Efficient Parallel Computation of the Galois Field Expressions for Ternary Logic Functions

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

***Abstract* – This paper presents a comparison of the time efficiency of parallel implementations of the Cooley-Tukey algorithm for computing Galois field expressions for ternary logic functions. The examined parallel implementations are an MPI (Message Passing Interface) implementation, processed on a multicore CPU (central processing unit), and CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) implementations, processed on the GPU (graphics processing unit). Program running times are compared using randomly-generated ternary functions with different number of variables. The experiments show that the MPI parallel implementation is up to 4.6 times faster than the C++ sequential implementation when the algorithm is performed on a quad-core CPU. The CUDA and OpenCL implementations on the GPU with 8 streaming multiprocessors are up to 30 and 22 times, respectively, faster than the C++ CPU implementation. These speed-ups are up to 7 and 4.5 times, respectively, when compared with the MPI CPU implementation. The CUDA GPU implementation is from to 2% to 53% faster than the OpenCL implementation, depending on the number of variables in the considered function.**

***Keywords* – Spectral transforms, multiple-valued logic, fast Fourier transform, Galois field expressions, Cooley-Tukey algorithm, parallel implementations, GPU computing, MPI, CUDA, OpenCL.**

I. Introduction

Moore's law has remained valid during the last decade only by a switch to multicore and manycore parallel processor architectures [4]. Today's computer systems are typically composed of multicore CPUs (central processing units) and manycore GPU (graphics processing units), but also feature specialized processors such as DSPs (digital signal processors). In order to efficiently use these heterogeneous and parallel computational platforms, existing parallel programming standards for CPUs, such as MPI (Message Passing Interface) [3, 12], were extended, and completely new computational and programming models for GPUs, such as CUDA [8, 10] and OpenCL [7, 9], were developed.

This paper is a case study on performance of MPI, CUDA, and OpenCL parallel implementations of the Cooley-Tukey algorithm for computing Galois field (GF) expressions for ternary logic functions. Since the time for computing a GF(3) expression is exponential in the number of variables of a given function, processing time is often a limiting factor for practical applications of GF-expressions, in spite of the existence of FFT (fast Fourier transform)-like algorithms [13]. Therefore, the development of parallel implementations of algorithms for computing Galois field expressions is an important task for the application of these expressions in practice. The case study presented in this paper aims at identifying the computing platform and programming framework which is most suitable for the considered computations in terms of processing times.

The paper is organized as follows. In Section II we briefly define the Galois field expressions for ternary logic functions and Cooley-Tukey algorithms for computing the coefficients in these expressions. To make this paper self-contained, basic information about MPI, CUDA, and OpenCL parallel programming models is provided in Section III. Details of mappings and implementations are discussed in Section IV. Section V presents the experimental results. Section VI offers some conclusions and directions for further work.

[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: dule.gajic@gmail.com, radomir.stankovic@gmail.com.

II. Galois Field Expressions for Ternary Functions

In this section we present the definition of the Galois field expressions for ternary logic functions and Cooley-Tukey algorithms for computing the coefficients in these expressions. For more detailed discussions of these topics, we refer to [13].

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

$$f(x_1, x_2, ..., x_n) = \sum_{i=0}^{3^n-1} g_i \phi_i, \quad (1)$$

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 $\mathbf{X}_{3GF}(n)$ defined as

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

The symbol $\otimes$ denotes the Kronecker product. Additions and multiplications are carried out modulo 3, i.e., in GF(3).

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

$$\mathbf{X}_{3GF}(1) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}. \quad (3)$$

In matrix notation, the coefficients g_i in the Galois field expression for a function f, specified by the function vector $\mathbf{F} = [f(0), f(1), ..., f(3^n-1)]^T$, are computed as

$$\mathbf{S}_{f,3GF} = \mathbf{G}_{3GF}(n)\mathbf{F}, \quad (4)$$

where

$$\mathbf{G}_{3GF}(n) = \bigotimes_{i=1}^{n} \mathbf{G}_{3GF}(1),\ \mathbf{G}_{3GF}(1) = (\mathbf{X}_{3GF}(1))^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix}. \quad (5)$$

Example 1. For n = 2, the Galois field transform matrix in $GF(3)$ is defined as

$$\mathbf{G}_{3GF}(2) = \mathbf{G}_{3GF}(1) \otimes \mathbf{G}_{3GF}(1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \quad (6)$$

Example 2. For a ternary function of two variables $f(x_1, x_2)$ given by the vector F = $[0,1,2,0,2,1,2,0,0]^T$, the $GF(3)$ transform matrix is as in Example 1.

From (4), the coefficients in the GF(3) expression for f are computed as

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

This computational complexity is $O(N^2)$, where $N = 3^n$ is the length of the function vector.

The same computation can be performed using the Cooley-Tukey FFT-like algorithm, with an $O(N\log N)$ asymptotical time complexity [13]. The algorithm is based on the factorization of the $GF(3)$ transform matrix as

$$\mathbf{G}_{3GF}(2) = \mathbf{C}_1\mathbf{C}_2. \quad (7)$$

where

$$\mathbf{C}_1 = \mathbf{G}_{3GF}(1) \otimes \mathbf{I}_3(1), \text{ and } \mathbf{C}_2 = \mathbf{I}_3(1) \otimes \mathbf{G}_{3GF}(1). \quad (8)$$

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 $\mathbf{S}_{f,3GF}$. The basic transform matrix $\mathbf{G}_{3GF}$ (1) specifies the elementary butterfly operation in the algorithm. Fig. 1 shows the Cooley-Tukey FFT for a function of two variables. The computation through this algorithm is performed as

$$\mathbf{S}_{f,3GF} = \mathbf{G}_{3GF}(2)\mathbf{F} = \mathbf{C}_1(\mathbf{C}_2\mathbf{F}). \quad (9)$$

The first step of the algorithm computes

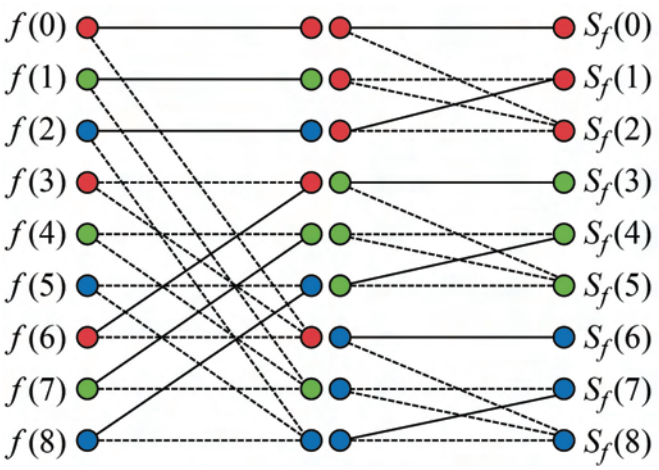

Fig. 1. Signal flow graph for the Cooley-Tukey algorithm in GF(3) for n = 2. Solid lines carry weight 1 and dashed lines carry weight 2.

$$\mathbf{S}_{f_1,3GF} = \mathbf{C}_2\mathbf{F} = [0,1,0,0,2,0,2,0,1]^T, \quad (10)$$

while the second step produces

$$\mathbf{S}_{f_{1,2},3GF} = \mathbf{C}_1\mathbf{S}_{f_1,3GF} = [0,1,0,2,1,1,1,0,2]^T. \quad (11)$$

III. Parallel Programming Frameworks

MPI has been, since its inception in 1991, the dominant programming model for parallel scientific and high-performance computing on CPUs [4]. It supports both shared and distributed memory systems, and enables efficient scaling of program execution from environments with a couple of processing cores to large computing clusters. For more details on MPI, we refer to [3, 12].

The computational approach based on using graphics processing units for performing general-purpose algorithms in a highly parallel manner, called the GPGPU (*general-purpose computing on GPUs*) or *GPU computing*, has recently attracted significant interest of researchers and proved its value in scientific and high-performance computing [5, 6]. GPU computing was made possible by the evolution of GPU architectures [1, 11]. The appearance of Nvidia CUDA and OpenCL programming frameworks made the GPU computational resources more accessible to researchers and programmers. For more details on GPU computing and CUDA and OpenCL frameworks, we refer to [2, 7, 8, 9, 10, 11].

IV. Mappings and Implementations of the Algorithm

The key issues in efficient mapping of the Cooley-Tukey algorithm to parallel computing architectures 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 both CPU and GPU parallel implementations of the Cooley-Tukey algorithm, the index l of the memory location which holds the first operand for each butterfly is computed as

$$l = b \,\%\, d + r \cdot d \cdot (b \setminus d), \quad (12)$$

where b is a positive integer that uniquely identifies the butterfly, r is the transform radix, d is the distance between the elements over which the butterfly performs the computations in the current step of the algorithm, and \ stands for integer division. The value of d for the k-th step is computed as $d = r^{n-k}$.

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

$$l_i = l + i \cdot d, \quad (13)$$

for $i = 1, 2, \ldots, r\text{-}1$. Notice that memory locations of the operands on which each butterfly operates change with each algorithm step.

In the MPI implementation, each processor core performs $N/(3*nprocs)$ butterflies, where *nprocs* is the number of active CPU processes. The number of processes typically equals the number of CPU cores. Switching between butterflies assigned to the same process is done as

count = *N*/(3**nprocs*),

start = *id* * *count*,

stop = *start* + *count*,

where *id* is the unique identifier of the process. These variables are used in the double *while* loop that implements the butterfly computations as

```
while (d >= 1)
{
        k = start;
        while (k < stop)
        {
                // data fetching, followed by
                // computations in the butterfly
                // defined by G3GF(1) and storing of results
                k = k + 1
        }
        d = d / 3
}
```

When computing the Galois field expressions on GPUs, a more fine-grained parallel computing model is used. In this model, each thread performs a single butterfly over different elements of the function vector **F**. Therefore, the variable b in (12) corresponds to a unique identifier of the thread, usually denoted as *tid*. Therefore, the computations are divided upon $N/3$ threads in total.

Both CUDA and OpenCL implementations are composed of the host program, which creates data structures and controls the execution, and the device program, which implements the butterfly operations. The device program contains the description of computations for a single thread, defined by $\mathbf{G}_{3GF}(1)$, and *tid* is used to distinguish the operands processed by each thread.

V. Experiments

A. Experimental Settings

The experiments reported in this section are performed on platforms specified in Table I.

Since the 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 ternary functions with different number of variables. The presented values for the CPU and GPU computation times are average values for 10 program executions. The source code for the C/C++ implementation is compiled for the *x64* platform using the maximum level of performance-oriented optimizations. The presented GPU processing times include the times for data transfer between the host and the device and vice versa, as well as times for setting kernel arguments and the kernel calls.

TABLE I
EXPERIMENTAL PLATFORM

CPU	*Intel Core i7-920 quad-core (2.66GHz)*
Memory	*12GB DDR3-2000*
Operating System	*Windows 7 Ultimate 64-bit*
Development Environment	*MS Visual Studio 2010 Ultimate*
Software Development Kit	
MPI	*Microsoft HPC Server 2008 –MPICH2*
CUDA OpenCL	*Nvidia GPU Computing 4.0*
GPU	*Nvidia GTX 650 Ti*
core frequency	*900 MHz*
memory	*1 GB GDDR5 4.2 GHz*
number of cores	*384*

B. Experimental results

The summary of the results considering processing times for different parallel implementations of the Cooley-Tukey algorithm for computing Galois field expressions for ternary functions is presented in Table II and Fig. 2.

The MPI implementation of the algorithm is up to 4.6 times faster than the respective C++ sequential implementation, when both implementations are processed on a quad-core CPU. This difference is almost constant throughout the considered range of functions. The speed-ups are larger than the number of available CPU processing cores due to Intel's Hyper Threading technology which allows 2 threads to exist simultaneously per each core. CUDA and OpenCL implementations on the GPU are up to 30 and 22 times, respectively, faster than the C++ CPU implementation. These speed-ups are up to 7 and 4.5 times, respectively, when compared with the MPI implementation. The CUDA GPU implementation is from to 2% to 53% faster than the respective OpenCL implementation. For $n \leq 15$, the difference in speed between CUDA and OpenCL is almost negligible, but as n increases CUDA becomes significantly faster.

TABLE II
COMPUTATION TIMES IN [SECONDS] FOR DIFFERENT IMPLEMENTATIONS

n		8	10	12	14	15	16	17
CPU	C++	0.001	0.005	0.029	0.302	0.973	3.122	10.025
	MPI	0.000	0.000	0.006	0.065	0.231	0.695	2.157
GPU	OpenCL	0.000	0.000	0.001	0.016	0.047	0.160	0.492
	CUDA	0.000	0.000	0.001	0.015	0.046	0.118	0.321

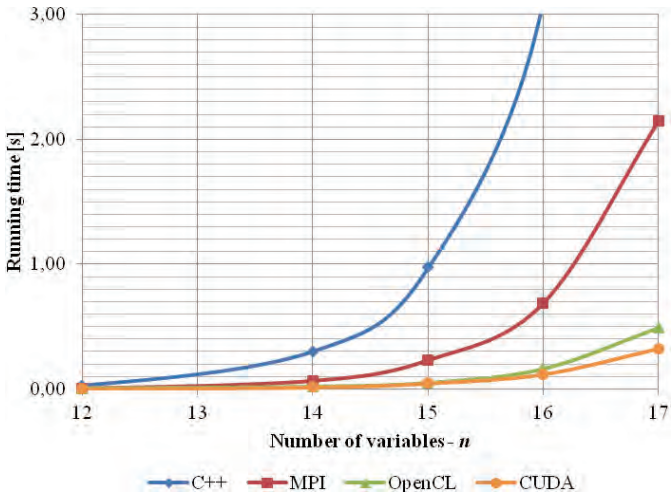

Fig. 1. Running times in [seconds] for different implementations of the GF(3) Cooley-Tukey FFT.

VI. CONCLUSIONS

In this paper, we presented a comparison of time efficiency of parallel algorithm implementations for computing Galois field expressions for ternary functions. The parallel implementation for CPUs was developed using Microsoft's MPI, while the two parallel implementations for GPUs were developed using CUDA and OpenCL. The experiments show that the GPU implementations outperformed the MPI implementation by a factor of up to 7, with CUDA implementation being faster than the OpenCL one by up to 53%. Therefore, the conclusion of the presented case study is that the GPU platform is better suited for performing the parallel implementations of the considered algorithm, with CUDA being the technology of choice when computing time is the most important limiting factor.

ACKNOWLEDGEMENTS

The research reported in this paper is partly supported by the Ministry of Education and Science of the Republic of Serbia, project OI 174026 (2011-2014).

REFERENCES

[1] T. M. Aamodt, "Architecting graphics processors for non-graphics compute acceleration", in *Proc. 2009 IEEE Pacific Rim Conf. Communications, Computers & Signal Processing*, Victoria, BC, Canada, 2009.

[2] Advanced Micro Devices, Inc., "AMD Accelerated Parallel Processing OpenCL Programming Guide", available from: *http://developer.amd.com /sdks/AMDAPPSDK*, [accessed 18 March 2013].

[3] Argonne National Laboratory, "MPICH – High Performance and Portable MPI", available from: *http://www.mpich.org*, [accessed 15 April 2013].

[4] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", UC Berkeley Technical Report No. UCB/EECS-2006-183, 2006.

[5] 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, 2013, pp. 4-13.

[6] D. B. Gajić, R. S. Stanković, "Computing spectral transforms used in digital logic on the GPU", in *GPU Computing with Applications in Digital Logic*, J. Astola, M. Kameyama, M. Lukac, R. S. Stanković (eds.), TICSP, Tampere, Finland, 2012, pp. 25-62.

[7] B. R. Gaster, L. Howes, D. Kaeli, P. Mistry, D. Schaa, *Heterogeneous Computing with OpenCL*, Elsevier, 2011.

[8] D. Kirk, W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

[9] Khronos,"OpenCL Specification 1.2", Khronos OpenCL Working Group, 2011.

[10] Nvidia, *Nvidia CUDA Programming Guide*, available from: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/ index.html* [accessed 18 March 2013].

[11] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, "GPU computing", *Proc. of the IEEE*, Vol. 96, No. 5, 2008, pp. 279–299.

[12] P. Pacheco, *An Introduction to Parallel Programming*, Elsevier, 2011.

[13] R. S. Stanković, J. T. Astola, C. Moraga, *Representation of Multiple-Valued Functions*, Morgan & Claypool Publishers, 2012.