Calculation of Dyadic Convolution Using Graphics Processing Units and OpenCL

Dušan B. Gajić[1] and Radomir S. Stanković[1]

Abstract – **Convolution is an important operation in many areas of science and engineering, including systems theory, signal processing, pattern recognition, switching theory, and logic design. In particular, when dealing with binary encoded signals and systems, the dyadic convolution is used, i.e., the convolution on finite dyadic groups consisting of binary *n*-tuples equipped with the addition modulo 2. Fast computation algorithms are essential for practical applicability of the dyadic convolution and algorithms based on it. These algorithms are defined by the application of the convolution theorem in the Walsh (Fourier) domain and then exploiting the Fast Fourier Transform (FFT).**

In this paper, we present a method for accelerating the computation of the dyadic convolution through a parallel implementation of the related algorithm on a Graphics Processing Unit (GPU). The architecture of the GPU is massively parallel, fully programmable, and it offers tremendous computational power and memory bandwidth. In order to be efficiently implemented, the fast algorithm for the dyadic convolution has to be suitably reformulated and adapted to the GPU resources. We present a solution to this problem using the Open Computing Language (OpenCL). Further, we consider several issues concerning the efficient mapping of the algorithm to the GPU architecture. Performance of the proposed implementation is compared with the referent C/C++ implementation processed on the Central Processing Unit (CPU). Experimental results show that significant speedups are achieved by the application of the proposed GPU calculation method.

Keywords – **Dyadic convolution, Fast Walsh transform, GPU parallel programming, OpenCL.**

I. INTRODUCTION

Convolution is a mathematical operation that expresses relationships between values of two signals (modeled by functions f and g) in points at a fixed distance. The convolution $C = f * g$ of two functions f and g is a function that resembles any one of them, modified by the other one. The convolution operation has an important place in efficient solutions to many problems in engineering and mathematics which are of both practical and theoretical importance [8].

When the finite dyadic group is used as an underlying algebraic structure on which the convolution operation is defined, we use the term dyadic (or logical or XOR) convolution (see Section 3 and also [4, 8, 9, 12]). Dyadic convolution coefficients can, in principle, be calculated by the brute force application of the equation that defines the operation (see Eq. (1) in Section 3). However, this method has exponential complexity in the number of inputs and is unfeasible in practice for large signals. Therefore, algorithms for the fast computation of convolution are derived by using the convolution theorem [8] on the corresponding algebraic structure.

In this paper, we present a technique for an accelerated calculation of the dyadic convolution through a parallel implementation of the fast algorithm derived from the convolution theorem. Due to this theorem, computation of the dyadic convolution converts into performing two direct and an inverse Walsh transform, which can be done by the corresponding FFT-like algorithms, i.e., the Fast Walsh Transform – FWT [4, 5, 8].

The proposed implementation is developed using the Open Computing Language (OpenCL) and processed in a highly parallel manner on a GPU. Experimental results and comparisons with the classical implementation confirm that the proposed method leads to significant computational speedups.

The rest of this paper is organized as follows. After a discussion of the related work in Section 2, in Section 3 we give a short introduction to the dyadic convolution and the fast algorithm for its calculation. Section 4 is devoted to the mapping of the fast algorithm to the GPU architecture and the design of the corresponding OpenCL implementation. In Section 5, we describe the experimental environment that we used to evaluate the method, and present the experimental results that we recorded. Section 6 offers some conclusions drawn from the presented research.

II. RELATED WORK

The fast algorithm for the dyadic convolution is based on the application of the Walsh transform which is the Fourier transform on finite dyadic groups. The implementation of various Fast Fourier Transforms (FFTs) on different technological platforms is a widely considered topic, see for instance [4, 5, 6, 7] and references therein. The calculation of dyadic convolution on classical Central Processing Units (CPUs) through the application of the convolution theorem, both on vectors and decision diagrams, is presented in [9]. Reference [4] presents an application of the dyadic convolution for the fast multiplication of hyper-complex numbers.

In recent years, the technique of performing General Purpose computations on the GPU (GPGPU) has proven to be a suitable approach in solving many computationally-intensive tasks [2, 3, 6, 7, 10]. In particular, the GPU-accelerated calculation of FFT algorithms using CUDA is

[1]Dušan B. Gajić and Radomir S. Stanković are with the University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: dusan.gajic@elfak.ni.ac.rs, radomir.stankovic@gmail.com.

described in [7, 11]. Reference [6] presents an OpenCL implementation of the FWT that uses the GPU hardware and leads to significant speedups over traditional CPU processing.

However, in our best knowledge, there are no papers discussing neither CUDA nor OpenCL GPU implementations of the fast algorithm for the calculation of dyadic convolution based on the convolution theorem. This fact, together with the intended applications of the dyadic convolution for particular problems in logic design [8], was the motivation for the research on the dyadic convolution calculation that is presented in this paper.

III. Dyadic Convolution

A. Dyadic convolution

The finite dyadic group of order *n* is defined as $C_2^n = \overset{n}{\underset{i-1}{\times}} C_2$, where $C_2 = (\{0,1\}, \oplus)$, $\oplus$ stands for the addition modulo 2, and $\times$ is the direct (Cartesian) product.

For two functions *f, g* $: C_2^n \to \mathbb{R}$, where $\mathbb{R}$ is the field of rational numbers, the dyadic convolution, at a distance $\tau = 0,1,...,2^{n-1}$, is defined as:

$$\mathrm{C}_{f*g}(\tau) = f * g = \sum_{x=0}^{2^n-1} f(x) \cdot g(x \oplus \tau). \qquad (1)$$

In binary notation, x is $x = (x_1, x_2, ..., x_n)$, and τ is $\tau = (\tau_1, \tau_2, ..., \tau_n)$, where $x_i, \tau_i \in \{0,1\}$.

B. Walsh transform and fast Walsh transform

The Walsh transform is defined by the Walsh matrix:

$$\mathbf{W}(n) = \bigotimes_{i=1}^{n} \mathbf{W}(1), \qquad (2)$$

where $\otimes$ denotes the Kronecker product and

$$\mathbf{W}(1) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (3)$$

is the basic Walsh matrix. Since $\mathbf{W}(n)$ is a self-inverse matrix up to the scalar 2^{-n}, the inverse Walsh transform is defined as:

$$\mathbf{W}^{-1}(n) = 2^{-n}\mathbf{W}(n). \qquad (4)$$

It follows that both the direct and the inverse Walsh transforms can be computed using the same algorithm.

The Walsh spectrum $\mathbf{S}_{f,h} = [S_{f,h}(0), S_{f,h}(1), \dots, S_{f,h}(2^n\text{-}1)]^T$ of a function $f : C_2^n \to \mathbb{R}$, specified by the function vector $\mathbf{F} = [f(0), f(1), \dots, f(2^n\text{-}1)]^T$, is defined as:

$$\mathbf{S}_{f,h} = \mathbf{W}(n)\mathbf{F}. \qquad (5)$$

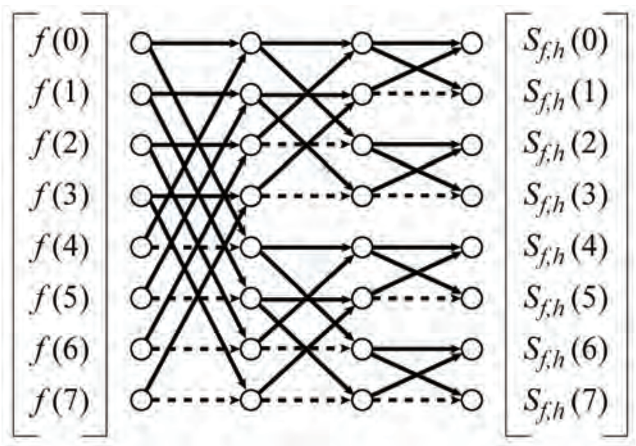

Fig. 1. Flow graph of the Cooley-Tukey FWT algorithm for $N = 8$ [6].

The spectral coefficients appear in natural (Hadamard) ordering, which is indicated by the index *h* in $\mathbf{S}_{f,h}$. The function *f* is reconstructed from the Walsh spectrum as:

$$\mathbf{F} = 2^{-n}\mathbf{W}^{-1}(n)\mathbf{S}_{f,h}, \qquad (6)$$

and (5) and (6) form the Walsh transform pair.

The computation of the Walsh transform based on its definition (Eqs. (2) and (5)) is inefficient, since it expresses the $O(N^2)$ time complexity, where $N = 2^n$ is the size of the input vector. Fortunately, more efficient algorithms based on the FWT [4, 8], with the time complexity of $O(N\log N)$, exist.

The fast Walsh transform (FWT) can be defined using the following factorization:

$$\mathbf{W}(n) = \prod_{i=1}^{n} \mathbf{C}_{w_i}(n)_i \qquad (7)$$

where

$$\mathbf{C}_{w_i}(n) = \bigotimes_{j=1}^{n} \mathbf{C}_{w_i}^{j}(1), \quad \mathbf{C}_{w_i}^{j}(1) = \begin{cases} \mathbf{W}(1), i = j, \\ \mathbf{I}(1), i \neq j. \end{cases} \qquad (8)$$

The matrix $\mathbf{C}_{w_i}(n)$ defines the partial Walsh transform and corresponds to the ***i***-th step of the FWT. The flow graph of the corresponding algorithm for $N = 8$ is given in Figure 1.

C. Convolution theorem

In the classical Fourier analysis, the convolution theorem states that the Fourier transform [4, 5, 8] of the convolution function $C = f * g$ is the componentwise product of Fourier transforms of *f* and *g*. In other words, a rather complex convolution operation in the original domain converts into a simple componentwise multiplication in the spectral domain.

In abstract harmonic analysis, the convolution theorem can be extended to the Fourier transform defined over locally compact Abelian groups [8]. For functions on the finite dyadic group C_2^n, the calculation of the dyadic convolution through the application of the convolution theorem is done as follows:

$$\mathbf{C}_{f*g} = 2^{-n}\mathbf{W}^{-1}(n)((\mathbf{W}(n)\mathbf{F})(\mathbf{W}(n)\mathbf{G})) \qquad (9)$$

Algorithm 1 FAST CALCULATION OF DYADIC CONVOLUTION $C = f * g$

```
1   Allocate buffers buffer1 and buffer2 in the global memory of the GPU device.
2   Transfer input vectors f and g from the host CPU memory to GPU buffers buffer1 and buffer2, respectively.
3   Perform the Walsh transform on vectors stored in buffer1 and buffer2 using the following in-place OpenCL
    implementation of the Cooley-Tukey algorithm for the FWT:
        a.  For each step of the FWT, from $step \leftarrow 0$ to $step \leftarrow (\log_2 N) - 1$, call the OpenCL kernel for the FWT with
            input parameters being the appropriate buffer in the GPU's global memory and the value of the current step
            $2^{step}$. The kernel is executed by N/2 threads in parallel on the GPU. Each thread reads two elements,
            determined by (10) and (11), from the buffer, performs the operations defined by the Walsh matrix $\mathbf{W}(1)$
            and stores back the results in the same locations.
4   After computing the FWT of both vectors, execute the OpenCL kernel for the componentwise multiplication of the
    two Walsh spectra with N threads executed in parallel. The resulting vector is stored in buffer1.
5   Perform the inverse FWT on buffer1 using the same kernel as for the FWT.
6   Scale the contents of buffer1 with the factor $2^{-n}$ using the OpenCL kernel with N threads executed in parallel.
7   Transfer the contents of the GPU buffer buffer1, which holds the resulting dyadic convolution coefficients, back to
    the host CPU memory.
```

Fig. 2. Algorithm for the fast calculation of dyadic convolution on the GPU.

where $\mathbf{W}$ is the Walsh transform, $\mathbf{W}^{-1}$ is the inverse Walsh transform, and $\mathbf{F}$ and $\mathbf{G}$ are function vectors for f and g, respectively.

Therefore, an efficient algorithm for the computation of the dyadic convolution can be developed in terms of the FWT.

IV. Mapping of the Algorithm and Implementation Details

The application of the convolution theorem, expressed in Eq. (9), leads to the following three-stage fast dyadic convolution calculation algorithm (shown in Figure 2):

Step 1. Perform the FWT on f and g and compute their Walsh spectra $\mathbf{S}_f$ and $\mathbf{S}_g$.
Step 2. Perform the componentwise multiplication of the two spectra $\mathbf{S}_f$ and $\mathbf{S}_g$.
Step 3. Perform the inverse FWT over $\mathbf{S}_f \cdot \mathbf{S}_g$ to obtain $\mathbf{C}_{f*g}$.

Since multiplication is done very fast on modern CPUs, the key issue in creating an efficient implementation of the above algorithm is the development of the fast implementation of the FWT. In order to perform the algorithm steps with the FWT and inverse FWT, we developed a kernel containing an OpenCL in-place implementation of the Cooley-Tukey algorithm for the FWT [6, 8]. As in all FFT-like algorithms, steps of the algorithm are executed sequentially and parallelism is used only within the steps. Within each of the steps, $N/2$ threads are executed in parallel. This large number of threads helps in hiding the data access latency to the GPU global memory [2, 3]. Each thread reads two elements from the GPU buffer with indices *op1* and *op2* calculated as:

$$op1 \leftarrow thread\_id \bmod step + 2 \times step \times (thread\_id / step), \quad (10)$$

$$op2 \leftarrow op1 + step. \quad (11)$$

Parameters *thread_id* and *step* are the global identifier of the thread and the identifier of the current step of the algorithm, respectively. All threads execute the elementary butterfly operation defined by the basic Walsh transform matrix $\mathbf{W}(1)$ and store the results back in the same locations in the GPU memory, as in other implementations of the in-place FWT algorithms.

The componentwise multiplication of vectors is also performed by the corresponding OpenCL kernel which is executed by N threads in parallel, with each thread multiplying the two corresponding elements of the input vectors.

After the multiplication, the inverse FWT is performed with the same kernel that is used for the direct transform, followed by scaling with 2^{-n}. The scaling is also performed in parallel through the execution of the corresponding OpenCL kernel.

Before executing any of the kernels, both input vectors are transferred from the main memory to the GPU global memory. After the calculations, the resulting convolution coefficients are transferred back to the host. These memory operations take a significant share of the total GPU running times as reported in Section 5.

V. Experimental Environment and Results

The test platform used to perform the experiments is an HP Pavilion dv7-4060us notebook computer (see Table I). The OpenCL kernels are developed using MS Visual Studio 2010 Ultimate and ATI APP SDK 2.3 [1]. ATI Stream Profiler 2.1 is used for GPU kernel performance analysis, in accordance with instructions provided in [2]. The referent C/C++ source code is compiled for the x64 platform with the maximum level of performance-oriented optimizations.

TABLE I Test machine specification

CPU	AMD Phenom II N830 triple-core (2.1GHz)
RAM	4GB DDR3
OS	Windows 7 (64-bit)
GPU	ATI Mobility Radeon 5650
- engine speed	650 MHz
- global memory	1 GB DDR3 800 MHz
- compute units	5
- processing elements	400
- price	~ 100$

TABLE II COMPUTATION TIMES FOR THE REFERENT CPU AND THE PROPOSED GPU IMPLEMENTATION

N	**Time [ms]**		
	CPU	**GPU**	
		Computation time	Memory time
16	0	0	0
256	0	0	1
1 024	0	1	1
65 536	15	3	2
262 144	53	6	3
1 048 576	108	24	9
8 388 608	1201	220	45
16 777 216	2512	469	86
33 554 432	5002	998	147

N – Number of elements in the input vector
CPU - C/C++ implementation, processed on the CPU
GPU - OpenCL C implementation, processed on the GPU

As in all FFT implementations over vectors, the resulting performance is independent of the function values. Therefore, we perform the experiments using randomly generated binary vectors. We present the results for the GPU calculation time and, also, the time for transfer of data to/from the GPU which offers a more complete perspective to the reader. In order to explore the performance of the proposed method, we performed a comparative analysis with respect to the classical C/C++ implementation of the same algorithm processed on the CPU.

The results of the experiments are presented in Table II and Figure 3. The OpenCL implementation processed on an inexpensive commodity GPU (see Table I) clearly outperforms the referent CPU implementation, by a factor of up to 5.5 × when only calculation times are compared, and by a factor of up to 4.5 × when total times, including memory transfers to/from GPU, are taken into account. The processing of the very same kernels on a more powerful GPU would directly lead to much larger speedups, which would not be the case if a more powerful CPU was used for processing the referent C/C++ implementation [3, 6].

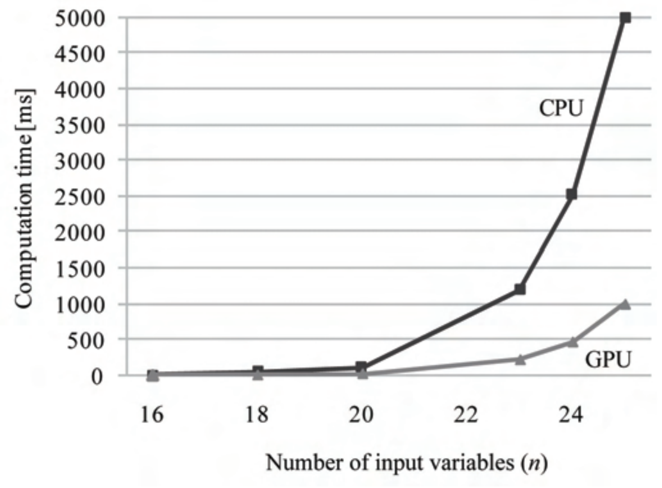

Fig. 3. Computation times for the referent (CPU) and the proposed (GPU) implementation.

VI. CONCLUSIONS

In this paper, we proposed a method for the fast calculation of dyadic convolution through an OpenCL parallel algorithm implementation which is processed on the GPU. We presented a comparative performance analysis of the proposed solution and the referent C/C++ implementation processed on a multicore CPU. A significant reduction of the calculation time is obtained due to an appropriate modification of the corresponding fast algorithm for the GPU implementation. Notice that, if $f(x) = g(x)$, then the dyadic convolution becomes the cross-correlation of a function with itself and produces coefficients which are referred to as the autocorrelation coefficients [8]. It follows that the same OpenCL implementation that is proposed in this paper can also be used for the fast calculation of both the dyadic convolution and the autocorrelation. The proposed method could, therefore, extend the area of applications of these operations to problems where algorithm running time is an essential and, often, limiting factor, since it allows time-efficient computations in systems theory, signal processing, pattern recognition, switching theory, and logic design.

REFERENCES

[1] *AMD Accelerated Parallel Processing SDK*, http://developer.amd.com/gpu/amdappsdk, AMD Inc., website last visited on 10/04/2011.
[2] *ATI OpenCL Programming Guide*, AMD Inc., 2010.
[3] T. M. Aamodt, ”Architecting graphics processors for non-graphics compute acceleration,” in *Proc. of the 2009 IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, Victoria, BC, Canada, August 23-26, 2009.
[4] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*, Springer, 2010.
[5] J. W. Cooley and J. W. Tukey, ”An algorithm for the machine calculation of complex Fourier series”, *Mathematics of Computation*, No. 90, 1965, 297-301.
[6] D. B. Gajić and R. S. Stanković, “Computing fast spectral transforms on graphics processing units using OpenCL,” in *Proc. of the Reed-Muller 2011 Workshop*, Tuusula, Finland, May 25-26, 2011, 27-36.
[7] N. K. Govindaraju et al., ”High performance discrete Fourier transforms on graphics processors,” in *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, IEEE Press, Austin, Texas, USA, November 15-21, 2008.
[8] M. G. Karpovsky, R. S. Stanković, and J. T. Astola, *Spectral Logic and Its Applications for the Design of Digital Devices*, Wiley-Interscience, 2008.
[9] M. M. Radmanović, “Calculation of dyadic convolution through binary decision diagrams,” *Facta Universitatis: Series: Automatic Control and Robotics*, Vol. 8, No. 1, 2009, 89 – 97.
[10] *NVidia CUDA: Compute Unified Device Architecture*, http://developer.nvidia.com/object/gpucomputing.html, NVIDIA Corp., website last visited on 30/03/2011.
[11] *NVidia CUDA CUFFT Library*, NVIDIA Corp., 2007.
[12] R. S. Stanković, M. Bhattacharaya, and J. T. Astola, “Calculation of dyadic autocorrelation through decision diagrams,” in *Proc. of the European Conference on Circuit Theory and Design*, August 2001, 337-340.
[13] *The OpenCL Specification 1.1*, Khronos OpenCL Working Group, 2010.