BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **18**, No 5
Special Thematic Issue on Optimal Codes and Related Topics
Sofia • 2018 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2018-0018

Parallel Fast Walsh Transform Algorithm and Its Implementation with CUDA on GPUs

Dusan Bikov[1], *Iliya Bouyukliev*[2]

[1]*Faculty of Computer Science, Goce Delchev University, Shtip, Macedonia*
[2]*Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria*
E-mails: dusan.bikov@ugd.edu.mk iliyab@math.bas.bg

***Abstract*: *Some of the most important cryptographic characteristics of the Boolean and vector Boolean functions* (*nonlinearity, autocorrelation, differential uniformity*) *are connected with the Walsh spectrum. In this paper, we present several algorithms for computing the Walsh spectrum implemented in CUDA for parallel execution on GPU. They are based on the most popular sequential algorithm. The algorithms differ in the complexity of implementations, resources used, optimization strategies and techniques. In the end, we give some experimental results.*

***Keywords*: *Walsh transform, CUDA C, GPU, Fast Walsh transform, Parallel algorithms.*

1. Introduction

The use of modern Graphics Processing Units (GPUs) has become attractive for scientific computing which is due to its massive parallel processing capability. The highly parallel structure of the modern GPUs makes them more effective than general-purpose CPU for algorithms where processing of large blocks of data is done in parallel [12, 13]. Compared with multi-core CPUs, new generation GPUs can have much higher computational power and memory bandwidth. Therefore they are attractive in many application areas [13, 16, 21].

The purpose of this paper is to assess the performance of the recent, inexpensive and widely used NVIDIA GPUs in computing the Walsh Spectrum of a Boolean function. The Walsh (Hadamard, Walsh-Hadamard, Walsh-Fourier) transform has a wide range of applications. It is used in cryptography, signal and image processing, image rendering, data compression algorithms, quantum computing, etc. Here we present several variants of an algorithm. Our motivation to create different algorithms is due to the fact that there are various optimization strategies and techniques with specific characteristics. The first algorithm is the basic and easiest one. In order to obtain a better performance we make more complex algorithms with implementing various optimization techniques. The last algorithm is designed for NVIDIA GPUs with compute capability 3.0 and higher [6].

There are many algorithms for computations connected with Walsh (Walsh-Hadamard) Transform (see for example [10, 11]), there are also many commonly used implementations, tools, libraries and mathematical software for CPU, for example Sage [23], Matlab [17], VBF Library [1], SET (S-box Evaluation Tool) [22], etc. GPU-based implementations of Walsh Transform for accelerating the decoding process on Low-Density Parity-Check codes (LDPC codes) is given in [2, 5]. There are a few GPU libraries that realize butterfly algorithms, like cuFFT [20] and BPLG [15], but they do not include the Walsh transform.

The paper is organized as follows. We describe the used GPU computing model with CUDA (Compute Unified Device Architecture) in Section 2. The main definitions connected with Boolean functions and Walsh spectrum are given in Section 3. We present there a sequential butterfly algorithm for calculating the Walsh spectrum using the binary representations of integers. Section 4 is devoted to the parallel algorithms. It includes six parallel realizations of the main algorithm. Some experimental results are given in Section 5. In the end we put a conclusion section.

2. GPU computing model with CUDA

GPUs are designed for efficient execution of thousands of threads in parallel on as many processors as possible at each moment. The computation processes are divided into many simple tasks that can be performed at the same time. This intensive multi-threading allows execution of various tasks on the GPU processors while data is fetched from or stored to the GPU global memory, ensures the scalability of the GPU computing model, and support parallel programming model [6].

A simple way to understand the difference between CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. This ability of a GPU with hundred and more cores to process thousands of threads can significantly accelerate the software over a CPU.

In other words, traditional CPUs designs are latency-oriented and they have powerful ALU, large caches, sophisticated control. Their goal is to minimize the running time of a single sequential program by avoiding task-level latency whenever possible. Techniques such as out-of-order execution, speculative execution, and sophisticated memory caches, have been developed to help to minimize latency. On the other hand GPUs have aggressively throughput-oriented design with small caches, simple control, energy efficient ALUs and require massive number of threads to tolerate latency. Broadly speaking, throughput-oriented processors rely on three key architectural features and put emphasis on many simple processing cores, extensive hardware multithreading, and use of single-instruction, multiple-data, or SIMD, execution. To hide latency of frequent movement of data and reach full utilization, GPU typically requires thousands of threads and larger data sets.

Modern NVIDIA GPU is a powerful platform developed for general purpose computing using CUDA [7]. It allows programmers to interact directly with the GPUs and run programs on them, thus effectively utilizing the advantages of parallelization. Depending on architecture CUDA cores can be organized into Streaming Multiprocessors (SMs), each having a set of registers, constants and texture caches, and on-chip shared memory as fast as local registers (one cycle latency). At any given cycle, each core executes the Same Instruction on different Data (SIMD), and the communication between multi-processors is performed through global memory. CUDA C [6] is essentially C/C^{++} language with a few extensions and a runtime library that allows executing parallel functions on the GPU. As a programming interface, CUDA C is a programming language close to C by syntax, but conceptually and semantically it is quite different from C. The source code for CUDA applications consists of a mixture of conventional C/C^{++} host code and GPU device functions. The CUDA C compiler, nvcc separates the device functions from the host code. Then it compiles the device functions and the host code, but for the latter it uses the available C/C^{++} host compiler. At the linking stage, specific CUDA runtime libraries are added for supporting explicit GPU manipulation.

The processing of the data flow has several steps. At the top level, we have a master process which runs on the CPU and performs the following steps: ini-tialises card, allocates memory in host (CPU) and on GPU (global) memory, copies data from the host (CPU) to GPU (global) memory, launches multiple instances of execution "kernel" on GPU, copies data from GPU (global) memory to host, deallocates all memory and terminates.

Data-parallel functions are written in units called kernels. The kernels are executed over the stream of data by many threads on a device in parallel. Thread is a process that performs series of independent programming instructions and it is a single instance of the kernel. Creating and destroying of threads barely require resources (time), therefore they don't have any significant impact on the performance. Threads are organized into blocks, which are sets of threads that can communicate and synchronize their execution. Maximum 1024 threads (512 threads for older GPU) per block can be launched. Each block is executed by a single SM, but SMs can execute multiple blocks simultaneously, depending on the specific GPU hardware [6]. First, a configuration on the kernel has to be made before to launch it. Triple angle brackets mark is used for kernel configuration. We define number of blocks and number of threads per block inside. Blocks and threads per block form a grid. All threads run the same code (the model SIMT – Single Instruction, Multiple Threads [14]). Each thread has an index (tID) that it uses to compute memory addresses and make control decisions. If there are M threads per block, the index of the current thread can be calculated from the index of the current block (blockIdx.x) and the number of the thread in that block (threadIdx.x) by the formula:

$$\text{tID} = \text{threadIdx.x} + \text{blockIdx.x}\cdot M.$$

The definition of the grid is given as follows:

```
mykernel <<< blocks per grid, threads per block >>> (…);
```

Each SM (Streaming Multiprocessor) organizes and executes threads in groups of 32 threads called "warps". The execution alternates between "active" warps, with warps becoming temporarily “inactive” while waiting for data. Each warp is treated separately. Usually all threads in the warp execute the same instruction at the same time.

What happens if different threads in a warp need to do different things? This is called warp divergence. Such a situation can come with if then else construction for example (Fig. 1). Then CUDA generates a correct code to handle this. But in that case there is an execution delay because any thread executes one of both conditional branches and waits until other threads execute the other branch, so the execution cost is a sum of the costs of both branches.

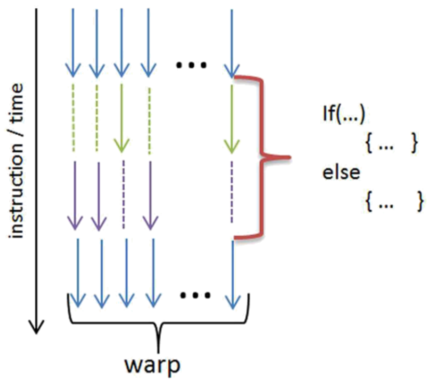


Fig. 1. Warp divergence

A memory hierarchy has to be considered while a parallel code is being written. The execution speed depends on the proper use of the memory hierarchy. Lower level memories are faster but more expensive and more limited. The registers are the fastest followed by local memory, shared memory and global memory. General overview on GPU memory model is shown in Fig. 2. Every thread has access to his local memory. The data in shared memory can be shared between all threads of the same block. All threads from all kernels can access the global memory. Since blocks are executed in an arbitrary order, if one block modifies a data element, no other block should read or write that data element in the global memory. Except these types of memory, there are additional memory and variable types. Data is copied to GPU global memory before launching the kernel.

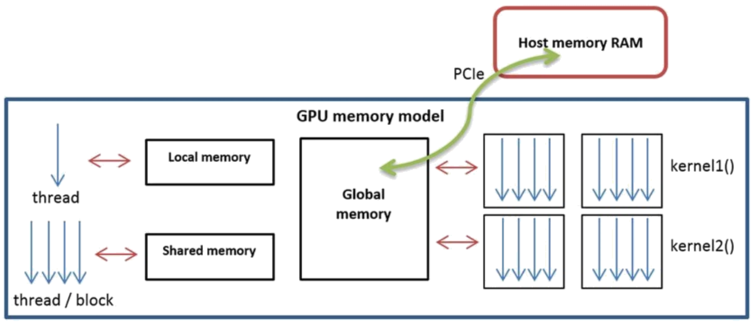


Fig. 2. GPU memory model

The memory model shows the interaction of the threads. All threads within one copy can access local shared memory but cannot see what the other copies are doing (even if they are on the same multiprocessor). There are no guarantees on the order in which the copies will execute. Therefore, for the correct execution of the kernel, in some cases a synchronization of the threads in each block is needed. Instruction __syncthreads(); inserts a “barrier” synchronization. No thread in a block is allowed to proceed beyond this point until the rest threads have reached it. Global synchronization of all threads can be performed across separate kernel launches or with Fast Barrier Synchronization [24].

3. Boolean functions and Walsh spectrum

Boolean functions play a critical role in cryptography, especially in the design of S-boxes which perform the substitutions in block ciphers. There are parameters of a Boolean function that are very important in this usage. Some of these parameters are connected with its Walsh spectrum [4].

A Boolean function f of n variables is a mapping from F_2^n into F_2, where $F_2 = \{0, 1\}$ is the field with two elements. The Truth Table TT(f) is the 2^n-dimensional vector which has the function values of f as coordinates. We can consider the vectors in F_2^n as binary representations of the integers in the interval $[0, \dots, 2^n - 1]$. This consideration is very useful if we try to describe and explain some transformations of Boolean functions and related algorithms. Here we present efficient algorithms for calculating the Walsh spectrum. The basis is a matrix and vector multiplication. In our case the considered matrices have not only recursive structure but this structure is quite specific and enables a very effective (*butterfly*) multiplication.

Walsh (Walsh-Hadamard) transform f^W of the Boolean function f is the integer valued function $f^W : F_2^n \to Z$ [4], defined by

$$f^{W}(a) = \sum_{x \in F_2^n} (-1)^{f(x) \oplus (a \odot x)} = \sum_{x \in F_2^n} (-1)^{a \odot x} (-1)^{f(x)},$$

where $a \odot x = a_1 x_1 \oplus a_2 x_2 \oplus \ldots \oplus a_n x_n$. The values of f^W are called *Walsh coefficients*. We show below that f^W can be calculated using the multiplication of an Hadamard matrix H_n by the Polarity Truth Table of the Boolean function.

Let $S = \{0, 1, 2, \dots, 2^n - 1\}$. To any integer $u \in S$ we correspond its binary representation, written as an n-dimensional binary vector, namely $\overline{u} = (u_1, u_2, \dots, u_n)$. This means that $u = u_1 2^{n-1} + u_2 2^{n-2} + \dots + u_{n-1} 2^1 + u_n$, $u_i \in F_2$. Let $S_{\text{set}} = \{\overline{0}, \overline{1}, \overline{2}, \dots, \overline{2^n - 1}\}$, and let $S_{\text{mat}}^{(n)}$ be the $2^n \times n$ matrix $S_{\text{mat}}^{(n)} = \{\overline{0}, \overline{1}, \overline{2}, \dots, \overline{2^n - 1}\}^{\mathrm{T}}$. The matrices $S_{\text{mat}}^{(n)}$ can be defined recursively in the following way:

$$S_{\text{mat}}^{(n+1)} = \begin{pmatrix} 0 & S_{\text{mat}}^{(n)} \\ 1 & S_{\text{mat}}^{(n)} \end{pmatrix}.$$

Consider the matrix $H_n^{+} = S_{\text{mat}}^{(n)} . \left(S_{\text{mat}}^{(n)}\right)^{\mathrm{T}}$. Obviously,

$$H_1^+ = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad H_2^+ = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

According to the matrix multiplication rule we have

$$H_n^+ = \begin{pmatrix} \overline{0} \odot \overline{0} & \overline{0} \odot \overline{1} & \cdots & \overline{0} \odot \overline{2^n - 1} \\ \overline{1} \odot \overline{0} & \overline{1} \odot \overline{1} & \cdots & \overline{1} \odot \overline{2^n - 1} \\ & & \ddots & \\ \overline{2^n - 1} \odot \overline{0} & \overline{2^n - 1} \odot \overline{1} & \cdots & \overline{2^n - 1} \odot \overline{2^n - 1} \end{pmatrix}.$$

It is easy to see that H_n^+ is a symmetric matrix. Its rows (and columns) are all vectors from an n dimensional vector subspace of $F_2^{2^n}$. In coding theory this subspace (without the zero coordinate) is known as a simplex code. This space together with its coset with representative (11 … 1) forms the first order Reed-Muller code [4].

Using the recurrence relation, we obtain

$$H_{n+1}^+ = S_{\text{mat}}^{(n+1)} . \left(S_{\text{mat}}^{(n+1)}\right)^{\text{T}} = \begin{pmatrix} 0 & S_{\text{mat}}^{(n)} \\ 1 & S_{\text{mat}}^{(n)} \end{pmatrix} \begin{pmatrix} 00...0 & 11...1 \\ \left(S_{\text{mat}}^{(n)}\right)^{\text{T}} & \left(S_{\text{mat}}^{(n)}\right)^{\text{T}} \end{pmatrix} = \begin{pmatrix} H_n^+ & H_n^+ \\ H_n^+ & \overline{H_n^+} \end{pmatrix},$$

where $\overline{H}$ is the ~~ne~~gation of the binary matrix *H*.

The polarity representation of a binary vector (matrix) is the vector (matrix) obtained after replacing 0 by $(-1)^0 = 1$ and 1 by $(-1)^1$. Denote by H_n the polarity representation of H_n^+. Since the negation for a binary matrix correspond to the multiplication by (–1) of its polarity representation, we have

$$H_0 = (1), \quad H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix} = H_1 \otimes H_{n-1} \quad \text{for} \quad n \geq 2.$$

The matrices H_n are Hadamard matrices of Sylvester type called also Sylvester matrices or Walsh matrices.

Let $f : F_2^n \to F_2$ be a Boolean function of *n* variables and TT(*f*) be its Truth Table, $\text{TT}(f) = (f(\overline{0}), f(\overline{1}), ..., f(\overline{2^n - 1}))$. The polarity representation of TT(*f*) is called the Polarity Truth Table and denoted by PTT(*f*). The foregoing implies that the Walsh spectrum of the Boolean function *f* is $W_f = H_n \cdot \text{PTT}(f)$ and its coordinates are the Walsh coefficients of *f* (in this equality PTT(*f*) and W_f are columns but we consider W_f also as a row depending on the particular case).

To calculate the Walsh spectrum of a Boolean function, one can use also the Fast Walsh transform which can be given by a butterfly diagram and the corresponding algorithm with complexity $O(n2^n)$ (Fig. 3 [10]). The theoretical base of the Fast Walsh transform is given by Good [9] and it follows from the factorization

$$H_n = (H_1 \otimes I_{2^{k-1}}) \cdot (I_2 \otimes H_1 \otimes I_{2^{k-2}}) \ldots (I_{2^{k-1}} \otimes H_1).$$

A vector t of length 2^n is transformed in n steps. In the beginning $t = t^{(0)} = \mathrm{PTT}(f)$. For the first step we partition $t^{(0)}$ into 2^{n-1} pairs (t_0, t_1), (t_2, t_3), etc. The new vector is

$$t^{(1)} = (t_0 + t_1, t_0 - t_1, t_2 + t_3, t_2 - t_3, \ldots, t_{2^n-2} + t_{2^n-1}, t_{2^n-2} - t_{2^n-1}).$$

In the s-th step $(1 \le s \le n)$ the vector (the table) is partitioned into intervals of length 2^s (called size in the step), and apply suitable calculations as follows:

(1) $$t_i^{(s)} = t_i^{(s-1)} + t_{i+j}^{(s-1)}, \quad t_{i+j}^{(s)} = t_i^{(s-1)} + t_{i+j}^{(s-1)}$$

for all i, $0 \le i < 2^n$, such that $i \equiv 0, 1, \ldots, 2^{s-1} - 1 \pmod{2^s}$, $j = 2^{s-1}$.

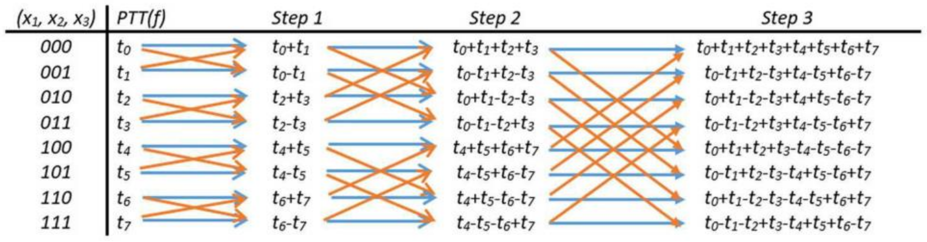


Fig. 3. Fast Walsh Transform

In [3], we have presented an algorithm (Algorithm 0) which uses the binary representations of the integers from the set S. This algorithm passes all elements of the matrix $S_{\mathrm{mat}}^{(n)}$ in n steps column by column starting from the last one. The array W_f is equal to $t^{(s)}$ after the s-th step. In the beginning, $W_f = t^{(0)} = \mathrm{PTT}(f)$. The integer j is equal to 2^{s-1} in the s-th step. The only coordinate equal to 1 in the vector $\bar{j}$ is in the position $n - s + 1$. Depending on the value in the i-th row and $(n - s + 1)$-th column of the matrix $S_{\mathrm{mat}}^{(n)}$ the algorithm calculates the next values for $W_f[i]$ and $W_f[i + j]$. After the n-th step, W_f is equal to the Walsh spectrum of the Boolean function.

Fast Walsh transform can be implemented in parallel, by using the base concept of Algorithm 0. For the parallel adaptation, we use CUDA C and make several algorithms using various optimization techniques, models and different memory types to get better performance and efficiency.

Algorithm 0. Fast Walsh Transform

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

```
$j \leftarrow 1$; $W_f \leftarrow$ PTT;
while ($j < 2^n$) do
  for $i = 0$ to $2^n - 1$ do
    if (($i \& j$) = 0) then
        temp $\leftarrow W_f[i]$;
        $W_f[i] \leftarrow W_f[i] + W_f[i+j]$;
        $W_f[i + j] \leftarrow$ temp $- W_f[i + j]$;
      end then
    end for
    $j \leftarrow 2*j$;
end while.
```

4. Parallel implementation of Fast Walsh transform

In this section we present several parallel algorithms realizing the Fast Walsh Transform (FWT). Each of them has improvements compared with the previous one. These algorithms are implemented in CUDA C.

We use the following notations, constants and variables in the pseudo codes which describe the algorithms:

- *a&b*, the bitwise AND operation of nonnegative integers *a* and *b*;
- *tID* is the index of the current thread in the grid;
- *bID* is the index of the current block;
- *tID_inblock* is the index of the thread in the current block;
- *block_size* shows the number of threads per block;
- *block_num* is the number of blocks in the grid;
- *grid_size* is the number of all threads in the grid,

$$grid\_size = block\_num \cdot block\_size.$$

Algorithm 1 is based on the sequential Algorithm 0 but with a suitable modification in order to implement it in parallel. All other algorithms are modifications of this algorithm.

Algorithm 1. Parallel Implementation of FWT

```
Input: The Polarity Truth Table PTT of the Boolean function f, with $2^n$ entries
Output: The Walsh spectrum $W_f$ of the Boolean function f, with $2^n$ entries
  Allocate memory for device copies and host copy
  Copy the input data from the host to the device
  Set a grid of blocks and threads
    size ← $2^n$;
    if (size <= 1024) then
        block_size ← size;
        block_num ← 1;
  end then
  else              /* if size > 1024 */
      block_num ← size=1024;
      block_size ← 1024;
  end else
j ← 1; r ← 0; $W_f$ ← PTT; temp ← 0;
while (j < size) do
  r ← r + 1;          /* r is the number of the current step */
    fwt_kernel($W_f$, temp, r, j)          /*Launch kernel*/
  j ← 2*j;              /* j is the size of the current step */
end while
if r is odd then $W_f$ ← temp;
Copy the result back to host
Cleanup memory
```

fwt_kernel(W_f, temp, r, j) Kernel, Algorithm 1

```
Input: The arrays $W_f$ and temp with $2^n$ entries, and the integers $r$ and $j$
Output: The arrays $W_f$ and temp
  Init tID;
  $i \leftarrow$ tID;                /* index of the thread */
  if $r$ is odd then       /* This determines where to save the intermediate result */
        if (($i$&$j$) = 0) then
            value $\leftarrow$ ($W_f[i]$ + $W_f[i + j]$);
        end then
        else
            value $\leftarrow$ ($-W_f[i]$ + $W_f[i - j]$);
        end else
      temp[$i$] $\leftarrow$ value;
  end then
  else
        if (($i$&$j$) = 0) then
            value $\leftarrow$ (temp[$i$] + temp[$i + j$]);
        end then
        else
            value $\leftarrow$ (–temp[$i$] + temp[$i - j$]);
        end else
      $W_f[i]$ $\leftarrow$ value;
  end else
```

The kernel (**fwt_kernel**) in Algorithm 1 uses two arrays, namely W_f and temp, and two integer variables r and j, where r is the number of the current step, and j is the size of the current step in the butterfly algorithm. In any step the kernel takes data from one of the arrays, depending on the step, calculates values for the current step of the butter y algorithm, and writes the result in the other array. It is impossible to use only one array in this simple implementation, because there is no global synchronization in the kernel between the threads in different blocks. In fact, the synchronization is achieved by launching the kernel n times from the main (sequential) program.

We have to mention that Algorithm 1 has not very good performance because the kernel uses global arrays at any step. Practically, the time for launching the kernel (creating and destroying threads) is negligible.

In some cases, warp divergence can lead to a big loss of parallel efficiency, and this is one of the things we should pay attention to. Therefore we propose Algorithm 2 which is a modification of Algorithm 1. The difference is only in the kernel where the statement if else is replaced by an algebraic expression to avoid warp divergence. The expression is

$$\text{value} \leftarrow (1 - \text{ii})*(W_f[i] + W_f[i + j]) + \text{ii}*(-W_f[i] + W_f[i - j]).$$

The variable ii takes two values, ii = 0 if i&j = 0, and ii = 1 otherwise.

fwt_kernel_no_if(W_f, temp, r, j) Kernel, Algorithm 2

```
Input: The arrays $W_f$ and temp with $2^n$ entries, and the integers $r$ and $j$
Output: The arrays $W_f$ and temp
  Init tID;
  $i \leftarrow$ tID;            /* index of the thread */
  ii $\leftarrow (i\&j)/j$;       /* Conditional variable */
  if $r$ is odd then
    value $\leftarrow (1 - ii)*(W_f[i] + W_f[i + j]) + ii*(-W_f[i] + W_f[i - j])$;
    temp[$i$] $\leftarrow$ value;
  end then
  else
    value $\leftarrow (1 - ii)*(temp[i] + temp[i + j]) + ii*(-temp[i] + temp[i - j])$;
    $W_f[i] \leftarrow$ value;
  end else
```

Algorithm 3 is a modification of Algorithm 1. In this algorithm, any thread executes more computations. The pseudo code of the parallel implementation is shown in Algorithm 3.

Algorithm 3 takes the same input as Algorithm 1, with an additional parameter *M*. In the previous algorithms, any thread calculates only one value for the current step in the butterfly algorithm. In Algorithm 3, any thread calculates *M* values. We would like to see how this affects the performance. This is the reason to present this new algorithm. Actually, we only add for loop in the kernel:

for $i = tID*M$ to $(tID + 1)*M - 1$ do.

In addition, the grid configuration depends on *M*. For computing the same Walsh spectrum, larger *M* means less number of threads and more work per thread. An example of the grid configuration is shown in pseudo code above.

Algorithm 3. Parallel Implementation of FWT

```
Input: The Polarity Truth Table PTT of the Boolean function $f$, with $2^n$ entries,
and an integer $M = 2^s$, $s < n$
Output: The Walsh spectrum $W_f$ of the Boolean function $f$, with $2^n$ entries
  Allocate memory for device copies and host copy
  Copy the input data from the host to the device
  Set a grid of blocks and threads, the grid con guration depends on $M$
     size $\leftarrow 2^n$;
     if (size/$M$ <= 1024) then
        block_size $\leftarrow$ size/$M$;
        block_num $\leftarrow$ 1;
     end then
     else       /* if size/$M$ > 1024 */
        block_num $\leftarrow$ (size/$M$)/1024;
        block_size $\leftarrow$ 1024;
     end else
     $j \leftarrow 1$; $r \leftarrow 0$; $W_f \leftarrow$ PTT;
```

```
    while ($j < 2^n$) do
     $r \leftarrow r + 1$;
        **fwt_kernel_M**($W_f$, temp, $r$, $j$, $M$)    /*Launch kernel*/
     $j \leftarrow 2*j$;
    end while
    if $r$ is odd then $W_f \leftarrow$ temp;
    Copy the result back to host
    Cleanup memory
```

```
**fwt_kernel_M**($W_f$, temp, $r$, $j$, $M$)  Kernel, Algorithm 3
*Input:* The arrays $W_f$ and temp with $2^n$ entries, and the integers $r$, $j$, $M$
*Output:* The arrays $W_f$ and temp
  Init tID;
  for $i$ from tID*$M$ to (tID + 1)*$M$ – 1 do
    if $r$ is odd then
      if (($i$&$j$) = 0) then
          value $\leftarrow$ ($W_f[i] + W_f[i + j]$);
      end then
      else
          value $\leftarrow$ ($-W_f[i] + W_f[i - j]$);
      end else
    temp[$i$] $\leftarrow$ value;
  end then
  else
      if (($i$&$j$) = 0) then
          value $\leftarrow$ (temp[$i$] + temp[$i + j$]);
      end then
      else
          value $\leftarrow$ (–temp[$i$] + temp[$i - j$]);
      end else
    $W_f[i] \leftarrow$ value;
  end else
end for
```

Algorithm 4 uses shared memory combined with the kernel from Algorithm 1. At first stage, we use shared memory for calculations until a certain step (depending on the shared memory limitation). After that we use global memory for computations. The pseudo code is shown in Algorithm 4.

Algorithm 4 has two kernels. The first one uses shared memory for calculations until a certain step. It can launch maximum 1024 threads per block. (because of the limitation of GPU) and the data is shared between all threads from the same block. If $n > 10$ the dimension of the vector considered is larger than 1024. Therefore the array of size 2^n can be partitioned into parts of length 1024. Each of these parts is copied into a special type of memory, and this is the shared memory of the corresponding block. The writing and reading in the shared memory is considerably faster.

Moreover, the writing and reading in this memory can be synchronized in CUDA terms. This allows the kernel to do all calculations for a fixed number of steps in the Fast Walsh Transform (which is 10 steps for 1024 threads). In the eleventh step, the FWT uses data from different parts of the array and therefore we go out of the size of the shared memory and we have to use the global memory again. For that we use the second kernel.

Algorithm 4. Parallel implementation of FWT

```
Input: The Polarity Truth Table PTT of the Boolean function $f$, with $2^n$ entries
Output: The Walsh spectrum $W_f$ of the Boolean function $f$, with $2^n$ entries
  Allocate memory for device copies and host copy
  Copy the input data from the host to the device
  Set a grid of blocks and threads
      size ← $2^n$;
      if (size <= 1024) then
          block_size ← size;
          block_num ← 1;
      end then
      else        /* if size > 1024 */
          block_num ← size/1024;
          block_size ← 1024;
      end else
$W_f$ ← PTT;
fwt_kernel_SM($W_f$, block_size)   /*Launch Shared memory, Kernel*/
$j$ ← 1024; $r$ ← 10;
while ($j < 2^n$) do
  $r$ ← $r$ + 1;
      fwt_kernel($W_f$, temp, $r$, $j$)        /*Kernel, Algorithm 1*/
  $j$ ← 2 * $j$;
end while
if $r$ is odd then $W_f$ ← temp;
Copy the result back to host
Cleanup memory
```

fwt_kernel_SM(W_f, block_size) Kernel, Algorithm 4

```
Input: The array $W_f$ with $2^n$ entries, and block size
Output: The array $W_f$
  Declare shared memory as the array tmpsdata of length block_size
  Init tID, tID_inblock;
  $i$ ← tID_inblock;
  value ← $W_f$[tID];    /*Local variable for every thread, taken from $W_f$ */
  $j$ ← 1;
  while $j$ < block_size do
    tmpsdata[$i$] ← value;
    __syncthreads();
```

```
        if (($i$&$j$) = 0) then
            value ← (value + tmpsdata[$i + j$]);
        end then
        else
            value ← (–value + tmpsdata[$i – j$]);
        end else
      __syncthreads();
      j ← 2*$j$;
    end while
    $W_f$[tID] ← value;
```

After starting the kernel, a shared memory is declared, and data is written from W_f to the shared memory. Each thread takes two elements from the shared memory, add or subtract them, and stores the result in a local variable. At the end of the step threads have to be synchronized.

After passing the first kernel, the algorithm launches the second kernel if needed (if $n > 10$). The second kernel is fwt_kernel from Algorithm 1.

Algorithm 5. Parallel Implementation of FWT

```
Input: The Polarity Truth Table PTT of the Boolean function $f$, with $2^n$ entries
Output: The Walsh spectrum $W_f$ of the Boolean function $f$, with $2^n$ entries
  Allocate memory for device copies and host copy
  Copy the input data from the host to the device
  Set a grid of blocks and threads
      size ← $2^n$;
      if (size <= 1024) then
          block_size ← size;
          block_num ← 1;
      end then
      else        /* if size > 1024 */
          block_num ← size/1024;
          block_size ← 1024;
      end else
$W_f$ ← PTT;
fwt_kernel_SM($W_f$, block_size)    /*Kernel, Algorithm 4*/
if (size > 1024) then         /* Shared memory + Memory pattern */
    fwt_kernel_SM_MP($W_f$, block_num)
end then
Copy the result back to host
Cleanup memory
```

Algorithm 5, we use shared memory combined with a memory pattern. The memory pattern is used for tracking the intermediate results from the steps of the calculation (this limitation comes from the shared memory). Here we obtain better performance compared with the other algorithms. This algorithm is designed for maximum 2^{20} entries or the maximum length of the used vector can be 2^{20}. We show the pseudo code in Algorithm 5.

Algorithm 5 has two kernels. The first kernel is fwt_kernel_SM – the first kernel of Algorithm 4. The difference between Algorithm 4 and Algorithm 5 comes after finishing this kernel. The second kernel is similar to the first kernel but with the adding a memory pattern. After the first kernel we have to sum the elements W_f [0] and W_f[1024], W_f[1] and W_f[1025], etc., but they are in different blocks, so in different shared memories. The memory pattern rearranges the shared memory in such a way that the memory elements from different blocks are set in order to perform FWT from the beginning.

The arrangement is given by the following expression:

ji = (tID_inblock)*block_num + bID.

The easiest way to explain it is by using a two dimensional array (a matrix). Suppose that we have put the elements of the grid into a block_num × 1024 matrix *G* such that the elements from the *i*-th block are set in the *i*-th row. Then the number of the rows coincides with the number of the blocks, and the number of the columns is the number of the threads in corresponding blocks. So the element in the *i*-th row and *j*-th column correspond to the *j*-th thread (*j* = tID_inblock) in *i*-th block (*i* = bID) which means that the thread with index tID reads W_f[tID]. It is not difficult to calculate the global index of this thread in the grid tID = 1024**i* + *j* = 1024*bID + tID_inblock. To rearrange the memory, we transpose the matrix *G*. Then the thread with index tID reads the value from W_f[ji], ji = block_num**j* + *i* = (tID_inblock)*block_num + bID, and writes it in the cell with number tID_inblock of the shared memory. After the last step the kernel writes the obtained result in W_f[ji].

In Fig. 4 we show memory movement for Boolean function *f* of 11 variables, which means that the vector W_f has 2048 entries.

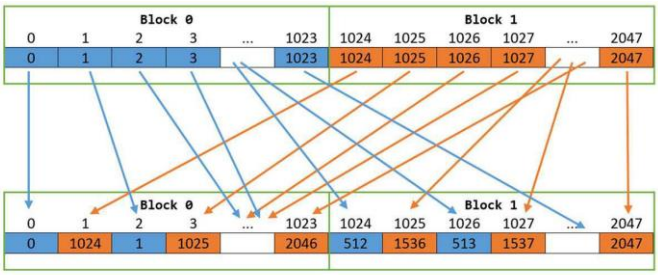
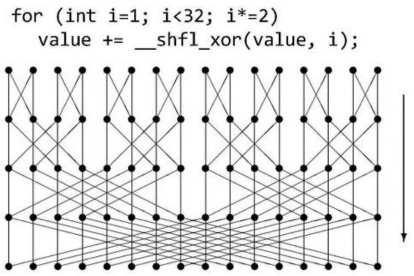


Fig. 4. Memory pattern for a Boolean function *f*, with 2048 entries

In Algorithm 6, we use warp shuffles. Warp shuffle is a machine instruction for NVIDIA GPUs with compute capability 3.0 and higher. Threads are executed in warps and any warp contains 32 threads. All threads in the warp execute the same instruction at the same time. Warp shuffles give a mechanism for moving data (values of local register variables) between threads in the same warp, without using any shared memory. There are 4 variants of warp shuffles, but for our need we use __shfl_xor(*a*, *i*) where *a* is a local register variable, and *i* is an integer, $0 < i < 16$ [8]. Then all threads in the warp are partitioned into 16 pairs, namely the threads with indexes (0, *i*), (1, *i* + 1), …, (31 – *i*, 31).

The value of the variable *a* in a thread becomes available for the other thread in the pair (Fig. 5).



```
for (int i=1; i<32; i*=2)
   value += __shfl_xor(value, i);
```

Fig. 5. Warp shuffles, *_shfl_xor*

Algorithm 6 has two kernels. The first kernel combines __shfl_xor and shared memory for calculations until certain steps. From the limitation of the shared memory, the combination of __shfl_xor and shared memory can calculate up to 10 step of FWT (minimum 5 steps). The first kernel is similar to the first kernel from Algorithm 4, with additional __shfl_xor part segment as it is shown below:

```
$j \leftarrow 1$
while $j < 32$ do
  ii ← $(i\&j)/j$;
  value ← ii*(__shfl_xor(value, $j$) – value)+(1–ii)*(__shfl_xor(value, $j$)+value);
   $j \leftarrow 2*j$
end while
```

In this segment, we make FWT on a level of warp. The wrap shuffle __shfl_xor does all the work here and the expression here is similar to expression from Algorithm 2.

The second kernel here is similar to the first kernel with additional memory pattern. We have already mentioned that the memory pattern is used to rearrange the memory in such a way that memory elements from different blocks are set in order to perform FWT from the first step. After a certain number of steps we do a rearrangement again. The additional parameter for_shfl is a check condition for the construction __shfl_xor in case if there are less than five steps of computations with the second kernel.

Algorithm 6. Parallel implementation of FWT

```
Input: The Polarity Truth Table PTT of the Boolean function $f$, with $2^n$ entries
Output: The Walsh spectrum $W_f$ of the Boolean function $f$, with $2^n$ entries
    Allocate memory for device copies and host copy
    Copy the input data from the host to the device
    Set a grid of blocks and threads
        size ← $2^n$;
        if (size <= 1024) then
            block_size ← size;
            block_num ← 1;
            end then
            else     /* if size > 1024 */
                block_num ← size/1024;
                block_size ← 1024;
                for_shfl ← block_num;
                if (block_num > 32) then
                    for_shfl ← 32;
                end then
            end else
$W_f$ ← PTT;
**fwt_kernel_shfl_xor_SM**($W_f$, block_size)   /* Warp shuffles-Shared memory*/
if (size > 1024) then
            **fwt_kernel_shfl_xor_SM_MP**($W_f$, block_num, for_shfl)
            /* Warp shuffles-Shared memory-Memory pattern */
end then
Copy the result back to host
Cleanup memory
```

fwt_kernel_shfl_xor_SM(W_f , block_size) Kernel, Algorithm 6

```
*Input:* The array $W_f$ with $2^n$ entries, and block_size
*Output:* The array $W_f$
Declare shared memory as the array tmpsdata of length block size
Init tID, tID_inblock;
$i$ ← tID_inblock;
value ← $W_f$ [tID];  /*Local variable for every thread, taken from $W_f$ */
$j$ ← 1;
while $j$ < 32 do
    ii ← ($i$&$j$)/$j$;
    value←ii*(__shfl_xor(value, $j$)–value)+(1–ii)*(__shfl_xor(value,$j$)+value);
    $j$ ← 2*$j$;
end while
while $j$ < block_size do
    tmpsdata[$i$] ← value;
    __syncthreads();
        if (($i$&$j$) = 0) then
                value ← (value + tmpsdata[$i + j$]);
        end then
        else
                value ← (–value + tmpsdata[$i – j$]);
        end else
    __syncthreads();
    $j$ ← 2*$j$;
end while
$W_f$ [tID] ← value;
```

```
fwt_kernel_sh_xor_SM_MP($W_f$, block_num, for_shfl)  Kernel, Algorithm 6
Input: The array $W_f$ with $2^n$ entries, and block_ size
Output: The array $W_f$
  Declare shared memory as the array tmpsdata of length block_size
  Init tID, tID_inblock;
  $i \leftarrow$ tID_inblock;
  ji = (tID_inblock)*block_num + bID;
  value $\leftarrow W_f$ [ji];    /*Local variable for every thread, taken from $W_f$ */
  $j \leftarrow 1$;
  while $j$ < for_shfl do
      ii $\leftarrow (i\&j)/j$;
      value $\leftarrow$ ii*(__shfl_xor(value, $j$) – value)+(1 – ii)*(__shfl_xor(value, $j$)+
      value);
      $j \leftarrow 2*j$;
  end while
  while $j$ < block_num do
      tmpsdata[$i$] $\leftarrow$ value;
      __syncthreads();
          if (($i\&j$) = 0) then
                  value $\leftarrow$ (value + tmpsdata[$i + j$]);
      end then
      else
                  value $\leftarrow$ (–value + tmpsdata[$i – j$]);
      end else
      __syncthreads();
      $j \leftarrow 2*j$;
end while
$W_f$ [ji] $\leftarrow$ value;
```

Table 1. Description of the test platforms

Environment	Platform 1	Platform 2
CPU	Intel i3-3110M	Intel Xeon E5-2640
Memory	4 GB DDR3 1333 MHz	48GB DDR3 1333 MHz
OS	Win7×64 SP1	Win7×64 SP1
Compiler	MSVC 2010	MSVC 2012
GPU	GeForce GT 740M	GeForce GTX TITAN
Driver	v347.62, SDK 7.0	v347.62, SDK 7.0

Experimental evaluations and the time efficiency of all algorithms are shown in the next section.

5. Experimental evaluation

In this section we present our experimental results. The test platforms that were used in our experiments are described in Table 1. Platform 1, a graphic card NVIDIA GeForce GT 740M [18], has 384 cores running at 0.9 GHz and 28.8 GB/s memory bandwidth. Platform 2, a graphic card NVIDIA GeForce GTX TITAN [19], has 2688 cores running at 837 MHz and 288.4 GB/s memory bandwidth. All algorithms are implemented in parallel computing platform and programming model CUDA [6]. We have used CUDA Toolkit 7.0 and development environment MS Visual Studio 2010 for Platform 1 and the same version CUDA Toolkit 7.0 but MS Visual Studio 2012 as a development environment for Platform 2. Program are executed in Active solution configuration-Release, and Active solution platform-Win32. We denote Platform 1 by P1 and Platform 2 by P2.

We run the programs with input array with 2^n entries for $n = 7, \ldots, 18$. All data resides in GPU device memory at the beginning of each test so there is no data transfer to CPU. This prevents interaction with other significant factors in this study.

For the purposes of comparison, we implement Algorithm 0 sequentially in programming language C^{++} using development environment MS Visual Studio 2010. All CPU examples are executed on Platform 1 (Intel i3-3110M) in Active solution configuration – Release, and Active solution platform Win32.

Fig. 6 shows the execution time for calculating the Walsh spectrum for different number of threads per block (Algorithm 1, Platform 1). The blue and the red lines show the execution time for calculating the Walsh spectrum of a Boolean function of 16 variables, but the red line presents the program which does not use synchronization (in some cases the program without synchronization does not give a right answer but the synchronization slows down the execution time). The green line shows the execution time for calculating the Walsh spectrum of a Boolean function of 15 variables (in this case the spectrum is a vector with 2^{15} coordinates). Looking at the graphic, we can conclude that we have the fastest execution time if we use 128 threads per block. But the increase in the execution time for 1024 threads per block is relatively small.

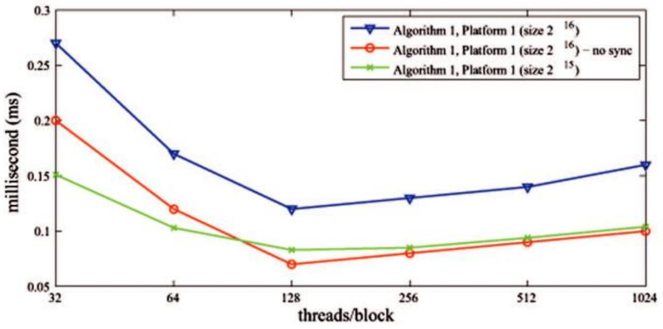


Fig. 6. Relation between time and number of threads per block (Platform 1)

The results for different values of the parameter *M* versus execution time (Algorithm 3, Platform 1) for a Boolean function with 16 variables is shown on Fig. 7. We see, that if there are less threads but any thread calculates more.

Walsh coefficients, the execution time increases. In Fig. 7, *M* = 32/2048 means that the grid has 2048 threads, and any thread calculates *M* = 32 coefficients.

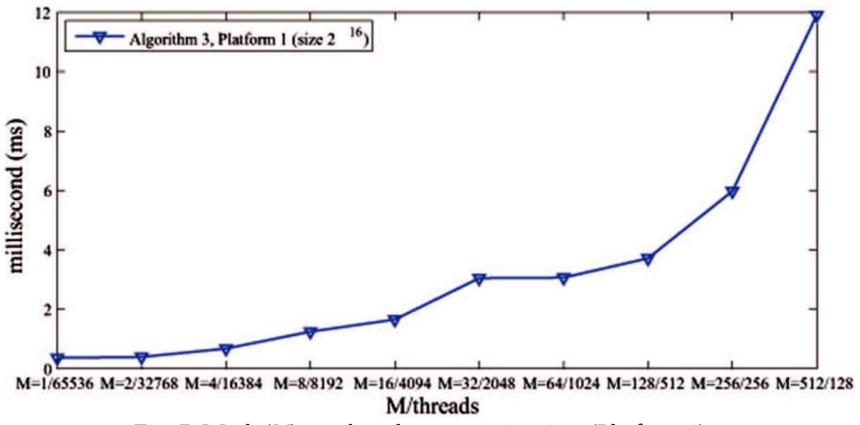


Fig. 7. Work (*M*) per thread vs. execution time (Platform 1)

The comparison between the CPU Algorithm 0, and the parallel algorithms (Platform 1) is shown in Fig. 8. We denote Algorithm 1 by A1, Algorithms 4, 5 and 6 by A4, A5 and A6, respectively. The blue line represents the CPU Algorithm 0's implementation in C^{++}, and the other lines shows the performance of Algorithms 1, 4, 5, 6. Obviously, there is a point in which the GPU implementation becomes faster (detailed results for the execution time are shown in Table 2).

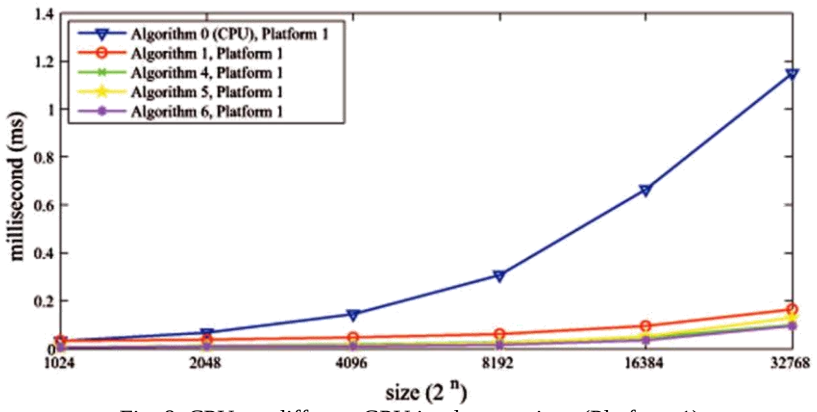


Fig. 8. CPU vs. different GPU implementations (Platform 1)

A comparison between the first, fourth, fifth and sixth parallel algorithms is shown in Fig. 9 (Platform 1). The parallel implementation algorithms are colored. As we have expected, every further algorithm has better execution time than the previous one for larger *n*.

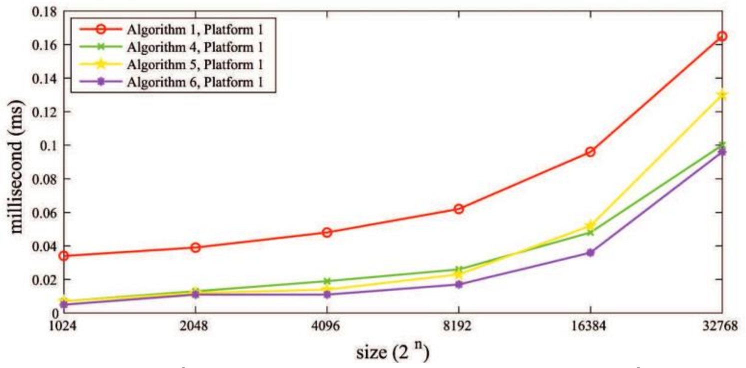


Fig. 9. Time for calculating W_f - GPU implementations (Platform 1)

One of our main goals is to achieve acceleration of the speedup between the sequential and parallel implementation where the speedup is given by the formula

$$S_p = \frac{T_0(n)}{T_p(n)}.$$

Here n is the size of the input data (in our case the number of variables of the Boolean function), $T_0(n)$ is the execution time of the fastest known sequential algorithm, and $T_p(n)$ is the execution time of the parallel algorithm.

Table 2 shows the execution times (in ms) of the implementations of CPU and GPU algorithms for different sizes of the input data, as well as the speedups for the different GPU implementations. The speedup of the parallel algorithm with number i is denoted by S_i, $i = 1, \dots, 6$. We see in the table that CPU is faster for small n. For larger n, more threads are used and therefore the computation is faster than in the case of sequential programming. The fourth, fifth and sixth algorithms shows that the size has to be at least 256 in order to have faster GPU than CPU implementation. However, there are limitations for the size of input data which depend on the problem, the algorithm, GPUs, the libraries, the model, etc.

Table 2. CPU vs. GPU implementations, Platform 1

Size	CPU (ms)	A1 (ms)	S_1	A4 (ms)	S_4	A5 (ms)	S_5	A6 (ms)	S_6
2^7	0.003	0.024	< 1	0.0066	< 1	0.0066	< 1	0.0056	< 1
2^8	0.007	0.026	< 1	0.0066	1.060	0.0066	1.060	0.0057	1.222
2^9	0.015	0.028	< 1	0.0069	2.272	0.0069	2.272	0.0058	2.547
2^{10}	0.033	0.034	< 1	0.0071	4.647	0.0071	4.647	0.0059	5.584
2^{11}	0.068	0.039	2	0.013	5.230	0.0124	5.483	0.0116	5.862
2^{12}	0.145	0.048	3.02	0.019	7.631	0.0147	9.863	0.0119	12.156
2^{13}	0.308	0.062	4.967	0.026	11.84	0.023	13.39	0.0174	17.647
2^{14}	0.665	0.096	6.927	0.048	13.85	0.052	12.78	0.0366	18.085
2^{15}	1.148	0.165	6.961	0.1	11.48	0.13	8.836	0.0965	11.901
2^{16}	3.116	0.366	8.513	0.24	12.98	0.28	11.12	0.2042	15.259
2^{17}	6.87	1.561	4.401	0.85	8.082	0.595	11.54	0.5379	12.771
2^{18}	14.81	3.571	4.149	2.058	7.207	1.207	12.27	1.1187	13.245

Another interesting observation (Table 2) for the fourth and fifth algorithms is about the intersection on time execution. In one point memory pattern has higher price (spend more time on memory movement) than shared memory computation. Also for sixth algorithm we can observe the time execution for 1024 to 2048 how climbing doubles. This duplication is due to the memory pattern or time spent to rearrange the memory.

Experimental results for Platform 2 are shown in Fig. 10. We included only tests for $n = 14, \ldots, 18$, in the graphic because there is no significant improvement for smaller n compared with Platform 1.

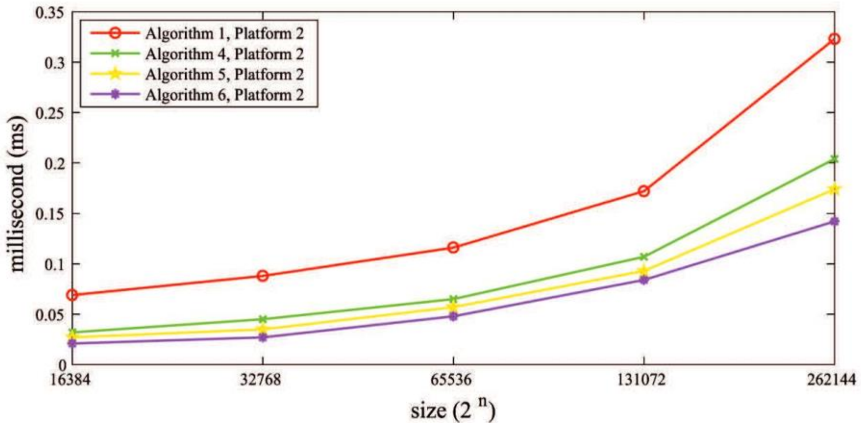


Fig. 10. Time for calculating $[W_f]$ GPU implementation (Platform 2)

Fig. 11 shows a comparison between Platform 1 and Platform 2 for the best Algorithm (Algorithm 6). As we can see in Fig. 11 the difference between execution times of both platforms increases with the size of data. In Table 3 we give some details for the comparison between Platform 1 and Platform 2 for Algorithm 6. This gap is due to the fact that Platform 2 GPU has better hardware performance.

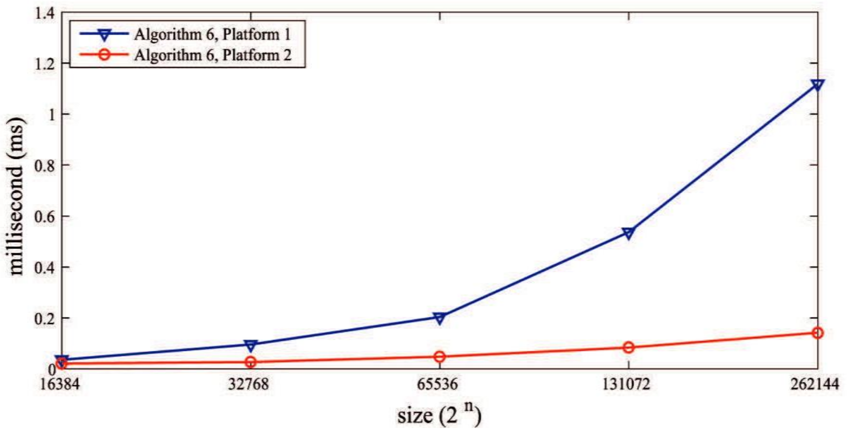


Fig. 11. Running time (Algorithm 6): Platform 1 vs. Platform 2

Table 3. Algorithm 6: Platform 1 vs. Platform 2 and CPU vs. Platform 2(A6)

Size	A6,P1(ms)	A6,P2(ms)	S_pA6:P1vsP2	CPU(ms)	S_p:CPUvs(A6)P2
2^{14}	0.036	0.021	1.71	0.665	31.66
2^{15}	0.096	0.027	3.55	1.148	42.51
2^{16}	0.204	0.048	4.25	3.116	64.91
2^{17}	0.537	0.084	6.40	6.87	81.78
2^{18}	1.118	0.142	7.87	14.81	104.30

6. Conclusion

In this paper, we present parallel algorithms for computing the Walsh spectrum of a Boolean function with widely used NVIDIA GPUs. We show how the basic algorithm can be improved in order to obtain better performance. Here we compare algorithms with and without synchronization, memory pattern, wrap shuffles, etc. By choosing proper optimization techniques and appropriate methods, the efficiency and performance can be increased.

To measure the execution time, we execute the program million times and take the average execution time. However, a deviation of ±5% may occur in next measuring of the execution time for the same size.

***Acknowledgements*:** This research is supported by Bulgarian Science Fund under Contract DN-02-2/13.12.2016.

References

1. A l v a r e z-C u b e r o, J., P. Z u f i r i a. A C++ Class for Analysing Vector Boolean Functions from a Cryptographic Perspective. – In: Proc. of International Conference on Security and Cryptography (SECRYPT'10), 2010, pp. 512-520.
2. A n d r a d e, J., G. F a l c a o, V. S i l v a. Optimized Fast Walsh-Hadamard Transform on GPUs for Non-Binary LDPC Decoding. – Parallel Computing, Vol. **40**, 2014, No 9, pp. 449-453.
3. B o u y u k l i e v, I., D. B i k o v. Applications of the Binary Representation of Integers in Algorithms for Boolean Functions. – In: Proc. of 44th Spring Conference of the Union of Bulgarian Mathematicians, Mathematics and Education in Mathematics, 2015, pp. 161-166.
4. C a r l e t, C. Boolean Functions for Cryptography and Error Correcting Codes. – In: C. Crama and P. Hammer, Eds. Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge University Press, 2010, pp. 257-397.
5. C o p e l a n d, A. D., N. B. C h a n g, S. L u n g. GPU Accelerated Decoding of High Performance Error Correcting Codes. – In: Proc. of 14th Annual Workshop on HPEC, Lexington, Massachusetts, USA, 2010.
6. CUDA C Programming Guide.
   https://docs.nvidia.com/cuda/cuda-c-programming-guide/
7. CUDA Homepage.
   http://www.nvidia.com/object/cuda home new.html
8. D e m o u t h, J. Kepler's Shuffle: Tips and Tricks. – GPU Technology Conference, 2013.
   http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf
9. G o o d, I. J. The Interaction Algorithm and Practical Fourier Analysis. – Journal of the Royal Statistical Society, Vol. **20**, 1958, No 2, pp. 361-372.

10. J o u x, A. Algorithmic Cryptanalysis. Chapman & Hall/CRC Cryptography and Network Security Series, 2009.
11. K a r p o v s k y, M. G., R. S. S t a n k o v i c, J. T. A s t o l a. Spectral Logic and Its Applications for the Design of Digital Devices. Wiley, 2008.
12. K i r k, D. B., W e n-m e i W. H w u. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier, 2013.
13. K u r z a k, J., D. A. B a d e r, J. D o n g a r r a. Scientific Computing with Multicore and Accelerators. CRC Press, 2010.
14. L i n d h o l m, E., J. N i c k o l l s, S. O b e r m a n, J. M o n t r y m. NVIDIA Tesla: A Unied Graphics and Computing Architecture. – IEEE Micro, Vol. **28**, 2008, Issue 2.
15. L o b e i r a s, J., M. A m o r, R. D o a l l o. BPLG: A Tuned Buttery Processing Library for GPU Architectures. – International Journal of Parallel Programming, Vol. **43**, 2015, No 6, pp. 1078-1102.
16. M a c i o l, P., K. B a n a s. Testing Tesla Architecture for Scientific Computing: The Performance of Matrix-Vector Product. – In: Computer Science and Information Technology, IMCSIT 2008, pp. 285-291.
17. MATLAB Platform for Solving Engineering and Scientific Problems. **https://www.mathworks.com/products/matlab/**
18. NVIDIA GeForce GT 740M Specification. **http://www.geforce.com/hardware/notebook-gpus/geforce-gt-740m**
19. NVIDIA GeForce GTX TITAN Specification. **http://http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan /specifications**
20. NVIDIA: CUDA cuFFT Library. **http://docs.nvidia.com/cuda/cufft/**
21. O w e n s, J. D., M. H o u s t o n, D. L u e b k e, S. G r e e n, J. E. S t o n e, J. C. P h i l l i p s. GPU Computing. – Proc. of IEEE, Vol. **96**, 2008, No 5, pp. 879-899.
22. P i c e k, S., L. B a t i n a, D. J a k o b o v i c, B. E g e, M. G o l u b. S-Box, SET, Match: A Toolbox for S-Box Analysis. – In: Information Security Theory and Practice. Securing the Internet of Things, Lecture Notes in Computer Science, Vol. **8501**, 2014, pp. 140-149.
23. Sage Mathematics Software. **http://www.sagemath.org/**
24. S h u c a i, X i a o, W u-c h u n F e n g. Inter-Block GPU Communication via Fast Barrier Synchronization. – In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10), 2010.

Received 28.09.2017; Accepted 30.11.2017