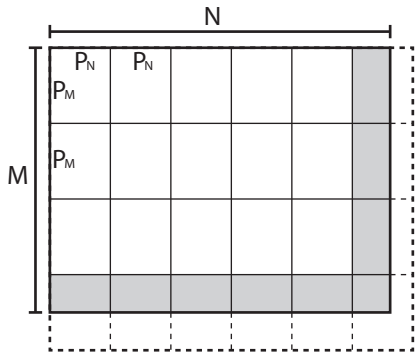


Figure 1.8: Memory division of a source/target area

Mr. Nedved deduced two findings from dividing, both seen in figure 1.8. First is processing of non-$P_M \times P_N$ parts located in right and bottom of the figure and marked in grey. An input source/target area can have any dimensions and it's not always possible to divide the area that there are no remainders. Ideally, we would like to optimize those reminding areas so that their light propagation would be as fast as a propagation of area $P_M \times P_N$. The easiest way to do so is by using properties of the cyclic convolution and complete the reminding areas with zeros up to area $P_M \times P_N$. Those expanded areas are implied by dashed lines.

Now Mr. Nedved needed to find out if he should either compute a propagation with reminding areas which take less memory but the computation for those dimensions are not optimized for speed, or with expanded areas which take more memory and are larger, but time of computation is optimized. After running some tests Mr. Nedved concluded that expanding areas *doesn't have noticeable negative impact* on computation speed. That means that it's not necessary to optimize reminding areas because we can expand areas.

Second is determination of the most efficient ratio of lengths P_M and P_N. Mr. Nedved assumes that various $P_M \times P_N$ parts will have different FFT computation speed. Those properties shall be measured with tests. If the differences will occur, it will be necessary to find lengths with faster computation speeds. If there won't be any differences in computation speeds, no ratio determination will be necessary. The ratio 1:1 would mean that both sides of a part are equally long and parts are square. Deviation from the ratio changes an area shape to rectangle. The more diversified the ratio, the more is an area rectangular.

After several tests Mr. Nedved determined that compared to optimization of reminding areas the effect of the ratio is *important*. In all cases where at least one part's side is considerably asymmetrical the FFT computation times are much longer than with regularly square or nearly square parts.

In the end Mr. Nedved writes that he succeeded in devising and implementing good enough solution of choosing input parameters for the problem of light propagation when only restricted amount of a computer memory is available. The confirmation of his theoretical reasonings were confirmed by test results.

The reason for existence of this work is the fact that the assumptions used in Mr. Nedved's research are not complete. For example he didn't consider all possibilities when choosing dimensions $2^{M} \times 2^{N}$ where $M, N \in \mathbb{N}$ of FFT input. Various FFT libraries have beside 2^{N} optimized other prime numbers such as 3, 5 or 7. Other fact is that I didn't want to base my conclusion on Mr. Nedved's dubious test results. The reason is that no measured values shown in resulting figures were provisioned and graphical representation of the measured values was overly simplified. As the last reason I would mention a lack of test results for very large matrices with areas up to several GB. Mr. Nedved only tested matrices with sizes up to 2^{16}, which is not enough for our needs.

In the section 2 we will write about an importance of the FFT and select acceptable candidates of FFT implementations that will fulfil our selected requirements. The final candidates are then tested in various benchmarks for their FFT computation speeds. After that the winning candidate is selected and its properties described. Section 3 continues with testing of the winning candidate. First is finding out the winner's behaviour for 1-D and 2-D data FFTs followed by measuring computation durations of rectangular areas of various sizes. Next test set is aimed at various winner's implementation settings and how their change manifest in results. Finally in section 4 I will describe my program that will for given input data determine divided "common tiles" dimensions of source and target. The determined dimensions will be "FFT friendly", meaning that the computation of the FFT should be optimized for speed.

2 FFTW characteristics

A computation of the fast Fourier transform (FFT) is an important element in the process of a light propagation. A propagation computation uses the FFT heavily and therefore trying to optimize this usage can be very fruitful. In this chapter the choice of the most suitable library with given parameters that computes the FFT is described. At first the library requirements are selected and reasoned. After that suitable library candidates are located and mutually tested for speed of FFT calculations. The library with the best tests results and the most favourable parameters wins the selection. At last the behaviour of the winning library is characterized.

The Fourier transform converts time or space to frequency and vice versa. The fast Fourier transform is an algorithm that rapidly computes such transformations. Many implementations of the FFT exist and we need to choose the one that will fulfil our requirements:

- **Computation speed** is a critical parameter because the propagation algorithm uses most of its computation time by calculating the FFT. Even small time improvements can create considerable speed-up.
- The implementation has to be able to compute *Two dimensional FFT transformations with complex numbers* which is a natural way to compute a light propagation.
- The light propagation algorithm itself supports *in parallel transforms* because source-target propagations are independent. From this fact we can choose two types of parallel operations:
  - By having the *ability to run several FFTs at once (in parallel)* we can handle the propagation code ourself. This approach has one disadvantage. If we were to run two FFTs parallelly each would need its own portion of a free computer memory to operate.
  - Or we can use *parallel (multi-threaded) FFT implementation*. One propagation would use given processing power to compute one FFT operation in parallel. To use this method a library should be able to support a parallel processing.
- The library's implementation has to be *multi-platform (C/C++ compatible)* because of already mentioned speed requirement. It should run on multiple operation systems, mainly Microsoft Windows and GNU/Linux.
- For propagations of large dimensions it's advantageous to have an option to choose from *non-power-of-two transform sizes* as well. Firstly, we are limited by provided computer memory and only power-of-two transform sizes grow very fast in higher indexes which restricts us. Secondly, a real FFT implementation and a computer hardware decreases the library's performance with large dimensions.
- Additionally the library has to have some sort of a *support*. A forum where problems can be posted and solved by a staff and an usable documentation. A big plus is an active development of the library. Generally libraries older than ten years and with inactive development are going to be disqualified.
- The library ought to be *free for non-commercial use*. These days many free implementations exist. If none free would suffice, libraries with different licensing would be found.

2.1 Selection of candidates

With requirements in place I needed to sort through implementations of the FFT libraries. I have found and used a comprehensive list [fft06] of very old and new FFT implementations along with a few others from [Rod09] and [Bla12]. In the following list the *disqualified implementations* are written along with an appropriate reason:

- **Non C/C++:** JTransforms
- **Non GNU/Linux, Windows:** Apple vDSP, MatrixFFT
- **Only 2^n transforms:** ffmpeg FFT, Ooura, FFTS
- **Only real numbers:** FFTReal
- **Old release:** FFTs for RISC 2.0 (1998), SciMark 2.0 (2000)
- **Specialization – GPU:** APPML-FFT, CUFFT, Nukada FFT library, OpenCL FFT
- **Specialization – 3D transforms:** OpenMM

- *Other:* GNU Scientific Library (GSL) – "For large-scale FFT work we recommend the use of the dedicated FFTW library by Frigo and Johnson" [gnu11]
- *Other:* Sparse Fast Fourier Transform (SFFT) [Rev12] – By using information from [Jas12] I came to conclusion that algorithms used in this library are not suitable for a light propagation
- *Other:* Spiral – Free version supports generation of only an 1-D array sizes up to 2^{15} in single precision [Pro12]

After the selection a following list of candidates remained:

- A FFT Package (FFTE) (Open Source, Fortran)
- AMD Core Math Library (ACML) (Free, Fortran)
- Fastest Fourier Transform in the West (FFTW) (GNU GPL, C)
- FXT (GNU GPL, C++)
- Intel Math Kernel Library (Intel MKL FFT) (Commercial – Royalty-free, C, C++)
- Intel Integrated Performance Primitives (Intel IPP) (Commercial – Royalty-free, C, C++)
- Jean-Marie Teuler FFT (JMFFT) (GNU GPL, Fortran90)
- Kiss FFT (BSD, C)

2.2 Testing of candidates – benchFFT

Those candidates should be tested for the last requirement, speed. The easiest way is to run benchmark(s) and compare results. Fortunately, the FFTW site hosts comprehensive benchmarks together with released source codes of tests and graphical representation of tests results.

The figure 2.1 [fft07c] shows data throughput during computation of 2-D data matrix, a typical operation for a light propagation. The horizontal axis represents lengths of a rectangle sides of matrices. The vertical axis represents speed of the computation in mflops. The benchFFT

[fft07a], a program to benchmark FFT software, describes this unit as

$$\text{mflops} = \frac{5N\log_2(N)}{\text{time for one FFT in microseconds}} \tag{3}$$

We see that "mflops" is a scaled version of the speed, where N is number of data points (the product of the FFT dimensions). For better comparison we would like to use real measure times, preferably µs or ms. With this knowledge I will transform the equation (3) into

$$\text{time for one FFT [µs]} = \frac{5N\log_2(N)}{\text{mflops}} \tag{4}$$

The result of the benchmark is shown in figure 2.1. The interpretation of the graph is as follows. Let us for example read values for 8×8 matrix for libraries fftw3 (in place calculation) and ooura-4f2d. They are 2 250 mflops for fftw3 in-place and 1 250 for ooura-4f2d. After substitution to the equation I have

$$\text{fftw3 in-place} : \frac{5 \times 8 \times 8 \times \log_2(8 \times 8)}{2\,250} \doteq 0.853 \text{ µs}$$

$$\text{ooura-4f2d} : \frac{5 \times 8 \times 8 \times \log_2(8 \times 8)}{1\,250} = 1.536 \text{ µs}$$

Therefore I can say that fftw3 in-place is approximately 2× faster than ooura-4f2d in computation of the FFT for matrix with dimensions of 8×8.

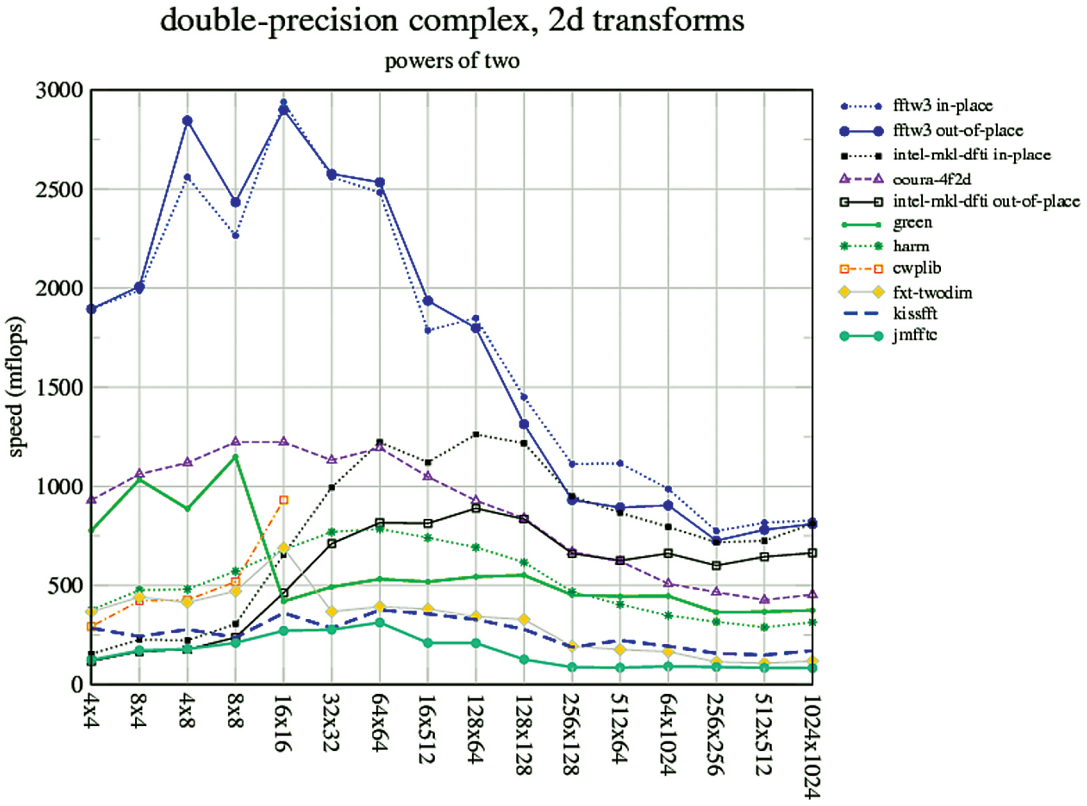


Figure 2.1: benchFFT, 2-D transformations benchmark of rectangular matrices. Test parameters: gcc-4.0.2, g++-4.0.2, gfortran-4.0.2, 2.4 GHz Intel Pentium 4 Single Core, 512 KB L2 cache. Linux 2.6.7, Intel Math Kernel Library Version 8.0.1

From figures 2.1 and 2.2 [fft07b] we can see that FFTW libraries have noticeably faster transforms for matrices with rectangle sides lengths up to 64×64. However in a light propagation we need to compute much larger values. Useful areas begin at about 512×512. After those sizes differences in speeds of contestants are greatly reduced.

From figure 2.2 it **seems** that computing fftw3 in-place library with dimensions 32×32 (10 850 mflops) is *only* ≈ 4× faster than the same library with dimensions 1024×1024 (2 400 mflops). Therefore I will convert speed results from mflops to ms.

$$\text{fftw3 in-place } 32 : \frac{5 \times 32 \times 32 \times \log_2(32 \times 32)}{10\,850} \doteq 4.719\ \mu\text{s} \tag{5}$$

$$\text{fftw3 in-place } 1\,024 : \frac{5 \times 1\,024 \times 1\,024 \times \log_2(1\,024 \times 1\,024)}{2\,400} \doteq 43\,691\ \mu\text{s} \tag{6}$$

The results of equations (5, 6) are much more believable. Fftw3 in-place 32×32 is ≐ 9 259× faster than fftw3 in-place 1024×1024. It follows that one can easily compare just performance of various libraries for a particular transform size. To compare various transform sizes, it is necessary to perform mflops to µs transform in a way we have shown.

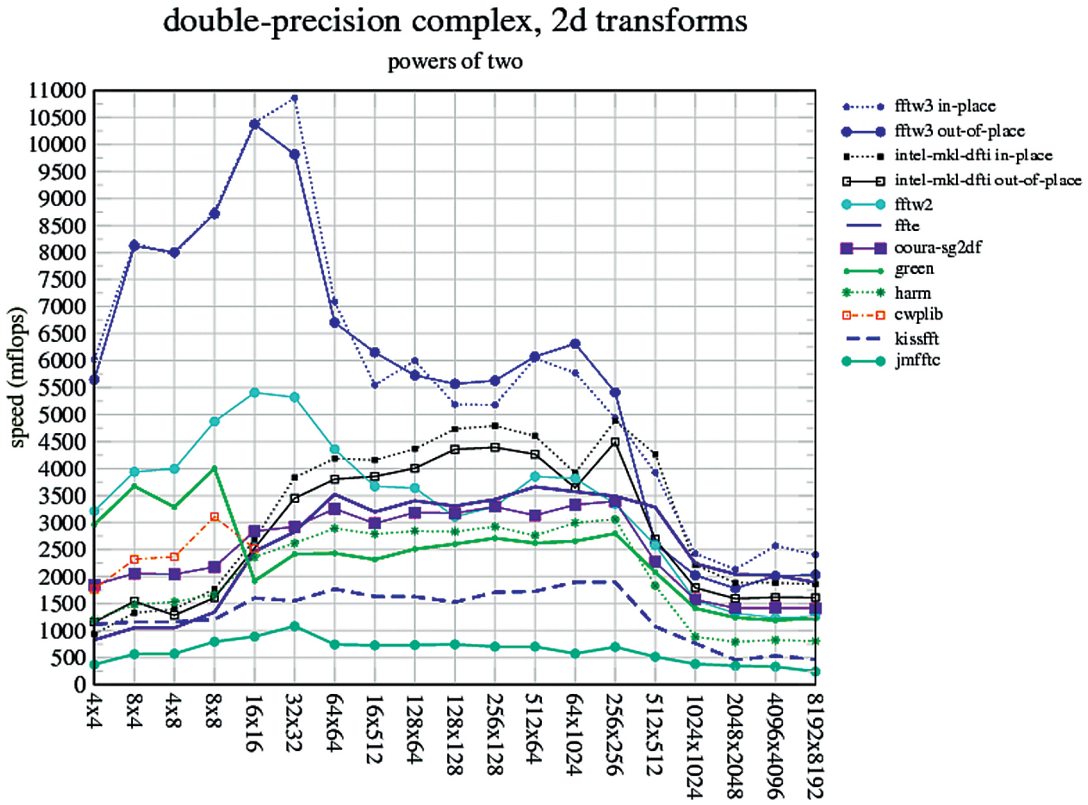


Figure 2.2: benchFFT, 2-D transformations benchmark of rectangular matrices. Test parameters: Intel C/C++ Compiler 9.1.043, Intel Fortran Compiler 9.1.037, 3.0 GHz Intel Xeon Core Duo, 4MB L2 cache, 64-bit mode. Linux 2.6.17, Intel Math Kernel Library Version 8.1.1

In figure 2.3 [fft07b] sizes with lengths that are not powers of two are used. The range of rectangular dimensions between 1 000 and 10 000 shows that all candidates have good speeds even for non-power-of-two values. I will explain how to read the graph for the values of the library Intel MKL FFT in-place. Computing the FFT of a matrix with dimensions 1 000×1 000 has speed of 2 750 mflops. A matrix 10 368×10 368 has speed of 1 700 mflops for the same library. It is necessary to confirm those result in real time:

$$\text{MKL FFT } 1\,000 : \frac{5 \times 1\,000 \times 1\,000 \times \log_2(1\,000 \times 1\,000)}{2\,750} \doteq 36\,239\ \mu\text{s} \tag{7}$$

$$\text{MKL FFT } 10\,368 : \frac{5 \times 10\,368 \times 10\,368 \times \log_2(10\,368 \times 10\,368)}{1\,700} \doteq 8\,436\,717\ \mu\text{s} \tag{8}$$

The results of equations (7, 8) have huge differences. Intel MKL FFT 1 000×1 000 is in round figures 233× faster than Intel MKL FFT 10 368×10 368.

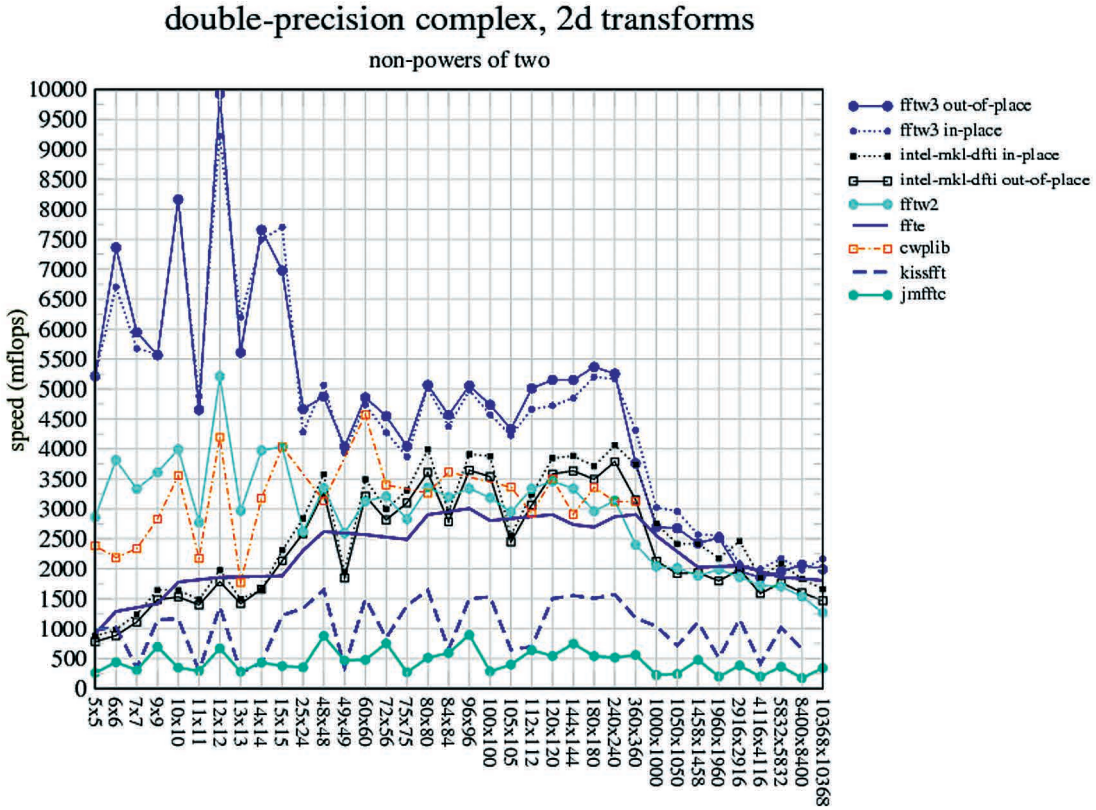


Figure 2.3: benchFFT, 2-D transformations benchmark of square matrices. Test parameters: Intel C/C++ Compiler 9.1.043, Intel Fortran Compiler 9.1.037, 3.0 GHz Intel Xeon Core Duo, 4MB L2 cache, 64-bit mode. Linux 2.6.17, Intel Math Kernel Library Version 8.1.1

Through our course of the benchmarking we could see that computation times of FFTs quickly increases with larger matrices. It would be well to check how much distinct are recorded FFT speeds from theoretical speed of the FFT. For an 1-D array a computation of the FFT we need

$O(N \log_2(N))$ arithmetical operations where N is number of elements of the array. In case of a 2-D matrix, computing the FFT means calculating an 1-D FFT for each row and column. A mathematical representation of this statement is shown in following equation

$$\begin{aligned}\text{FFT operations} : \text{number of rows} \times \text{row length} \times \log_2(\text{row length}) + \\ \text{number of columns} \times \text{column height} \times \log_2(\text{column height})\end{aligned} \tag{9}$$

Figure 2.4 show such an example. I transformed speed results of fftw3 in-place library from figure 2.3 into µs using equation (4). Next I used the same matrix dimensions as in figure 2.3 and computed theoretical speed of the FFT using equation (9). To position the theoretical FFT line along with the computed one I had to divide the theoretical results by magic number 1000. The horizontal axis represents total area of matrices. The vertical axis represents duration of the FFT computation for given matrix area in real time milliseconds. From figure 2.4 we can see that measured values approximately follow $O(N^2 \log_2(N))$ complexity. I will read an example of measured values from figure 2.4. The FFT duration of the matrix with total area 100 is 0.4 µs. Selection of additional computed results are shown in table 2.1.

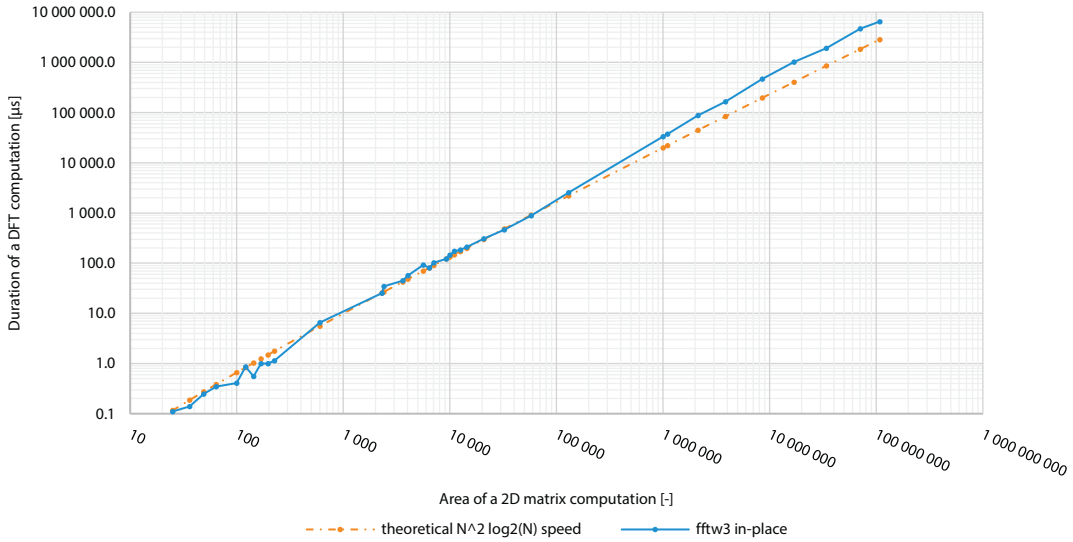


Figure 2.4: Deviation of the FFT computation speed of fftw3 in-place library in figure 2.3 from theoretical speed of $O(N^2 \log_2(N))$

Matrix dimensions	fftw3 in-place [mflops]	fftw3 in-place [µs]
10×10	8200	0.41
25×24	4260	6.50
96×96	5000	121.37
240×240	5150	884.34
5832×5832	2200	1 934 023
10368×10368	2200	6 518 058

Table 2.1: Selection of computed FFT speed results from figure 2.4

2.3 Testing of candidates – independent

Since the benchFFT program is maintained by same people as FFTW the results can be biased. Therefore some other independent test should be used to test the candidates.

The figure 2.5 shows test case used in benchFFT but ran at the computers of SGI-IZO by staff of Campus de Excelencia Internacional [Exc09]. Resulting graphical representation of this test is similar to the result shown in figure 2.1. Speeds of the contestants is generally higher because the hardware used for the test is faster than the one used in figure 2.1.

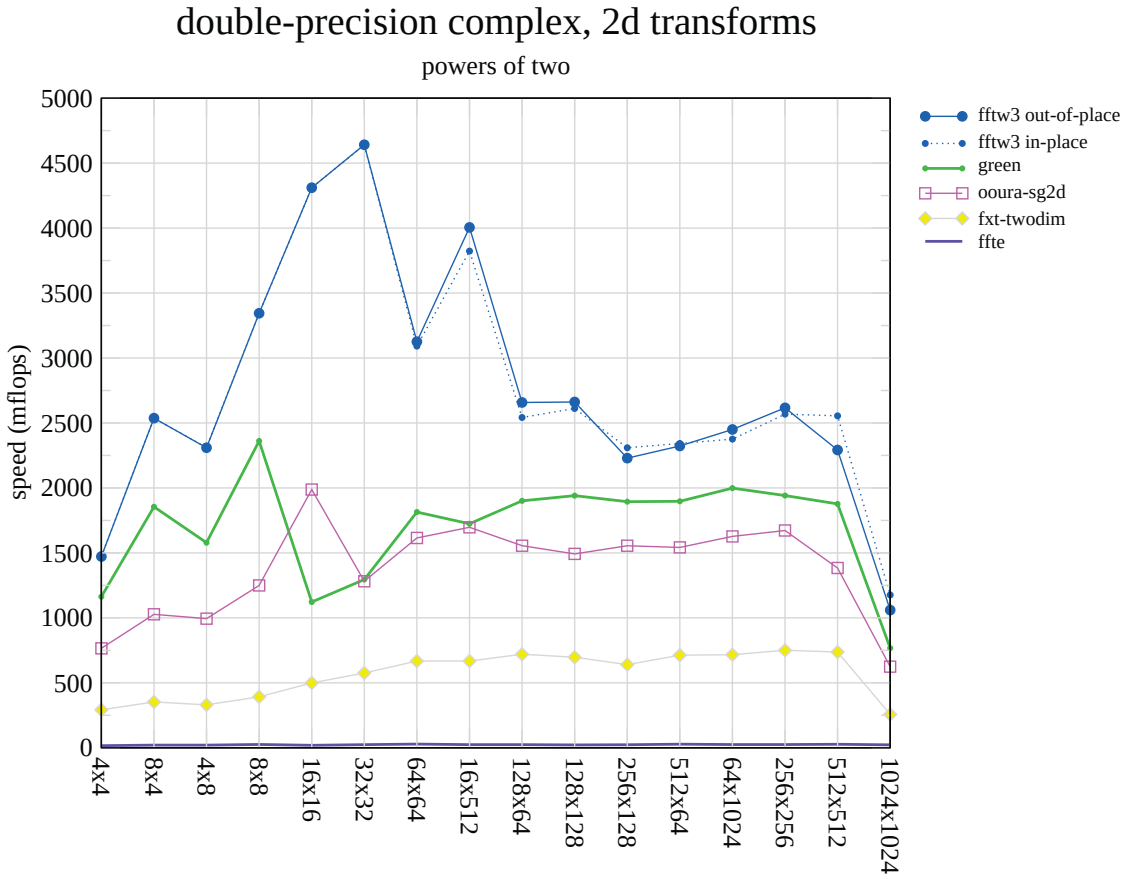


Figure 2.5: SGI-IZO, 2-D transformations benchmark of square matrices. Test parameters: Dual core Itanium2, CPU 1.6 Ghz, 18 MB L3 cache

Table 2.2 shows benchmarks for the transformation of 64×64 64-bit real arrays on an computer 833MHz Alpha EV6, obtained using benchFFT from the FFTW website in 2005. The timings in the table were reproduced at University of Tasmania [Phi06, p. 89] in mflops. I have transformed the times using formula

$$\text{time for one FFT } [\mu s] : \frac{2.5N\log_2(N)}{\text{mflops}} = \frac{2.5 \times 64 \times 64\log_2(64 \times 64)}{\text{mflops}} = \frac{122\,880}{\text{mflops}}$$

into µs. Resulting values show the superior performance of FFTW.

Package	Author / Vendor	Forward transforms [µs]	Backward transforms [µs]
FFTW 2.X	FFTW	74.5	77.72
CXML	Hewlett-Packard	123.56	130.89
Ooura FFTs	Takuya Ooura	128.03	128.03
FFTs for RISC 2.0	John Green	161.05	163.91
JMFFT	Jean-Marie Teuler	325.68	325.44

Table 2.2: FFTW benchmarks ran on the hardware on which Mk3L [Phi06] was developed, 2005

Last figure 2.6 from [Bau10] shows speed comparison of FFTW and Intel MKL libraries. The horizontal axis represents dimensions of a 2-D matrix ranging from matrices 8×8 to 8 192×8 192. The vertical axis plots result times of FFT computation but the representation of the axis is in "speedup" multiplier. With those units the figure can be read as follows. For example the difference between FFTW 8 cores and Intel MKL 8 cores for the matrix $2^3 \times 2^3$ has the speed-up of one, meaning Intel MKL 8 cores FFT was computed two times faster than FFTW 8 cores. Alternatively we can say that time needed for the FFT computation was halved. Another example is for matrix $2^9 \times 2^9$ with 8 cores of Intel MKL and FFTW. Mutual speed-up multiplier is 4 (5.25 − 1.25) meaning FFTW was five times faster than Intel MKL.

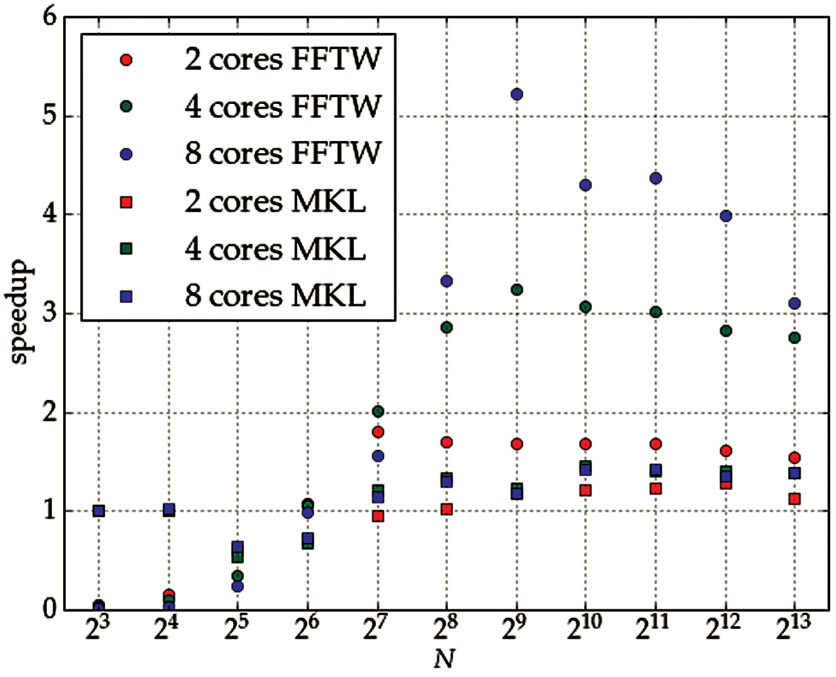


Figure 2.6: Parallel speed up of parallel FFT of four interwoven two-dimensional matrices of size N×N with FFTW library version 3.2.2 and Intel MKL library bundled with Intel compiler suite version 11.1.056. Test parameters: two Quad Core Intel Xeon CPUs (E5345) at 2.5GHz

From figure 2.6 can be concluded that

- the Intel MKL FFT routines show a very poor parallel speed-up. Even with eight cores, the speed-up never exceeds two, whereas FFTW library reaches a reasonable speed-up.
- FFTW library also excels Intel MKL in terms of absolute computing time.
- Only for very small matrices, Intel MKL is superior.

From results of benchmarks and other requested parameters the library fftw3 (Fastest Fourier Transform in the West version three) was chosen as best candidate for a light propagation.

2.4 Fastest Fourier Transform in the West

FFTW is an implementation of the discrete Fourier transform (DFT) that adapts to the hardware in order to maximize performance [FJ05]. The FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly. The only difference is that the FFT is faster. Important characteristic of this library is a way of creating an optimization *plan* for execution of the FFT. The planning algorithm generates plans according with rules that repeatedly simplify a problem into simpler sub-problems. When the problem becomes "sufficiently simple", FFTW produces a plan that uses generated optimal line of processing instructions that solves the problem directly. These fragments are called **codelets** in FFTW's jargon. For example, a codelet might be specialized to compute the DFT of real data (as opposed to complex). FFTW's speed depends therefore on two factors [FJ05]:

- The decomposition rules must produce a sufficient number of plans that is rich enough to contain "good" plans for most machines. The standard FFTW distribution contains a set of about 150 pre-generated codelets that cover most common uses.
- Codelets must be fast because they perform all the essential work.

Codelets are written in C language and the more is a codelet optimized the higher is an efficiency of the codelet. Creating a codelet is easy but optimizing it is very error-prone [FJ98]. Therefore we can use a special compiler for their generation. The generator is written in Caml Light (dialect of ML functional language). It is not desirable for me to write my own codelets, because the code is then not portable into other FFT implementations. Also codelets are mainly for specialised cases that happen often and not for my broad use of any input and output data.

It's necessary to describe how to use FFTW because different operational parameters can greatly change a characteristic of a computation. One-dimensional and multi-dimensional transforms [fft12b] work much the same way:

1. Allocate arrays of type `fftw_complex` (16 bytes, 8 for real and 8 for imaginary part) preferably using `fftw_malloc` (for an automatic data aligning),
2. create an `fftw_plan` (combination of codelets),

3. execute it arbitrarily with `fftw_execute(plan)` (launching the DFT),
4. and clean up with `fftw_destroy_plan(plan)` and `fftw_free` (cleaning memory).

All of the planner routines in FFTW accept an integer argument called flags (for our use of 2-D complex planning the command is `fftw_plan_dft_2d(ROWS,COLUMNS,…,FLAGS)`). These flags control the rigour (and time) of the planning process [fft12c]. A final computation time of the DFT can be greatly modified by the used flag. From all available flags I will try (and use) those four:

- `FFTW_ESTIMATE` specifies that, instead of actual measurements of different algorithms, a simple heuristic is used to pick a (probably sub-optimal) plan quickly.
- `FFTW_MEASURE` tells FFTW to find an optimized plan by actually computing several FFTs and measuring their execution time. Depending on a machine, this can take some time (often a few seconds).
- `FFTW_PATIENT` is like `FFTW_MEASURE`, but considers a wider range of algorithms and often produces a "more optimal" plan (especially for large transforms), but at the expense of several times longer planning time (especially for large transforms).
- `FFTW_EXHAUSTIVE` is like `FFTW_PATIENT`, but considers an even wider range of algorithms, including many that we think are unlikely to be fast, to produce the "most optimal" plan but with a substantially increased planning time.

3 FFTW testing

By running a set of benchmarks we have chosen the best suited FFT library candidate, Fastest Fourier Transform in the West. In this chapter we will further our testing of the library. At first the environment used for testing will be described. After that various tests aimed at diverse properties of FFTW will be run. Our objective is to learn the library's behaviour and draw conclusions that will help us increase an efficiency of the library.

In some test cases I will run a test several times and write only one set of result or average of results into this document. The whole computed data with all test results and figures are saved on the enclosed CD.

To fully test FFTW I will perform tests on various computers with different configurations:

1. Windows 7 Enterprise SP1 x64, Intel Core i5-2400S, x64 CPU 2.5 GHz, 4 Cores, 4 threads, 8 GB physical memory. This set will be used in absolute majority of tests.
2. Windows 7 Professional SP1 x64, Intel Core2 Duo, x64 CPU 2.26 GHz, 2 Cores, 2 threads, 4 GB physical memory. Used mainly for confirmation of measured results of configuration 1.
3. Windows XP Professional SP3 x86, AMD, x86 CPU 1.91 GHz, 1 Core, 1 thread, 1.5 GB physical memory. Used specifically for one test because HW settings in this set allowed the processor cache to be disabled.

Additionally to ensure that the tests are the least influenced by other programs and computer hardware parts, I modified all testing environments with following changes:

- Disabled disk swapping
  - \+ No I/O disk activity (stable test results, faster computation times mainly for larger data matrix)
  - − Maximal data sizes up to a computer physical memory
- Disabled the indexing, the superfetch, anti-virus systems, firewalls
- Disabled all Internet connections, physically unplugged computer from a network
- Closed all other running programs

As I ran an operation system Windows as my target platform I needed to measure running times of the FFT. I had to use two different counters described in table 3.1. The cycle-counter was used to count durations of very fast operations from range of μs to tens of seconds. After exceeding the upper time interval the counter overflowed. The discovery of this problem forced me to find an alternative, which is the second tool for measuring time, Query Performance Counter. QPC allows measuring time from ms to days.

FFTW cycle-counter (CC)	The high-precision timer provided by FFTW that uses hardware cycle counter available in modern CPUs. The cycle counter works on various processors and most common compilers. The elapsed time is in arbitrary units, not seconds or anything similar. When using measured values of this counter in figures' axes, I marked the times as *CPU cycles*. The only use for this counter is for performance comparison. I have tried to find out if there is any way to transform those cycles into time. On *one computer* and the *unchanging settings* (computer settings 1) I stably measured that $$2\,000\,000\,000 \text{ CPU cycles} \approx \text{one real time second} \quad (10)$$ Through this document I will be using conversion (10) to roughly compare speeds of various test results.
WINAPI Query Performance Counter (QPC)	The high-resolution performance counter provided by Microsoft. I have used this counter for measuring speeds from several milliseconds up to few real time hours. Because the resolution is system-dependent, there are no standard units that it measures. I had to divide the difference by the parameter Query Performance Frequency to determine the number of milliseconds elapsed.

Table 3.1: Counter types used in all tests

Unless stated otherwise all test have run in four threads with a FFTW plan created by the planner with flag `ESTIMATE`. Combination of four threads and flag `ESTIMATE` was chosen because measurement of FFT durations with such setting produced the fastest results from all options. Reason for using four threads is justified in figure 2.6 and verified in section 3.7. Comparison of different planner flags is tested in section 3.4.

3.1 Codelets influence in 1-D arrays

We already know that FFTW uses codelets to create a FFT plan and right choice of the codelets is a key element in FFT computation speed. So as a next step I wanted to know how much codelets influence the computation speed of the FFT. By testing FFT computations durations of an 1-D array with variable lengths, I expected the results to look similarly to figure 2.4 but with lower computation duration.

Figure 3.1 shows results of FFT computation with an 1-D array. The horizontal axis represents lengths of an array filled with random generated complex numbers, whereas the vertical axis stands for a number of CPU cycles needed to compute the FFT of the array with specific length. For example computing the FFT of an array length 101 took 10 400 CPU cycles.

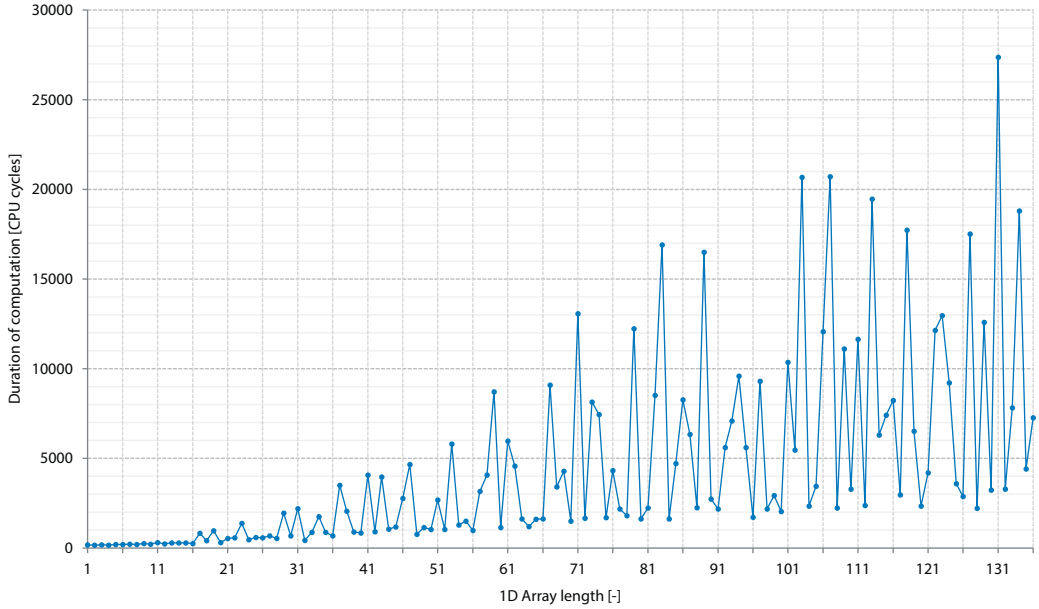


Figure 3.1: Selections of lengths where FFTW computes the FFT of an 1-D array filled with random complex numbers

We can see that unlike in figure 2.4 the relief of figure 3.1 resembles "mountains". After some search I found the reason for this behaviour in the FFTW documentation [fft12a]. The documentation states that arrays of sizes

$$2^a \times 3^b \times 5^c \times 7^d \times 11^e \times 13^f \tag{11}$$

where e+f is either 0 or 1, and the other exponents are arbitrary, will produce much faster transform results. I will call numbers created from the formula as **fftw-friendly**. By picking some array lengths from figure 3.1 I will test if the stated formula is right:

- **121** = 11^2
- **122** = no decomposition
- **123** = no decomposition
- **124** = no decomposition
- **125** = 5^3
- **126** = $2^1 \times 3^2 \times 7^1$
- **127** = no decomposition
- **128** = 2^7
- **129** = no decomposition
- **130** = $2^1 \times 5^1 \times 13^1$
- **131** = no decomposition
- **132** = $2^2 \times 3^1 \times 11^1$

When comparing the picked lengths with measured CPU cycles in figure 3.1 we see that the documentation was right. All chosen numbers that can be decompositioned have much shorter FFT computation durations. The time difference between numbers 131 and 132

is approximately *sevenfold*. Those tendencies continue analogously to times of higher lengths which can be see in figure 3.2.

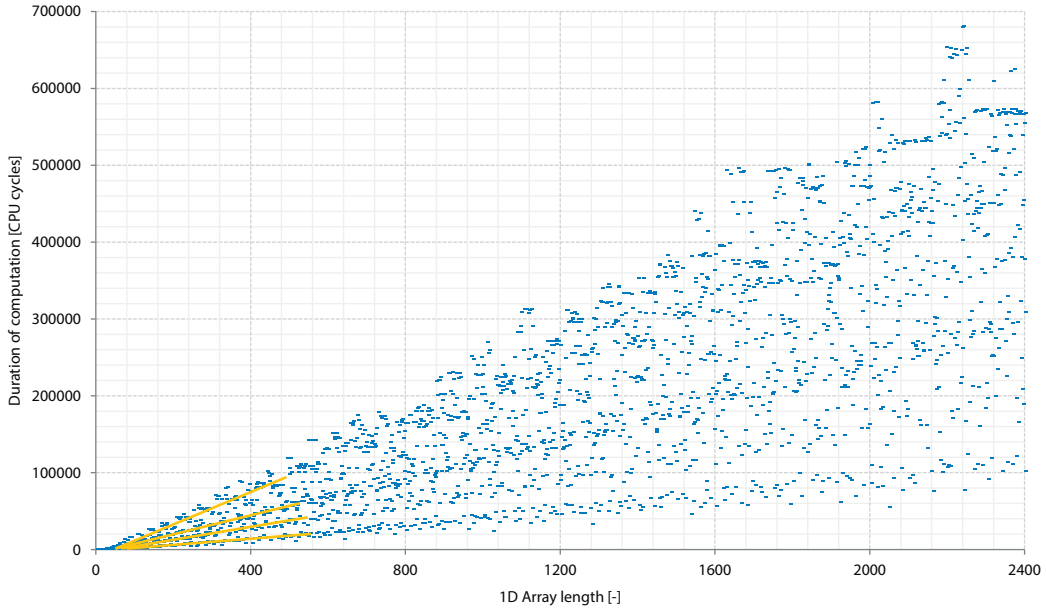


Figure 3.2: The full 1-D array filled with random complex numbers where FFTW computes the FFT

Figure 3.2 is the same as figure 3.1 but with longer 1-D array length interval. The durations of FFT computations in the figure are fanned out into four "streams" highlighted in yellow lines. The most important stream is the bottom one with the lowest cycle durations. All array lengths in the bottom stream meet the formula (11); i.e. decomposition of a number into prime numbers.

A last unexplained effect in figures 3.1 and 3.2 is very fast (seemingly linear) time needed for computing an array lengths 1–16. As written in [FJ98, ch. 2], special codelets, that handle integers in stated range, were specifically created for those array lengths.

We found out that not all numbers in FFTW are "equal". Certain numbers that we will call *fftw-friendly* have up to seven times faster computation speed. We will further focus on finding and using fftw-friendly values.

3.2 Codelets influence in 2-D matrices

Testing 1-D arrays proved useful but a light propagation uses 2-D data. I needed to know whether the result I measured in figure 3.1 for 1-D arrays will be similar for 2-D matrices. To find out such results I will compute the FFT of 2-D matrices with various dimensions. On basis of chapter 3.1 and the fact that a 2-D FFT is computed as set of an 1-D FFTs, I will be expecting the results to be as follows:

- Transforms of 2-D matrices with fftw-friendly sides lengths will be the fastest.
- Transforms of 2-D matrices with non-fftw-friendly sides lengths will be the slowest.

I ran approximately 11 000 FFT computations of 2-D matrices with various dimensions ranging from 1000×1000 to 1023×1500 and saved the FFT durations. Table 3.2 show excerpt from the saved durations.

M\N	**1000**	**1001**	**1002**	**1003**	**1004**	**1005**	**1006**	**1007**	**1008**	**1009**
1350	208	324	424	484	341	347	369	427	195	521
1351	298	424	514	593	431	433	459	531	281	621
1352	208	325	436	500	358	359	382	437	200	523
1353	329	465	559	647	475	488	513	576	322	666

Table 3.2: Duration of FFT computation of 2-D matrices measured in rounded CPU cycles×10^6. Column length M and row length N are horizontal and vertical lengths of a matrix

Computation times in table 3.2 varies greatly. For better idea I created figure 3.3 as a graphical representation of FFT computation from table 3.2. Matrices have ten column lengths (M in range from 1000 to 1009) and four row lengths (N in range from 1350 to 1353). I will use an example of reading the measured values from the figure. Duration of the FFT for the matrix 1000×1350 is 208×10^6 CPU cycles.

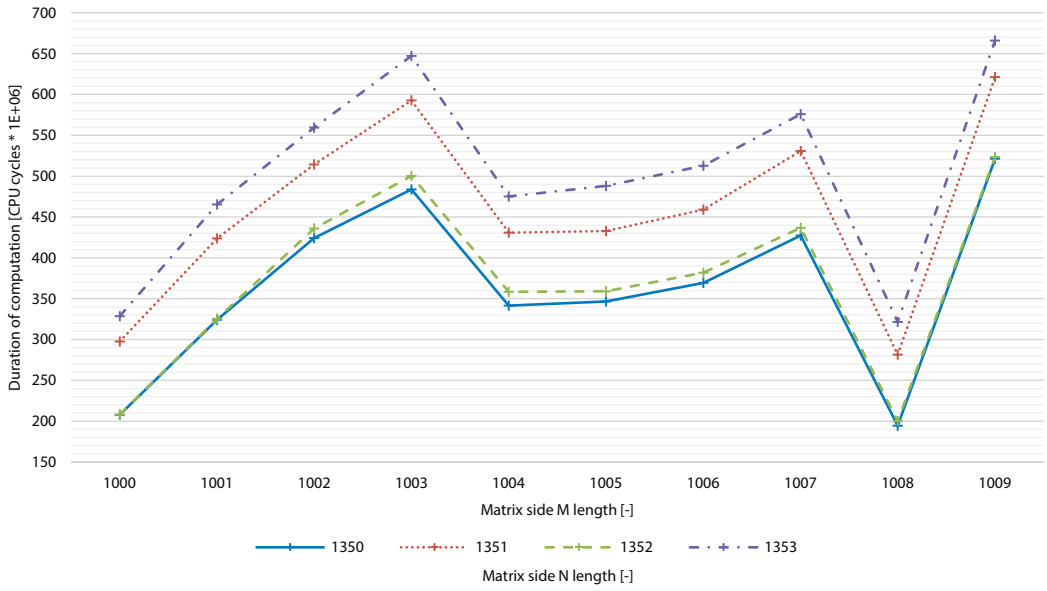


Figure 3.3: Time needed for FFTW to compute the FFT of matrix with complex numbers

First I will locate fftw-friendly numbers in table 3.2 and write them into a list:

- `N`: **1350** = $2^1 \times 3^3 \times 5^2$
- `M`: **1000** = $2^3 \times 5^3$
- `M`: **1008** = $2^4 \times 3^2 \times 7^1$

As we predicted, the fastest times in figure 3.3 are with combinations of fftw-friendly numbers. Particularly matrix 1350×1008 with 195×10^6 CPU cycles and 1350×1000 with 208×10^6 CPU cycles. Figure 3.3 revealed one additional characteristic of FFTW that we didn't know. Why has dimension 1352 very good computation times even when it's not a fftw-friendly number by equation (11)?

Equation (11) states that when an exponent e+f is either 0 or 1 a computation will be much faster. I have found this statement not entirely true. The dimension 1352 that can be decompositioned as $2^3 \times 13^2$ which doesn't fulfil the equation's conditions e+f 0 or 1, indicating a non-fftw-friendly number. But the results claim the opposite. In all `M` dimensions combinations the number 1352 is nearly on par with the legitimate fftw-friendly number 1350. Even more so when the `M` combination includes both fftw-friendly numbers. Because of this discovery I included all numbers with any e or f exponent value into a fftw-friendly category. With this change I will update the fftw-friendly list:

- `N`: **1350** = $2^1 \times 3^3 \times 5^2$
- `N`: **1352** = $2^3 \times 13^2$

- `M`: **1000** = $2^3 \times 5^3$
- `M`: **1001** = $7^1 \times 11^1 \times 13^1$
- `M`: **1008** = $2^4 \times 3^2 \times 7^1$

With those information and results of figure 3.3 we can conclude following outcome. Combination of:

- Two non-fftw-friendly values (e.g. 1353 and 1003) results in longest computation times.
- One non-fftw-friendly value and one fftw-friendly value (e.g. 1351 and 1000) results in medium-term computation times.
- Two fftw-friendly values (e.g. 1350 and 1008) results in shortest computation times.

Results of this test – fftw-friendly values fastest and non-fftw-friendly values slowest – corresponds with our expectations. Additionally we extended the fftw-friendly formula and gained additional fftw-friendly numbers. In following chapters we will be using only arrays and matrices of **fftw-friendly** lengths.

3.3 FFT computation speed diversity in one matrix area

I will begin new test set containing 21 test cases that will tell whether FFT computation speed of matrices with equivalent areas (e.g. 1×1024 (an area 1024) and 32×32 (same area, 1024)) are going to be similar or will differ greatly. Additionally if a speed differs, I will need to find reason(s). Results of those tests are necessary to find out whether it's advantageous to compute the FFT of squarish or either horizontal or vertical rectangular areas and in what ratio. The results measured up to now indicate that areas created by multiplying fftw-friendly values are going to be approximately similar in computation speeds.

The test set contains matrices ranging from areas 5 308 416 to 10 617 750. I used such interval because of sufficient amount of used memory (where 16 bytes stands for required memory space for one complex number)

- $5\,308\,416\times16$ bytes $=$ 81 MB
- $10\,617\,750\times16$ bytes $\doteq$ 162 MB

and simultaneously low enough that computations won't take unacceptable amount of time on the testing machines. Additionally all chosen areas are fftw-friendly. The reason for this is found out in section 3.2. I will list some of the test areas:

- **5 308 416** = $2^{16} \times 3^4$
- 6 054 048 = $2^5 \times 3^3 \times 7^2 \times 11^1 \times 13^1$
- **7 584 759** = $3^5 \times 7^4 \times 13^1$
- 8 000 000 = $2^9 \times 5^6$
- 9 528 750 = $2^1 \times 3^2 \times 5^4 \times 7^1 \times 11^2$

  …

I will describe just two areas highlighted in **bold** print. The other tests had similar results. In each test case I chose a test `AREA` from the list and created a matrix $\mathcal{A}$ with dimensions `M` $\times$ `N`, where `M` represents column lengths and `N` row lengths. The final matrix is created with conditions

- `(N := 1..AREA) && (AREA % N == 0)`
- `M := AREA / N`

Row length `N` is increased by one until a modulo with `AREA` returns zero. For example with `AREA` 35, row length `N` will sequentially gain values 1, 5, 7 and 35. For every `N` the appropriate `M` will be 35, 7, 5 and 1. Unsurprisingly, the values are in reversed order. A multiplication `M`$\times$`N` is therefore always equal to `AREA`. Table 3.3 shows every combination of column length `M` and row length `N` and respectively created matrices with area 1024.

M	1024	512	256	128	64	32	16	8	4	2	1
N	1	2	4	8	16	32	64	128	256	512	1024

Table 3.3: An interpretation of matrices with various side lengths `M`, `N`. A multiplication of column length `M` and row length `N` always returns 1024

The test with area 5 308 416 in figure 3.4 was run four times to confirm results from previous runs. The horizontal axis represents row length `N` of the matrix. To gain column length `M` it's necessary to divide `AREA` with row length `N`. The vertical axis represents duration of FFT computations for given matrix dimension.

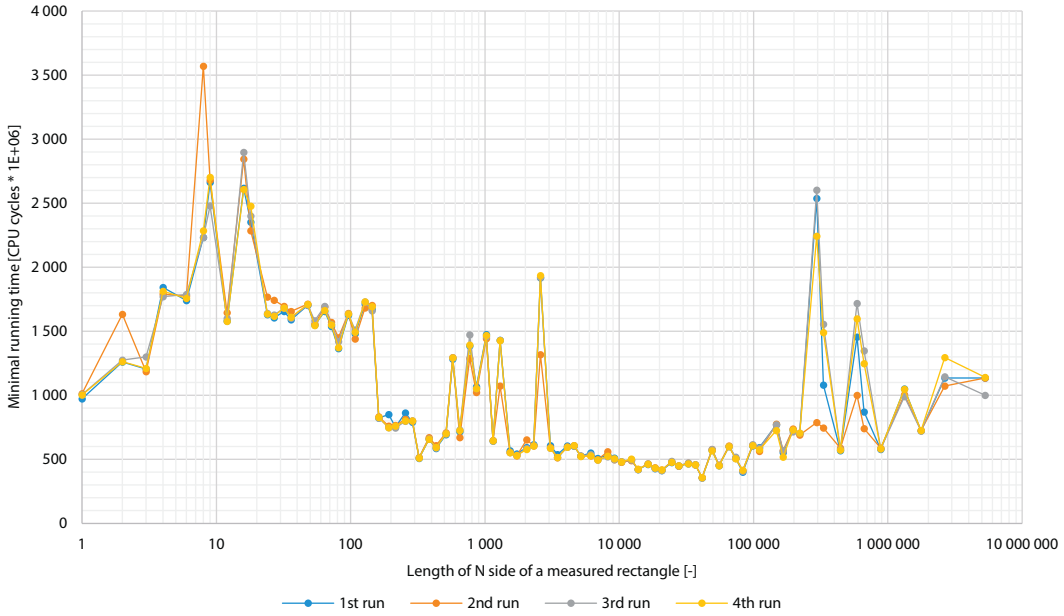


Figure 3.4: FFT speed test of a rectangular area 5 308 416

I will read an example of measured values. The matrix with row length `N` 2592 (the middle spike) has its other length

$$\text{column length } \texttt{M} := \frac{\texttt{AREA}}{\texttt{N}} = \frac{5\,308\,416}{2592} = 2048 \tag{12}$$

FFT computation duration of the matrix 2048×2592 was measured three in a four times as $\approx 1920 \times 10^6$ CPU cycles and single time as 1320×10^6 CPU cycles. The reason for this difference is not obvious. In my opinion it's caused by the planner that had chosen different set of codelets resulting in better computation time. However, we will accept the worse time because it's more probable that the planner will choose "slower" codelets set three in a four times.

The measured times were expected to be in central symmetry but as we can see in figure 3.4, they aren't. Reason for this is found in the FFTW documentation [fft12d]. It says that FFTW doesn't work similarly with matrix's rows and columns. Its algorithms expect array to be stored

as a single contiguous block in *row-major order*. The matrix $\mathcal{A}$ should have very **long rows** (large row length `N`) and have only **a few of them** (small column length `M`). That is a reason why row lengths `N` from 1 to 2592 have such slow computation times. Other half of the graph from row length `N` 3000 has much lower and more regular computation times. The lowest measured CPU cycles in figure 3.4 are located roughly between row lengths `N` 3000 and 100 000.

Figure 3.4 shows one additional anomaly. At the row length `N` $\sim$ 300 000 the computation duration is deviating from other measured times by a great margin. The area was made from a multiplication of column length `M` 18 and row length `N` 294 912, so a row-major order is kept. An explanation for this phenomenon is that the planner that uses flag `ESTIMATE` has chosen a bad set of codelets to compute the FFT of the given matrix. The anomaly occurred through the tests of the test set randomly with non-specific matrix sizes. Because of those random fluctuations I created a test in chapter 3.4 that is looking into various planner modes and their impact on the FFT computation results.

To confirm the need of a row-major order and the lowest measured computation times I ran next test with area 7 584 759. The test was reran three times. The results of the test are displayed in figure 3.5 and were similar to test results 3.4. The horizontal axis represents row length `N` of the matrix. The vertical axis represents duration of FFT computations for given matrix dimension. I will read an example of measured values. FFT computation duration of the matrix with row length `N` 3087 and computed column length `M` 2457 was measured three times as 769×10^6 CPU cycles.

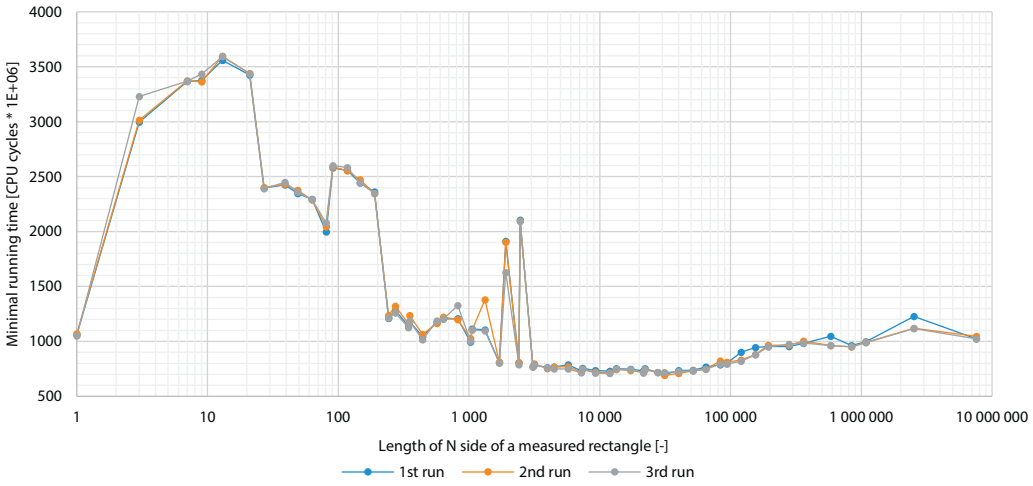


Figure 3.5: FFT speed test of a rectangular area 7 584 759

The best measured times (lowest FFT computation durations) are again located between 3000 and 100 000 row length `N`. In every other FFT test of the test set such location with very low computation durations also exists. I call those locations of low durations **bathtubs** because of a graphical resemblance to cross-section of a bathtub. Now when I know such locations exist I will try to define a principle that will choose such a row length `N` that will in most cases find bathtub.

I took results of all tests from the test set and calculated intervals of bathtubs' minima and maxima that were measured visually for each test from the test set. An example of bathtub's minimum in figure 3.5 is the row length `N` 3159. The maximum in the same figure is the row length `N` 83 349. Put into one record, bathtub of figure 3.5 is in the row length `N` interval [3159; 83 349].

Now I have row lengths `N` of bathtubs' minima and maxima for all tests in the test set. It would be useful to find common denominators of minima and maxima so we can determine where usually minima starts and maxima ends. This is not feasible just by comparing measured lengths because every matrix has different beginning and ending length of bathtub. What is comparable are *ratios of minima* lengths and *ratios of maxima* lengths. The ratio represents how much should be row length `N` longer than column length `M` before the minimum / maximum of the figure's bathtub will be reached.

$$\text{Ratio} := \frac{\text{row length } \texttt{N}}{\text{column length } \texttt{M}} \tag{13}$$

As an example I will compute bathtub minimum ratio for the FFT computation time with matrix area 7 584 759. We already know that bathtub's minimum (figure 3.5) is in the row length `N` 3159. Using substitution in equation (12) we gain

$$\text{column length } \texttt{M} := \frac{\texttt{AREA}}{\texttt{N}} = \frac{7\,584\,759}{3159} = 2401$$

With both row length `N` and column length `M` known, we can substitute variables in equation (13) as

$$\text{Ratio}_{7\,584\,759,min} := \frac{\text{row length } \texttt{N}}{\text{column length } \texttt{M}} = \frac{3159}{2401} \doteq 1.32$$

The gained ratio 1.32 means that to compute the FFT at the beginning of bathtub I need to have a row length `N` 1.32× longer than column length `M`. Table 3.4 shows first ten minima of the tests' bathtubs.

Matrix area	column length `M`	row length `N`	Minimum ratio
5 308 416	1728	3072	1.78
5 948 800	880	6760	7.68
5 976 432	693	8624	12.44
6 054 048	792	7644	9.65
6 141 096	729	8424	11.56
6 328 125	1125	5625	5
6 879 600	588	11 700	19.90
7 474 194	702	10 647	15.17
7 584 759	2401	3159	1.32
7 844 067	693	11 319	16.33

Table 3.4: First ten minima of the tests' bathtubs

To better represent the computed values I created figure 3.6 that shows minima ratios of the whole test set bathtubs. The horizontal axis represents each matrix's area. The vertical axis shows `N:M` ratio. An example of reading the figure is that for the matrix with area 7 584 759 the ratio is 1.32.

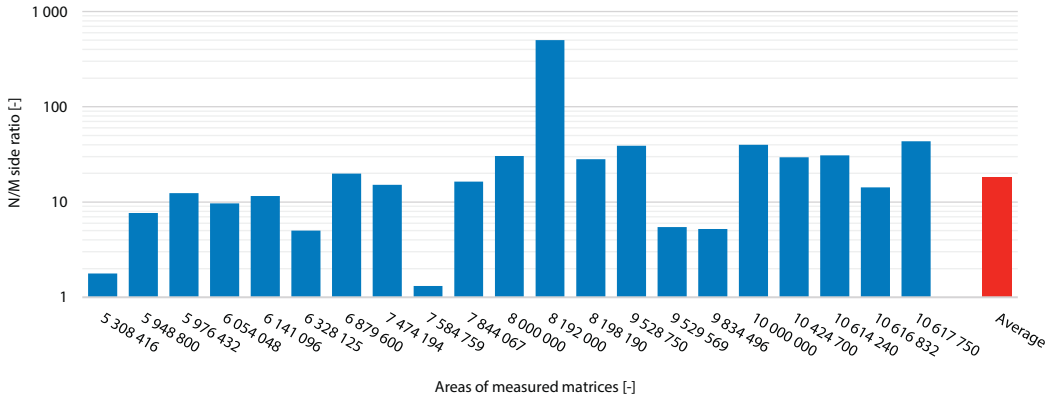


Figure 3.6: N/M sides ratios of bathtubs' minima

Last column in figure 3.6 shows average value of measured ratios. Because of high variance of computed ratios I excluded 30 % of border values from the average ratio; 15 % from top and 15 % from bottom. Precisely I used method `TRIMMEAN` that takes two input arguments. First is list of values to be processed, second is percentage of border values to be excluded from the computations. From our 21 results of the test set I have decided that three values from top and three from bottom will be excluded. Put into equation I find out needed

$$\text{excluded percentage} := \frac{3\ (\text{top}) + 3\ (\text{bottom})}{\text{number of tests}} = \frac{6}{21} \doteq 0.3 = 30\ \% \tag{14}$$

With 30 % of ratios excluded, final minima average ratio is 18.39. As we already described the gained ratio 18.39 means that to compute the FFT at the beginning of bathtub I need to have a row length `N` approximately **19× longer** than column length `M`.

I will apply the same procedure I used to find out average minima ratio to compute average maxima ratio. Table 3.5 shows first ten maxima of the tests' bathtubs.

Matrix area	column length `M`	row length `N`	Maximum ratio
5 308 416	64	82 944	1296
5 948 800	1	5 948 800	5 948 800
5 976 432	36	166 012	4611.44
6 054 048	6	1 009 008	168 168
6 141 096	1	6 141 096	6 141 096
6 328 125	45	140 625	3125
6 879 600	45	152 880	3397.33
7 474 194	26	287 469	11 056.50
7 584 759	91	83 349	915.92
7 844 067	7	1 120 581	160 083

Table 3.5: First ten maxima of the tests' bathtubs

As with minima I created figure 3.7 that shows maxima ratios of the whole test set bathtubs. The horizontal axis represents each matrix's area. The vertical axis `N:M` ratio. An example of reading the figure is that for the matrix with area 7 584 759 the ratio is 915.92.

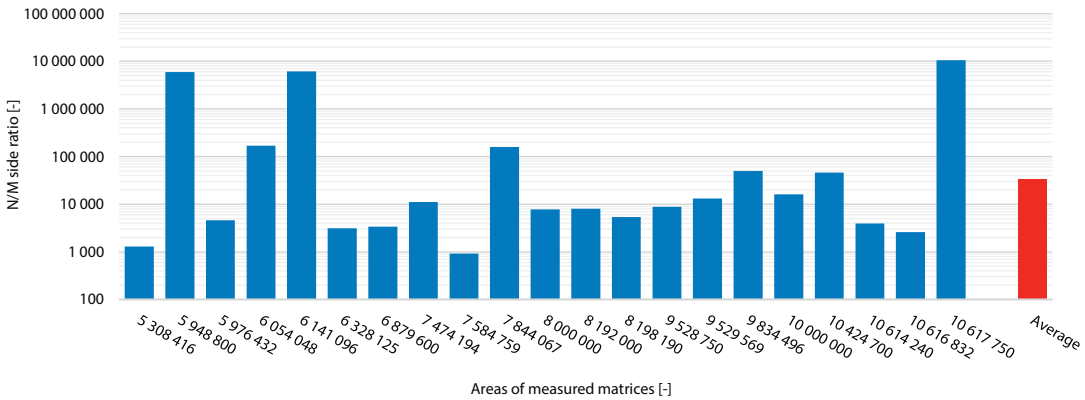


Figure 3.7: N/M sides ratios of bathtubs' maxima

Last column in figure 3.7 shows average value of measured ratios. As with minima I excluded three border values from top and three from bottom from average ration. With 30 % of ratios excluded (equation (14)) final maxima average ratio is 33 993.3. The gained ratio 33 993.3

means that to compute the FFT at the end of bathtub I need to have a row length N approximately **34 000× longer** than column length M.

Unfortunately the variance of the results is large and is not stable across the tests. With that said, I will conclude following outcome based mainly on results of average ratios:

- Row length N should be *at least* 19× larger than column length M
- and row length N should be *at most* 34 000× longer than column length M.

3.4 Planner flags significance

From previous test results I found out that the FFT computation times for some dimensions are much longer than they could be. After some reading I reached the opinion that this behaviour is related to the *planner*. The planner (described along with FFTW flags in section 2.4) takes the FFTW flag as a parameter and generates a plan. Up to now we used flag ESTIMATE which generates a plan in the fastest time.

I will create a new test set that will deal with settings of other various planner flags. By using other flags we can achieve better FFT computation speeds and more predictable results without computation durations fluctuations. If my estimation is correct, setting any other planning flag but ESTIMATE (heuristic, no measuring) should create a "smoother" graphical representation with overall faster FFT computation times.

3.4.1 Flags for area 2^{16}

As a first test I will compute the 2-D FFT of an area 2^{16} with FFTW planning flags ESTIMATE, MEASURE, PATIENT and EXHAUSTIVE. Such small area was chosen because of long computation time needed to create a plan for non-ESTIMATE flags as stated in chapter 2.4. Based on computed results from section 3.3 I expect that FFT durations measured with flag ESTIMATE will have bathtub. As for other flags we will see how much will be the computation results faster than for ESTIMATE flag.

Graphical representation of measured durations are shown in figure 3.8. The horizontal axis represents row length N of the matrices. The vertical axis represents duration of the FFT computation for given matrix dimension. I will read an example of measured values for row length N 1024 in descending order. FFT computation with flag ESTIMATE was finished in 680×10^3 CPU cycles, MEASURE in 560×10^3 CPU cycles, PATIENT and EXHAUSTIVE were finished equally in 510×10^3 CPU cycles.

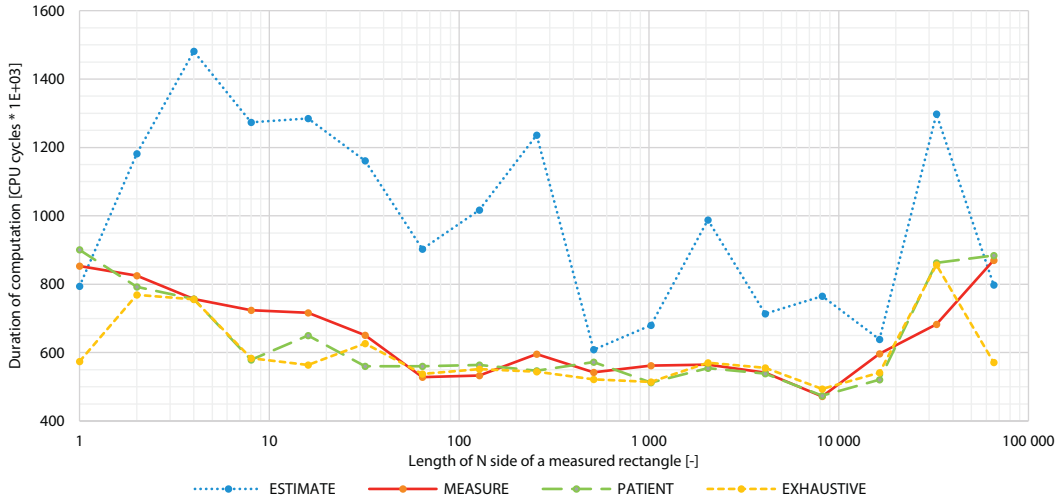


Figure 3.8: FFT computation speed test of matrices with area 2^{16}

Non `ESTIMATE` durations in figure 3.8 are lower and my prediction was nearly correct. I say nearly, because at the beginning and the end of the figure we can see that the planner with any flag except for `EXHAUSTIVE` is not able to optimize FFT time completely. What's more, at row length `N` 2^{15} (32.8×10^3) we can see `PATIENT` and `EXHAUSTIVE` FFT computation times worse than with flag `MEASURE`. Other important fact is that measured bathtub with the `ESTIMATE` flag is not present with other flags. We can state that the planner with `ESTIMATE` flag creates plans with unfavourable performance.

The obvious question is why should we use the planner with the flag `ESTIMATE` at all and what are other differences between the flags except for various computation results speeds. Figure 3.9 shows times the planner needs to create a plan with a flag. Measured area 2^{16} is the same as in figure 3.8. The horizontal axis represents row length `N` of the matrix. The vertical axis represents duration of the FFT computation for given matrix dimension in real time milliseconds instead of CPU cycles. Reason for the change of units is described at the beginning of chapter 3. I will read an example of the figure's values. The planner with `ESTIMATE` flag needed <1 ms for row length `N` 1024 to finish the plan. The planner computation with `MEASURE` flag took 163 ms, `PATIENT` flag took 10 291 ms and `EXHAUSTIVE` flag 151 533 ms.

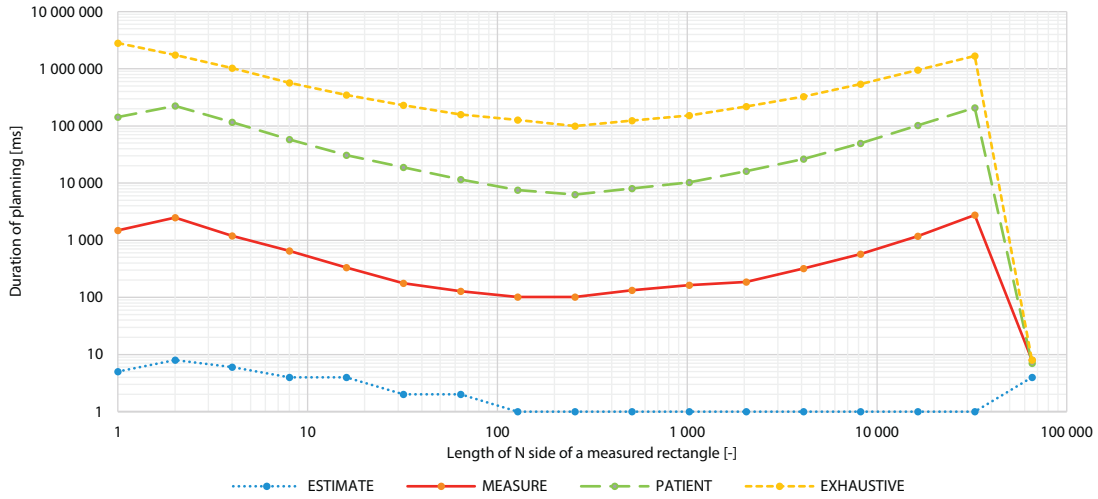


Figure 3.9: Computation speed test of the planner of matrices with area 2^{16}

In figure 3.9 measured durations of a plan creation with flag `ESTIMATE` in range [2^7 (128); 2^{15} (32.8×10^3)] are not exact, because the used counter has a standard millisecond deviation. Measured time in row length `N` 2^{16} (65.5×10^3, rightmost value) is interesting. The reason the planner have similar very low times for all flags is because of measured dimensions 1 `M`$\times2^{16}$ (65.5×10^3) `N`. FFTW recognized this as an 1-D array and the planner used this information to speed-up the calculation.

To compare the result times of the FFT and the planner I will need to convert measured FFT times into milliseconds with conversion procedure described in table 3.1. The highest CPU cycle count from the whole test is `ESTIMATE` flag 16 384 `M`$\times4$ `N` with 1 480 941 measured CPU cycles. If 2 000 000 000 CPU cycles $\approx$ 1000 milliseconds, 1 480 941 cycles is under one millisecond. According to this conversion I created table 3.6 with converted measured FFT computations and planning durations.

	FFT time [ms]	Planner time [ms]
`ESTIMATE`	$\approx$1	<1
`MEASURE`	$\approx$1	163
`PATIENT`	$\approx$1	10 291
`EXHAUSTIVE`	$\approx$1	151 533

Table 3.6: Combination of the FFT and the planner times from results of figures 3.8 and 3.9 for row length `N` 1024

From the results of table 3.6 it's clear that even though `ESTIMATE` flag has the worst performance of all flags, it produced the FFT result in the shortest total time. If we would look at it from the other side we would see that to produce FFT computation result that takes one millisecond, the planner with `PATIENT` flag needed ten seconds and with `EXHAUSTIVE` flag even more than two minutes. In such a small area as 2^{16} (65.5×10^3) ($2^{16}\times16$ bytes

for complex number = 1 MB) the long planning times indicate that those two flags are unusable for our needs. In order not to hastily exclude those flags I searched the FFTW documentation [fft12c] which states that `PATIENT` and `EXHAUSTIVE` flags produce "more optimal" plan especially for *larger* matrices. Therefore I will run next test in section 3.4.2 for large matrices.

3.4.2 Flags for areas 2^{10} – 2^{28}

By following results of section 3.4.1 I will measure the FFT computation duration and times of planning of areas ranging from 2^{10} to 2^{28} with FFTW planning flags `ESTIMATE`, `MEASURE`, `PATIENT` and `EXHAUSTIVE`. My objective is to see whether computations with non `ESTIMATE` flags will be noticeably faster for larger areas. 2^N series was chosen because there is a low number of FFT computations required (area 2^{28} will be computed in only 29 computation sequences) and results will be equidistantly divided in measured area. In this test I expect that FFT computations measured with flags `PATIENT` and `EXHAUSTIVE` will have much lower durations especially for very large areas. I obtained measured values for the tests as follows.

I have ran speed tests on a rectangular areas ranging from 2^{10} to 2^{28} similar to tests in section 3.3 for all four planner flags. The area 2^{28} was the largest I could test. With 8 GB of a physical memory and 16 bytes needed for a representation of a complex number I made use of 4 GB of the memory. Other memory was either unused or reserved by the OS.

Finally after running all test I ended up with 60 test files with various areas and used planning flags (60 total files with measured area 2^{10} to 2^{28}, 19 `ESTIMATE`, 19 `MEASURE`, 14 `PATIENT`, 8 `EXHAUSTIVE`). To meaningfully compare the results I took one measured value from every test result chosen with formula

$$\texttt{RoundUpToNearestNValue}(\sqrt{\text{MeasuredArea}} \times \text{MinRowLengthMultiplier}) \qquad (15)$$

for comparison with the other chosen values. I chose the `MinRowLengthMultiplier` as 10 instead 19 (result of test 3.3) to simulate worse choice of the planner with `ESTIMATE` flag. With this setup the measured values of the `ESTIMATE` flag should lie near a beginning of bathtub so results shouldn't be slowed down much. Other flags don't rely on bathtubs but have generally better results when respecting row-major order so multiplier 10 should be sufficient.

I will show an example of choosing measured value for area 2^{15} (32.8×10^3) flag `MEASURE`. At first I measured FFT and planning speeds and saved them into table 3.7.

Column length M	Row length N	Planner time [ms]	FFT duration [CPU cycles $\times 10^3$]
32768	1	1423	334
16384	2	1090	409
8192	4	604	383
4096	8	222	367
2048	16	168	370
1024	32	115	354
512	64	70	366
256	128	58	344
128	256	87	280
64	512	94	356
32	1024	104	286
16	**2048**	155	260
8	4096	287	252
4	8192	503	249
2	16384	1287	363
1	32768	3	331

Table 3.7: Combination of the FFT and the planning times for area 2^{15} (32.8×10^3) flag MEASURE

After that I needed to find out the measured value. I filled the formula

$$\begin{aligned}\text{ChosenValue}_{2^{15},\mathit{MEASURE}} &:= \lceil\sqrt{\text{MeasuredArea}}\times 10\rceil_{\text{NearestN}}\\ &= \lceil\sqrt{2^{15}}\times 10\rceil_{\text{NearestN}}\\ &= \lceil 1810\rceil_{\text{NearestN}}\\ &= 2048\end{aligned}$$

and found out my searched row length N is 2048 and extracted computation times, i.e. 155 ms planning time and 260×10^3 CPU cycles. I applied this procedure to all measured results and gained final test data. Graphical representation of the data is shown in figure 3.10. The horizontal axis represents areas of the matrices in 2^N sequence where $N\in[10; 28]$. The vertical axis represents duration of the FFT computation for given matrix dimension and all flags. I will read an example of measured values for area N 2^{10} (1024, leftmost value). Computation of the FFT for flag ESTIMATE was finished in 49×10^3 CPU cycles, for MEASURE flag in 60×10^3 CPU cycles, for PATIENT flag in 11×10^3 CPU cycles and plan for flag EXHAUSTIVE was finished in 10×10^3 CPU cycles.

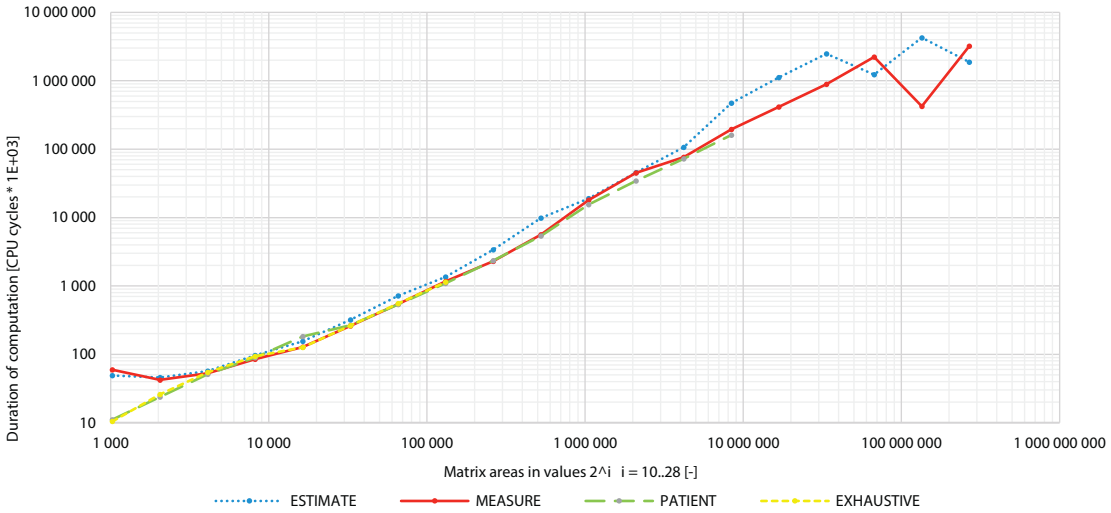


Figure 3.10: Measured FFT computation speeds of various FFTW flags of matrices with areas 2^{10} (1024) to 2^{28} (268×10^6)

At the beginning of figure 3.10 we can see a bit of a performance distortion. We have already dealt with such small number of CPU cycles in the first test of chapter 3.4. Next useful information is that FFT durations of all flags excluding `ESTIMATE` have similar running times through all tests. To show the times in linear way I changed representation of figure 3.10 partly to figures 3.11 and 3.12.

Figure 3.11 shows half of the same data as in figure 3.10 but in durations of computations up to 10×10^6 CPU cycles ($\approx$ 5 ms). The horizontal axis represents areas of the matrices in 2^N sequence where N$\in$[10; 19]. The vertical axis represents duration of the FFT computation for given matrix dimension and all flags. I will read an example of measured values for area `N` 2^{18} (262×10^3). FFT computation with planner flag `ESTIMATE` was finished in 3.4×10^6 CPU cycles, `MEASURE` and `PATIENT` in equally in 2.25×10^6 CPU cycles, and computation with flag `EXHAUSTIVE` wasn't for this area measured.

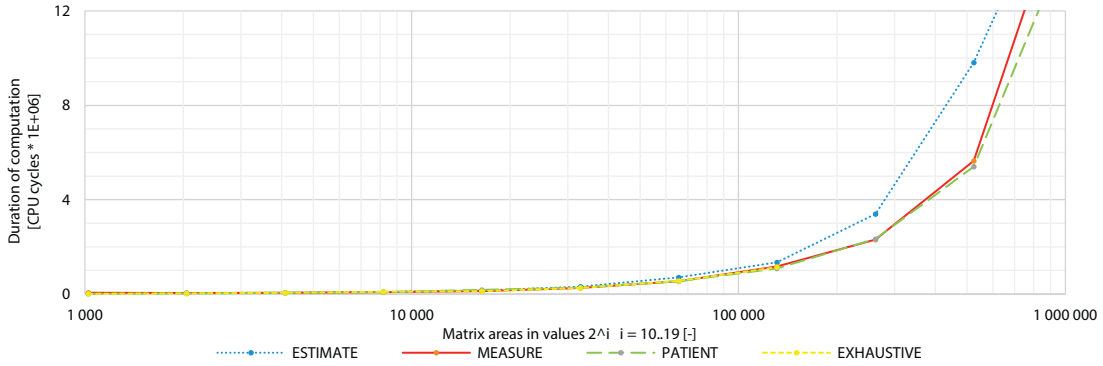


Figure 3.11: FFT computation speed test of various FFTW flags of matrices with areas ranging from 2^{10} (1024) to 2^{19} (524×10^3)

The computation with EXHAUSTIVE flag ends at the area of 2^{17} (131×10^3) because time needed for planning with such area exceeded six minutes (see figure 3.13). With such high required planning time and low CPU cycles the planner with EXHAUSTIVE flag was unfit for our needs and was excluded from next tests.

Figure 3.12 shows the other half of the same data in a high computations durations area beginning at 10×10^6 CPU cycles. The horizontal axis represents areas of the matrices in 2^N sequence where $N\in[20; 28]$. The vertical axis represents duration of the FFT computation for given matrix dimension and remaining flags. I will read an example of FFT measured values for area N 2^{24} (16.8×10^6). FFT computation with planner flag ESTIMATE was finished in 1110×10^6 CPU cycles, MEASURE in 400×10^6 CPU cycles, and computation with flag PATIENT wasn't for this area measured. The planner with flag PATIENT and planning time over seven minutes suffered the same fate as the planner with EXHAUSTIVE flag and was excluded from future tests.

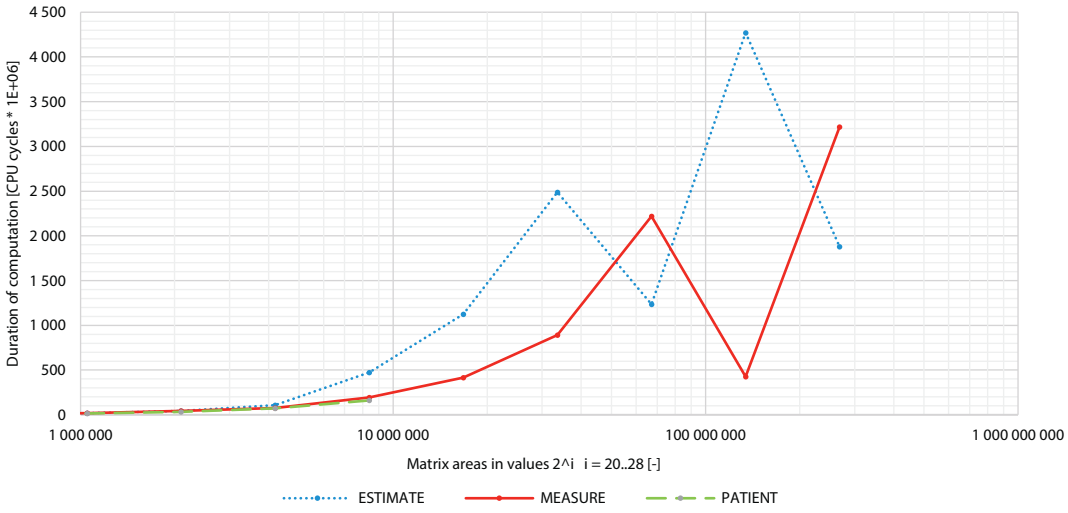


Figure 3.12: FFT computation speed test of various FFTW flags of matrices with areas 2^{20} (1 048 576) to 2^{28} (268 435 456)

The FFT durations of the planner with ESTIMATE and MEASURE flags began to differer at area 2^{22} (4.2×10^6) continuing up to 2^{28} (268×10^6). Measured values are shown in table 3.8. The rightmost column holds computed time differences between measured FFT durations of flags ESTIMATE and MEASURE in milliseconds. To perform such conversion I used equation (10). The '–' sign at areas 2^{26} (67×10^6) and 2^{28} (268×10^6) symbolize that the ESTIMATE FFT computation was faster than MEASURE computation.

We can see the stable grow of areas ranging from 2^{22} (4.2×10^6) up to 2^{26} (67×10^6) where the first time change occurs. When looking at area 2^{26} (67×10^6) in figure 3.12 we would be expecting the measured ESTIMATE FFT duration to be located somewhere around 3300×10^6 CPU cycles. The fact that it is not there means that the planner with ESTIMATE flag was "lucky" and chose exceptionally good set of codelets.

Second and the largest time change occurs with area 2^{27} (134×10^6). The FFT computation difference is nearly two seconds. This time was "lucky" the planner with `MEASURE` flag and chose codelets exceptionally well. On the other hand FFT duration with `ESTIMATE` flag turned out as expected.

The last, third, time change happens with area 2^{28} (268×10^6) and the results are similar to the first time change.

Area `M`×`N`	`ESTIMATE` FFT duration [CPU cycles $\times10^6$]	`MEASURE` FFT duration [CPU cycles $\times10^6$]	`ESTIMATE` − `MEASURE` duration difference [ms]
2^{22} (4.2×10^6)	107	76	15
2^{23} (8.4×10^6)	473	195	139
2^{24} (16.8×10^6)	1123	415	354
2^{25} (33.5×10^6)	2487	891	798
2^{26} (67×10^6)	1234	2221	−493
2^{27} (134×10^6)	4271	426	1923
2^{28} (268×10^6)	1880	3217	−668

Table 3.8: Combination of the FFT durations for areas 2^{22} – 2^{28} flags `ESTIMATE` and `MEASURE`

How much time was needed for computation of plans for all four flags is shown in figure 3.13. The horizontal axis represents areas of the matrices in 2^N sequence where $N\in[10; 28]$. The vertical axis represents duration of planning for given matrix dimension. I will read an example of measured values for area `N` 2^{16} (65.5×10^3), the last area with all four FFTW flags. Creation of a plan with planner flag `ESTIMATE` was finished in less than one millisecond, `MEASURE` planning took 320 ms, `PATIENT` planning took 20 600 ms and planning with flag `EXHAUSTIVE` was finished in 320 000 ms. We can see that all times grow in a similar shape and direction.

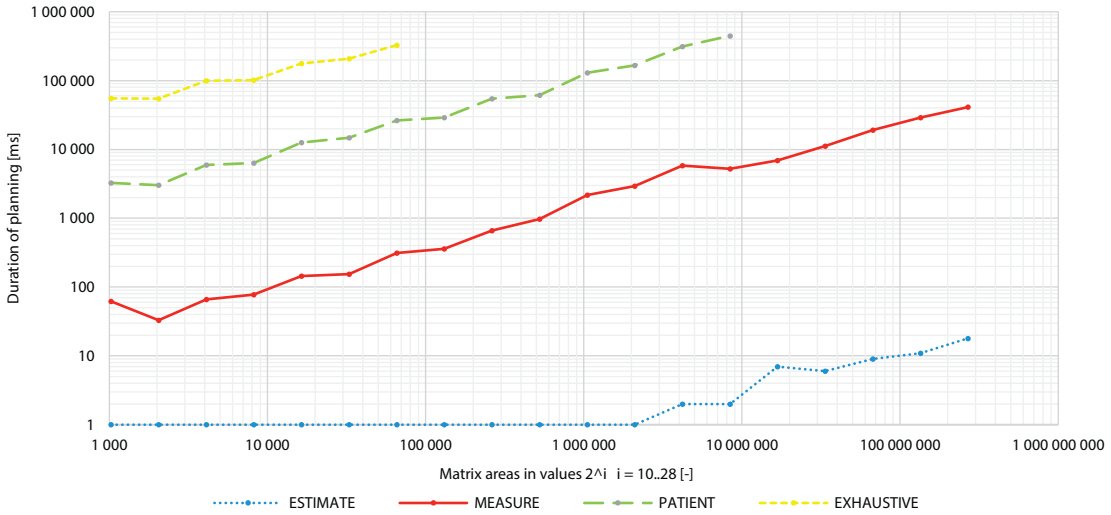


Figure 3.13: Planning speed test of matrices' areas 2^{10} to 2^{28} with various FFTW flags

To conclude the test set results I created table 3.9. Most interesting values were in higher areas of the test so only the measured planning durations from flags `ESTIMATE` and `MEASURE` will go in the table.

Area of computation	Planner `ESTIMATE` [ms]	Planner `MEASURE` [ms]
$\mathbf{2^{25}}$ (33.5×10^6)	6	11 179
$\mathbf{2^{26}}$ (67×10^6)	9	19 085
$\mathbf{2^{27}}$ (134×10^6)	11	29 064
$\mathbf{2^{28}}$ (268×10^6)	18	41 191

Table 3.9: Combination of the most interesting planning durations for areas 2^{25}–2^{28} flags `ESTIMATE` and `MEASURE`

To compare speeds of FFT computations and planning durations I converted the CPU cycles to milliseconds into table 3.10. The planner with `MEASURE` flag should generally have better FFT computation times (confirmed in figure 3.10) but has much slower planning (confirmed in figure 3.13). From the final computation times in the converted table is clear that using the flag `ESTIMATE` yields the FFT result multiple times faster than using the `MEASURE` flag. This conclusion is confirmed up to computations using 4 GB of a computer memory. My hypothesis is that when using more memory the planner's times will grow accordingly with figure 3.13.

Area of computation	Total computation time `ESTIMATE` [ms]	Total computation time `MEASURE` [ms]
$\mathbf{2^{25}}$	1243 + 6 = 1249	446 + 11 179 = 11 625
$\mathbf{2^{26}}$	617 + 9 = 626	1110 + 19 085 = 20 195
$\mathbf{2^{27}}$	2136 + 11 = 2147	213 + 29 064 = 29 277
$\mathbf{2^{28}}$	940 + 18 = 958	1608 + 41 191 = 42 799

Table 3.10: An excerpt from total computed results of the test section 3.4

Conclusion of the chapter 3.4 is that planner flags `PATIENT` and `EXHAUSTIVE` are not usable for our needs and don't have noticeably faster results than `MEASURE` flag as seen in figure 3.10.

Additionally in figure 3.12 we could see that for larger areas the difference between planner flags is significant. The flag `MEASURE` yields faster plans than flag `ESTIMATE` but its planning duration growth is too steep especially with larger areas. What we couldn't control is the choice of codelets resulting in unpredictability of the FFT computation duration. Unfortunately we couldn't test even larger areas because of a lack of usable free memory. And so with the available result data I made the decision that following tests will use the planner flag `ESTIMATE`.

3.5 2^n matrices

This test is based on deduction of Mr. Nedved who wrote that noticeable FFT computation speed-up can be achieved with input matrix dimensions $2^{\mathtt{M}} \times 2^{\mathtt{N}}$ where $\mathtt{M}, \mathtt{N} \in \mathbb{N}$. The FFTW documentation [fft12a] contains the same information in exact wording, "transforms whose sizes are powers of two are especially fast". If the deduction is correct I'm going to prefer powers of two to choose matrix sizes for FFT computations.

To prove the speed-up is present, I created the test where the goal is to compare FFT computation speeds of matrices with total areas of three neighbour fftw-friendly numbers. For middle number I chose area $2^{24} = 16\,777\,216$ because this area is large enough for longer durations of FFT computations. I expect that FFT durations for matrix with area 2^{24} will be faster than for neighbouring matrices. Now I needed to choose the neighbour fftw-friendly numbers, one that is lesser than 16 777 216 and other larger than 16 777 216. I found out that nearest fftw-friendly numbers together with 2^{24} are

- **16 773 120** = $2^{12} \times 3^2 \times 5^1 \times 7^1 \times 13^1$; labeled *2^24−1*,
- **16 777 216** = 2^{24}; labeled *2^24* and
- **16 793 868** = $2^2 \times 3^1 \times 7^2 \times 13^4$; labeled *2^24+1*.

After running the test I found out interesting facts about measured data. Even when not intentional, every number has different properties.

- I have computed 312 values for various combinations of column lengths `M` and row lengths `N` for number *2^24−1*. From those 312 values I equidistantly chose only 26 that will be put into a graphical representation. Those 26 measured values have either row lengths `N` $2^{\texttt{N}}$ or column lengths `M` $2^{\texttt{M}}$. The other value is always fftw-friendly but neither $2^{\texttt{N}}$ nor $2^{\texttt{M}}$. In other words every matrix has one dimension $2^{\texttt{N}}$ or $2^{\texttt{M}}$ and one non-$2^{\texttt{M}}$, non-$2^{\texttt{N}}$ dimension. An example can be measured matrix with dimensions M×N 8190×2^{11}.

- *2^24* has all 25 measured row lengths `N` as $2^{\texttt{N}}$ and column lengths `M` as $2^{\texttt{M}}$. For example a measured matrix has dimensions M×N $2^{17}\times2^{7}$.

- For the last number *2^24+1* I have computed 90 values from that I chose 84 that will be put into a graphical representation and don't have any dimension $2^{\texttt{M}}$ or $2^{\texttt{N}}$. Put differently every matrix has two non-$2^{\texttt{M}}$ non-$2^{\texttt{N}}$ lengths in any dimension. As an example we can use a measured matrix with dimensions M×N 24 843×676.

Figure 3.14 shows results of the FFT computation duration for all three tested areas. The horizontal axis represents row length `N` of the matrices. The vertical axis represents duration of the FFT computation for given matrix dimension. I will read an example of measured values for row length `N` approximately 1000 from fastest to slowest. *2^24+1* has FFT duration of 1070×10^6 CPU cycles, *2^24−1* 1400×10^6 CPU cycles and *2^24* 1715×10^6 CPU cycles. That means the fastest time was computed with *2^24+1*, *2^24−1* comes second and the slowest was *2^24*. This is in contradiction to the assumption of Mr. Nedved.

Additionally I computed average FFT computation duration for the all measured values of the three areas. The fastest is *2^24−1* with average 1224×10^6 CPU cycles. Second is *2^24+1* with 1277×10^6 CPU cycles. The difference between those measured values of *2^24−1* and *2^24+1* is only $1277-1224 = 53\times10^6$ CPU cycles. The last is *2^24* with 1409×10^6 CPU cycles. Difference between *2^24+1* and *2^24* is $1409-1277 = 132\times10^6$ CPU cycles which is $\doteq 2.5\times$ larger than the result of first difference 53×10^6 CPU cycles.

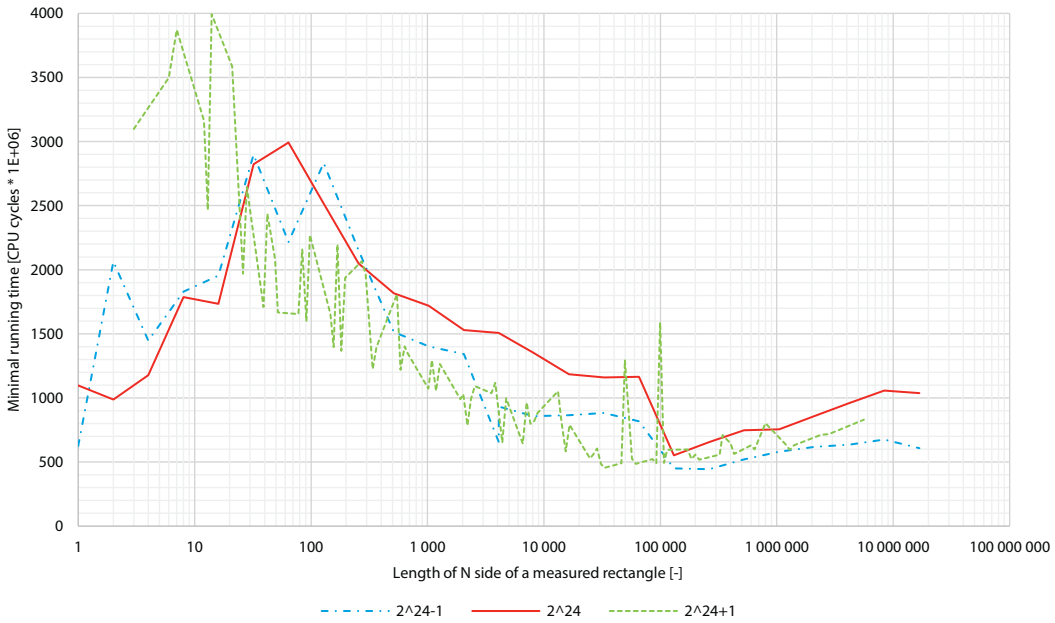


Figure 3.14: FFT computation time differences between fftw-friendly areas of matrices *2^24−1*, *2^24*, *2^24+1*

From figure 3.14 we can see that even though we would be expecting the *2^24* to be noticeably faster or at least equally fast with other numbers, the *opposite is true*. The results were a bit surprising. Area *2^24+1* without any $2^{N|M}$ fftw-friendly values had the fastest FFT results in row length `N` interval [26; 100 000] where *2^24−1* takes over. Only interval where *2^24* is better than others is in row length `N` [1; 16]. At last we can state that our expectation were not met and FFT durations for matrix with dimensions 2^{24} is **slower** than for neighbouring matrices.

3.6 Speed-up of bathtub

This test is based on discovery of Mr. Nedved (see chapter 1) who wrote that in all cases where at least one side of division part $P_M \times P_N$ is considerably asymmetrical, FFT computation times are much longer than with regularly square or nearly square parts. I will transform such statement into our terms. We have a rectangular area with dimensions: column length `M` and row length `N`. If the column length `M` is approximately same as row length `N` the area is squarish and its computation times should be much shorter than in cases where column lengths `M` are much different from row length `N`.

With results of test set 3.3 and one additional test I will try to challenge such statement. I will measure FFT computation speed of a matrix with area 40 435 200 $\approx$ 6359×6359. I have chosen this area because

- it's fftw-friendly and its decomposition $2^9 \times 3^5 \times 5^2 \times 13^1$ contains four prime numbers and their various exponents,

- takes up large enough memory space $40\,435\,200 \times 16 \doteq 617$ MB to compute the in-place FFT and
- is at least four times larger than tests in chapter 3.3 so we will cover more areas.

I expected the results to be similar to test set results from chapter 3.3. Row length `N` with its range [1; 40 435 200] will have in interval [1; 6359] slow measured FFT times and in interval [6359; 40 435 200] we will find bathtub (the fastest measured FFT times). Other result will be that squared or nearly squared area won't have the fastest FFT computation results, because we already know that the fastest results are located in bathtub, which is in our case minimally $19 \times 6359 = 120\,821$ row length `N`.

Figure 3.15 shows results of FFT computation speed-up. The horizontal axis represents row length `N` of the matrix. The vertical axis represents *speed-up multiplier* of FFT computation for given matrix dimensions. The multiplier was created this way. FFT computation of matrix 6359×6359 (square) took 1955×10^6 CPU cycles. This result is not measured but had to be interpolated from results of matrices 6400×6318 and 6318×6400 because number 6359 and its adjacent numerical neighbours are not fftw-friendly. I have assigned base multiplier *one* to this computed FFT duration. Other measured FFT durations have their multiplier computed in proportion to the base multiplier. In other words if other measured FFT durations have lower (better) result times, the multiplier is lower than one and vice versa. In table 3.11 we can see some of the measured FFT durations and their respective multipliers.

computed area dimensions (`M`×`N`)	FFT duration [$\times 10^6$ CPU cycles]	Multiplier
6912×5850	1830	0.936
6656×6075	2132	1.090
6480×6240	2628	1.344
6400×6318	2171	1.110
6359×6359	1955	1
6318×6400	1739	0.890
6240×6480	1707	0.873
6075×6656	1929	0.986
5850×6912	3311	1.694

Table 3.11: Measured FFT durations for given areas and their respective multipliers

The combination of measured results and computed multipliers are shown in figure 3.15. I will read an example of measured values. The matrix with row length `N` 6480 and computed column length `M` 6240 has FFT duration multiplier of 1.34. It means that duration of FFT computation was 1.34× longer than duration of FFT computation for square matrix 6359×6359 with multiplier one.

The results of this test roughly turned out as expected. Figure 3.15 has standard shape with measured values in its left half slower than in the right half with existing bathtub. Additionally our results doesn't confirm Mr. Nedved's statement of best computation times with square matrices. Square or nearly square matrices have speed-up multiplier 0.9, whereas speed-up multiplier in bathtub is around 0.7.

Figure 3.15 shows one additional characteristic that hasn't occurred in any other test result. Some FFT durations with row length `N` in interval [1; 1152] have been computed outstandingly fast. The fastest calculation was performed with row length `N` 27 gaining multiplier 0.001 37. Which means that duration of FFT computation for row length `N` 27 was 730× faster than duration of FFT computation for square matrix with dimensions 6359×6359. Same as in section 3.3, the reason for this difference in speed is not obvious. In my opinion it's caused by the planner that had chosen different set of codelets resulting in better computation time. We cannot use this finding because its occurrence can not be guaranteed.

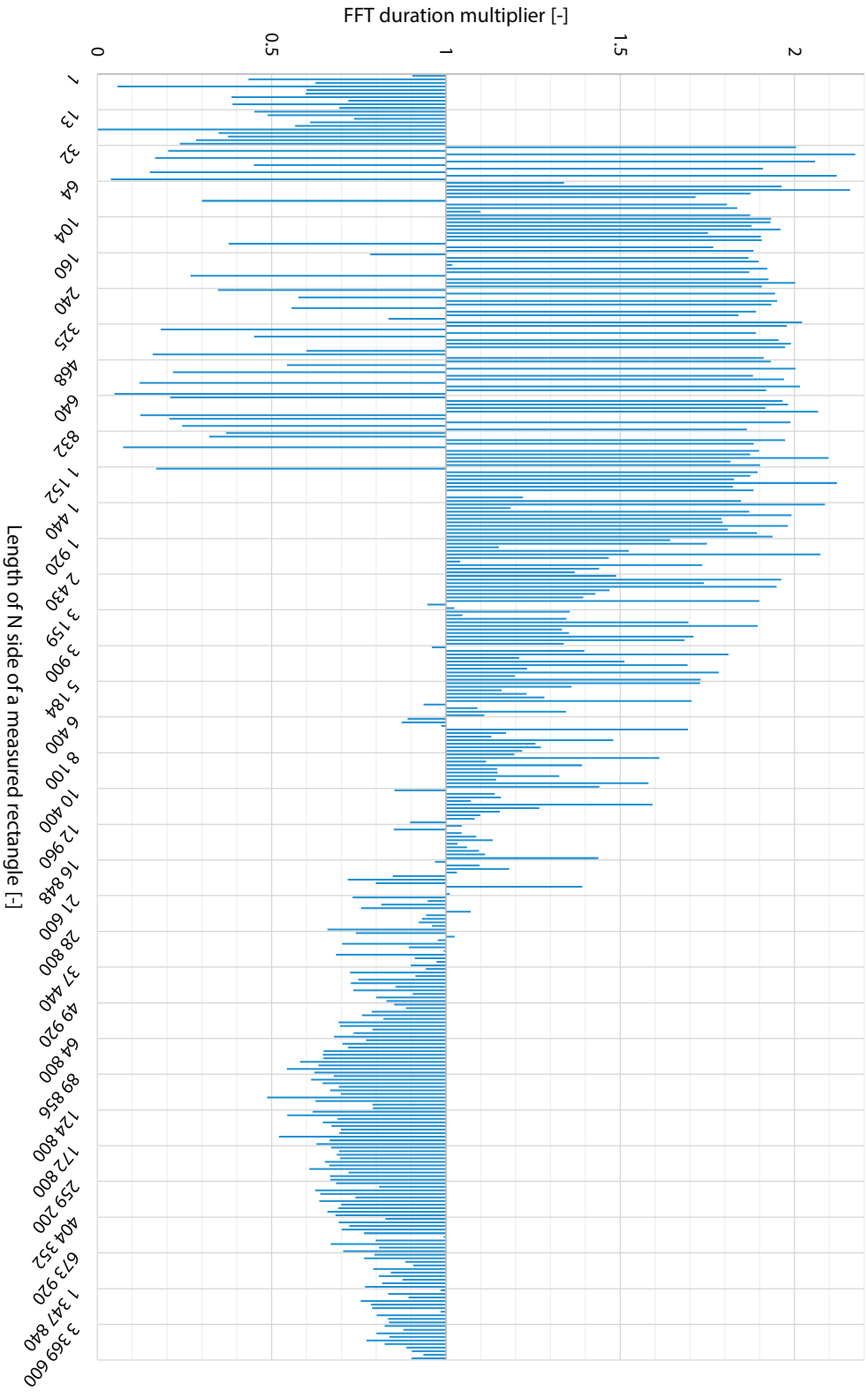


Figure 3.15: FFT computation duration multiplier of the rectangular area 40 435 200. The figure is represented as various multipliers with a basis of the square area of 6359×6359 that has a multiplier of value *one*

3.7 Parallel calculation

In chapter 2 we discussed different ways of a parallel processing of input data and stated, that the winning library should support at least one of the ways. In this section I will describe and test options of a parallel processing available in FFTW. We emphasise the importance of an in parallel processing because nowadays it's common to have a computer hardware capable of processing at least eight threads concurrently. It would be wise of us to use such opportunity to considerably fasten FFT computations.

As we have seen in chapter 1, the process of a light propagation is heavily based on the use of forward and backward FFTs. If we would run FFTs fully in parallel, we would speed-up the whole light propagation process. Further parallelism wouldn't have large impact on a speed-up of a light propagation process.

Before searching for manual ways of processing threads, such as library `pthreads` or `Intel TBB`, I looked into FFTW documentation and found a brief manual [fft12f] with a description about a native support of an in parallel FFT processing in FFTW. The procedure of enabling a parallel processing in FFTW consists of initializing the parallel FFTW `pthreads` engine and after that setting a required number of threads with function `fftw_plan_with_nthreads`. Apart from the initialization of threads at the beginning and freeing taken resources at the end of a process, all other FFTW operations function the same way as when not using any thread functions.

When reading more information I found a note that says, "unfortunately, the arrays being transformed by different processors are interleaved in memory, resulting in more memory contention than is desirable. We are investigating ways to alleviate this problem" [fft12e]. We won't look into this issue any longer because our implementation of threads would also have to solve such problem. The issue will be dealt with in future versions of FFTW.

3.7.1 Native FFTW parallel processing

I'm going to run a test with a various number of FFTW threads measuring FFT computation speeds of matrices with area 40 435 200 chosen for the same reasons already described in section 3.6. My objective is to see how much performance I will gain when using one, two, three or four threads. I expect that durations of FFT computations for one thread will be the slowest (the longest needed time) and for four threads the fastest. During adding threads into computation the general shape of computed values should stay the same, but the computation time will decrease. All FFTW plans for computations will be created with flag `ESTIMATE` so I expect bathtubs will be presented.

Graphical representation of measured values for all numbers of threads are shown in figure 3.16. The horizontal axis represents row length `N` of the matrix. The vertical axis represents durations of FFT computations for given matrix dimension and number of threads. I will read an example of measured values. Duration of FFT computations for matrix with row length `N` 5 for

- one thread is 1630×10^6 CPU cycles,
- for two threads is 2400×10^6 CPU cycles,
- for three threads is 620×10^6 CPU cycles and
- for four threads is 1180×10^6 CPU cycles.

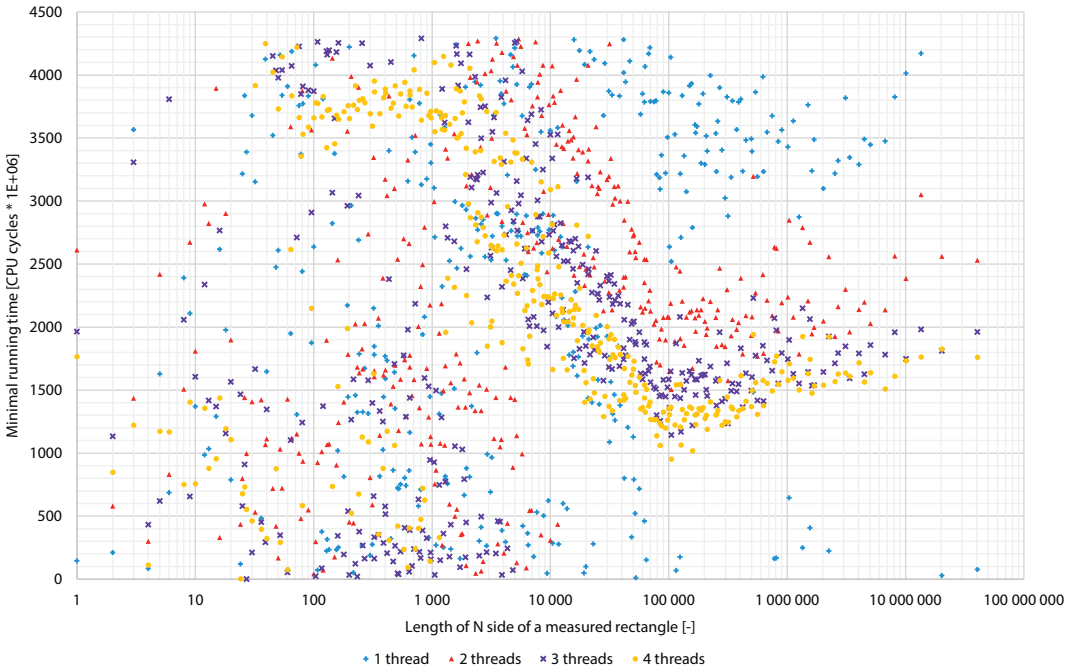


Figure 3.16: Duration of FFT computation of area 40 435 200 for one, two, three and four threads

We can separate measured values from figure 3.16 into two areas. The first one is in lower left half of the figure with measured values up to row length `N` 10 000. Measured durations in that area are very diverse for all thread counts. Only discernible fact is that nearly all measured times for three threads are located in row length `N` interval [1000; 5000] with very short FFT durations.

Second area is in top right half of the figure in row length `N` interval [2000; 100 000 000]. Measured FFT durations for two, three a four threads are much more stable. Those threads contain bathtubs with the centre in row length `N` 180 000. Results of one thread are rather chaotic. It doesn't seem that bathtub for one thread exists; the measured values don't form a solid shape and their mutual positions are too distant. What is even more interesting is that measured FFT durations were computed in very short times that even the other test runs with

more threads didn't reach. For example duration of FFT with one thread for row length `N` 52 000 (located under bathtubs) took only 15×10^6 CPU cycles which is **100× faster** than for four threads. I will further investigate this behaviour in section 3.7.2.

After measuring the durations I wanted to compute a performance gain against durations of one thread. But after seeing the unpredictability of its measured times I computed a performance gain of three and four threads against measured FFT durations of two threads. Table 3.12 shows excerpt from the computed data.

row	FFT duration [$\times10^6$ CPU cycles]			speed-up multiplier	
length N	2 threads	3 threads	4 threads	3 threads	4 threads
1	2611	1964	1766	0.75	0.68
2	580	1134	848	1.96	1.46
3	1435	3308	1223	2.31	0.85
4	300	432	113	1.44	0.38
5	2419	619	1174	0.26	0.49
6	832	3808	1169	4.57	1.40
8	1506	2057	752	1.37	0.50
9	2674	656	1406	0.25	0.53

Table 3.12: Computed performance multipliers of three and four threads from durations of FFT computations

I will show an example of how I got the two performance multipliers for row length `N` 1. The multipliers were computed as

$$\text{Multiplier}_{3\text{threads}} := \frac{\text{Duration of 3 threads}}{\text{Duration of 2 threads}} = \frac{1964}{2611} \doteq 0.75$$

$$\text{Multiplier}_{4\text{threads}} := \frac{\text{Duration of 4 threads}}{\text{Duration of 2 threads}} = \frac{1766}{2611} \doteq 0.68$$

Result of $\text{Multiplier}_{3\text{threads}}$ represents the fact that the time needed to compute the matrix with row length `N` 1 is 0.75× of time that would be needed for computation with two threads. For $\text{Multiplier}_{4\text{threads}}$ is required time 0.68× of time that would be needed for computation with two threads.

I have put all measured values into figure 3.17. The horizontal axis represents row length `N` of the matrix. The vertical axis represents performance multiplier for given matrix dimension and number of threads. The lower the multiplier, the shorter the duration of FFT and vice versa. I will read an additional example of computed multipliers from figure 3.16 that is different from the example for table 3.12. The multiplier for the matrix with row length `N` 10 for three threads is 0.88× of FFT duration of two threads and for four threads is 0.42× of FFT duration of two threads.

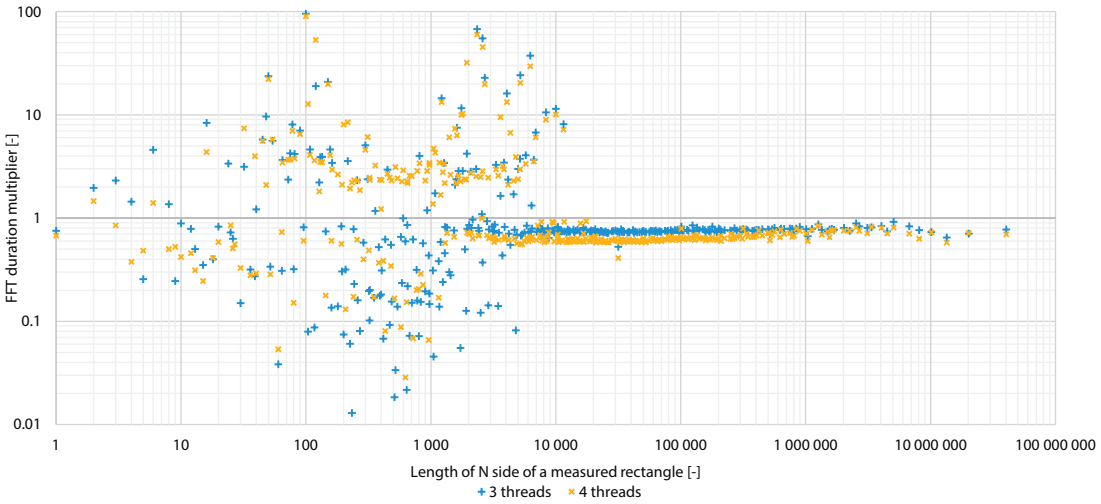


Figure 3.17: Performance multipliers of three and four threads based on FFT durations of two threads of area 40 435 200

Figure 3.17 provides us with the better representation of a performance multiplier than table 3.12. We can see that the figure can be divided into left and right part. In the left part the measured values have a great spread. More than half the durations for three threads are located in the lower part (multiplier <1) and more than half the durations for four threads are located in the upper part (multiplier >1) of the figure. The right part have consistent durations for both tests; the average multiplier is 0.77 for three threads and 0.65 for four threads.

We have seen in figures 3.16 and 3.17 that two, three and four threads contain bathtubs. Because the bathtub of four threads has the lowest stable FFT durations, I wanted to know how much can be the result enhanced by using a different planning flag. Figure 3.18 shows durations of FFT computations for four threads with planning flags `ESTIMATE` and `MEASURE`. The horizontal axis represents row length `N` of the matrix. The vertical axis represents durations of FFT computations for given matrix dimension and planning flags with four threads. I will read an example of measured values. Duration of FFT computations for matrix with row length `N` 10 with flag `ESTIMATE` is 700×10^6 CPU cycles and for flag `MEASURE` is 4020×10^6 CPU cycles.

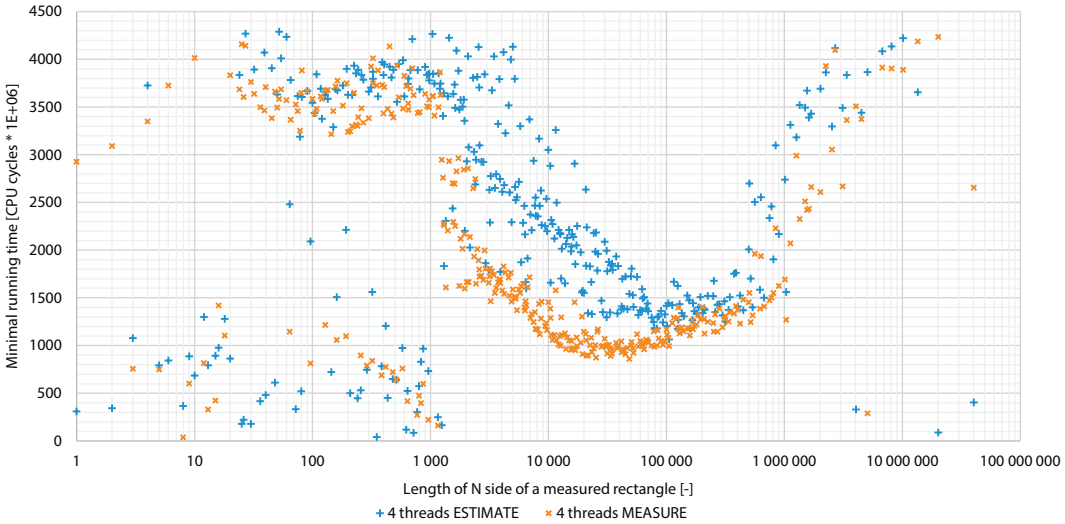


Figure 3.18: Durations of FFT computations for four threads with planning flags `ESTIMATE` and `MEASURE` of area 40 435 200

In chapter 3.4 we found that results of planning with flag `MEASURE` produced better computation speeds than with `ESTIMATE` flag. I would expect the results seen in figure 3.18 to have similar outcome. That's why measured results surprised me a bit. Differences of durations between both flags in range [1; 1000] are very low. The range [1000; 1 000 000] contains bathtub. As expected, in the first half of the bathtub flag `MEASURE` has better results. In the second half the differences between measured times are very low again.

If we would try to find out best row length `N` with equation (15) and multiplier 19 (chosen as the result of chapter 3.3) to be used in the planner, we would choose row length `N` 124 416 with `ESTIMATE` duration 1625×10^6 CPU cycles and `MEASURE` duration 1286×10^6 CPU cycles, both located at the bottom of the bathtub amongst shortest FFT durations.

The outcome of this section is that FFTW supports native way of in parallel processing of FFTs and using the native way is advantageous for us. Test results of two, three and four threads came out as expected; the more threads used in computation, the higher speed-up of computation is. Additionally bathtubs are present for more than one thread. We can combine this discovery with results of chapter 3.3 to gain even faster results of FFT computations. Unstable results of one thread will be examined in section 3.7.2.

3.7.2 Single thread behaviour

Next set of tests will follow results shown in figure 3.16 and will be focused on figuring out strange FFT results spread of one thread. As already stated in section 3.7, some of the FFT durations of one thread had much shorter time than four threads. If we would find some steps that would stably find those short times, we could use only one thread instead of four (or more) and still substantially speed-up FFT computations. I expect that the findings won't help me create faster FFTs but I will gain some knowledge of one thread operations.

At first I needed to find out if measured values are stable across multiple test runs. Figure 3.19 was created the same way as figure 3.16 but with only one thread and ran four times. The horizontal axis represents row length `N` of the matrix. The vertical axis represents durations of FFT computations for given matrix dimension. I will read an example of measured values for matrix with row length `N` 16. All four test runs were finished in 2600×10^6 CPU cycles. The absolute majority of measured FFT durations are similar for all runs.

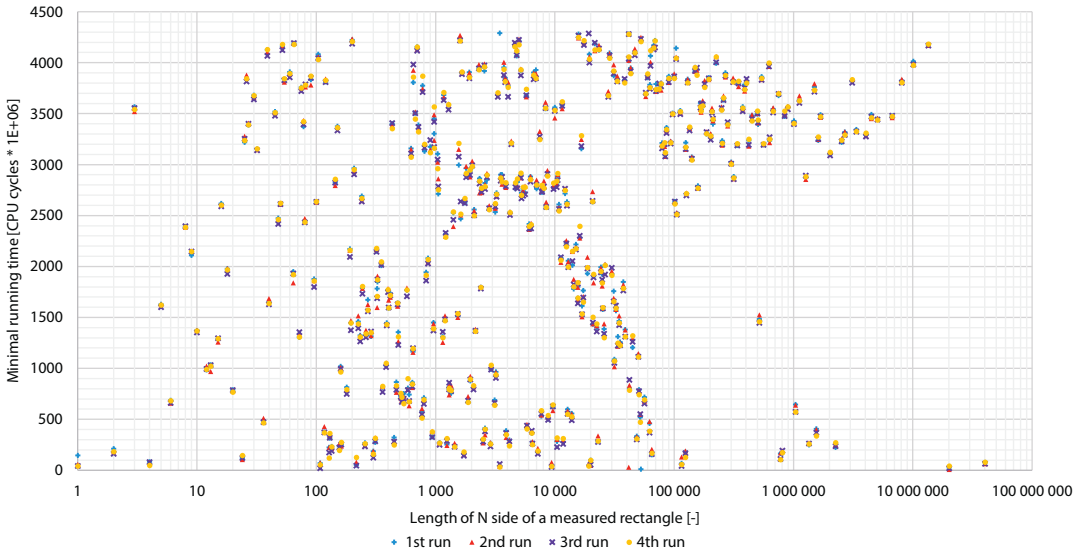


Figure 3.19: Multiple measures of FFT durations of area 40 435 200 with one thread

In section 3.7 we found from figure 3.16 that one thread seemingly doesn't have bathtub. Next test will prove or disprove such statement. Results of the test in figure 3.20 shows durations of FFT computations for one thread with planning flags `ESTIMATE` and `MEASURE`. The horizontal axis represents row length `N` of the matrix. The vertical axis represents durations of FFT computations for given matrix dimension and planning flags with one thread. I will read an example of measured values. Computation of FFTs for matrix with row length `N` 10 with flag `ESTIMATE` took 1560×10^6 CPU cycles and for flag `MEASURE` took 3515×10^6 CPU cycles.

Figure 3.20 clearly shows that with flag `MEASURE` bathtub exists. We can estimate the existence of bathtub for flag `ESTIMATE` but its presence is not shown because the planner created the plan with different set of codelets and measured values didn't form visual bathtub.

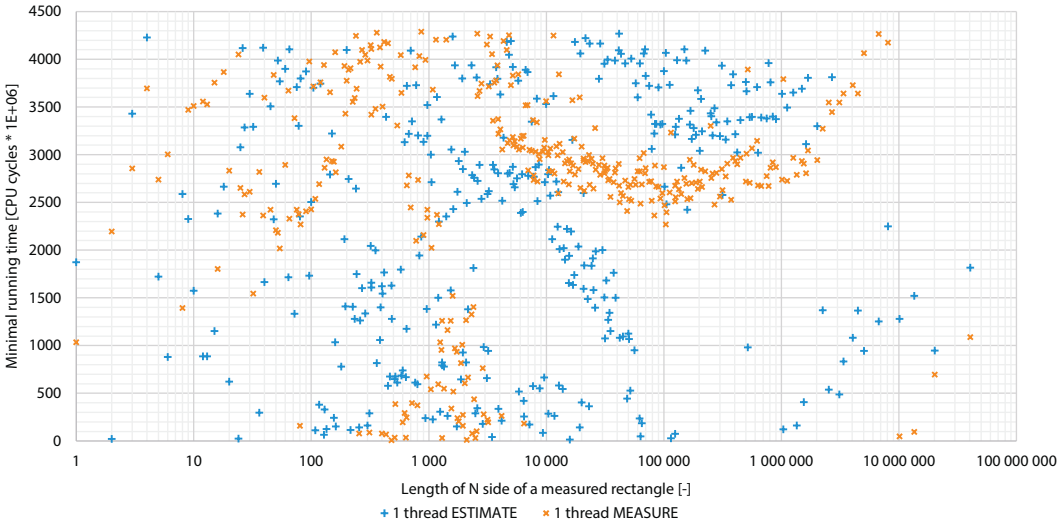


Figure 3.20: Durations of FFT computations of area 40 435 200 for one thread with planning flags `ESTIMATE` and `MEASURE`

As for the last test of this section I wanted to test one thread with flag `ESTIMATE` with disabled processor cache. The reason was that one thread has whole shared processor cache for itself and it doesn't have to share it with other threads. That could explain such low measured FFT durations. The test was run on test environment 3 (see beginning of chapter 3). The measured area is much smaller than in previous tests, only 106 470. This change was necessary because the computer from environment 3 has much lower computing power than computers in environments 1 or 2 and FFT computations in environment 3 are about two orders slower.

Results of the test in figure 3.20 show durations of FFT computations for one thread with enabled and disable 64 kB L1, L2 processor cache. The horizontal axis represents row length `N` of matrices with area 106 470. The vertical axis represents durations of FFT computations for given matrix dimension. I will read an example of measured values. Computation of FFTs for matrix with row length `N` 30 with disable cache took 1580×10^6 CPU cycles and for enabled cache took 42×10^6 CPU cycles.

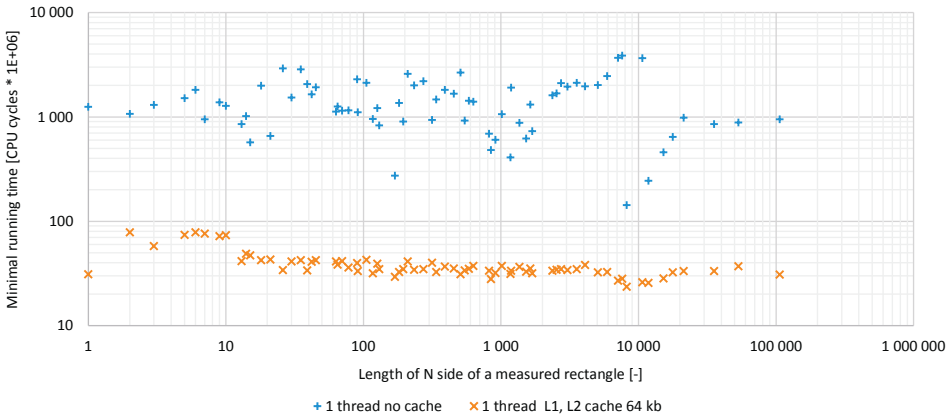


Figure 3.21: Durations of FFT computations for one thread with/without enabled processor cache

The results of test shown in figure 3.21 imply three facts. First fact is that the shape and FFT durations of measured values are completely different. It seems that FFTW natively expects an use of a processor cache. Without the cache, measurement is loosing significance. Second fact is that the computation without the cache was 40× slower than with cache. This value was computed as average of divisions of measured durations. I would like to note that equation (10) for transformation of CPU cycles to seconds doesn't work in this test because it is not set for test environment 3.

The results confirmed my expectations about not finding the right reason for substantial speed-up of FFT durations. Combined with results of test set in chapter 3.3 I know that fast results of one thread planner are dimension bound and are quite stable in course of multiple test runs.

3.8 Primes speed

In this test I want to determine which prime number from formula (11) is computed in the fastest time. I can compute decomposition of any number and see how many exponents each prime number get. If one prime number would be computed faster then the others I would favour it when choosing suitable fftw-friendly numbers for FFT computations. I expect sequence 2^a to be faster than the others because as written in section 3.5, "transforms whose sizes are powers of two are especially fast".

Similarly to test in section 3.4.2 I have ran speed tests on a rectangular areas ranging from 2^{10} to 2^{28} for every prime number contained in formula (11). The area 2^{28} was the largest I could test. With 8 GB of a physical memory and 16 bytes needed for a representation of a complex number I made use of 4 GB of the memory. Other memory was either unused or reserved by the OS.

From every test I had to choose only one measured value. I used equation (15) with the `MinRowLengthMultiplier` value 19 (result of test 3.3) to simulate the best choice of the planner with `ESTIMATE` flag. With this setup the measured values should have the lowest FFT durations.

Results of the test in figure 3.22 show durations of 2-D FFT computations for each prime number. The horizontal axis represents row length `N` of the matrix. The vertical axis represents durations of FFT computations for given matrix dimension. I will read an example of measured values in row length `N` interval [100 000; 200 000] from left to right. Measured FFT computation

- row length `N` 7^6 (117 649), took 1529×10^3 CPU cycles,
- row length `N` 2^{17} (131 072), took 1459×10^3 CPU cycles,
- row length `N` 11^5 (161 051), took 6213×10^3 CPU cycles and
- row length `N` 3^{11} (177 147), took 2872×10^3 CPU cycles.

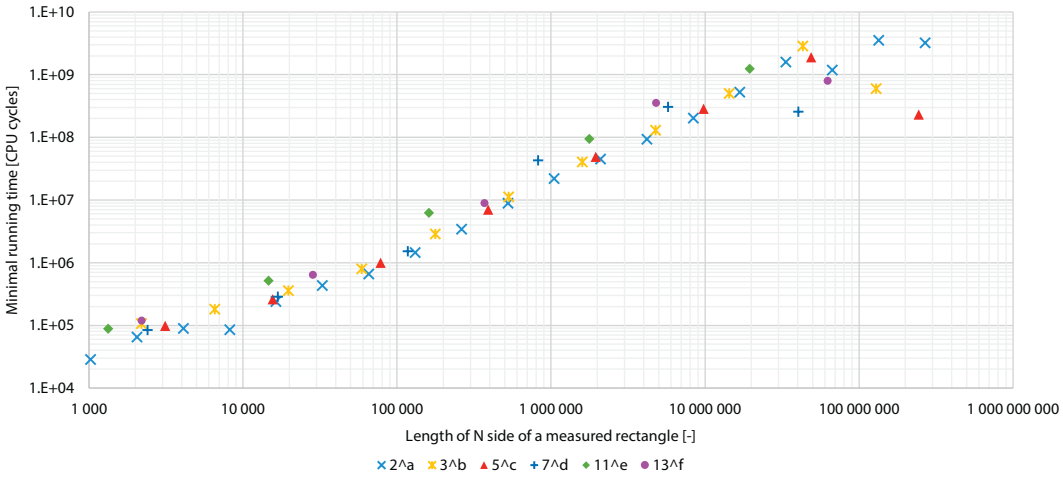


Figure 3.22: Durations of 2-D FFT computations for sequences 2^a, 3^b, 5^c, 7^d, 11^e and 13^f

From figure 3.22 we can see that sequences 2^a, 3^b, 5^c and 7^d were computed with very similar times and our expectations weren't fulfilled. Additionally sequences 11^e and 13^f are a bit slower than the sequences with lower prime numbers. At the end of the measured values we can see noticeable variances of FFT durations. The low times are caused by the planner that chose more efficient set of codelets for computations. This effect is described in more details in section 3.4.2. The conclusion of this test is that there are no computation benefits from preferring the tested prime number sequences.

4 Optimization planner

In this chapter I will describe my program *Optimization Planner* (OP). Its goal is to find the "right" number of divisions of *source* and *target* and "right" dimensions of divided parts resulting in the fastest FFT computation times. To obtain such results OP uses all knowledge we learned in previous chapters about how FFTW works and what results can we expect. At first we will describe which steps OP uses. After that some of the steps will be examined in more detail. At the end we will run a test that will tell how much efficient OP is.

OP successively passes through several steps:

1. Loading and parsing input parameters
2. Creation of data structures
3. Reduction of memory requirements
4. Search for the optimal division
5. Compaction of results

As the first step we will load data into OP. It expects five parameters on input. Two dimensions of a discretized source M×N, two dimensions of a discretized target P×Q and a one parameter for maximal available memory a light propagation process can use (already described in chapter 1).

If all input parameters were successfully parsed, we can continue with step 2 and create needed data structures. Simplified scheme of the data structures is shown in figure 4.1. The *propagation process* that already parsed input parameters will create *source* and *target*. We can see from the figure that *source* and *target* are composed of one to four *locations*. Every location can be divided into numerous parts (see figure 4.3b) but OP registers them only as one *tile* that aggregates information about divided parts. Detailed description of locations and their tiles can be found in section 4.1.

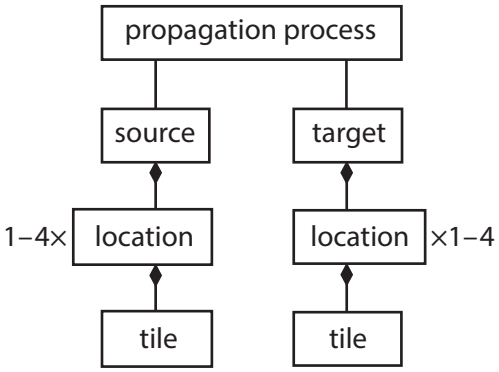


Figure 4.1: High level architecture of *Optimization Planner*

4.1 Four locations

In this section I will describe what a location is, why have we introduced them and how are they used. A location represents some area of *source* or *target*. Every single location has name, i.e. `Main`, `Top`, `Right` and `Small` and unique characteristic. From a programme standpoint a location stores information about the number of its horizontal and vertical divisions. Positions and shapes of all locations can be seen in figure 4.2.

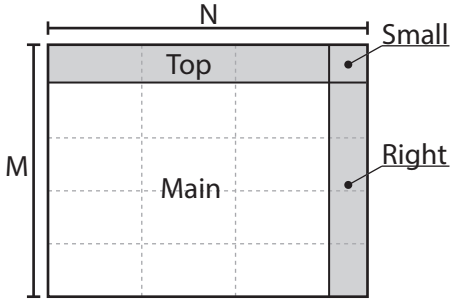


Figure 4.2: Division of *source*/*target* area into four locations: `Main`, `Top`, `Right` and `Small`

I will go through examples of all possible combinations of the locations. All following examples in this section will be presented for *source*. For all examples to be valid for *target*, replace all occurrences of words [*source*; M×N] with words [*target*; P×Q].

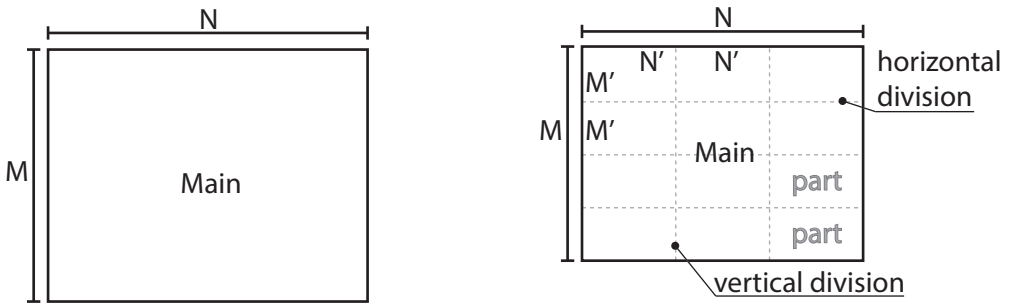
Figure 4.3a shows the standard situation for every newly created *source*. *Source* with dimensions M×N is composed of one location `Main`. In case we will be able to propagate a light from M×N *source* onto P×Q *target* in one go (without the need to split either one into parts), both *source* and *target* will have only one `Main` location with one part (see figure 1.2).

If the required memory is too large, we will split *source* into parts (figure 4.3b). The exact process of splitting will be described in section 4.2. For now, let us assume we know it. *Source* has still only one `Main` location that has been divided into 4×3 parts but **only one** tile. A tile stores information about the parts' dimensions and number of parts. For example if we have *source* with dimensions M×N 120×120 and its `Main` location will be divided into 12 parts M'×N' 30×40, information stored for `Main` location will be

- horizontal divisions: 3,
- vertical divisions: 2.

Information stored for tile of `Main` location will be

- height: 30,
- width: 40,
- number of parts: 12.

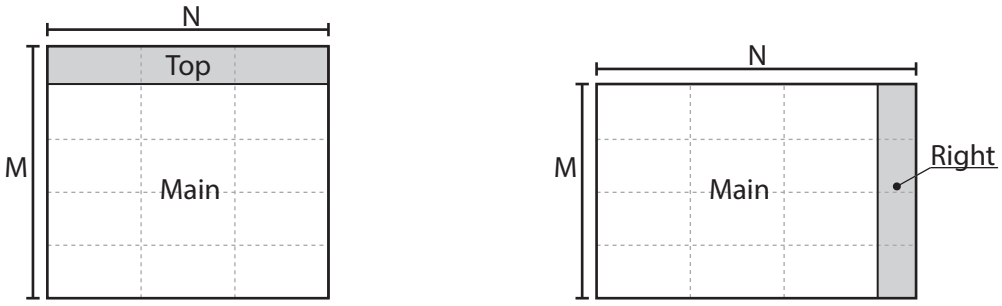


(a) *Source* with one undivided location `Main`, one part and one tile

(b) *Source* with one location `Main` divided into 4×3 parts and one tile

Figure 4.3: Possible divisions of location `Main`

If we would try to divide *source* and it wouldn't be possible to do so without a remainder, locations `Top` (fig. 4.4a) or `Right` (fig. 4.4b) can be established for reminding area. The tile for `Top` location have always the same width as tile for `Main` location. Initial height of the `Top` tile is the remainder after the division. Alternatively the tile for `Right` location have always the same height as tile for `Main` location. Initial width of the `Right` tile is the remainder after the division. Reasons for those conditions are described in section 4.3.



(a) Establishment of location `Top`

(b) Establishment of location `Right`

Figure 4.4: Establishment of locations `Top` and `Right`

After dividing `Main` location we can find that remainders are present for both width and height and we need `Top` and `Right` locations. The presence of the locations implies that at imaginary intersection of the locations (top right corner of *source*) will be established new `Small` location. This location has the smallest area of all locations and its initial weight and height are set by the height of `Top` tile and the width of `Right` tile. Specificity of `Small` location is that it cannot be divided and always has only one part (its tile has parameter "number of parts" set to one). The setup with all four locations has been already shown in figure 4.2.

4.2 Reduction of memory requirements

Now we know all architectural parts of OP and can continue with a description of the optimization process. In this step we will divide `Main` locations of *source* and *target* in such

way that a light propagation will use maximum of an allowed memory. All operations in this section submit to `Main` location because its area is the largest (i.e. the most important) of *source* and *target*. Dimensions of other locations are computed indirectly and change with any modification of `Main` locations.

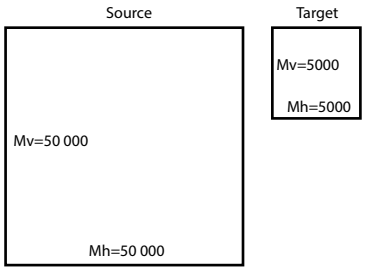
The figures of *source* and *target* in this chapter (such as figure 4.5a) are not exact models with correct dimension ratios but rather are meant as an illustration of a division process. I will describe used symbols found in the figures. In figure 4.5a we can see *source* and *target* areas with sides labelled as `Mv` and `Mh`. The first letter denotes a location for which the label belongs to: `M` for `Main`, `T` for `Top`, `R` for `Right` and `S` for `Small`. The second letter denotes the dimension of the location's tile. The letter `v` stands for a vertical dimension and `h` for a horizontal dimension. For example label `Mv` denotes the length of vertical dimension of `Main` tile.

I will show the process of the reduction on the example from chapter 1. I will quickly sum up and expand the example. We will try to propagate a light from *source* onto *target*. We have five input parameters; *source* `Main` tile M×N 50 000×50 000 samples, *target* `Main` tile P×Q 5000×5000 samples (fig. 4.5a) and our computer has 16 GB of an available memory from which we can only use 14 GB. With the knowledge that an one complex number C_n takes up 16 bytes of memory, the propagation of *source* and *target* will need

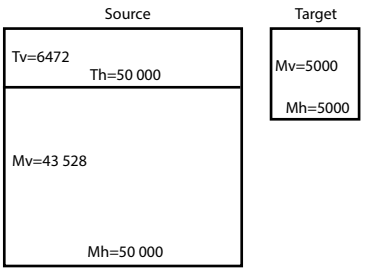
$$(M+P-1)\times(N+Q-1)\times 2\times C_n = 54\,999\times 54\,999\times 2\times 16 \approx 90\,\text{GB}$$

of an available memory. 90 GB is much more than we can accommodate and it's necessary to perform a division. OP will come to the same conclusion and will continue with additional division details:

1. $54\,999\times 54\,999\times 2\times 16 \approx 90\,\text{GB} > 14\,\text{GB}$; we have to divide.
2. *Source* `Main` area 50 000×50 000 > *target* `Main` area 5000×5000; we will divide *source* `Main` area.
3. *Source* `Main` horizontal dimension (50 000) ≦ *source* `Main` vertical dimension (50 000 × 19 = 950 000); we are going to divide *source* `Main` area horizontally.



(a) Initial setup of *source* M×N 50 000×50 000 and *target* P×Q 5000×5000



(b) First division of *source* into two parts, `Main` and `Top`

Figure 4.5: Beginning steps in the division of *source*

The multiplier of 19 (average minimal bathtub ratio from test 3.3) was added to ensure parts of `Main` have optimal dimensions for processing by FFTW. Now we will find out how the horizontal division is performed. A division have to be performed fast and in such way that we use the most of our allowed memory. The question is, how large should the division be. Figure 4.6 shows some of possible division factors:

- If we would divide *source* dimension in half (i.e. 50 000, 25 000, 12 500,…) the reduction of length is too steep and after the second division we have only a quarter of the original length. So the division factor of two is too large. We have to use a smaller one.
- After a bit of thinking I chose division factor of $\sqrt[5]{2} \doteq 1.15$. At first I wanted to divide the length as $50\,000/(\sqrt[5]{2}i)$, where $i \in \mathbb{N}$. The results with the multiplied root weren't as good as I expected because the reduction of the length was still very steep as in the previous case.
- After that I combined both approaches. Starting with $50\,000/\sqrt[5]{2}$ I have first division result 43 528. The second division is computed as $43\,528/\sqrt[5]{2}$ and so on. From figure 4.6 we can see that this method produces more gradual results. This division process is satisfactory and is used on *source* and *target*.

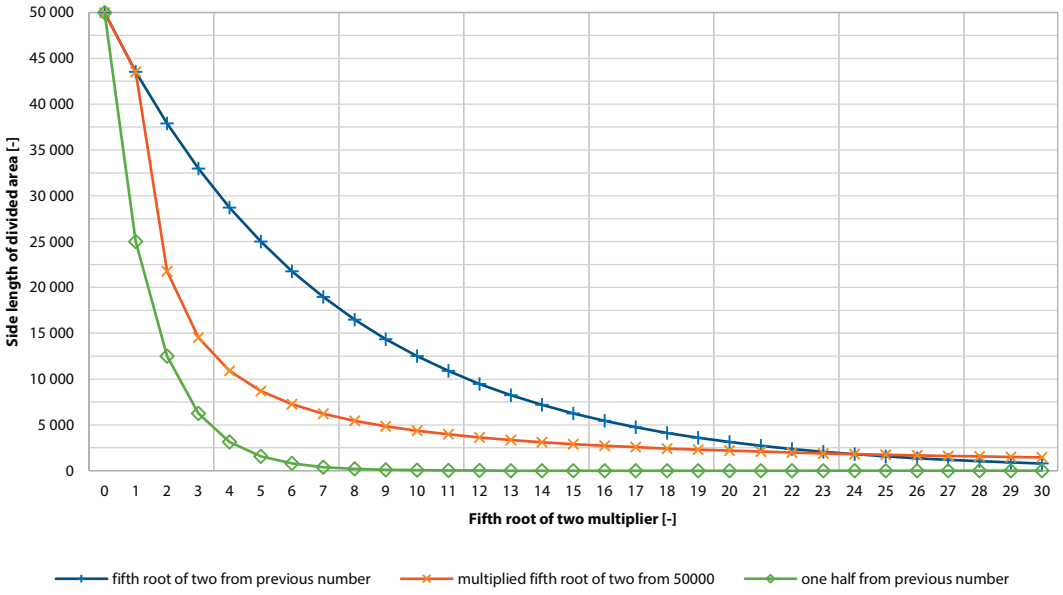


Figure 4.6: Three possible methods of division of length 50 000. Only points "multiplied fifth root of two from 50000" use multiplier for their calculation. The others are multiplier independent. All of the three results can be in one figure because every point of the horizontal axis represents "one next division" regardless of an used computation procedure

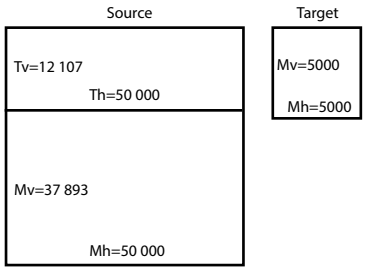
Now that we know the division process we can finally divide *source* into two parts. The result of the division is shown in figure 4.5b. *Source* became divided into two locations, `Main` and `Top`. Both locations consist of one part so the tiles have the same dimensions as the parts.

The division process is not finished and we will continue with next steps. With the current state of *source* and *target* shown in figure 4.5b OP chooses following steps:

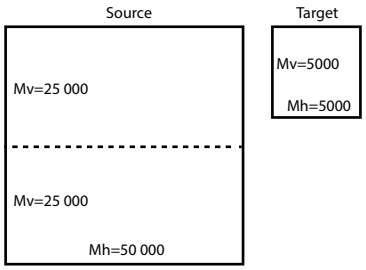
1. $48\,527 \times 54\,999 \times 2 \times 16 \approx 79.5\,\text{GB} > 14\,\text{GB}$; we have to divide.
2. *Source* `Main` area $43\,528 \times 50\,000$ > *target* `Main` area 5000×5000; we will divide *source* `Main` area.
3. *Source* `Main` horizontal dimension ($50\,000$) $\leqq$ *source* `Main` vertical dimension ($43\,528 \times 19 = 827\,032$); we are going to divide *source* `Main` area horizontally.

Figure 4.7a shows results of the second division. The vertical dimension of `Main` tile has decreased and the vertical dimension of `Top` tile has increased. No other changes were made.

After several additional divisions important changes occur. We can see in figure 4.7b that `Top` location has been removed and `Main` location divided into two parts. If we wouldn't do such change the vertical dimension of `Top` tile would have equal length to vertical dimension of `Main` tile. Such operation is forbidden because all non-`Main` areas are *remaining*; that means `Top` tile has to have shorter vertical dimension than `Main` tile.



(a) The second division of *source* into two locations, `Main` and `Top`



(b) Change of *source* after several divisions. `Top` location has been removed and `Main` divided into two parts

Figure 4.7: Middle steps in the division of *source*

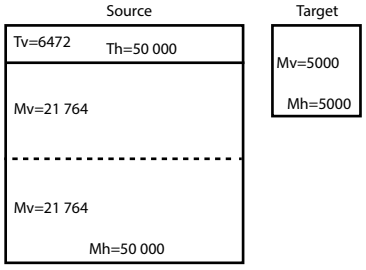
Continuing with the division from figure 4.7b OP chooses:

1. $25\,000 \times 54\,999 \times 2 \times 16 \approx 41\,\text{GB} > 14\,\text{GB}$; we have to divide.
2. *Source* `Main` area $25\,000 \times 50\,000$ > *target* `Main` area 5000×5000; we will divide *source* `Main` area.
3. *Source* `Main` horizontal dimension ($50\,000$) $\leqq$ *source* `Main` vertical dimension ($25\,000 \times 19 = 475\,000$); we are going to divide *source* `Main` area horizontally.

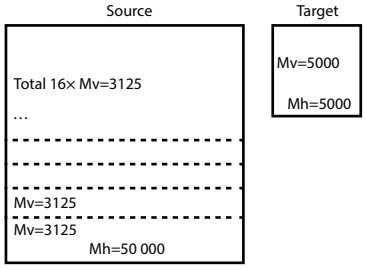
The result of this division can be seen in figure 4.8a. By changing `Main` tile's vertical dimension, dimensions for all `Main` parts are also changed. Apart from return of `Top` location nothing has changed.

After many more divisions we gain final result as seen in figure 4.8b. *Source* with only `Main` location has been divided into 16 parts each with height 3125 and width 50 000. *Target* with only `Main` location hasn't been divided, meaning only one part with height 5000 and width 5000 exists. If we will try to run OP on such setup, we find out

1. $8124 \times 54\,999 \times 2 \times 16 \approx 13.3\,\text{GB} < 14\,\text{GB}$; **no need** to divide. Algorithm finished.



(a) Further division of `Main` location of *source*



(b) Final division of *source* and *target* into two locations and 17 parts (16 *source* and one *target*)

Figure 4.8: Final steps in division of *source*

At the beginning of this section we had our *source* and *target* each with one `Main` location and one tile with dimensions as large as *source* and *target*. After many divisions we have prepared *source* and *target* to be processed in a shared memory. In following chapter 4.3 we will see how OP prepares *source* and *target* for fast transformations.

4.3 Search for the optimal division

With *source* and *target* divided, OP will find the best fftw-friendly dimensions for all locations' tiles. The reason for the use of fftw-friendly numbers is lower computation times of FFTs (result of tests 3.1 and 3.2). The algorithm varies according to the number of locations in *source* and *target*. If each have just `Main` location, only one optimization $source_{\texttt{Main}}$ to $target_{\texttt{Main}}$ will be held. On the other hand if both *source* and *target* will have all four locations, OP will have to independently optimize 4×4 pairs of tiles.

I will described `Main` to `Main` optimization (first location is meant for *source*, second for *target*). After the division process `Main` tiles of *source* and *target* can have non-fftw-friendly dimensions. This state is not desirable so the first step is to find whether *all* dimensions of `Main` tiles

$$(source\ \texttt{Mv} + target\ \texttt{Mv} - 1) \times (source\ \texttt{Mh} + target\ \texttt{Mh} - 1)$$

are fftw-friendly. If not, it will be necessary to transform them into fftw-friendly. The transformation is performed by **reduction of lengths** of `Main` tiles' dimensions. The following example shows the algorithm that reduces the lengths by *one* point shaping the tiles into bathtub optimal shape.

I will use the previous example in figure 4.8b to show one step of how OP optimizes `Main` areas:

1. $8124 \times 54\,999 = 446\,811\,876$; such number is not fftw-friendly, try to transform them into fftw-friendly.
2. *Source* `Main` area $3125 \times 50\,000 >$ *target* `Main` area 5000×5000; reduce *source* dimension
3. *Source* `Main` horizontal dimension ($50\,000$) $\leqq$ *source* `Main` vertical dimension ($3125 \times 19 = 59\,375$); we are going to reduce *source* `Main` vertical length by one point and continue from first step.

For the same reason as in section 4.2 the multiplier of 19 (average minimal bathtub ratio from test 3.3) was added to ensure parts of `Main` have optimal dimensions for processing by FFTW. After many iterations we finally find fftw-friendly dimensions for both *source* and *target*. One of such results is shown in figure 4.9. We can see that our original division from figure 4.8b was noticeably different from our current state.

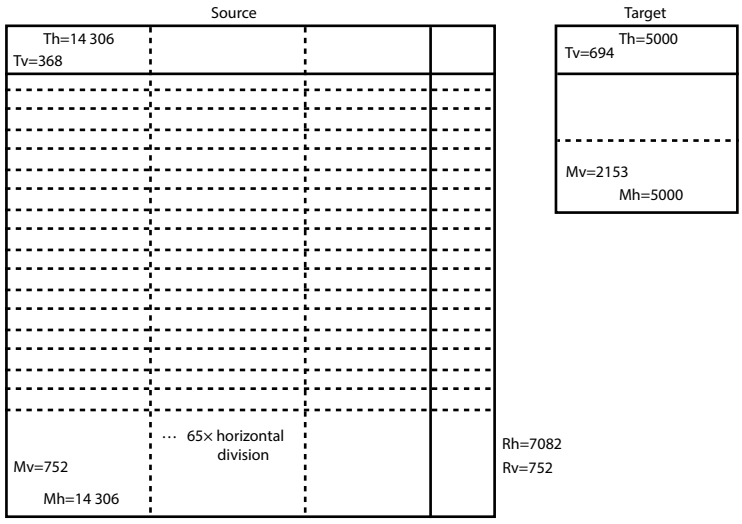


Figure 4.9: Possible result of transformation of *source* and *target* dimensions into fftw-friendly ones

Additionally for *source* and *target* from figure 4.8b we needed

$$8124 \times 54\,999 \times 2 \times 16 \approx 13.3\,\text{GB}$$

of available memory. After the fftw-friendly reduction we need only

$$2904 \times 19\,305 \times 2 \times 16 \approx 1.7\,\text{GB}$$

of available memory which is approximately eight times less than in non-optimized case. The result of `Main` reduction generally varies greatly. With only small change of *source* or *target* dimensions, the reduction results can use from 0.5 GB to 12 GB of available memory. The culprits are fftw-friendly numbers that are more sparse in higher values than for lower values. This implies the probability for choosing the higher fftw-friendly number is lower.

After the `Main` tiles are reduced to fftw-friendly dimensions, they are **locked**. That means further manipulation with the dimensions of `Main` tiles of *source* and *target* are not permitted. With this result the `Main` to `Main` optimization is complete. From figure 4.9 we can see that other optimizations that awaits OP are

- *Source* `Top` to *target* `Main`,
- *Source* `Right` to *target* `Main`,
- *Source* `Small` to *target* `Main` and
- *Source* `Main` to *target* `Top`.

Up to this moment, only `Main` tile pairs were fftw-friendly optimized. The other locations are not optimized and we will try to optimize them.

In first four cases one location from the pair is locked `Main`. We will represent first case "*Source* `Top` to *target* `Main`" in figure 4.10 to show the next optimization procedure. In section 4.1 we said that horizontal dimension of *source* `Top` tile cannot be manually changed and is set automatically by *source* `Main` tile's horizontal dimension. From the figure we can see that both `Main` locations are locked and that means horizontal dimension of *source* `Top` location is also locked. Fortunately we don't need to change those dimensions in any way because both dimensions of `Main` tile are optimized and horizontal dimension of *source* `Top` tile was set just as for `Main` tile, so it's also optimized.

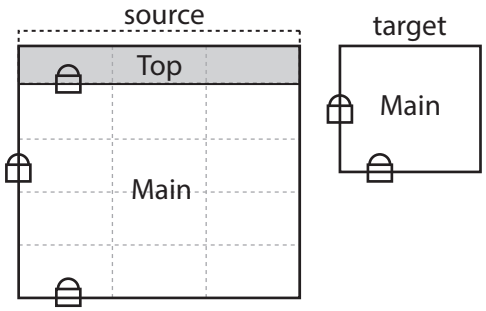


Figure 4.10: Optimization of locked `Main` tile and `Top` tile

Only side that is not optimized is *source* `Top` tile vertical dimension. As opposed to the `Main` to `Main` transformation, this transformation is performed by **addition of lengths** to `Top` tile's unlocked dimension. The process of addition can be described in following steps:

1. OP checks that actual dimensions produce fftw-friendly results. If not,
2. OP adds one point to *source* `Top` tile unlocked dimension and checks if the changed dimensions are fftw-friendly. We can allow to add one point to the dimension because `Top` tile can expand upwards (into dashed area shown in figure 4.10) and fill empty spaces with zeros without any change to the data. This process is repeated until fftw-friendly dimensions are found or
3. lengthened dimension reaches length of `Main` tile vertical dimension. We have already noted in section 4.2 that *remaining* locations' tiles cannot have the same dimensions as `Main` tile. This case is an exception because `Main` dimensions are locked and won't be changed. And because now the optimized `Main` tile has the same dimensions as `Top` tile, the `Top` tile is also optimized.

We have described optimization process for our first case from the list. For other three in the same category is the process analogous only with different unlocked dimension.

The second category of optimization pairs are shown in following list and have only non-`Main` location. In comparison with first category the non-`Main` pairs have two unlocked dimensions; One for *source* and other for *target*. Optimization process is analogous to first category but instead of adding only to one unlocked dimension, algorithm chooses which side to lengthen.

- *Source* `Top` to *target* `Top`,
- *Source* `Right` to *target* `Top`,
- *Source* `Small` to *target* `Top`.

When all pairs are optimized OP condenses all results and offers them for use. With this last step finished, the program ends.

4.4 Planner efficiency

In this test I want to determine effectiveness of *Optimization Planner* (OP). The test will consist of two parts. In the first part I will perform FFT computation from a source M×N onto a target P×Q with restricted access to available memory. This propagation will be computed by brute force with every permutation of dimensions on small areas; Otherwise I won't be able to compute all possibilities. In the second part I will start OP with the same setting as for the first part and compare results of brute force run. The reason for this test is to see how much different will be the optimal division of brute force run and "optimum" returned by OP. I expect the result returned by OP to be distant at most one hundred places from measured optimal speed.

First part was ran with parameters source M×N 48×48, target P×Q 48×48 with memory 8192 (2^{13}). The test ran 4 hours and 17 minutes and computed 153 902 permutations. The number is so low because many combinations weren't allowed because of insufficient allowed memory. The fastest division is source M×N 8×16, target P×Q 8×16 with duration 5 938 688 CPU cycles. The slowest division is source M×N 2×2, target P×Q 39×2 with duration 700 481 536 CPU cycles. The slowest computation was approximately 118× slower than the fastest calculation. An excerpt from measured values is shown in table 4.1.

Place	Column length M	Row length N	Column length P	Row length Q	FFT duration [CPU cycles]
1st	8	16	8	16	5 938 688
2nd	16	8	16	8	5 967 744
3rd	7	16	9	16	7 427 920
4th	9	16	7	16	7 453 440
5th	16	9	16	7	7 456 000
249th	4	35	3	35	14 392 576
253rd	4	16	11	16	14 425 536
255th	4	32	2	32	14 427 520
277th	4	33	3	32	14 678 136
299th	4	32	3	33	14 745 720
153 899th	2	2	2	39	666 865 664
153 900th	2	39	2	2	671 649 792
153 901st	39	2	2	2	698 982 400
153 902nd	2	2	39	2	700 481 536

Table 4.1: Excerpt from measured values of source M×N 48×48, target P×Q 48×48 with memory restricted to 8192 bytes

Second part was ran in OP with the same parameters as in the first part. The resulting "optimum" was computed by OP in source M×N 4×32, target P×Q 2×32. This duration of those dimensions were computed in 14 427 520 CPU cycles. By comparing computed dimensions with results from the first test the OP's result is placed 255th fastest from 153 902 measured permutations.

My expectations weren't fulfilled because I didn't expect so large variety of measured durations. Still being only approximately 2.4× slower than the best permutation is satisfying and 255th place from 153 902 is very good result.

5 Conclusion

At the end of the description of the light propagation process we were acquainted with a lack of an available computer memory needed for a propagation of high quality computer generated holograms. It's known that by dividing a source and a target into smaller parts we are able to run a light propagation process in terms of the number of parts and utilize the available memory. What we didn't know is how should be the division performed, so that it's as fast as possible. This problem already solved Mr. Nedved in his research. After the examination of his results I reached the conclusion that some of his assumptions are not (entirely) correct and test results too vague. For those reasons I conducted my own research and later retested Mr. Nedved's results.

Because of a heavy use of fast Fourier transforms in a light propagation we needed to find the best library that would fulfil our diverse requirements. I searched for more or less known FFT libraries and created the list of candidates. After disqualifying all candidates that didn't meet our requirements only two most prospective libraries "Fastest Fourier Transform in the West" (FFTW) and "Intel Math Kernel Library and its Fast Fourier Transform Routines" (Intel MKL) remained. By converting possibly biased and unbiased measured speeds from various tests into microseconds we were able to compare speeds of the libraries. At the end FFTW library was proclaimed as the winner. Through various tests we examined behaviour of FFTW and found many interesting features:

- Only fftw-friendly numbers chosen by the specific FFTW formula are computed noticeably faster than non-fftw-friendly numbers. Because of this discovery we used only fftw-friendly number in following tests.
- The computation speed depends on height to width ratio of dimensions of a source and a target. With certain height to width ratio our speed results were especially fast. We started to call the dimensions with such ratios as "bathtubs" and found a generic formula so we could always find them.
- FFTW provides several planning flags that change behaviour and computation speed results. We tested several of those flags and determined all except one as unsatisfactorily slow. We couldn't test flags for matrices > 4 GB because of hardware limitation.
- Even through FFTW documentation stated 2^N transforms to be especially fast we disproved the statement and determined that non-2^N transforms with similar transform areas to 2^N were computed faster.
- With use of FFTW native parallel engine we tested transforms with various number of threads. For two or more threads the results turned out as expected; The more threads used for computation the shorter transform durations. Results for one thread were chaotic and even after several tests we didn't find out why.
- Lastly we tested transforms of prime numbers from fftw-friendly formula and found out that all were computed in roughly same time. Favouring one prime number over another is meaningless.

All gained knowledge of FFTW behaviour was utilised in making Optimization Planner, the program that finds optimal divisions of a source and a target so that the durations of fast Fourier transforms take the shortest time. The function of the program is demonstrated by the test results that shows a high efficiency of the program.

Acronyms Overview

DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West; the library that implements FFT
GPL	GNU General Public License
IDFT	Inverse Discrete Fourier Transform
SLM	Spatial Light Modulator; a display

References

[Bau10] Heiko Bauke. *Parallel FFT performance*. Mar. 6, 2010. URL: http://numbercrunch.de/blog/2010/03/parallel-fft-performance/.

[Bla12] Anthony Blake. *Existing Libraries*. Connections. Version 1.1. July 15, 2012. URL: http://cnx.org/content/m43809/1.1/.

[Exc09] Campus de Excelencia Internacional. *FFT benchmark. SGI-IZO*. Mar. 27, 2009. URL: www.ehu.es/sgi/ARCHIVOS/fft_benchmark.pdf.

[fft06] fftw.org. *Benchmarked FFT Implementations*. The year 2006 is the last update to the list. 2006. URL: http://www.fftw.org/benchfft/ffts.html.

[fft07a] fftw.org. *BenchFFT Home*. 2007. URL: http://www.fftw.org/benchfft/.

[fft07b] fftw.org. *FFTW Intel Core Duo benchmarks*. 2007. URL: http://www.fftw.org/speed/CoreDuo-3.0GHz-icc64/.

[fft07c] fftw.org. *FFTW Pentium 4 benchmarks*. 2007. URL: http://www.fftw.org/speed/Pentium4-2.4GHz-gcc/.

[fft12a] fftw.org. *FFTW Complex DFTs best handling sizes*. Version 3.3.3. Nov. 25, 2012. URL: http://www.fftw.org/doc/Complex-DFTs.html.

[fft12b] fftw.org. *FFTW Complex Multi-Dimensional DFTs*. Version 3.3.3. Nov. 25, 2012. URL: http://www.fftw.org/doc/Complex-Multi_002dDimensional-DFTs.html.

[fft12c] fftw.org. *FFTW Planner Flags*. Version 3.3.3. Nov. 25, 2012. URL: http://www.fftw.org/doc/Planner-Flags.html.

[fft12d] fftw.org. *FFTW Row-major Format*. Version 3.3.3. Nov. 25, 2012. URL: http://www.fftw.org/doc/Row_002dmajor-Format.html.

[fft12e] fftw.org. *How the parallel FFTW transforms work. Multi-dimensional Parallel FFT (shared memory)*. Nov. 25, 2012. URL: http://www.fftw.org/parallel/.

[fft12f] fftw.org. *Usage of Multi-threaded FFTW*. Version 3.3.3. Nov. 25, 2012. URL: http://www.fftw.org/doc/Usage-of-Multi_002dthreaded-FFTW.html.

[FJ05] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* **93** (2 2005), pp. 216–231.

[FJ98] Matteo Frigo and Steven G. Johnson. “FFTW: An Adaptive Software Architecture for the FFT”. In: *ICASSP conference proceedings* vol. 3 (1998), pp. 1381–1384.

[gnu11] gnu.org. *References and Further Reading*. 2011. URL: https://www.gnu.org/software/gsl/manual/html_node/FFT-References-and-Further-Reading.html.

[Jas12] R. Jason. *What is the sparse Fourier transform?* May 8, 2012. URL: http://dsp.stackexchange.com/questions/2317/what-is-the-sparse-fourier-transform.

[Lob12] Petr Lobaz. *Computer generated holography: 3D vision and beyond*. University of West Bohemia. Pilsen, Czech Republic, July 2012. 104 pp.

[Ned12] Ondřej Nedvěd. “Rychlá simulace 3D holografického displeje”. Czech. Semestrální práce KIV/VAM. ZČU Plzeň, Dec. 13, 2012. 13 pp.

[Phi06] Steven J. Phipps. *The CSIRO Mk3L Climate System Model*. Antarctic Climate & Ecosystems CRC. Oct. 2006.

[Pro12] Spiral Project. *DFT/FFT IP Core Generator*. Carnegie Mellon University. Version 1.1. 2012. URL: http://www.spiral.net/hardware/dftgen.html.

[Rev12] MIT Technology Review. *A Faster Fourier Transform*. 2012. URL: http://www2.technologyreview.com/article/427676/a-faster-fourier-transform/.

[Rod09] Bernard Rodrigue. *c/c++ FFT library with non GPL license*. Jan. 20, 2009. URL: http://stackoverflow.com/questions/463181/c-c-fft-library-with-non-gpl-license.

List of Figures

List of Tables