Graphical Abstract

A new set of efficient SMP-parallel 2D Fourier subroutines

Alexander O. Korotkevich
Highlights

A new set of efficient SMP-parallel 2D Fourier subroutines
Alexander O. Korotkevich

- New set of efficient SMP-parallel DFTs
- Free software license
- Significant performance boost for large sizes of arrays.
A new set of efficient SMP-parallel 2D Fourier subroutines

Alexander O. Korotkevich\textsuperscript{a,b}

\textsuperscript{a}Department of Mathematics \& Statistics, The University of New Mexico, MSC01 1115, 1 University of New Mexico, Albuquerque, New Mexico, 87131-0001, USA
\textsuperscript{b}L. D. Landau Institute for Theoretical Physics, 2 Kosygin Str., Moscow, 119334, Russian Federation

Abstract

Extensive set of tests on different platforms indicated that there is a performance drop of current standard \textit{de facto} software library for the Discrete Fourier Transform (DFT) in case of large 2D array sizes (larger than $16384 \times 16384$). Parallel performance for Symmetric Multi Processor (SMP) systems was seriously affected. The remedy for this problem was proposed and implemented as a software library for 2D out of place complex to complex DFTs. Proposed library was thoroughly tested on different available architectures and hardware configurations and demonstrated significant ($38 \sim 94\%$) performance boost on vast majority of them. The new library together with the testing suite and results of all tests is published as a project on GitHub.com platform under free software license (GNU GPL v3). Comprehensive description of programming interface as well as provided testing programs is given.

\textit{Keywords:} FFT, GPL, 2D DFTs
\textit{PACS:} 89.20.Ff, 07.05.Tp, 07.05.Bx, 47.27.er

1. Introduction

Since 1990s the standard \textit{de facto} for scientific high performance computations involving discrete Fourier transform (DFT) is FFTW library \cite{1}. Author used it during his whole research career, from early 2000s, when it was still version 2. This library provides self tuned perfected version of fast Fourier transform algorithm, which was initially introduced for broad use in \cite{2}. From the very beginning of the usage of the FFTW, author
was fascinated by the elegant design of the software and convenient application programming interface (API). Author used FFTW for many research codes (e.g. \cite{3, 4, 5, 6, 7, 8, 9, 10, 11}). Some of them involved 2D FFTs, but for very modest sizes of arrays. For quite a long time to get a computational workstation with many computing central processing unit (CPU) cores was prohibitively expensive, while distributed memory computations for these sizes of arrays was utterly useless due to communication cost. Fortunately, situation changed drastically. Now even several tens of CPU cores SMP configurations are readily available for a manageable cost. Available amount of memory also have risen to hundreds of gibibytes. It made some long standing problems from Physics and Applied Mathematics approachable. For example, turbulence simulations require wide range of scales, in other words large wavenumbers arrays, like in the recent work \cite{12}. In order to check some theoretical predictions for propagation of the laser radiation in a turbulent atmosphere, like in paper \cite{13}, one needs 2D array of at least \((43000)^2\) size. Taking into account previously described availability of systems with high number of CPU cores, usage of shared memory program architecture makes algorithms both simpler to implement and more efficient, as it allows to avoid communication between parallel parts of the program due to direct access to the whole program memory by any part (thread) of the code. FFTW has multithreaded parallel functions for all kinds of DFTs during practically all time of its existence. At the same time, when performance tests were done, author was disappointed to notice drastic drop of performance of multithreaded DFTs with growth of the array sizes (see Figures 4-6 below). It should be specified, that author was mostly interested in 2D DFTs. While parallelization of 2D DFT looks obvious and one would expect linear or near to linear growth of performance with the number of parallel threads, the problem is not that simple for small transformation sizes, as inevitable overhead for threads initialization and synchronization can eliminate all the gain if amount of work for every parallel part is not large enough. But for large, tens of thousands points in every direction 2D arrays situation supposed to be pretty simple and speedup from parallelization looks inevitable. On the contrary, parallel FFTW performance degradation was increasing with the size of the arrays (as it is demonstrated later in the paper). A remedy for the situation was found, nearly linear speedup in number of used CPU cores was achieved. A set of 2D DFTs (out of place, double precision complex numbers) is published \cite{14} under free software license (GNU GPL v3) for use by community and/or incorporation into existing software.
2. Proposed solution

Author decided that most probably architecture of FFTW is tuned for efficient memory handling for relatively small transformations. This is why it was decided to use 1D FFTW subroutines and efficient CPU cache memory handling techniques in order to try to improve the situation. For one of the previous codes [8] author developed a parallel version of block matrix transposition, which allowed to avoid unnecessary penalties for cache memory misses (when accessed memory parts are not going one after another, which results in access to the parts of memory (RAM) not mapped into CPUs cache memory, causing latency delays). As a result, the 2D DFT was implemented in the following way: parallel 1D DFTs (using 1D DFTs from FFTW) in the direction of array which goes “along the memory” (in C programming language matrix organization is so called “row major format”, which means that matrix is stored in memory as row after row, unlike FORTRAN’s “column major format” for a matrix storage, so we first compute 1D DFTs along rows) are computed; then array is transposed, resulting in other dimension being along the memory (in C columns become rows); now parallel DFTs are computed for now fast dimension (in C it will correspond to 1D DFTs along rows of transposed matrix, meaning columns of original one); then we transpose array again. Operations of array transposition look unnecessary, as one could just perform 1D DFTs along second dimension without them, but in such a case distance in memory between successive elements of corresponding 1D array will be large (for large arrays) and nearly every operation with these elements would result in CPU “cache miss”, meaning the CPU will wait for data from RAM instead of number crunching. As it was mentioned above we avoid unnecessary “cache misses” by using block array transposition. For large array sizes (when array doesn’t fit into CPU cache memory) effect of efficient memory handling could easily results in order of magnitude speedup of transposition operation in comparison with naive version of it. Block transposition is a pretty standard technique demonstrating importance of thoughtful use of memory especially for large array sizes (some discussion of even more advanced memory handling techniques used internally in FFTW can be found, for example, here [15]). Simple set of subroutines for pool of threads organization and synchronization was used [16] (included with the new library). Because author created this set of 2D DFT subroutines specifically for his set of research tasks, only complex-to-complex out-of-place DFTs were implemented. Other types of DFTs could be implemented by demand
or if new research needs will arise. Published code was not specifically optimized for speed or memory usage, with exception of memory block size for transposition subroutine, which was tested on several architectures to provide slight performance gain with respect to some generic value (only two different values: for AMD® and Intel® CPUs). Application programming interface (API) was made as close to FFTW’s one as possible. Is described in details in Appendix A, together with complete description of benchmark program and proposed performance test procedure.

3. Performance of the code

All systems which participated in benchmark are briefly described in Appendix B. In all the tests version 3.3.8 (the latest at the moment of writing the article) of FFTW was used. Configuration of the library for different CPUs and machines is described in Appendix C. For evaluation of performance the same methodology as described for official FFTW benchmark utility (described in [17]) was used, which means that for complex transforms number of operations is estimated as $5N \log_2 N$ (here $N$ is the total number of harmonics/points), time (up to nanoseconds) is measured for a series of DFTs which takes at least several (usually tens or hundreds) of seconds to compute, then this procedure is repeated 10 times and smallest time (meaning the best performance) expressed in microseconds was used to divide estimated number of operations in order to get performance in MFLOPs (Mega-FLOPs or millions of floating point operations per second). Two series of tests were performed: fixed array size and variable number of parallel threads, fixed number of threads (usually the number of threads which gives the best performance for FFTW in the previous test was chosen) and variable size of the array in order to determine at what value performance drop happens.

3.1. Fixed array size. Performance scaling with number of threads

In all the cases double precision complex array of the size $32768 \times 32768$ was used. Out of place (means different input and output arrays are used) complex to complex DFTs were evaluated for both new library and FFTW. Benchmark program is included in the library package. Its usage together with methodology of the tests are explained in the end of Appendix A. For all but one system (exception was System 2b) the full test was completed twice with several days between runs to avoid influence of system processes.
etc. Best results of all runs were used for evaluation. Benchmark results are represented in Figures 1-3.

![Graph showing performance dependence as a function of number of parallel threads.](image)

**Figure 1**: Array size $(32768)^2$, performance dependence as a function of number of parallel threads. Red: new library, green: FFTW. **System 1** was equipped with two Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz (Haswell), which results in $2 \times 12$ CPU cores, $2 \times 24$ HyperThreading cores.

As one can see, for this size of the array new library gives performance boost for all but one system (exception is **System 3**). See Table 1 for a summary of results. Performance improvement ranges for most systems

| MFLOPS       | Sys 1 | Sys 2a | Sys 2b | Sys 3 | Sys 4 | Sys 5 |
|--------------|-------|--------|--------|-------|-------|-------|
| FFTW         | 42109 | 17640  | 24241  | 44918 | 9078  | 15961 |
| new library  | 59175 | 34292  | 42130  | 38628 | 14986 | 21993 |
| boost        | 41%   | 94%    | 74%    | −14%  | 65%   | 38%   |
| best/core/GHz| 986   | 1021   | 675    | 501   | 407   | 529   |

Table 1: Comparison of best performances for a problem $32768 \times 32768$ double precision complex array. Numbers for FFTW and new library are given in MFLOPS. It should be noted that these numbers could be achieved at different numbers of used parallel threads for FFTW and new library. “Boost” shows speedup of the best results of the new library with respect to the best result of FFTW. Also we provide estimation of efficiency for different systems, computed as best performance (regardless of the library) divided by number of real CPU (not HyperThreading) cores and by base frequency in GHz.
Figure 2: Array size $(32768)^2$, performance dependence as a function of number of parallel threads. Red: new library, green: FFTW. *(Left panel)* **System 2a** was equipped with two Intel® Xeon® Silver 4110 CPU @ 2.10GHz (Skylake), which results in $2 \times 8$ CPU cores, $2 \times 16$ HyperThreading cores. **System 2b** was equipped with two Intel® Xeon® Gold 6126 CPU @ 2.60GHz (Skylake), which results in $2 \times 12$ CPU cores, $2 \times 24$ HyperThreading cores. *(Right panel)* **System 3** was equipped with two Intel® Xeon® Gold 6242 CPU @ 2.80GHz (Cascade Lake), which results in $2 \times 16$ CPU cores, $2 \times 32$ HyperThreading cores.

Figure 3: Array size $(32768)^2$, performance dependence as a function of number of parallel threads. Red: new library, green: FFTW. *(Left panel)* **System 4** was equipped with two AMD® Opteron™ Processor 6276 @ 2.3GHz (Bulldozer), which results in $2 \times 8$ CPU cores, $2 \times 16$ HyperThreading cores. *(Right panel)* **System 5** was equipped with two Intel® Xeon® CPU E5-2670 0 @ 2.60GHz (Sandy Bridge EP), which results in $2 \times 8$ CPU cores, $2 \times 16$ HyperThreading cores.
from 38% to 94%. The only system which behaved completely differently (see Figure 2, right panel) is **System 3**. Although architectures of **Systems 2a,b** and **System 3** are very close, the dependence changed drastically after some number of parallel threads. As one can see, until number of threads close to 16 everything looks pretty similar to other systems. Some ideas why it can be so are given in the end of [Appendix B]. Also, one could notice that 16 is exactly the number of real CPU cores on one of the two processors. As a result, absence of speedup after adding more parallel threads can be attributed to some problems of communication between two CPUs in the system. The source(-s) of these problems can range from architecture flaws and/or hardware configuration to configuration of Linux kernel or even motherboard (BIOS) software. The fact that there is no noticeable influence of this issue on FFTW’s performance could be the lower usage of memory throughput due to slightly lower performance which does not create some kind of issue in the kernel and/or CPUs. As a summary, the behavior of **System 3** is still a mystery for the author. Even on this system if we limit number of parallel threads by 16 (one real CPU) new library demonstrates improvement in performance.

### 3.2. Fixed number of threads. Performance scaling with array size

In order to test the performance of two libraries under consideration as a function of the size of an array, author performed another series of tests. Now the number of parallel threads was fixed and only the size of the problem was changing. Once again, only out of place double precision complex DFTs were tested. Number of threads was determined from the tests described in the previous Subsection 3.1. Namely, author used number of parallel threads which gave the highest (or close to that) performance for FFTW library on every architecture. All arrays were square and the problem size was starting from $4096 \times 4096$ with continuation in integer (natural) multiples of 1024 (for example, $(20 \times 1024)^2$). The same testing program included in the library package was used. Results of the tests are represented in Figures 4-6.

As can be seen from the measurements, for every architecture the size of the array when FFTW experiences a dramatic drop in performance is slightly different. At the same time the performance of the new library relatively weakly depends on the size of the problem. Observed oscillations are the consequence of different efficiency of 1D FFTW DFTs for different sizes of the problems ("out of the box" FFTW is the most efficient for sizes which can be represented as products of integer (natural) powers of few lowest prime
Figure 4: Performance dependence as a function of array size $N$, full 2D-array size if $N^2$. Number of threads was chosen as giving the best performance to FFTW in the previous series of tests. Red: new library, green: FFTW. **System 1** was equipped with two Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz (Haswell), which results in $2 \times 12$ CPU cores, $2 \times 24$ HyperThreading cores, number of threads is 47.

Figure 5: Fixed number of threads (giving best performance to FFTW in the previous series of tests), performance dependence as a function of size (length of the side) of the square array. Red: new library, green: FFTW. **(Left panel) System 2a** was equipped with two Intel® Xeon® Silver 4110 CPU @ 2.10GHz (Skylake), which results in $2 \times 8$ CPU cores, $2 \times 16$ HyperThreading cores, number of threads is 32. **System 2b** was equipped with two Intel® Xeon® Gold 6126 CPU @ 2.60GHz (Skylake), which results in $2 \times 12$ CPU cores, $2 \times 24$ HyperThreading cores, number of threads is 48. **(Right panel) System 3** was equipped with two Intel® Xeon® Gold 6242 CPU @ 2.80GHz (Cascade Lake), which results in $2 \times 16$ CPU cores, $2 \times 32$ HyperThreading cores, Number of threads is 33.
numbers 2, 3, 5, and 7, although there is a possibility to make FFTW to be as efficient for larger primes as well \cite{18}). As it can be noted, dips and peaks of the performance of new library perfectly coincide with ones for FFTW, which is inevitable as the same 1D DFT subroutines are used. See Table 2 for a summary of results. In all the cases there is a significant 25 – 40% performance degradation when size of the array exceeds the threshold close to 16384 × 16384 (slightly different for different systems). Again we see different behavior of System 3 with respect to other systems. The first drop in performance by 25% happens, as problem size exceeds 16 × 1024,
the second one, twice stronger (50% if we are using relative measure, but approximately the same amount in MFLOPS), after approximately double of the first threshold, i.e. at $33 \times 1024$. Taking this fact into account, one could see, that if the previous performance tests were done not for array’s size $32 \times 1024$, but for $33 \times 1024$, perhaps we would get result similar to other systems.

4. Conclusion

The significant degradation of performance of current standard de facto DFT library, using multiple parallel threads on SMP systems, motivated the author to perform extensive set of tests on different platforms. Results of these tests indicated that there is performance drop for array sizes exceeding some threshold (slightly different for different CPUs and architectures). The author implemented a relatively simple algorithm as a library using 1D DFT subroutine from FFTW together with parallel block transposition subroutine. The new library demonstrated significant ($35 - 94\%$) performance boost on all but one configurations. Even in the exceptional case slight increase in performance was achieved for some limited number of parallel computational threads. The new library together with the testing suite and results of all tests are published as a project on GitHub platform under free software license (GNU GPL v3). Comprehensive description of programming interface as well as provided testing programs is given in the Appendix A.

Author would like to mention that domination of Intel® CPUs in the list of tested systems was not intentional. AMD® Opteron™ CPU of Bulldozer family (and similar) are infamous for their dreadful performance. Newer CPUs of the same company appeared few years ago. Author tried to contact AMD® through different means of communication and requested a remote access to any SMP machine for 24 hours. Author’s request was declined, it was proposed to contact vendors. Unfortunately, during the time of tests no high performance computing vendors offered servers equipped with recent AMD® CPUs.

Current version of the library provides only 2D out of place complex to complex DFT subroutines. It comes with some limitations and not optimized for memory consumption. Some straightforward ways of improvement include different pools of threads for transposition and 1D DFTs, elimination of current limitations on dimensions of the arrays (currently they have to be multiples of relatively small number, like 64), and optimization of memory
usage. Further development will depend on author’s future research needs as well as demand from potential users. Author would prefer the new library to be included as a possible engine for large arrays into existing libraries and will try to work in this direction. Another option will be internal tuning of the existing libraries, perhaps based on tests presented in this paper, in order to eliminate existing significant performance degradation for large arrays. Any of these two variants will make life of users (including the author) way easier as there will be no need to choose different libraries for different sizes of the problems.

Acknowledgments

The author has performed this work for the project supported by the Simons Collaboration on Wave Turbulence and author is grateful for this support. Some of the systems used for development and testing (System 1 and System 4) were funded by NSF grant OCE 1131791. Author would like to express his gratitude to A.O. Prokofiev, who explained him block transposition in early 2000s. Also author would like to thank developers of FFTW [1] for this beautiful and free example of software engineering.

Appendix A. Library description

Appendix A.1. Current library packaging

As currently there is only one subroutine for out of place 2D complex to complex DFT, author left the library as just mere C-files, which have to be compiled into object files and linked to the users program in the usual way. As the interest level to the library is not yet clear, author would prefer to invest the time into creation of auto-configuration and installation script only if requested by other users.

The new library is dependent on FFTW v3. Author strongly recommends to compile users local version of the library in the user’s directory, configured to use system specific SIMD instructions as explained in [Appendix C] as performance gain can be substantial. After that Makefile has to be edited in order to provide paths to header files and library files. Currently, static linking is used. If user prefers to use dynamic linking the option -static in LDFLAGS has to be erased. Usual command make clean all will compile the library (object files) and the benchmark program.
Appendix A.2. Library API

Library API was intentionally made as close to FFTW’s one as possible. In the description we suppose that the reader is familiar with FFTW’s API, detailed description of which can be found at FFTW’s web-site \[1\]. First of all, one needs to initialize threads (here we initialize 32 threads):

```c
unsigned long int threads_number=32;

my_fft_init (threads_number);
```

Here `threads_number` is the number of threads to be initialized in the pool of threads. Maximum number of threads to be available for the computations. This value has to be more or equal to zero.

Then the usual creation of a DFT’s plan has to be performed:

```c
my_fft_plan plan_fwd;

plan_fwd = my_fft_plan_dft_2d (in, out, out2, NX, NY, +1, FFTW_EXHAUSTIVE, threads_number);
```

Here `in` and `out` are input and output arrays similarly to FFTW’s API. Array `out2` is a scratch array of the same size which is necessary for the present version as the array transposition subroutine does not support in place functionality. This limitation can be easily eliminated in the next version. All other parameters are passed directly to the 1D DFT API of the FFTW library. Here `threads_number` is actual number of threads to be used for DFT (has to be less or equal to number of threads previously specified in the `my_fft_init`). If it is equal to zero, nonparallel (linear) version of subroutine is used.

**Important note:** Currently library supports only sizes when both `NX` and `NY` are multiples of variable `BLOCK_SIDE_SIZE` defined in the file `my_fft_lib.h`. This variable defines a size of the square block used for transposition. This limitation can be eliminated if needed. Even in its present form it hardly pose any practical issue, as sizes of arrays when this library become beneficial are usually $16 \times 1024$ or larger for `NX` and `NY`, as a result to change it to the nearest multiple of even 64 (optimal case for all tested Intel® CPUs, for AMD® it is equal to 16) is not a serious problem.

In order to perform planned DFT one needs to call the subroutine `my_fft_execute` with the previously created plan:
my_fft_execute (plan_fwd);

In the case if the user needs to free the resources allocated for the previously created plan, one needs to call the following subroutine:

my_fft_destroy_plan (plan_fwd);

In order for the functions definitions to be included into the *.c-file one needs to include my_fft_lib.h:

#include "my_fft_lib.h"

Here it is supposed that file my_fft_lib.h is located in the same directory.

If you would like to use FFTW concept of wisdom (information about previously computed optimal plans, significantly accelerates creation of new plans), just load wisdom from file using standard FFTW subroutine. It should be noted, that new library uses only 1D linear DFTs and parallel and linear wisdom are not compatible. Please, load only usual (nonparallel) wisdom, not the one produced as a result of plans creation for parallel DFTs.

Appendix A.3. Benchmark program

The simple benchmark program my_fft_test is provided together with the library files. It allows to perform a performance test for specified size of an array using both new library’s and FFTW’s subroutines. Only out of place complex to complex 2D DFTs are compared. Result is reported in MFLOPS. Here is an example of usage of the program:

$ ./my_fft_test
Usage is the following:
./my_fft_test NX NY num_of_repeats number_of_threads [output_file]
$ ./my_fft_test 8192 8192 10 8
NX = 8192, NY = 8192, num_reps = 10, threads_number=8
8 Threads initialized.
Wisdom successfully imported.
Exporting wisdom to file...
Time for 10 Fourier transforms using my_fft is 1.09193526530e+01 seconds.
......... (here will be output for 10 runs)
Time for 10 Fourier transforms using my_fft is 1.11323441190e+01 seconds.
MFLOPS = 8062.030694
Using FFTW_MEASURE option for FFTW plan. Usually quite fast.
Time for 10 Fourier transforms using FFTW is 9.5567851900e+00
seconds.
......... (here will be output for 10 runs).
Time for 10 Fourier transforms using FFTW is 9.55813624100e+00
seconds.
MFLOPS = 9256.224182
Normed L_infty norm: ||Mismatch between my_fft and fftw||_infty
/(NX*NY) = 5.218373960228812e-17.

Input parameters are: sizes of the array in X and Y dimensions, number of
repeated DFTs for every measurement of time (for good results should give
at least few seconds per measurement, several tens of seconds even better),
number of threads to use, optional output file name. If the output file name
is provided, result is written to a file (a line is appended to the end of file)
as seven columns: X-size, Y-size, number of threads used, performance in
MFLOPS for new library, performance in MFLOPS for FFTW, best measured
time for new library, best measured time for FFTW. Also shell-scripts for
testing are provided. They will be discussed in the next subsection
together with methodology of testing.

Important note 1: Author used FFTW_MEASURE option for creation of
FFTW parallel plans for large arrays, although FFTW benchmarking manual
[17] recommends to use FFTW_PATIENT. The reason is extremely
long time necessary for creation of plans with this option. For example, on System 1
creation of a plan for problem size 32768 x 32768 using 47 threads
with FFTW_PATIENT option took around 30 hours. No significant advantage
for large array sizes was obtained. On System 2b creation of a plan for
problem size 32768 x 32768 using 24 threads using FFTW_PATIENT option took
more than 24 hours. Again, no significant performance boost for large array
sizes was noticed. If one would like to try it, variable FFTW_PATIENT_PLAN
has to be defined in the file my_fft_test.c.

Important note 2: As it was mentioned above, 10 time measurement
runs are performed and the best (smallest) time is used for evaluation of
performance. The number of runs can be changed by redefining variable
RUNS_NUMBER in the file my_fft_test.c.

Program tries to load FFTW wisdom from the file fftw.wisdom located
in the same directory. If the file is found, the program reads the wisdom and
after creation of a plan for the new library writes the cumulative wisdom to the same file. If program cannot find `fftw.wisdom` file, it creates plans without it and saves the accumulated wisdom to `fftw.wisdom` file if possible. Unfortunately, as it was already mentioned above, for FFTW v3 wisdom information for linear and parallel planning subroutines are incompatible. Even more, if linear planning subroutine was used in the program, parallel wisdom cannot be used in the same program. If this situation will change in future releases, all necessary parts of the code can be initiated by defining variables `READ_THR_WISDOM_FOR_FFTW` and `SAVE_THR_WISDOM_FOR_FFTW` in the file `my_fft_test.c`. For parallel wisdom program will try to use file `fftw.thr.wisdom` located in the same directory.

### Appendix A.4. Installation and testing methodology

When benchmark program is compiled and working properly, one could try to fine tune the only changeable parameter in the library, namely the variable `BLOCK_SIDE_SIZE` in the file `my_fft_lib.h`, which determines the side length of the square block used for transposition. By author’s experience, the optimal values are: 64 for Intel® CPUs and 16 for AMD® CPUs (author had an access only to relatively old AMD® CPUs). Author recommends to use powers of 2. Of course, after every change of this value, library and test program has to be recompiled.

After that one can start scanning system for performance dependence on number of used threads with fixed array size. Initially, it is good to determine the largest array which fits into system’s RAM memory. Due to limitation mentioned above author recommends to use multiples of $2^{10} = 1024$. Let us say that the maximum size of the array is $(20 \times 1024)^2$ and the system has 16 logical CPU cores (usually it means 8 real CPU cores and 8 more HyperThreading cores). There is a simple shell-script `thr_test.sh` provided with the library. This is how it can be tuned for the situation described above:

```bash
#!/bin/bash

for i in {0..8}; do
   ./my_fft_test 20480 20480 5 $i 20k.User_thr_test.dat
done

for i in {9..16}; do
```
Here we use 5 DFTs for every time measurements for small number of parallel threads and 10 for presumably faster configurations with larger number of parallel threads. The output file format is described above in details.

After completing the previous scan, one should find the number of threads which gives the best performance for FFTW. This number of threads will be used to determine the performance dependence on the size of the problem when number of parallel threads is fixed. For example, let us suppose that the best performance is achieved by FFTW using 15 threads. Then one can slightly modify simple script `size_test.sh` provided with the library in the following way:

```bash
#!/bin/bash

for i in {4..20}; do
    size=$(echo $i*1024| bc)
    ./my_fft_test $size $size 10 15 User_size_test.15thr.dat
done
```

Here we scan in integer (natural) multiples of 1024 as the side length of a square array. Size is changing from $4 \times 1024$ till $20 \times 1024$. As we determined previously that this is the largest size which fits into system’s RAM. The output file format is described above.

This is the method which was used in all measurement presented in this paper.

Appendix  B. Tested systems

All systems were working under different flavors of GNU Linux distributions ranging from Ubuntu and Debian to Gentoo. GNU Compiler Collection (gcc) was used as a C compiler in all the cases. Here are short hardware descriptions for every system.

- **System 1**
  CPU: 2× Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz (Haswell), 2x12 CPU cores, 2x24 HT cores
  RAM: 128 GiB
• **System 2a**
  CPU: 2x Intel® Xeon® Silver 4110 CPU @ 2.10GHz (Skylake), 2x8 CPU cores, 2x16 HT cores
  RAM: 64 GiB

• **System 2b**
  CPU: 2x Intel® Xeon® Gold 6126 CPU @ 2.60GHz (Skylake), 2x12 CPU cores, 2x24 HT cores
  RAM: 64 GiB

• **System 3**
  CPU: 2x Intel® Xeon® Gold 6242 CPU @ 2.80GHz (Cascade Lake), 2x16 CPU cores, 2x32 HT cores
  RAM: 384 GiB

• **System 4**
  CPU: 2x AMD® Opteron™ Processor 6276 CPU @ 2.3GHz (Bulldozer), 2x8 CPU cores, 2x16 HT cores
  RAM: 64 GiB

• **System 5**
  CPU: 2x Intel® Xeon® CPU E5-2670 0 @ 2.60GHz (Sandy Bridge EP), 2x8 CPU cores, 2x16 HT cores
  RAM: 128 GiB

It should be noted, that all systems, except **System 3**, were producing very consistent results in all the tests. The **System 3** behaved very inconsistently, with significantly different results between benchmarking runs (each run took about 24 hours, difference could be as high as 10%). This is the newest CPU in the tested systems and it was released already after recent cache memory associated vulnerabilities reported for Intel® CPUs. Perhaps, temporary and/or urgent patches to the cache memory system and Linux kernel were the reason for such a behavior. Author has no other explanation.

**Appendix C. Configuration of FFTW**

For most of the systems (**Systems 1-3**) participated in benchmarking the following line was used during configuration of FFTW library:

```
$ ./configure --enable-threads --disable-fortran --with-gnu-ld
         --enable-avx2 --enable-fma --enable-sse2
```
Of course, if you need FORTRAN interface, please, remove `--disable-fortran` from the line. As Systems 4-5 do not support AVX2 SIMD (single instruction multiple data) instructions, slightly different configuration line was used:

```
$ ./configure --enable-threads --disable-fortran --with-gnu-ld
  --enable-avx --enable-fma --enable-sse2
```

It should be noted, that although Systems 2a,b-3 support AVX512 SIMD instruction set, it is not recommended to use configuration option `--enable-avx512` as it results in drastic drop in performance. Comparison of different configurations can be performed using `bench` program supplied with FFTW library. The following command line was used for final testing: `./bench --onthreads=24 --opatient --s oc32768x32768`, initial testing was done on smaller $4096 \times 4096$ array.

As an another note, author do not recommend to supply any additional `CFLAGS` during compilation. In his experience it resulted only in performance drop. Default compilation flags are well tuned and (nearly?) optimal.

References

[1] M. Frigo, S. G. Johnson, The design and implementation of FFTW 3, Proc. IEEE 93 (2) (2005) 216–231.

URL [http://fftw.org](http://fftw.org)

[2] J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19 (90) (1965) 297–301.

[3] doi:10.2307/2003354.

[3] A. I. Dyachenko, A. O. Korotkevich, V. E. Zakharov, Decay of the monochromatic capillary wave, JETP Lett. 77 (9) (2003) 477–481.

[4] arXiv:physics/0308100.

[4] A. I. Dyachenko, A. O. Korotkevich, V. E. Zakharov, Weak turbulence of gravity waves, JETP Lett. 77 (10) (2003) 546–550.

[5] arXiv:physics/0308101.

[5] A. I. Dyachenko, A. O. Korotkevich, V. E. Zakharov, Weak turbulent kolmogorov spectrum for surface gravity waves, Phys. Rev. Lett. 92 (13) (2004) 134501.

[6] arXiv:physics/0308099.
[6] A. O. Korotkevich, Simultaneous numerical simulation of direct and inverse cascades in wave turbulence, Phys. Rev. Lett. 101 (7) (2008) 074504. arXiv:0805.0445.

[7] A. O. Korotkevich, A. I. Dyachenko, V. E. Zakharov, Numerical simulation of surface waves instability on a homogeneous grid, Physica D 321-322 (2016) 51–66. arXiv:1212.2225.

[8] A. O. Korotkevich, P. M. Lushnikov, Proof-of-concept implementation of the massively parallel algorithm for simulation of dispersion-managed wdm optical fiber systems, Optics Lett. 36 (10) (2011) 1851–1853. arXiv:1101.0086.

[9] A. O. Korotkevich, P. M. Lushnikov, H. A. Rose, Beyond the random phase approximation: Stimulated brillouin backscatter for finite laser coherence times, Phys. of Plasmas 22 (2015) 012107. arXiv:1105.2094.

[10] S. A. Dyachenko, P. M. Lushnikov, A. O. Korotkevich, The complex singularity of a stokes wave, JETP Letters 98 (11) (2013) 767–771. arXiv:1311.1882, doi:10.7868/S0370274X13230070.

[11] S. A. Dyachenko, P. M. Lushnikov, A. O. Korotkevich, Branch cuts of Stokes wave on deep water. Part I: Numerical solution and Padé approximation, Studies in Applied Mathematics 137 (4) (2016) 419–472. arXiv:1507.02784, doi:10.1111/sapm.12128.

[12] G. Falkovich, N. Vladimirova, Cascades in nonlocal turbulence, Phys. Rev. E 91 (2015) 041201. doi:10.1103/PhysRevE.91.041201. URL https://link.aps.org/doi/10.1103/PhysRevE.91.041201.

[13] I. Kolokolov, V. Lebedev, P. M. Lushnikov, Statistical properties of a laser beam propagating in a turbulent medium, Phys. Rev. E 101 (2020) 042137. doi:10.1103/PhysRevE.101.042137. URL https://link.aps.org/doi/10.1103/PhysRevE.101.042137.

[14] A. O. Korotkevich, Set of SMP-parallel subroutines for 2D out of place complex to complex DFTs. (2019). URL https://github.com/w5kao/my_fft_lib.

[15] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, Proc. IEEE Symp. on Foundations of
A. O. Korotkevich, Library for initialization and synchronization of pool of threads (2009).
URL https://github.com/w5kao/my_thr_lib

M. Frigo, S. G. Johnson, FFT benchmark methodology (2011).
URL http://www.fftw.org/speed/method.html

M. Frigo, S. G. Johnson, Introduction to FFTW v3 (2011).
URL http://www.fftw.org/fftw3_doc/Introduction.html