Dariusz PUCHAŁA[1], Kamil STOKFISZEWSKI[1], Bartłomiej SZCZEPANIAK[2], Mykhaylo YATSYMIRSKYY[1]

Institute of Information Technology (1), Institute of Applied Computer Science, Technical University of Lodz (2)

doi:10.15199/48.2016.07.14

Effectiveness of Fast Fourier Transform Implementations on GPU and CPU

Abstract. *In this paper, we present the results of comparison of the effectiveness of selected variants of radix-2 Fast Fourier Transform (FFT) algorithms implemented on both Graphics (GPU) and Central (CPU) Processing Units. The considered algorithms differ in memory consumption and the arrangement of data-flow paths which affects the global memory coalescing and cache memory exploitation. The obtained results allow to indicate the variants of FFT algorithms which are best suited for GPU and CPU architectures, to confirm the advisability of GPU oriented calculations of FFT and to formulate a guideline for implementations of fast algorithms of various linear transforms.*

Streszczenie. *W niniejszej pracy przedstawiono wyniki porównania efektywności wybranych wariantów algorytmów szybkiej transformaty Fouriera (FFT) typu radix-2 realizowanych zarówno dla procesorów graficznych (GPU) jak i typowych jednostek centralnych (CPU). Rozważane algorytmy różnią się zapotrzebowaniem pamięciowym oraz postaciami grafów przepływu danych, które mają wpływ na spójność wykorzystania pamięci globalnej oraz pamięci cache jednostek GPU i CPU. Uzyskane wyniki pozwalają na wskazanie wariantów algorytmów FFT, które są najlepiej dostosowane dla architektur GPU i CPU, pozwalają też potwierdzić celowość realizacji implementacji FFT zorientowanych na wykorzystanie jednostek GPU, a także sformułować ogólne wytyczne dla implementacji zorientowanych na wykorzystanie jednostek GPU algorytmów szybkich przekształceń liniowych.* ***(Porównanie efektywności wybranych wariantów algorytmów szybkiej transformaty Fouriera (FFT) realizowane na procesorach graficznych (GPU) i jednostkach centralnych (CPU).)***

Keywords: Fast Fourier Transform, parallel computations, general purpose GPU computations.
Słowa kluczowe: szybkie przekisztłcenie Fouriera, obliczenia równoległe, obliczenia ogólnego przeznaczenia na GPU.

Introduction

In recent years, we could witness a growing interest of scientific community in parallel GPU implementations of a variety of classical computation algorithms, e.g. [1], [2]. This is due to the fact that GPUs have become an attractive tools for general-purpose computations delivering a low-cost hardware solutions combined with relatively high performance [2]. The class of algorithms of a particular interest are the fast algorithms for discrete Fourier transform (DFT) since DFT plays an important role in many engineering areas such as: digital signal processing, pattern recognition, cryptography, lossy data compression, etc. (see [3]-[5]). Hence, many different variants of FFT implementations (e.g. [3], [7], [8]) which are well suited for various hardware architectures, including the consumer segment GPUs (cf. [2], [9] and [10]), were already constructed.

In this paper, we present the results of comparison of the effectiveness of two selected variants of radix-2 Fast Fourier Transform (FFT) algorithms. The first one is a well known Cooley-Tukey algorithm (see [6]) and the second one described in [7] can be characterized by highly uniform structure and hence is more convenient for parallel GPU realizations. Both algorithms were implemented on GPU and CPU using two different ways of input data arrangement. The results of experiments are presented and interpreted. On its basis variants of FFT best suited for GPU and CPU architectures are indicated and in addition the authors formulate a guideline for future implementations of fast algorithms of various discrete linear transforms.

Fast Fourier Transform

The term fast Fourier transform (FFT) refers to the class of computationally efficient algorithms for the calculation of the discrete Fourier transform (DFT) which is defined as:

$$(1) \quad X(k) = DTF_N\{x(n)\} = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}kn}$$

for $k = 0, 1, \ldots, N-1$, where N is the size of transformation, $x(n)$ is a sequence of complex numbers and i stands for imaginary unit. FFT reduces the original computational complexity from $\mathcal{O}(N^2)$ (cf. eq. (1)) to $\mathcal{O}(N \log_2 N)$ level. The reduction of complexity by one order of magnitude is achieved by apt usage of divide-and-conquer strategy which allows to rewrite formula (1) in the form:

$$(2) \quad X(2k) = DFT_{N/2}\{x(n) + x(n+N/2)\},$$
$$X(2k+1) = DFT_{N/2}\{(x(n) - x(n+N/2))W_N^n\}$$

for $k = 0, 1, ..., N/2-1$ and $W_N^n = \exp(-i2\pi n/N)$. Then the decomposition formulas (2) can be applied recursively for $N = 2^p$, where p is an integer number, resulting in radix-2 Cooley-Tukey (CT) type FFT with decimation in frequency domain (DIF). In Fig. 1 the data-flow graph of $N = 16$ point CT DIF FFT is presented with definitions of butterfly operations used to its description.

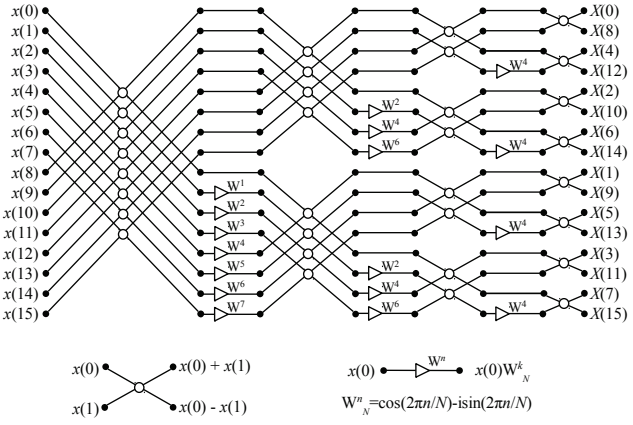

Fig. 1. The data-flow graph for 16-point CT DIF FFT

As mentioned earlier different hardware platforms may have different requirements regarding memory coalescing and cached memory access. For modern CPUs data coalescing is not an issue while strided memory access can significantly reduce data processing bandwidth. For the first generation GPUs (with no cached global memory) proper data coalescing is an essential requirement while strided memory access will result in critical reduction of the processing bandwidth regardless the generation of GPUs.

If we take a look at the structure of CT DIF FFT (see Fig. 1) it can be figured that for huge values of N and at the initial stages such a structure may suffer from strided memory access since butterfly operators require distant data locations with offset of N/2 elements. This situation improves with suc-

cessive stages and is negligible at the last one. Due to that fact, we take into consideration the second variant of radix-2 DIF FFT described in paper [7]. It can be easily proved that it is equivalent to CT DIF FFT and the only difference lies in rearrangement of locations of inputs/outputs for butterfly operators in the following stages. The data-flow graph of such FFT algorithm (referred to as MY FFT) for $N = 16$ points in shown in Fig. 2.

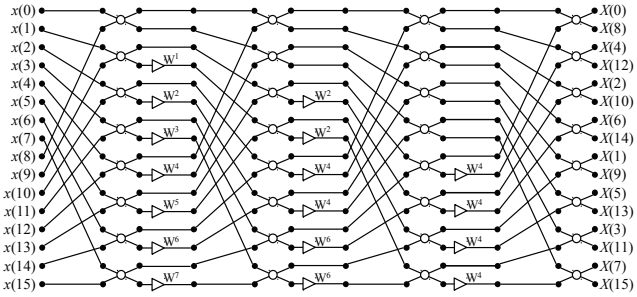

Fig. 2. The data-flow graph for 16-point DIF FFT from [7]

Such a structure should exhibit better memory access properties because outputs of all butterfly operators at each stage are strictly localized. The only drawback of MY FFT is that it requires additional memory buffer since now calculations cannot be realized in-place.

It should be noted that DFT operates on complex numbers. Thus an additional consideration of the way of memory storage of real and imaginary parts of input data is needed. In this paper, we consider two approaches: real and imaginary parts stored as pairs in one data buffer (1B); two separate buffers used for storage of real and imaginary parts (2B).

Implementation of FFT on GPU and CPU

For the implementation of the considered algorithms on GPU cards we assumed the same approach regardless of a variant of FFT structure. It consists in calculating the following stages sequentially by separate kernel functions. Within a single stage butterfly operators are assigned to separate threads which are grouped into blocks. Thus, individual threads are responsible for calculation of operations needed by a single butterfly operator. In case of MY DIF FFT calculations cannot be realized in place, thus we use two buffers which are switched sequentially between successive stages.

The CPU implementation involved two nested loops where the outer was responsible for sequential iteration in order left to right between stages while the inner was responsible for sequential iteration in top-down for butterfly operators.

Experimental research

The experiments involved testing the considered variants of FFTs of different sizes operating on random data with two data samples arrangements: 1B and 2B. The obtained results in the form of execution times for CPU, GPU and their ratios, averaged over 50 trials, are presented in Figs. 3-5 respectively. The tests were performed using Intel Core i7, 3.5 GHz, processor with 16 GB RAM and NVidia GeForce GTX 970 graphics card, with 4GB DRAM. All FFT calculation procedures for CPU and GPU were originally implemented by the authors in C language using Microsoft™ Visual C++® 2012 compiler with code optimization options set up for time performance efficiency. For GPU code authors have used NVIDIA® CUDA® Toolkit's 7.5 NVCC C++ language compiler with standard code optimization options. In Figs. 3 and 4 one can see the absolute averaged execution times of all eight (i.e. CPU and GPU) implementations of the analyzed FFT

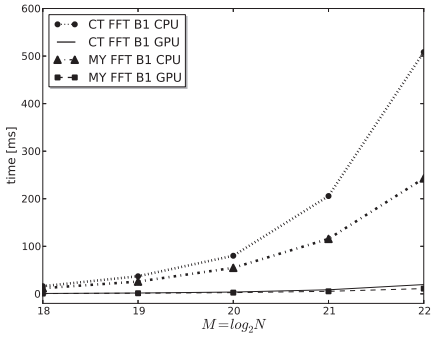

Fig. 3. Execution time comparison for B1 FFTs

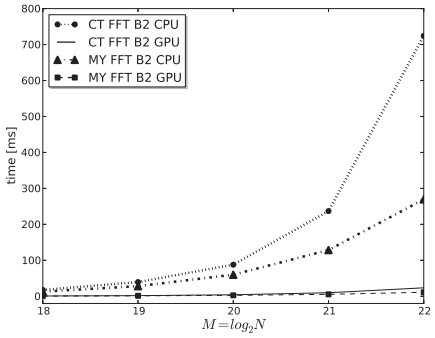

Fig. 4. Execution time comparison for B2 FFTs

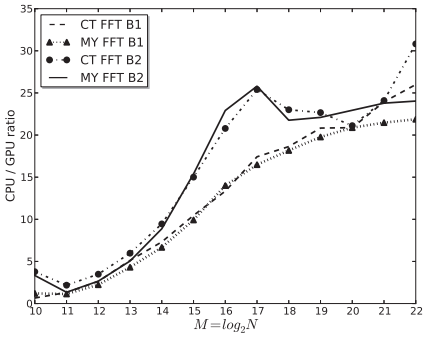

Fig. 5. GPU to CPU acceleration ratios between FFTs

algorithms, where their B1 variants are considered in Fig. 3 and B2 variants are considered in Fig. 4 In Fig. 5 time ratios between CPU and GPU implementations are shown for all of the four considered FFT algorithms. It can be seen from Figs. 3 - 5 that the overall execution time results achieved by the GPU implementations for large sizes of N of the two considered fast transforms were significantly better than their respective CPU implementations for both their B1 and B2 variants, which strongly proves the advisability of implementation of the FFT algorithms on GPUs. For example in Fig. 5 one might observe that for transform size $N = 2^{22}$ B2 CT FFT GPU is over 30 times faster than its CPU implementation while for equivalent B2 MY FFT case it's CPU/GPU time ratio reaches almost 25. From Figs. 3 and 4 it can be seen that the best absolute result among all considered implementations for large values of N was achieved by B1 variant of GPU implementation of MY FFT. Among CPU implementations the B1 MY FFT algorithm is also the fastest one outperforming CPU CT FFT B1 and B2 variants over 2 and 2.5 times respectively. To sum up, for both CPU and GPU implementations of the considered FFT algorithms MY FFT B1 achieves the shortest execution times.

The following part of experimental study involves the times of calculation of FFT phase coefficients, i.e.: W_N^n for $N = 2^p$, $n = 0, 1, \ldots, N/2 - 1$, both on GPU ($t_{GPU}$) and CPU ($t_{CPU}$). In GPU implementation a single kernel call was used to calculate the coefficients, where each of $N/2$ threads calculated one coefficient in accordance with its definition (cf. Fig. 1). In case of CPU the following coefficients were calculated sequentially using $\mathrm{fsincos}$ instruction of x86 family microprocessors. That instruction calculates both $\sin$ and $\cos$ functions of the same angle with time at about 25% shorter than separate calculations. The obtained experimental results are shown in Table 1.

Table 1. Experimental results in times of computation of phase coefficients on GPU and CPU

$p =$	11	12	13	14	15	16	17
t_{GPU} [ms]	0.004	0.004	0.005	0.005	0.005	0.007	0.009
t_{CPU} [ms]	0.221	0.361	0.390	0.682	0.924	1.373	2.682
$p =$	18	19	20	21	22	23	24
t_{GPU} [ms]	0.013	0.022	0.039	0.074	0.146	0.280	0.554
t_{CPU} [ms]	5.227	10.36	20.74	41.49	83.23	166.0	333.1

Analysis of experimental results shows a huge advantage of GPU-oriented approach. The resulting acceleration over CPU realization was in the range of 50 (for $N = 2^{11}$) to 600 (for $N = 2^{24}$) times. It should be noted that the possible improvement of CPU-oriented approach may take advantage of trigonometric identities for $\sin$ and $\cos$ functions of the sum of angles. However, in such approach the computational error accumulates sequentially and hence that approach is not practical for transform sizes N higher than 1024 points.

Last issue considered during the experimental study regarded a bit-reversal permutation of the FFT output coefficients which must be performed if after FFT calculation the resulting data has to be directly used by the following stages of a given computational process. Two types of implementations of such permutation were considered, namely, standard work-efficient algorithm (see [11]) performed on CPU and radix-2 step-efficient algorithm implemented on GPU. The first algorithm works in-place and its computational complexity can be shown to be $\mathcal{O}(N)$ [11] while the second one, depicted in Fig. 6, uses two N-complex-element buffers and needs $\log_2 N$–1 parallel steps. The obtained execution times of the considered algorithms are shown in Fig. 7. One can see that for data sizes ranging from $N = 2^{18}$ to $N = 2^{25}$ GPU implementation of a bit-reversal FFT coefficient permutation outperforms that of CPU by 2.6 to 9.1 times respectively.

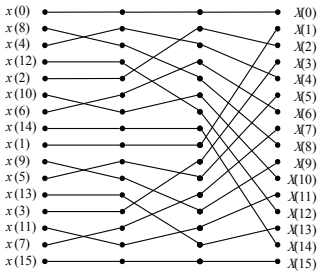

Fig. 6. Data-flow graph of 16-pt bit-reverse step-efficient algorithm

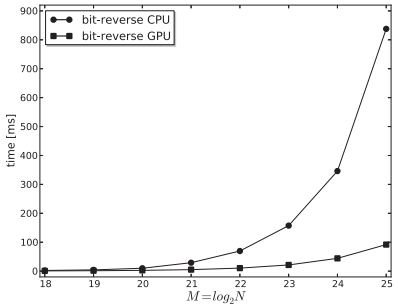

Fig. 7. Execution time comparison for bit-reverse implementations

Conclusion

In the paper the results of comparison of the effectiveness of selected variants of radix-2 Fast Fourier Transform algorithms implemented on both Graphics and Central Processing Units were presented. Experimental results have shown that GPU implementation of FFT algorithms may achieve over 30 times the acceleration with respect to the CPU FFT realizations for sufficiently large data sizes. It's also shown that GPU implementations of the FFT phase coefficients calculation and bit-reversal permutation stages of the FFT algorithm hugely outperform their standard CPU implementations. Moreover the results indicate that algorithms characterized by unified structures (having identical stages) are better suited for both CPU and GPU implementations. In addition it can be concluded that structures that are simpler in the sense of indices calculations are also more computationally efficient.

***Authors**: Ph.D. Dariusz Puchała, Ph.D. Kamil Stokfiszewski, Prof. Mykhaylo Yatsymirskyy, Institute of Information Technology, Faculty of Technical Physics, Computer Science and Applied Mathematics, Technical University of Lodz, ul. Wólczańska 215, 90-924 Łódź, Poland, email: mykhaylo.yatsymirskyy@p.lodz.pl, M.Sc. Bartłomiej Szczepaniak, Institute of Applied Computer Science, ul. Stefanowskiego 18/22, 90-924 Łódź, Poland, email: bartlomiej.szczepaniak@p.lodz.pl*

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. Purcell, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[2] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, *High Performance Discrete Fourier Transforms on Graphics Processors*, Proc. 2008 ACM/IEEE Conf. on Supercomputing, IEEE Press Piscataway, NJ, USA 2008.

[3] U. N. Ahmed, K. R. Rao, *Orthogonal Transforms for Digital Signal Processing*, Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 0-387-06556-3.

[4] D. Puchala, K. Stokfiszewski, *Parametrized Orthogonal Transforms for Data Encryption*, Computational Problems of Electrical Engineering Journal, vol. 3, no. 1, pp. 93-97, 2013.

[5] M. Yatsymirskyy, K. Stokfiszewski, P. S. Szczepaniak, *Image compression using fast transforms realized through neural network learning*, Modelling of Computer Science Technologies, Ukrainian National Academy of Sciences vol. 23, pp. 95–99, 2003.

[6] J. W. Cooley,J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation vol. 19, no. 90, pp. 297–301, Apr. 1965.

[7] M. Yatsymirskyy, *Fast algorithms for orthogonal trigonometric transforms' computations*, (in Ukrainian), Lviv Academic Express, ISBN 966-7094-16-2, Lviv, 1997.

[8] T. Wiechno, M. Yatsymirskyy, *Two-stage Fast Fourier and Hartley Transform of Real Data Sequence*, Przegląd Elektrotechniczny, vol. 86, no. 1, pp. 41-43, 2010.

[9] K. Moreland, E. Angel, *The FFT on a GPU*, in Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware, 2003, pp. 112–119.

[10] J. Spitzer, *Implementing a GPU-efficient FFT*, SIGGRAPH Course on Interactive Geometric and Scientific Comput. with Graphics Hardware, 2003.

[11] J. J. Rodriguez, *An Improved FFT Digit-Reversal Algorithm*, IEEE Trans. on Acoustics, Speech and Signal Processing, vol. 37. no. 8, pp. 1298-1300, 1989.