FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 24, no. 3, December 2011, 483-499

GPU Accelerated Computation of Fast Spectral Transforms

Dušan B. Gajić and Radomir S. Stanković

Abstract: This paper discusses techniques for accelerated computation of several fast spectral transforms on graphics processing units (GPUs) using the Open Computing Language (OpenCL). We present a reformulation of fast algorithms which takes into account peculiar properties of transforms to make them suitable for the GPU implementation. A special attention is paid to the organization of computations, memory transfer reductions, impact of integer and Boolean arithmetic, different structure of algorithms, etc. Performance of the GPU implementations is compared with the classical C/C++ implementations for the central processing unit (CPU). Experiments confirm that, even though the spectral transforms considered involve only simple arithmetic, significant speedups are achieved by implementing the algorithms in OpenCL and performing them on the GPU.

Keywords: Spectral transforms; Fast Fourier Transform (FFT); GPGPU; OpenCL.

1 Introduction

In traditional computer systems, graphics processing units (GPUs) have been used as fixed-function hardware for rendering graphics. The recent evolution of their architecture towards support of general algorithmic tasks has lead to the technique of general purpose computing on GPUs (GPGPU) which offers increased computational power and memory bandwidth for many practical applications [1–4].

Switching theory and logic design often require computations with large vectors. In particular, this is the case when spectral transforms are used for analysis

Manuscript received August 15, 2011. An earlier version of this paper was presented at the Reed-Muller 2011 Workshop, May 25-26, 2011, Tuusula, Finland.

The authors are with the Department of Computer Science, Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, 18 000 Niš, Serbia (e-mails: `dule.gajic@gmail.com` and `radomir.stankovic@gmail.com`).

Digital Object Identifier: 10.2298/FUEE1103483G of logic functions and design of corresponding networks [5, 6]. Performing spectral transforms of large functions is a computationally intensive task in spite of the existence of fast algorithms [5, 6].

In this paper, we investigate GPU implementations of several spectral transforms, often used in switching theory and logic design, and perform an analysis of their efficiency. The motivation for the research comes from the following considerations. In [7, 8], it is shown that the Fast Fourier Transform (FFT) [8], is extremely well suited for processing on GPUs because it involves intense complex number arithmetic and transcendental operations. The Walsh transform [5, 6], that is the Fourier transform on finite dyadic groups, reduces to performing additions and subtractions. We show that in spite of the simplicity of the arithmetic operations involved in the Walsh transform, a considerable speedup can be achieved by a simple adaptation of algorithms to the GPU architecture.

We also consider the Reed-Muller transform and the arithmetic transform [5,6], that are based on the basis vectors of the same form, however involving integer and Boolean operations, respectively. The idea behind the selection of these transforms is to compare the performance of their implementations since GPUs do not natively support Boolean vectors and on the hardware level interpret them as integers [9–11].

The Walsh, the Reed-Muller, and the arithmetic transforms have the same time complexity of $O(N \log_2 N)$, where $N = 2^n$ is the size of the vector and n is the number of variables in the function. These spectral transforms have transform matrices that are Kronecker product representable. Therefore, we also implemented and analyzed the Haar transform [5, 6, 12], that is not a Kronecker representable transform, but has a layered-Kronecker structure [12] and is characterized by the time complexity of $O(N)$. Although this transform offers less data parallelism and is computationally less demanding than the Walsh transform, and considerably less than the FFT, the experimental results confirm that even in this case significant speedups can be achieved.

For the Walsh and the Haar transforms, we also compared GPU implementations based on two different classes of algorithms, the in-place fast algorithms of the Cooley-Tukey type and the so-called fast algorithms with constant geometry [5, 6]. This comparison was done in order to explore various approaches in implementing these families of different algorithms for the GPU and develop techniques to improve performance. For the implementation, functions and spectra are represented by vectors, stored in memory as arrays, which are a data structure well suited for the underlying GPU hardware. Computations in the FFT-like algorithms discussed are performed componentwise over elements of arrays.

2 Related Work

Implementation of various FFT algorithms on different technological platforms is a widely considered subject, see for instance [5,6,8] and references therein. In particular, the GPU-accelerated calculation of FFT algorithms using CUDA is described in [7, 13]. NVIDIA provides an FFT library called CUFFT, as well as CUDA SDK [14] examples of the Walsh and the Haar transforms on single-precision floating point numbers, which all use CUDA platform for improving program performance. A GPU-based CUDA implementation of low-density parity-check codes (LDPC) that uses the Walsh transform and the inverse Walsh transform in accelerating the decoding process is presented in [2]. The method in [2] reduces the LDPC decoding time from 9 days on the CPU to less than 6 minutes on an array of high-performance GPUs.

OpenCL is a more recent development in GPGPU than CUDA. AMD Accelerated Parallel Processing SDK [15] has examples of the Walsh and the Haar transforms on single-precision floating point numbers, but these implementations are very limited and could not be used for comparison (e.g., the Haar transform from [15] offers GPU processing only for vectors with $N \leq 512$). To the best of our knowledge, except for the earlier version of this paper [16], there are no published discussions of OpenCL GPU implementations of the spectral transforms considered here.

3 GPU Computing Model with OpenCL

The GPU computing model follows the GPU architecture which is based on a parallel array of many programmable processors. The architecture of many-core GPUs is quite different from multi-core CPUs [17]. GPUs are designed for efficient execution of thousands of threads in parallel on as many processors as possible at each moment. Thus, the computational processes in the GPU computing are divided into many simple tasks that can be performed at the same time. This intensive multi-threading allows execution of various tasks on GPU processors whilst data is fetched from and/or stored to the GPU global memory. It also ensures the scalability of the GPU computing model, since processors are abstracted as threads, and provides support for fine-grained parallel programming models [9, 11, 17, 18].

3.1 GPU Architecture

Figure 1 gives the details of the GPU architecture and an overview of the algorithm processing flow on GPUs.

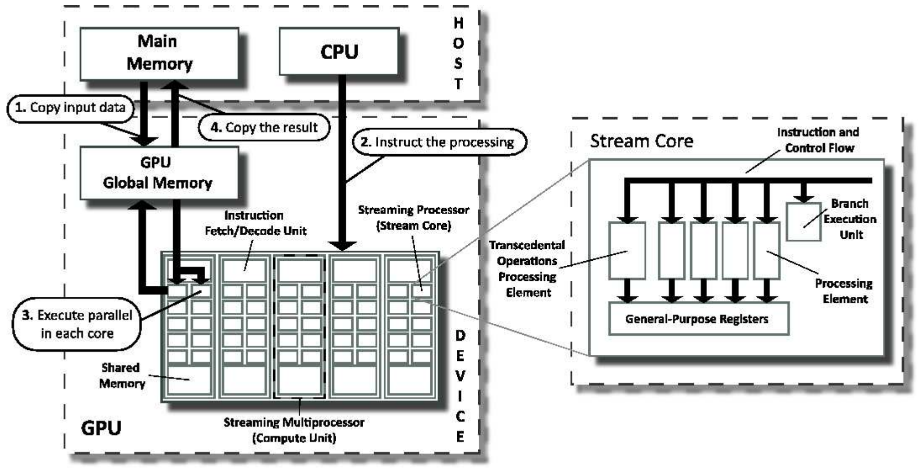
Main Memory
CPU
HOST
1. Copy input data
4. Copy the result
2. Instruct the processing
GPU Global Memory
Instruction Fetch/Decode Unit
Streaming Processor (Stream Core)
3. Execute parallel in each core
Shared Memory
GPU
Streaming Multiprocessor (Compute Unit)
DEVICE
Stream Core
Instruction and Control Flow
Branch Execution Unit
Transcedental Operations Processing Element
Processing Element
General-Purpose Registers

Fig. 1. Details of the GPU architecture and the GPGPU processing flow.

The fundamental building block of the GPU is a single instruction, multiple data (SIMD) streaming multiprocessor (SM). Each SM consists of several streaming processors (SPs) but only one instruction fetch/decode unit. This implies that all processors in an SM must execute the same instruction simultaneously. A streaming processor is typically arranged as a four or five-way very long instruction word (VLIW) processor [9, 11], allowing execution of several scalar operations simultaneously. Processing elements can execute either integer operations or single and double precision floating point operations. One processing element in each SP can perform transcendental operations such as sine, logarithm, etc. Each streaming multiprocessor has registers and a small on-chip shared memory. Main storage is located in a high-latency off-chip GPU global memory. Each access to the GPU global memory is relatively slow and takes from 400 to 800 clock cycles [11]. In contrast, computations on the GPU are performed very fast (up to 32 basic instructions per clock cycle per SM [9, 11]). Therefore, for optimal performance, GPUs use many active threads to fully utilize streaming multiprocessors while data is transferred from/to the global memory.

3.2 GPU Computing Model

In stream processing, a single data parallel function, called a kernel, is executed over a stream of data by many threads in parallel. A thread (also called a work-item in OpenCL) is the smallest execution entity and represents a single instance of the kernel. Threads are organized into blocks, which are sets of threads that can communicate and synchronize their execution. Each block is executed by a single

SM, but due to an existence of specific GPU hardware, an SM can execute multiple blocks simultaneously [9]. To support programs involving data dependent control flow, GPUs use a variant of SIMD called a single instruction, multiple threads (SIMT) model [9, 11, 17]. A SIMT multiprocessor is capable of executing individual threads independently, in contrast to traditional SIMD vector architectures (e.g., x86 SSE) in which all threads are always executed in synchronous groups. When programming for SIMD systems, data parallelism must be expressed explicitly on the software level for each vector instruction. With the GPU SIMT architecture, data parallelism between independent threads is discovered automatically on the hardware level. Differences between SIMT and SIMD models are elaborated in more detail in [11, 17, 18].

3.3 OpenCL Framework

For the development of GPU implementations, we had a choice between different application programming interfaces (APIs), like NVIDIA CUDA [14] or a more recent standard Open Computing Language - OpenCL [10]. CUDA is a vendor specific technology and supports only NVIDIA GPU hardware. Therefore, advantage was given to the OpenCL since it is a hardware agnostic and open standard that is strongly supported by many key industry players such as Apple, Google, Intel, AMD, NVIDIA, ARM, Nokia, etc. Furthermore, the OpenCL C programming language, included in the OpenCL framework [10], allows development of code that is both accelerated and portable across various hardware platforms (GPUs, field programmable gate arrays - FPGAs, digital signal processors - DSPs, embedded processors). This language represents a subset of the ISO C99 language with certain restrictions (e.g., recursion is not allowed) and special extensions for parallel programming. It is in many aspects similar to CUDA C and the transition from one language to the other is almost straightforward.

OpenCL offers a computing model that is an abstraction of the underlying GPU architecture. OpenCL abstractions for GPU streaming multiprocessors and processors are called compute units (CUs) and stream cores (SCs), respectively [10, 18]. The OpenCL program execution model is defined by the way the kernels are processed [10]. An OpenCL program consists of two parts:

1. Host (CPU) code that creates the context and, among else, makes the kernel calls, and
2. Device (GPU) code that implements the kernel.

A context for the execution of the kernels is defined by the host. The context includes resources like devices, kernels, and program and memory objects. The host creates a data structure called a command-queue to coordinate the execution of the kernels on the devices. The host then places commands into the command-queue which are afterwards scheduled onto the devices that exist within the context. When a kernel is submitted for execution by the host, an index space is defined. A single instance of the kernel, called a work-item or a thread, is executed for each point in the index space [10, 18]. A number representing the global identifier of a work-item is assigned to it based on the corresponding point in the index space to distinguish the data to be processed by each work-item. Every time a kernel is launched, many work-items (a number specified by the programmer) are created. Each work-item executes the same code, but the specific path through the code and the data operated upon can vary for each of the work-items. Work-items are grouped into work-groups to provide communication and cooperation between them.

4 Fast Spectral Transforms

Spectral transforms are an efficient tool in solving many tasks in switching theory and logic design [11, 20]. The spectra are usually computed by FFT-like algorithms with the time complexity of $O(N \log_2 N)$.

In this paper, we discuss two different kinds of spectral transforms:

1. Kronecker transforms [5], represented by the Walsh, the arithmetic, and the Reed-Muller transforms.
2. Layered-Kronecker transforms [12], represented by the Haar transform.

In matrix notation, Kronecker transforms can be defined in a unified way as:

$$\mathbf{T}(n) = \bigotimes_{i=1}^{n} \mathbf{T}_i(1), \qquad \mathbf{T}_i(1) = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{1}$$

where n is the number of variables in the function and parameters a, b, c, and d are specified in Table 1 for transforms discussed in this paper.

The Haar transform is defined as:

$$\mathbf{H}(n) = \begin{bmatrix} \mathbf{H}(n-1) \otimes \begin{bmatrix} 1 & 1 \end{bmatrix} \\ \mathbf{I}(n-1) \otimes [1 - 1] \end{bmatrix}, \qquad \mathbf{H}(0) = [1], \tag{2}$$

where n is the number of variables in the function to be processed and $\mathbf{I}$ stands for the identity matrix.

Notice that this is the definition of the non-normalized Haar transform that is usually used in spectral logic, however, the same method can be directly extended to the normalized Haar transform [5, 6].

Table 1. Entries in basic transform matrices for Kronecker transforms.

Transform	**Entry value**			
	a	*b*	*c*	*d*
Walsh	1	1	1	-1
Reed-Muller	1	0	1	1
Arithmetic	1	0	-1	1

Different ways of factorization of transform matrices $\mathbf{T}(n)$ and $\mathbf{H}(n)$ yield different fast algorithms [5, 6]. In this paper, we consider the Cooley-Tukey algorithms and the so-called algorithms with constant geometry (see Figures 2(b) and 2(f)) [5, 6]. This selection was made in order to compare the efficiency of in-place implementations (Cooley-Tukey algorithms) and out-of-place implementations (algorithms with constant geometry) on the GPU. Figure 2 shows the fast algorithms for the considered transforms for $N = 8$. The solid and the dotted lines carry positive (+1) and negative (-1) coefficients, respectively. Computations for the fast Walsh transform (FWT), the fast arithmetic transform (FAT) and the fast Haar transform (FHT) are over integers, while for the fast Reed-Muller transform (FRMT) we use Boolean operations. It should be noted that the spectral coefficients in Figures 2(a), 2(c), and 2(d) are in the natural (Hadamard) order, while in other figures they are in the sequency order [5, 6].

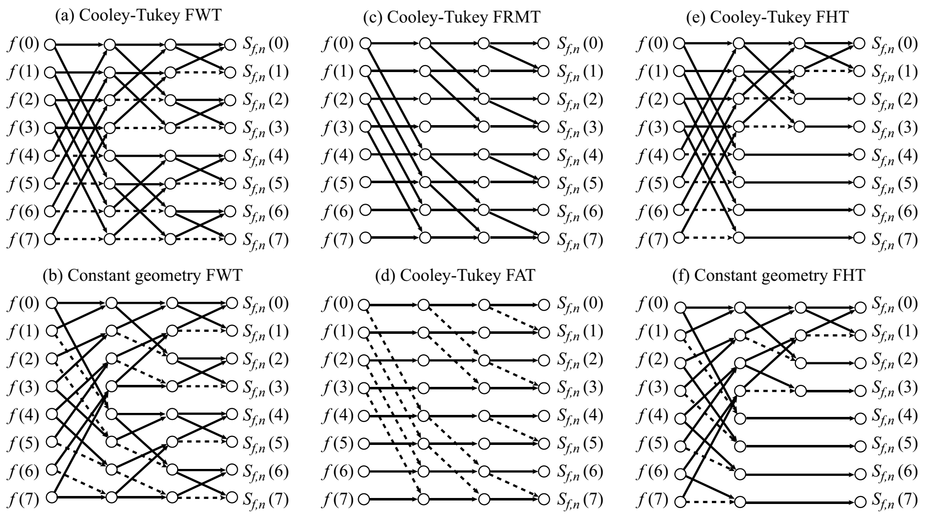

Fig. 2. Spectral transform flow graphs for $N = 8$.

5 Mapping of Algorithms

Mapping of an algorithm to a targeted hardware technology is hardly ever directly possible. A careful tailoring of algorithms for the implementation on a concrete technology permits to fully exploit all the favorable properties of both the fast algorithms and the hardware. In the case considered, this particularly concerns the organization of the computations, memory transfer reductions, impact of the integer and the Boolean arithmetic, structure of algorithms and other related issues.

5.1 Kernel Design and Optimization

Figure 3 presents a general overview of the computing model that we devised for mapping of the spectral transforms to the GPU architecture.

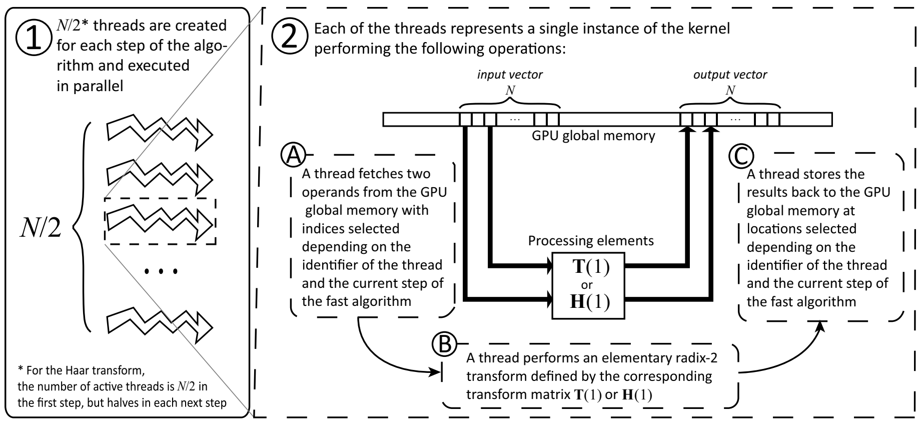
1
N/2* threads are created for each step of the algorithm and executed in parallel
N/2
* For the Haar transform, the number of active threads is N/2 in the first step, but halves in each next step
2
Each of the threads represents a single instance of the kernel performing the following operations:
input vector
N
output vector
N
GPU global memory
A
A thread fetches two operands from the GPU global memory with indices selected depending on the identifier of the thread and the current step of the fast algorithm
Processing elements
T(1)
or
H(1)
C
A thread stores the results back to the GPU global memory at locations selected depending on the identifier of the thread and the current step of the fast algorithm
B
A thread performs an elementary radix-2 transform defined by the corresponding transform matrix T(1) or H(1)

Fig. 3. Mapping of fast spectral transforms to the GPU architecture and computing model.

As in all FFT-like algorithms, the algorithm steps are executed sequentially and parallelism is exploited only within the steps. In every step of the Kronecker transforms we perform the elementary butterfly operations over pairs of numbers, therefore, $N/2$ threads are created and executed in parallel. For the Haar transform, the number of active threads is $N/2$ for the first step, but halves after each step. In all of the OpenCL fast algorithm implementations presented in this paper, a single thread performs the following tasks: first, it fetches two numbers from the GPU memory, then it performs the elementary butterfly operation according to the respective transform, and, finally, stores the results back to the GPU global memory. Computation of the spectrum is divided into many active threads in order to keep the GPU compute units active while the data is being fetched from and stored to the GPU global memory.

Before the device code execution starts and after the computations are complete, data also needs to be transferred between the main memory of the host and the global GPU memory. These memory operations take from 25% to 75% of the total GPU running times, as reported in Section 6. An important conclusion is that the GPU processing makes sense only for problem sizes large enough to make the price of buffer creation and transfers acceptable (in the case of fast spectral transforms the experiments show that this is approximately for $n \geq 18$).

Since speed of memory transfers can be considered as a bottleneck in GPU computing, memory coalescing is an important optimization technique [9, 14]. The application of this technique to the GPU implementations results in simultaneous memory accesses to the GPU global memory by multiple threads in a single memory transaction. Achieving efficient coalescing on the GPU requires techniques that are completely opposite to the methods of parallel programming for the CPU [9]. In the case of AMD GPUs and OpenCL, coalescing has less impact on performance than for NVIDIA GPUs and CUDA, but still significantly affects it [9]. For this reason, we organize the computations so that threads with consecutive global identifiers access consecutive memory locations.

Optimization of the kernel code in the reported research includes a technique of replacing integer divide and modulo operations, involved in Eq.(3), with the corresponding bitwise operations. If i is an integer that is a power of two and j is any integer, the integer division of j with i is equivalent to $j >> \log_2 i$ and the modulo operation $(j \bmod i)$ can be replaced with the bitwise AND operation $(j\&(i-1))$ [19]. Integer divide and modulo operations are costly since they are implemented in hardware through tens of basic GPU instructions, while subtraction, logical operations, and base-2 logarithm are all included in the basic set of GPU hardware instructions and have a throughput of up to 32 operations per clock cycle per SM [11, 17].

5.2 Cooley-Tukey Fast Algorithms for the Kronecker Transforms

Algorithm 1 presents the general outline of the FFT-like algorithms for the Kronecker transforms.

Each thread in the OpenCL implementation of the Cooley-Tukey algorithms for the Kronecker transforms reads two elements from the input GPU buffer. Indices of the elements to be fetched, $op1$ and $op2$, for all three spectral transforms discussed, are calculated as follows:

$$op1 \leftarrow thread\_id \bmod step + 2 \cdot step \cdot (thread\_id/step), \tag{3}$$

$$op2 \leftarrow op1 + step. \tag{4}$$

Parameters *thread_id* and *step* are the global identifier of the thread in the index space and the current algorithm step, respectively. All threads execute the elementary butterfly operation defined by the corresponding transform $(\mathbf{T}_i(1))$ and then store the results back in the same locations in the GPU global memory.

Algorithm 1 Cooley-Tukey Fast Algorithm for the Kronecker Transforms

```
1: Allocate a buffer buff in the global memory of the GPU device.
2: Transfer the input vector input from the main memory to the buffer buff.
3: For each step of the transform from $step = (\log_2 N) - 1$ to $step = 0$, with decrement of 1:
    a. Call the OpenCL kernel for the appropriate transform (FWT, FAT or FRMT) with input parameters being the GPU buffer buff and the value of the current step $2^{step}$.
    b. The kernel is executed by $N/2$ threads in parallel on the GPU. Each of the threads reads two elements, determined by (3) and (4), from the buffer buff, performs the defined operations and stores back the results in the same locations.
4: Transfer the contents of the GPU buffer buff, holding the resulting spectral coefficients, back to the main memory.
```

The FRMT kernel operates on Boolean values, while the kernels for other two transforms operate on integers. The Reed-Miller transform kernel does not have better performance on the GPU than the kernels working with integer numbers, because contemporary GPUs, on the hardware level, interpret Boolean values as integers [9, 10]. Boolean buffers are not even officially supported by the OpenCL standard specification [10] and, therefore, it is recommended to treat Boolean vectors as integers in the OpenCL application code. As a consequence, the speedups for the Reed-Muller transform, which are still acquired through GPU processing, are not as large as for the other transforms performed. Since the goal of the research presented in this paper was to develop a unified approach to the GPU implementations of fast spectral transforms, we did not consider potential optimizations in the bit-level implementation of the Reed-Miller transform. Bitwise techniques [19] may provide additional gains for the FRMT, however, that would require a completely different implementation approach. This approach should take into account that the minimal size of data transactions on the GPU that allows optimal performance is 32 bits [9]. This imposes development of additional procedures, working on the bit level, for efficient execution of operations and packing and unpacking of data.

5.3 Fast Algorithms with Constant Geometry for the Kronecker Transforms

Algorithms with constant geometry for Kronecker transforms [5](see Figure 2(b)) read from one pair of vector elements and write the results into another pair and, therefore, cannot be implemented in-place. We describe the mapping and report experimental results for the FWT (Algorithm 2) since for the FRMT and the FAT the difference is just in the butterfly operations.

For this class of algorithms, we need two separate buffers, one for reading the input vector and the other for writing the output. The corresponding kernels have three arguments: the input buffer, the output buffer, and the current step. The values of indices *op1* and *op2* of the elements fetched from the input buffer depends only on the value of the global thread identifier:

$$op1 \leftarrow 2 \cdot thread\_id, \tag{5}$$

$$op2 \leftarrow 2 \cdot thread\_id + 1. \tag{6}$$

The indices of the locations *dst1* and *dst2* in the output buffer, where the results of the butterfly operation performed by the kernel are stored, are calculated by the same equations as in the case of the in-place algorithm (Eqs. (3) and (4)).

Algorithm 2 Constant Geometry Fast Algorithm for the Kronecker Transforms

```
1: Allocate two buffers buff1 and buff2 on the GPU device.
2: Transfer the input vector input from the main memory to buffers buff1 and buff2.
3: For each step of the transform from $step=(\log_2 N)-1$ to $step = 0$, with decrement of 1:
   a. If $step \bmod 2 = 0$, then call the OpenCL kernel for the appropriate transform (FWT,
      FAT or FRMT) with input parameters in the order: buff1, buff2 and the value of
      current step $2^{step}$. The kernel is executed by $N/2$ threads in parallel on the GPU.
      Each thread reads two elements determined by (5) and (6) from buff1, performs the
      operations defined by the kernel and stores the results in the locations, determined
      by (3) and (4), in buff2.
   b. Else if $step \bmod 2 \neq 0$, call the OpenCL kernel for the appropriate transform (FWT,
      FAT or FRMT) with the order of the first two input arguments swapped: buff2, buff1
      and the value of current step $2^{step}$. The kernel is then executed in the same way as in
      the case a, except for the buff1 and the buff2 exchanging roles.
4: If $(\log_2 N)-1 \bmod 2 = 0$, transfer the contents of buff1 to the vector output in the main
   memory, else transfer the contents of buff2.
```

For the Walsh transform, there is also a difference between the operations performed by threads with even and odd global identifier numbers in all transform steps except the last one (see Figure 2(b)). This is done to avoid the shuffling of elements. Threads with even global identifiers perform operations $u+v$ and $u-v$, while the odd numbered threads perform the operations $u+v$ and $v-u$, where u and v are operands fetched from the input buffer. In the last step, all of the threads perform the $u+v$ and $u-v$ operations.

The problem here is that not only memory space requirements have doubled, but buffer transfers occupy the bandwidth and thus are very expensive performance-wise. However, if we add a simple check of the pass order number, we can execute the kernel with arguments for the input and the output swapped with every loop pass. After completing the transform, we just check whether the last pass is odd or even numbered and then copy only the appropriate buffer back to the host. Implementing the algorithm with constant geometry on the GPU now requires adding just two condition checks in the host code and one extra buffer in device memory and no extra bandwidth occupation.

5.4 Fast Algorithms for the Haar Transform

The operations in the in-place Cooley-Tukey algorithm for the Haar transform (see Figure 2(e) and Algorithm 3) [5,6,12], are the same as in the FWT, except that after the first step the number of butterflies is halved in each step. Therefore, the kernel for the FWT can be used in FHT, however, the number of active threads is halved with each step of the transform, starting from $N/2$ active threads and ending with only one thread for the final step.

Algorithm 3 Cooley-Tukey Fast Algorithm for the Haar Transform

```
1: Allocate a buffer buff in the global memory of the GPU device.
2: Transfer the input vector input from the main memory to the buffer buff.
3: For each step of the transform from step=$(\log_2 N) - 1$ to step=0, with decrement of 1:
    a. Call the OpenCL kernel for the FHT with input parameters being the GPU buffer
       buff and the value of current step $2^{step}$.
    b. The kernel is then executed in parallel on the GPU. The number of active threads is
       $N/2$ for the first step, but halves in each next step. Each of the threads reads two
       elements, determined by (3) and (4), from the buffer, performs the defined operations
       and stores back the results in the same locations.
4: Transfer the contents of the GPU buffer buff, holding the resulting Haar coefficients,
   back to the main memory.
```

Formulas for fetching the operands and writing the results in the algorithm with constant geometry (see Figure 2(f) and Algorithm 4) are the same as for the FWT. This algorithm needs to be implemented out-of-place and therefore the technique of argument swapping is used again. Both buffers *buff1* and *buff2* contain different parts of the spectrum, since the number of elements modified in each step is halved. A simple algorithm can be devised for reading the resulting spectrum from these two buffers. Alternative we used is to add a third GPU buffer and write the results of each step of the algorithm both in that buffer and in the buffer currently set as the output. This third buffer will finally contain the whole resulting spectrum. When the computation is completed, we copy the contents of this buffer back to the host.

Algorithm 4 Constant Geometry Fast Algorithm for the Haar Transform

```
1: Allocate three buffers buff1, buff2 and buff3 on the GPU device.
2: Transfer the input vector input from the main memory to buffers buff1 and buff2.
3: For each step of the transform from step=$(\log_2 N) - 1$ to step=0, with decrement of 1:
    a. If step mod 2 = 0, then call the OpenCL kernel for the FHT with input parameters
       in the order: buff1, buff2, buff3 and the value of current step $2^{step}$. The kernel is then
       executed on the GPU. The number of active threads is $N/2$ for the first step, but
       halves in each next step. Each of the threads reads two elements, determined by (5)
       and (6), from buff1, performs the operations and stores the results in the locations,
       determined by (3) and (4), in buff2 and buff3.
    b. Else if step mod 2 $\neq$ 0, call the OpenCL kernel for the FHT with arguments list that
       has the first two elements swapped: buff2, buff1, buff3 and the value of current step
       $2^{step}$. The kernel is then executed in the same way as in the case a, except for the
       buff1 and the buff2 exchanging roles.
4: Transfer the contents of the GPU buffer buff3, holding the resulting Haar coefficients, to
   the output in the main memory.
```

6 Experiments

The experiments were performed using an AMD Phenom II N830 triple-core CPU with 4 GBs of DDR3 RAM and an ATI Mobility Radeon 5650 GPU with 1GB of DDR3 RAM. This GPU is composed of 5 compute units, has 400 processing elements in total, and belongs to the lower-middle performance class. The OpenCL kernels were developed using MS Visual Studio 2010 Ultimate and ATI Accelerated Parallel Processing SDK 2.3 [15]. The graphics card driver is ATI Mobility Catalyst 10.12. ATI Stream Profiler 2.1 was used for performance analysis of OpenCL kernels, in accordance with instructions provided in [9]. The OpenCL host code and the C/C++ referent implementations were compiled for the x64 platform and optimized during the compilation for the maximum level of performance.

In order to conduct the experiments, a C/C++ test environment was developed. As in all FFT implementations over vectors the algorithm time complexity is independent of the function values. Therefore, we perform experiments on randomly generated binary vectors, in the same way as in [7, 13]. No architecture-dependent GPU code optimizations are applied in order to preserve code portability.

The sequential C/C++ implementations of Kronecker transforms for the CPU require a careful handling of the memory access patterns. For example, in the classical radix-2 FFT, swapping of the inner loops which control the order of computations within the algorithm steps, reduces the number of trigonometric operations which in certain situations improves the overall performance [20]. Unlike the FFT, the spectral transforms considered in this paper do not involve transcendental computations. As a consequence, the loop order that can improve the FFT performance brings no benefit here and results only in a highly non-local memory access patterns. This poor spatial locality leads to an inefficient use of the cache memory (cache thrashing) [13, 20, 21]. This effect is invisible when computing with small vectors that fit in the cache. However, swapping of the inner loops that define the order of the butterfly operations within a step, followed by a slight modification of the entire code results in speedups of 30$\times$ or more, as it can be clearly seen from the experiments. We decided to address this issue here, and include these two different CPU implementations in the experiments, after coming across several fast spectral transform implementations that neglected the importance of spatial locality. Even the AMD APP SDK [15] has a C/C++ implementation of the FWT included as a referent example that violates this principle and as a result has very poor performance.

6.1 Experimental Results

The first set of experiments is designed in order to explore the techniques for implementing the fast Walsh transform on the GPU (Figure 4(a)).

It is clear that the memory access pattern and implementation design have a huge impact on the performance of the C/C++ implementations. The CPU implementation labeled CPU B in Figure 4(a) is up to 21$\times$ faster than the referent implementation provided in [15]. But, the OpenCL implementation of the Cooley-Tukey algorithm performed on a commodity GPU clearly outperforms both the slower CPU implementation (labeled CPU A in Figure 4(a)), by a factor of 104$\times$, and the faster CPU implementation (labeled CPU B in Figure 4(a)), by a factor of 5$\times$, when the calculation time is compared. These factors are 78$\times$ and 3.7$\times$, respectively, when the total time, including memory transfers to/from the GPU, is taken into account. Further, the application of the technique of argument swapping for the implementation of the algorithm with constant geometry results in performance that is equal to the in-place algorithm in terms of calculation times and only 16% to 19% slower when we add memory times. After applying this simple technique, the algorithm with constant geometry can be implemented on the GPU by adding one more buffer and with no extra bandwidth occupation.

The performance for the fast Reed-Muller and the fast arithmetic transforms are presented in Figures 4(b) and 4(c), respectively. The conclusions for the two different CPU implementations of the FWT case are also valid here, with speedups going up to 26$\times$ for the arithmetic transform, and up to 38$\times$ for the Reed-Muller transform. The OpenCL implementations of both transforms again clearly outper-

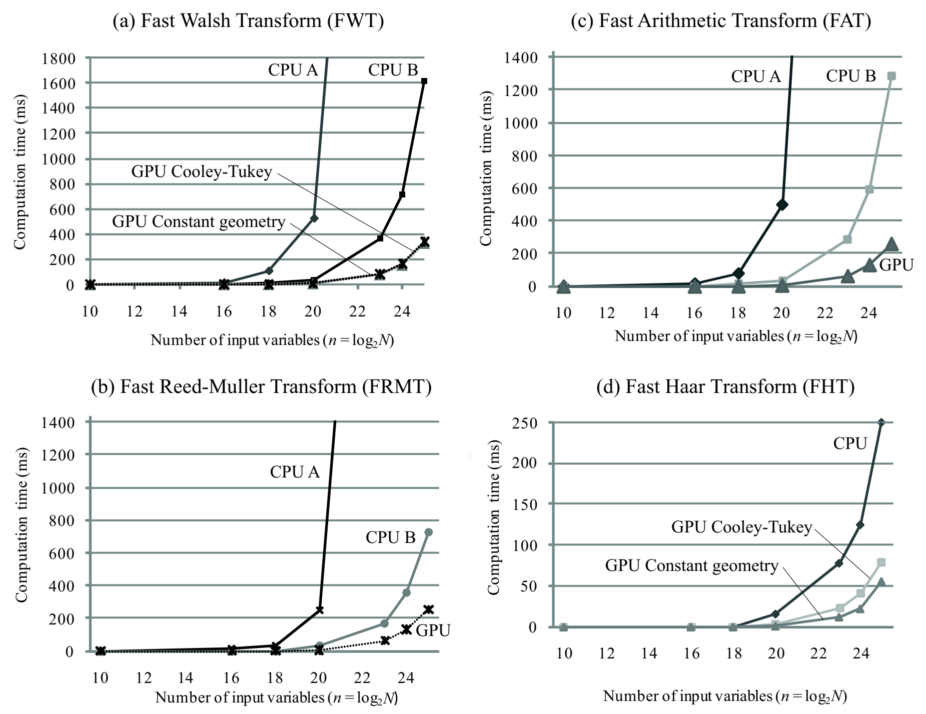

Fig. 4. Computation times for: (a) FWT, (b) FRMT, (c) FAT, (d) FHT.

form their CPU counterparts, although the speedup factors are a bit smaller in the case of the Reed-Muller transform, because the GPU on the hardware level interprets Boolean values as integers [10]. For the FRMT, speedups are up to 109$\times$ and 77$\times$, against the slower CPU code (labeled CPU A in Figure 4(b)), and 2.8$\times$ and 2$\times$, against the faster CPU code (labeled CPU B in Figure 4(b)). For the arithmetic transform, speedups in terms of the calculation time and the total time, respectively, are up to 131$\times$ and 93$\times$, against the slower CPU code (labeled CPU A in Figure 4(c)), and 5$\times$ and 3.5$\times$, against the faster CPU code (labeled CPU B in Figure 4(c)).

The final set of experiments considers the Cooley-Tukey algorithm and the algorithm with constant geometry for the fast Haar transform (Figure 4(d)). The Haar transform offers a smaller amount of computational parallelism than the Kronecker transforms and the number of active parallel threads in the respective OpenCL implementation is halved in every step of the algorithm. Because of the linear time complexity of the Haar transform, the sequential C/C++ code on the CPU performs much better here than in the case of the Kronecker transforms, which have the

$O(N\log_2 N)$ complexity. However, speedups of up to $3\times$ in the calculation time and $1.4\times$ in the total time for the Cooley-Tukey and up to $5\times$ in the calculation time and $1.2\times$ in the total time for the algorithm with constant geometry are still achieved. Times for memory transfers to/from GPU dominate over the GPU calculation times for the FHT, especially for the algorithm with constant geometry.

7 Conclusions

We considered the efficient implementation of the fast algorithms for spectral transforms on GPUs using OpenCL and presented a comparative analysis with the referent C/C++ implementations on the CPU. The acceleration is obtained by an appropriate modification of the fast algorithms for the GPU processing through massively parallel execution of the OpenCL kernels. Experimental results show that, even in the case of transforms not involving floating point and complex number arithmetic, a computational speedup ranging from $3\times$ up to $131\times$, depending on the referent implementation, is obtained on a lower-middle performance class GPU. Processing of the same kernels on a more powerful GPU (with more streaming multiprocessors and a higher memory bandwidth) would directly lead to much larger speedups, due to the inherent scalability of the GPU parallel programming model. We believe that the methods presented here could, therefore, widen the area of applications of spectral transforms in switching theory and logic design.

Acknowledgment

The authors are very grateful to the reviewers for the constructive comments that were useful in improving the presentation in this paper, as well as for pointing to interesting topics for further work in this area.

References

[1] A. Buluc, J. Gilbert, and C. Budak, "Solving path problems on the GPU," *Parallel Computing*, vol. 36, no. 5-6, pp. 241–253, 2010.

[2] A. D. Copeland, N. B. Chang, and S. Lung, "GPU accelerated decoding of high performance error correcting codes," in *Proc. 14th Annual Workshop on HPEC*, Lexington, Massachusetts, USA, Sep. 2010.

[3] M. Lukac, M. Perkowski, P. Kerntopf, and M. Kameyama, "GPU acceleration methods and techniques for quantum logic synthesis," in *Proc. 9th Int. Workshop on Boolean Problems*, Freiberg, Germany, Sep. 2010, pp. 125–132.

[4] E. Paul, B. Steinbach, and M. Perkowski, "Application of CUDA in the Boolean domain for the unate covering problem," in *Proc. 9th Int. Workshop on Boolean Problems*, Freiberg, Germany, Sep. 2010, pp. 133–142.

[5] M. G. Karpovsky, R. S. Stanković, and J. T. Astola, *Spectral Logic and Its Applications for the Design of Digital Devices*. Wiley-Interscience, 2008.

[6] M. A. Thornton, R. Drechsler, and D. M. Miller, *Spectral Techniques in VLSI CAD*. Springer, 2001.

[7] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, Austin, Texas, USA, Nov. 2008.

[8] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial Mathematics, 1992.

[9] AMD, *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 2011.

[10] Khronos, *The OpenCL Specification 1.1*, Jun. 2010.

[11] NVIDIA, *OpenCL Programming Guide for the CUDA Architecture*, 2011.

[12] B. J. Falkowski, "Relationships between arithmetic and Haar wavelet transforms in the form of layered Kronecker matrices," *Electronic Letters*, vol. 35, no. 10, pp. 799–800, 1999.

[13] V. Volkov and B. Kazian, *Fitting FFT onto G80 architecture*, UC Berkeley CS258 Project Report, 2008.

[14] NVIDIA, "NVidia CUDA: Compute Unified Device Architecture," Aug. 2011. [Online]. Available: http://developer.nvidia.com/object/gpucomputing.html

[15] AMD, "AMD Accelerated Parallel Processing SDK," Aug. 2011. [Online]. Available: http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx

[16] D. B. Gajić and R. S. Stanković, "Computing fast spectral transforms on graphics processing units using OpenCL," in *Proc. Reed-Muller 2011 Workshop*, Tuusula, Finland, May 2011, pp. 27–36.

[17] J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2008.

[18] B. Gaster *et al.*, *Heterogeneous Computing with OpenCL*. Elsevier, 2011.

[19] D. Knuth, *The Art of Computer Programming - Volume 4A, Combinatorial Algorithms*. Addison-Wesley, 2011.

[20] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 2010.

[21] R. E. Bryant and D. R. OHallaron, *Computer Systems: A Programmers Perspective*. Addison Wesley, 2010.