SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 11, No. 1, February 2014, 97-109

UDC: 004.421:517.443
004.421.2:512.623

DOI: 10.2298/SJEE131201009G

Computation of Galois Field Expressions for Quaternary Logic Functions on GPUs

Dušan B. Gajić[1]

Abstract: Galois field (*GF*) expressions are polynomials used as representations of multiple-valued logic (MVL) functions. For this purpose, MVL functions are considered as functions defined over a finite (Galois) field of order p - $GF(p)$. The problem of computing these functional expressions has an important role in areas such as digital signal processing and logic design. Time needed for computing *GF*-expressions increases exponentially with the number of variables in MVL functions and, as a result, it often represents a limiting factor in applications. This paper proposes a method for an accelerated computation of $GF(4)$-expressions for quaternary (four-valued) logic functions using graphics processing units (GPUs). The method is based on the spectral interpretation of *GF*-expressions, permitting the use of fast Fourier transform (FFT)-like algorithms for their computation. These algorithms are then adapted for highly parallel processing on GPUs. The performance of the proposed solutions is compared with referent C/C++ implementations of the same algorithms processed on central processing units (CPUs). Experimental results confirm that the presented approach leads to significant reduction in processing times (up to 10.86 times when compared to CPU processing). Therefore, the proposed approach widens the set of problem instances which can be efficiently handled in practice.

Keywords: Spectral techniques, Galois field expressions, Finite fields, Fast Fourier transform, FFT, Parallel programming, GPU computing.

1 Introduction

Functional expression over finite fields, often called Galois field (*GF*) expressions, are polynomials used as analytical representations of multiple-valued logic (MVL) functions. In this sense, MVL functions are viewed as functions defined over finite (Galois) fields $GF(p)$, where p is the non-necessarily prime order of the field [13].

The presented research is based on a notion that time needed for computing coefficients in *GF*-expressions can be significantly shortened by adapting the fast Fourier transform (FFT)-like algorithms for their computation, developed

[1]Dept. of Computer Science, Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, 18000 Niš, Serbia; E-mail: dusan.b.gajic@gmail.com

through their spectral interpretation, to the architecture and programming model of modern graphics processing units (GPUs) [5, 8].

The motivation for the use of GPUs for computational purposes in the presented research is the following. The approach based on using GPUs for performing general-purpose non-graphics algorithms in a highly parallel manner, called general-purpose computing on GPUs (GPGPU) or GPU computing, has recently attracted significant interest of researchers [3, 12]. The fast-growing field of GPU computing was made possible by the evolution of GPU single-instruction, multiple-threads (SIMT) architectures [1, 12], followed by the appearance of Nvidia CUDA [9] and OpenCL [2, 11] programming frameworks, which made the immense GPU computational resources more accessible to researchers. Each step of the FFT-like algorithms involves performing the same elementary butterfly operations over different subsets of operands [4]. These butterfly operations are independent of each other and, therefore, can be performed in parallel, without any need for communication and synchronization within a step. These properties of the FFT-like algorithms make them a suitable match to the nature of the GPU hardware.

The remainder of the paper is organized as follows. In Section 2, we discuss the theoretical background of the paper. The proposed mappings of the Cooley-Tukey and the constant geometry fast algorithms for computing Galois field expressions for quaternary logic functions to the architecture and programming model of GPUs are considered in Section 3. Their implementation for efficient processing on graphics processors is discussed in Section 4. Section 5 presents the experimental environment, used for the verification of the proposed solutions, and the recorded experimental results. The paper is closed with some conclusions, as well as a short discussion of possible directions for further work.

2 Galois Field Expressions for Quaternary Logic Functions

In this section, we present the definition of the Galois field expressions for quaternary (four-valued) logic functions and the Cooley-Tukey and the constant geometry FFT-like algorithms for computing coefficients in these expressions. For more detailed discussions of these topics, we refer to [13, 15].

Each quaternary function of n variables can be represented as a polynomial of the form

$$f(x_1, x_2, ..., x_n) = \sum_{i=0}^{4^n-1} g_i \phi_i \ , \tag{1}$$

where $g_i \in \{0, 1, 2, 3\}$, and ϕ_i are the product terms defined in the natural (Hadamard) order as elements of the vector $\boldsymbol{X}_{4GF}(n)$ defined as

$$\boldsymbol{X}_{4GF}(n) = \bigotimes_{i=1}^{n} \boldsymbol{X}_{4GF}(1), \quad \boldsymbol{X}_{4GF}(1) = [x_i^0 \; x_i^1 \; x_i^2 \; x_i^3], \tag{2}$$

where $\otimes$ denotes the Kronecker product. Since 4 is a non-prime number, additions and multiplications in $GF(4)$ are carried out as defined in **Table 1**. Notice that, when p is a prime number, ordinary operations of addition and multiplication modulo p can be used as operations in $GF(p)$ [13].

Table 1

Addition and multiplication in GF(4).

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

·	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

The set of basic functions for $n = 1$ is given by the columns of the matrix

$$\boldsymbol{X}_{4GF}(1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 3 & 2 & 1 \end{bmatrix}. \tag{3}$$

In matrix notation, the coefficients g_i in the Galois field expression for a function f, specified by the function vector $\boldsymbol{F} = [f(0), f(1), ..., f(4^n - 1)]^T$, are computed as

$$\boldsymbol{S}_{f,4GF} = \boldsymbol{G}_{4GF}(n)\boldsymbol{F}, \tag{4}$$

where

$$\boldsymbol{G}_{4GF}(n) = \bigotimes_{i=1}^{n} \boldsymbol{G}_{4GF}(1), \; \boldsymbol{G}_{4GF}(1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 3 & 2 \\ 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{5}$$

Example 1. *For a quaternary logic function with two variables $f(x_1, x_2)$, given by the function vector* $\boldsymbol{F} = [3, 1, 2, 0, 2, 1, 2, 2, 0, 3, 1, 0, 0, 2, 3, 2]^T$, *the GF transform matrix in GF*(4) *can be computed directly from the definition by using* (5). *Then, using this matrix and* (4), *the coefficients in the GF*(4)*-expression for f are computed by matrix-vector multiplication which yields*

$$\boldsymbol{S}_{f,4GF} = \boldsymbol{G}_{4GF}(2)\boldsymbol{F} = [3, 0, 2, 0, 2, 2, 3, 3, 2, 1, 0, 2, 1, 0, 2, 2]^T. \tag{6}$$

However, this approach has a computational complexity of $O(N^2)$, *where* $N = 4^n$ *is the length of the function vector.*

The same computations can be performed using the FFT-like algorithms, with an $O(N\log N)$ *asymptotical time complexity* [13, 15]. *In this paper, we discuss the Cooley-Tukey and the constant geometry fast algorithms, based on different factorizations of the transform matrix* [10, 15].

In the case considered in this example, the Cooley-Tukey algorithm stems from the factorization of the GF(4) *transform matrix taking the following form*

$$\boldsymbol{G}_{4GF}(2) = \boldsymbol{C}_1\boldsymbol{C}_2 . \tag{7}$$

where

$$\boldsymbol{C}_1 = \boldsymbol{G}_{4GF}(1) \otimes \boldsymbol{I}_4(1),\ \boldsymbol{C}_2 = \boldsymbol{I}_4(1) \otimes \boldsymbol{G}_{4GF}(1). \tag{8}$$

$\boldsymbol{I}_4(1)$ *in* (8) *represents the 4× 4 identity matrix. Each of the matrices* $\mathbf{C}_1$ *and* $\mathbf{C}_2$ *describes a step in the Cooley-Tukey FFT-like algorithm for computing the coefficients in* $\boldsymbol{S}_{f,4GF}$. *The basic transform matrix* $\mathbf{G}_{4GF}$ (1) *specifies the elementary butterfly operation in the algorithm.*

The computation through this algorithm is performed as

$$\boldsymbol{S}_{f,4GF} = \boldsymbol{G}_{4GF}(2)\boldsymbol{F} = \boldsymbol{C}_1(\boldsymbol{C}_2\boldsymbol{F}). \tag{9}$$

The first step of the algorithm computes

$$\boldsymbol{S}_{f_1,4GF} = \boldsymbol{C}_2\boldsymbol{F} = \left[\,3,\,1,\,2,\,0,\,2,\,0,\,0,\,1,\,2,\,1,\,2,\,3,\,1,\,1,\,2,\,0\right]^T, \tag{10}$$

while the second step produces

$$\boldsymbol{S}_{f_{1,2},4GF} = \boldsymbol{C}_1\boldsymbol{S}_{f_1,4GF} = \left[\,3,\,0,\,2,\,0,\,2,\,2,\,3,\,3,\,2,\,1,\,0,\,2,\,1,\,0,\,2,\,2\right]^T. \tag{11}$$

The same coefficients can be computed using the constant geometry fast algorithm, based on the factorization of the transform matrix into identical factor matrices in the following way

$$\boldsymbol{G}_{4GF}(2) = \boldsymbol{A}^2, \tag{12}$$

where

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_0 \\ \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \boldsymbol{A}_3 \end{bmatrix},\ \boldsymbol{A}_i = diag(\boldsymbol{a}_i, \boldsymbol{a}_i, ..., \boldsymbol{a}_i),\ i = 0,1,2,3, \tag{13}$$

where $\mathbf{a}_i$ *are the row-vectors of* $\boldsymbol{G}_{4GF}(2)$

$$\boldsymbol{a}_0 = [1,0,0,0],\quad \boldsymbol{a}_1 = [0,1,3,2],\quad \boldsymbol{a}_2 = [0,1,2,3],\quad \boldsymbol{a}_3 = [1,1,1,1], \tag{14}$$

Therefore, the factor matrix $\boldsymbol{A}$, *in the case of factorizing* $\boldsymbol{G}_{4GF}(2)$, *has the form*

$$\boldsymbol{A}=\begin{bmatrix}\boldsymbol{A}_0\\ \boldsymbol{A}_1\\ \boldsymbol{A}_2\\ \boldsymbol{A}_3\end{bmatrix}=\begin{bmatrix}\boldsymbol{a}_0 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{a}_0 & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_0 & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_0\\ \boldsymbol{a}_1 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{a}_1 & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_1 & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_1\\ \boldsymbol{a}_2 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{a}_2 & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_2 & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_2\\ \boldsymbol{a}_3 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{a}_3 & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_3 & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{a}_3\end{bmatrix}, \tag{15}$$

where $\boldsymbol{0}=[0, 0, 0, 0]$.

Then,

$$\boldsymbol{G}_{4GF}(2)=\boldsymbol{A}^2. \tag{16}$$

The use of constant geometry algorithms for computing coefficients in GF-expressions was first proposed in [14].

In this case, the first step of the algorithm yields

$$\boldsymbol{S}_{f_1,4GF}=\boldsymbol{AF}=\left[3, 2, 0, 0, 0, 3, 0, 3, 2, 3, 1, 2, 0, 3, 2, 3\right]^T, \tag{17}$$

while the second step produces the same final coefficients

$$\boldsymbol{S}_{f_{1,2},4GF}=\boldsymbol{AS}_{f_1,4GF}=\left[3, 0, 2, 0, 2, 2, 3, 3, 2, 1, 0, 2, 1, 0, 2, 2\right]^T. \tag{18}$$

Fig. 1 shows the signal flow graphs for the Cooley-Tukey and the constant geometry FFT-like algorithms for computing coefficients in a *GF*(4)-expression for a quaternary function of two variables. In this figure, solid lines correspond to weights of 1, dashed lines carry weights 2, and dotted lines represent weights of 3. Operands on which different elementary butterflies operate are identified by different colors in the figure.

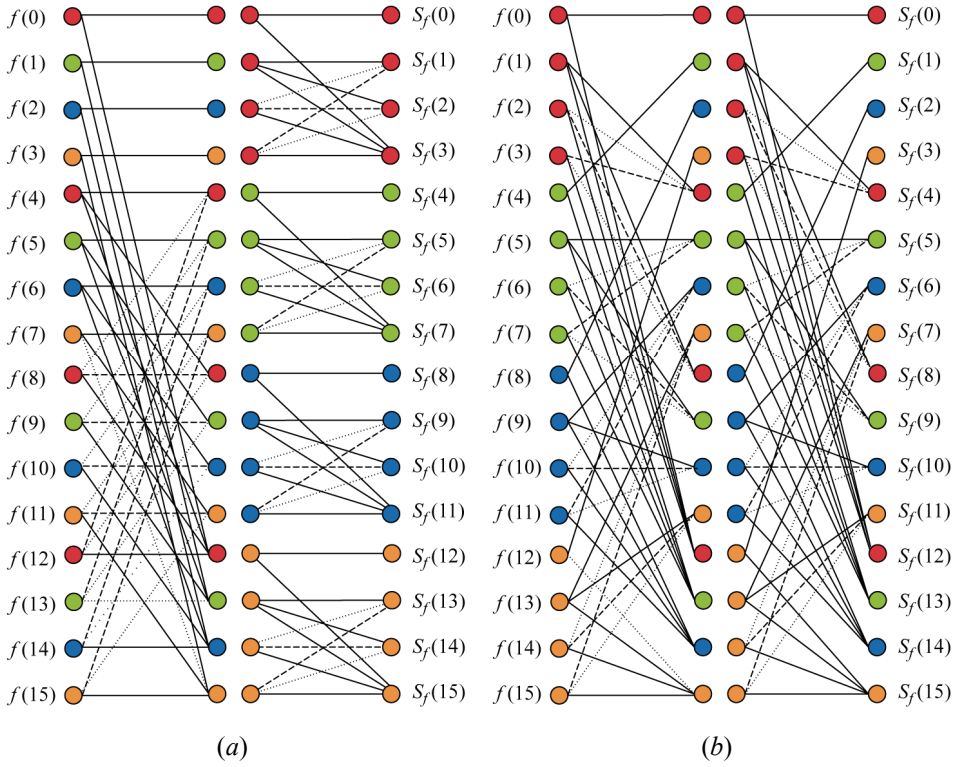

Fig. 1 – *Signal flow graphs for the Cooley-Tukey* (a) *and the constant geometry* (b) *algorithms for computing a GF(4)-expression for a quaternary function with* $n = 2$.

Both the Cooley-Tukey and the constant geometry algorithms are characterized by identical asymptotical time complexity and involve performing the same elementary butterfly operations, defined by the basic transform matrix. However, differences in these two factorizations of the *GF*-transform matrix lead to distinct memory access patterns, which can have significant effects on processing times needed for computing coefficients in *GF*-expressions.

3 Mapping of the Algorithms

The key issues in efficient mapping of the FFT-like algorithms to parallel computing architectures, such as GPUs, are:

1. The implementation of address arithmetic, i.e., computations needed for data fetching and storing, and
2. Organization of computations, i.e. distribution of the butterfly operations over processing elements.

In the proposed mappings, computations involved in the elementary butterflies of the considered fast algorithms are mapped to device programs, processed on the GPU, while the rest of the procedures are performed as parts of the CPU-processed host programs [6, 9]. Each thread performs a radix-4 butterfly over different input function values and the number of threads in each algorithm step is, therefore, $N/4$. This organization of computations, in which each butterfly is mapped to a single GPU thread, allows having a large number of threads performing the same operations on different data simultaneously, which is a good match to the nature of the GPU hardware [3].

In both CPU and GPU implementations of the Cooley-Tukey algorithm, the index l of the memory location which holds the first operand for each butterfly is computed as

$$l = b\%d + p \cdot d \cdot (b \setminus d), \tag{19}$$

where b is a positive integer that uniquely identifies the butterfly, d is the distance between the elements over which the butterfly performs the computations in the current step of the algorithm, % represents the modulo operation and \ stands for integer division. The value of d for the k-th step is computed as $d = p^{n-k}$. Notice that both the modulo operation and the integer division are not implemented directly in CPU or GPU hardware and, therefore, are performed by using routines based on simpler arithmetic operations such as addition and multiplication [2, 9].

The other operands for a butterfly are fetched from the locations with indices l_i computed as

$$l_i = l + i \cdot d\ , \tag{20}$$

for i = 1, 2,…, p–1. Notice that memory locations of the operands on which each butterfly operates change with each algorithm step.

For the constant geometry algorithm, the index l of the memory location which holds the first operand for each butterfly is computed as

$$l = b \cdot p\ , \tag{21}$$

while the rest of the operands are then fetched from locations l_i computed as

$$l_i = l + i\ , \tag{22}$$

for i = 1, 2,…, p–1. Notice that memory locations of the operands on which each butterfly operates are independent of the current algorithm step.

Since, in the case of the constant geometry algorithms, memory locations in which results of computations are stored differ from the ones from which the input data were read, these locations need to be computed in the following way

$$o_i = b + i \cdot p^{n-1}, \quad i = 1, 2, ..., p-1\ . \tag{23}$$

After comparing (19) and (20) with (21)–(23), it can be concluded that the address arithmetic in the case of constant geometry algorithm involves simpler operations, which are directly implemented in hardware and, therefore, can be performed faster. As a consequence, processing times for these algorithms are shorter than for the Cooley-Tukey algorithms, as reported in Section 5.

4 Implementation of the Algorithms

The Cooley-Tukey FFT-like algorithms allow in-place implementations using a single array in memory, since input data are fetched from the same locations where the results are stored. However, these algorithms require more complex address arithmetic than the constant geometry algorithms, since the locations of the operands change in each step of the algorithm.

Constant geometry algorithms require an out-of-place implementation with separate memory arrays for input data and output results, since operands are read from one set of locations and results are stored in another set of locations. On the other hand, these algorithms involve simple address arithmetic.

As a consequence, GPU implementations of these two algorithms require different amounts of GPU resources, in terms of registers, number of instructions, and occupancy [6, 14].

The operations of addition and multiplication in $GF(4)$, defined as in **Table 1**, need to be implemented on the program level, since the syntax of programming languages includes only modulo operations which can be used for computing $GF(p)$-expressions solely for p prime. This is most efficiently done by using look-up tables, represented as matrices. The result of performing the operation on two operands x and y is the element of the matrix stored at location with indices x and y.

5 Experimental Results

5.1 Experimental settings

The experiments reported in this section are performed on platforms specified in **Table 2**. Since computations are performed over function vectors, the running time of the implementations is independent of function values. Therefore, the experiments are performed using randomly generated quaternary functions with different number of variables. The presented values for the CPU and GPU computation times are average values for 10 program executions. The discussed algorithms were also implemented as sequential C/C++ programs processed on CPUs. These implementations were developed solely to measure baseline processing times needed to establish the speed-ups achieved by transferring computations to GPUs. The source code for referent C/C++ implementations is compiled for the *x64* platform using the maximum level of

performance-oriented optimizations. Proposed GPU implementations of the considered algorithms were developed using CUDA (for Nvidia GPU on platform **Intel/Nvidia**) and OpenCL (for AMD GPU on platform **Intel/AMD**).

Table 2
System specifications for the experimental platforms.

Platform	Intel/Nvidia	Intel/AMD
CPU		
type	Intel Core i7-920	Intel Core i7-2600K
frequency	2.66 GHz	3.4 GHz
number of cores/threads	4/8	4/8
RAM	12 GB DDR3 2000 MHz	16 GB DDR3 1600 MHz
GPU		
Type	Nvidia GTX 560 Ti	AMD Radeon HD 6750
Number of cores	384	720
Core frequency	900 MHz	800 MHz
Processing power	1.26 TFLOPs	1 TFLOPS
Graphics RAM	1 GB GDDR5 4.0 GHz	1 GB GDDR5 1.1 GHz
Memory bandwidth	128 GB/s	73.6 GB/s
Operating system	Windows 7 Ultimate (64-bit)	
IDE	MS Visual Studio 2012 Ultimate	
SDK	Nvidia GPU Computing 5.0	AMD APP 2.9
GPU driver	Nvidia 320.57	AMD Catalyst 13.9
Profiling tools	Nvidia Parallel Nsight 3.1	AMD APP Profiler 2.5

5.2 Experimental results

The results regarding processing times for different implementations of the considered fast algorithms for computing coefficients in Galois field expressions for quaternary functions are presented in **Table 3**, as well as in Figs. 2 and 3.

On the **Intel/Nvidia** platform, the CUDA GPU implementations of the Cooley-Tukey and the constant geometry algorithms are up to 10.22 and 10.86 times, respectively, faster than their C/C++ implementations processed on CPUs. The CPU implementation of the constant geometry algorithm is up to 13% faster than the respective Cooley-Tukey FFT implementation, while gains on the GPU are up to 7.5%. On the **Intel/AMD** platform, the OpenCL GPU implementations outperform their referent CPU counterparts by a factor of 3.48, for the Cooley-Tukey algorithm, and 3.97, for the constant geometry algorithm. On this platform, the CPU implementation of the constant geometry algorithm is up to 12.5% faster than the respective Cooley-Tukey algorithm implementation, while on the GPU the improvement in processing time is up to 16.7%.

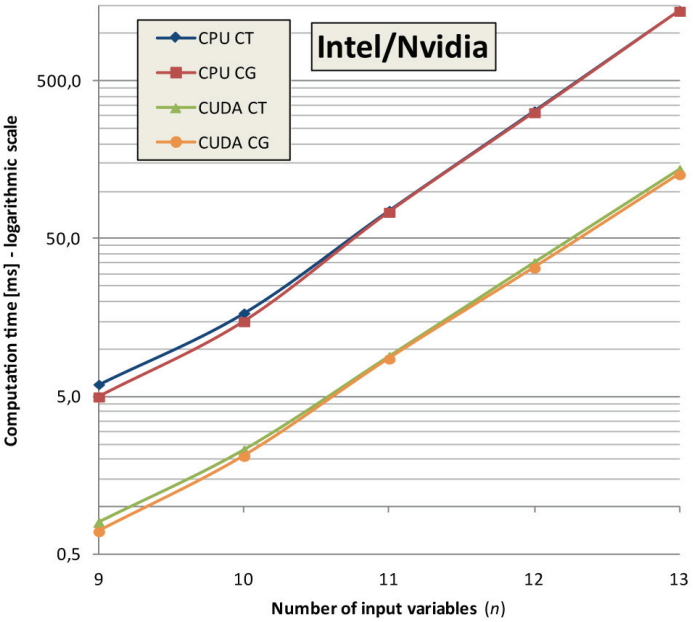

Fig. 2 – *Computation times for different implementations on the* ***Intel/Nvidia*** *platform.*

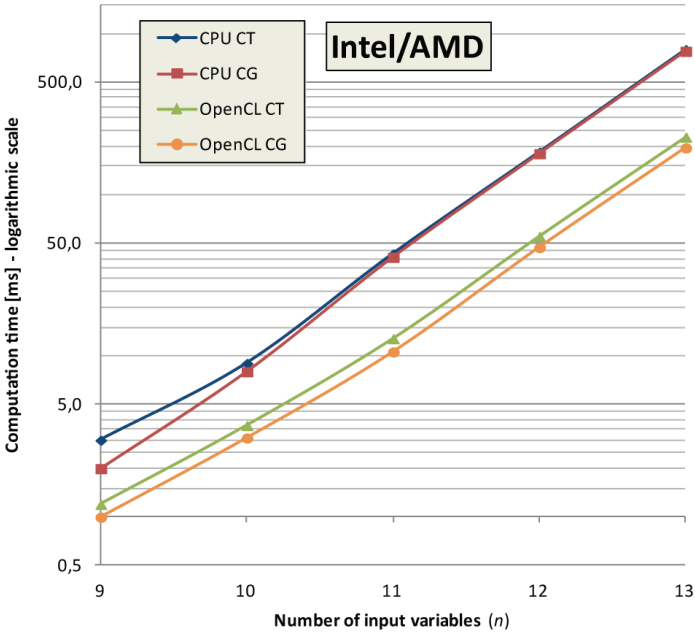

Fig. 3 – *Computation times for different implementations on the* ***Intel/AMD*** *platform.*

Table 3
Computation times in [ms] for different implementations of the Cooley-Tukey (CT) and the constant geometry (CG) algorithms for computing GF(4)-expressions.

N	Intel/Nvidia				Intel/AMD			
	CPU/C++		GPU/CUDA		CPU/C++		GPU/OpenCL	
	CT	CG	CT	CG	CT	CG	CT	CG
9	6	5.5	0.8	0.7	3	2	1.2	1
10	17	15	2.3	2.1	9	8	3.7	3.1
11	76	74	9	8.7	43	41	12.8	10.6
12	325	317	35.4	32.8	184	180	55.2	47.1
13	1409	1394	137.8	128.3	796	777	228.1	195.6

The following can be concluded after analyzing times needed for performing individual steps of the examined algorithms, shown in **Table 4** for the case of the largest considered quaternary functions ($n = 13$). Processing times for the CUDA implementation of the Cooley-Tukey FFT show a significant rise in the last two steps, leading to lower overall performance. This increase in processing times can be explained by the change in memory access pattern for storing results of computations, with writes occurring at evermore closer locations which can result in memory access conflicts. The constant geometry algorithm implementation is characterized by almost constant processing times per step, which are, at start, a bit slower than times for the Cooley-Tukey algorithm, but avoid later rise and, thus, achieve better overall performance.

Table 4
Computation times [ms] for different steps of CUDA implementations of the Cooley-Tukey (CT) and the constant geometry (CG) algorithms, in the case $n = 13$.

step	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
CT	9.64	9.72	9.66	9.67	9.64	9.69	9.57	9.77	9.70	9.77	9.97	12.53	18.46
CG	9.87	9.87	9.86	9.88	9.86	9.87	9.87	9.85	9.88	9.86	9.85	9.86	9.88

The profiling tools used for measuring and optimizing GPU program performance (see **Table 2**) also allow an insight into further causes of shorter processing times for implementations of the constant geometry algorithm than for the implementations of the Cooley-Tukey algorithm. The improved processing times can also be explained by lower use of resources per thread in the GPU implementations of the constant geometry algorithm. The OpenCL implementation of the Cooley-Tukey requires 27 registers and 100 instructions

per thread, while the respective constant geometry algorithm implementation needs 24 registers and only 65 instructions per thread. For CUDA implementations, the ratio of per thread register use is 23:20 in favor of the constant geometry algorithm, leading to effective multiprocessor occupancy of 66.7%, for the Cooley-Tukey, and 100% for the constant geometry algorithm implementations.

6 Conclusion

In this paper, we proposed a method for accelerated computation of *GF*(4)-expressions for quaternary logic functions, based on the application of GPUs. The method stems from the spectral interpretation of *GF*(4)-expressions, which permits the use of FFT-like algorithms, which are, in the presented research, adapted for highly parallel processing on GPUs. We consider two classes of FFT-like algorithms – the Cooley-Tukey and the constant geometry algorithms. We show that the simpler address arithmetic in the constant geometry algorithm leads to lower use of resources, and, as a consequence, better performance on both CPUs and GPUs. Experimental results confirm that the presented approach of using GPUs for computations leads to significant reduction in processing times (up to 10.86 times when compared to referent CPU implementations). In this way, the application of the proposed method widens the set of problem instances which can be efficiently handled in practice. Plans for further work include development of hybrid algorithm implementations, which would use both CPUs and GPUs for performing computations in parallel. In addition, one of the goals for future research is the construction of implementations of external memory algorithms for GPUs, which would allow processing of larger MVL functions than the ones considered in this paper.

Acknowledgement

The author would like to thank Prof. Radomir S. Stanković for his contribution to the research reported in this paper. The presented research is partly supported by the Ministry of Education and Science of the Republic of Serbia, projects ON174026 (2011-2014) and III44006 (2011-2014).

8 References

[1] T.M. Aamodt: Architecting Graphics Processors for Non-graphics Compute Acceleration, IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Victoria, BC, Canada, 23 – 26 Aug. 2009, pp. 963 – 968.

[2] AMD Accelerated Parallel Processing OpenCL Programming Guide, Available Online: http://developer.amd.com/sdks/AMDAPPSDK.

[3] A.R. Brodtkorb, M.L. Sætra, T.R. Hagen: Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing, Journal of Parallel and Distributed Computing, Vol. 73, No. 1, Jan. 2013, pp. 4 – 13.

[4] J.W. Cooley, J.W. Tukey: An Algorithm for the Machine Calculation of Complex Fourier Series, Mathematics of Computation, Vol. 19, No. 90, Apr. 1965, pp. 297 – 301.

[5] J. Astola, M. Kameyama, M. Lukac, R. S. Stanković: GPU Computing with Applications in Digital Logic, TICSP, Tampere, Finland, 2012.

[6] D.B. Gajić, R.S. Stanković: The Impact of the Address Arithmetic on the GPU Implementation of Fast Algorithms for Computing the Vilenkin-Chrestenson Transform, IEEE 43rd International Symposium on Multiple-valued Logic, Toyama, Japan, 22 – 24 May 2013, pp. 296 – 301.

[7] B.R. Gaster, L. Howes, D. Kaeli, P. Mistry, D. Schaa: Heterogeneous Computing with OpenCL, Elsevier, NY, USA, 2012.

[8] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli: High Performance Discrete Fourier Transforms on Graphics Processors, International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, 15 – 21 Nov. 2008, pp. 1 – 12.

[9] D. Kirk, W.M.W. Hwu: Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, Burlington, MA, USA, 2013.

[10] M.G. Karpovsky, R.S. Stanković, J.T. Astola: Spectral Logic and Its Applications for the Design of Digital Devices, John Wiley and Sons, HOBOKEN, NJ, USA, 2008.

[11] The OpenCL Specification 2.0, Khronos OpenCL Working Group, 2013.

[12] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips: GPU Computing, Proceedings of the IEEE, Vol. 96, No. 5, May 2008, pp. 879 – 899.

[13] R.S. Stanković, J.T. Astola, C. Moraga: Representation of Multiple-valued Functions, Morgan and Claypool Publishers, San Rafael, CA, USA, 2012.

[14] R.S. Stanković, J.T. Astola, C. Moraga, D.B. Gajić: Constant Geometry Algorithms for Galois Field Expressions and Their Implementation on GPUs, IEEE 44th International Symposium on Multiple-valued Logic, Bremen, Germany, 19 – 21 May2014. (To be published).

[15] M.R. Stojić, M.S. Stanković, R.S. Stanković: Discrete Transforms in Applications, Nauka, Belgrade, 1993. (In Serbian).