GPU Accelerated Construction of Characters of Finite Abelian Groups

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

Abstract – **In group theory and Fourier analysis on finite Abelian groups, the group characters are an essential concept. In many applications, as for instance, spectral processing of logic functions (binary or *p*-valued), it is often required to construct the table of group characters for the specified group. This can be a computationally demanding task, both in terms of space and time, when dealing with large groups, since the group characters are viewed in matrix notation as rows of ($p^m \times p^m$) matrices, where *p* is the cardinality of the set where the given logic function and its variables take values, and *m* is the number of variables. The graphics processing unit (GPU), as a highly parallel computational platform, may facilitate this complex task.**

This paper discusses the application of the GPU processing to the construction of tables of group characters for finite Abelian groups represented as a direct product of cyclic subgroups of order *p*. We exploit the Kronecker product structure of these tables permitting redistribution of the related computing tasks over GPU resources. Experimental results confirm that the presented solution offers a considerable speed-up over the C/C++ implementation of the same character construction method processed on the central processing unit (CPU).

Keywords – **Abstract harmonic analysis, finite Abelian groups, group characters, Kronecker product, GPU computing, OpenCL.**

I. INTRODUCTION

Abstract harmonic analysis is a mathematical discipline that evolved from the classical *Fourier analysis* by the replacement of the real group *R* with an arbitrary locally compact Abelian or compact non-Abelian group [3, 6, 9, 14, 15]. This implies the transition from the exponential functions, used in classical Fourier analysis and viewed as the group characters of *R*, to the *group characters*, in the case of Abelian groups, and the *group representations*, in the case of non-Abelian groups [6, 9]. Abstract harmonic analysis provides foundations for the formulation of many methods with significant applications in electrical engineering and computer science [9, 16, 17, 18, 19]. In these methods, it is often required to construct the group characters of various Abelian groups and use them in further computations. With that motivation, this paper presents a method for an efficient construction of group characters of finite Abelian groups using the graphics processing unit (GPU). This choice of hardware is made due to the fact that contemporary GPUs are highly parallel computing engines which can simultaneously serve as programmable graphics processors and scalable parallel computational platforms [1, 8, 13]. For a given group *G*, the construction of group characters can be expressed in terms of the Kronecker product of characters of its subgroups of smaller orders. In this formulation, the algorithm for the construction of group characters expresses a substantial inherent parallelism and, therefore, the GPU is a natural choice of hardware for the implementation of this algorithm. The experimental comparisons of the proposed implementation on the GPU and the C/C++ implementation of the same algorithm processed on the central processing unit (CPU) confirm this assumption.

The rest of the paper is organized as follows. The background theory is introduced in Section 2. In Section 3, we propose a mapping of the algorithm for the construction of group characters to the GPU and discuss the details of the respective programming implementation. The experiments are discussed in Section 4. We close the paper with Section 5, by presenting some conclusions and possible directions for further research.

[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. BACKGROUND THEORY

In this section, we give a brief introduction to the theoretical background of the paper. For more detailed discussion of these topics, we recommend classical works such as [3, 15, 17], or more recent references [6, 9, 14].

We consider finite Abelian groups of the form $G = C_p^m = (\{0,1,...,p-1\}^m, \oplus_p)$, where C_p is the cyclic group of order *p*, and $\oplus_p$ is the componentwise addition modulo *p*.

The group characters $\chi_\omega^{(p)}(z)$, z = 0, 1,..., p^m-1, of the group *G* are defined as [9, 16, 17]:

$$\chi_\omega^{(p)}(z) = \exp\left(\frac{2\pi}{p} i \sum_{s=0}^{m-1} \omega_{m-1-s} z_s\right), \qquad (1)$$

where $i = \sqrt{-1}$, $\omega_s, z_s \in \{0,1,...,p-1\}$, and

$$\omega = \sum_{s=0}^{m-1} \omega_s p^{m-1-s}, \quad z = \sum_{s=0}^{m-1} z_s p^{m-1-s}. \qquad (2)$$

Example 1 The group character tables, for the cyclic groups C_p of orders p = 2, 3, and 4, are given in Table I, where $i = \sqrt{-1}$, $e_1 = -0.5 \cdot (1 - i\sqrt{3}) = \exp(2\pi i/3)$, and $e_2 = e_1^* = -0.5 \cdot (1 + i\sqrt{3}) = \exp(4\pi i/3)$.

TABLE I CHARACTER TABLES OF CYCLIC GROUPS

Cyclic group	C_2	C_3	C_4
Character table	$\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$	$\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix}$	$\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & -1 & -i\\ 1 & -1 & 1 & -1\\ 1 & -i & -1 & i\end{bmatrix}$

The group $G = C_p^m$ is the direct product of m elementary cyclic subgroups C_p. It follows, see for instance [3, 6, 15], that the character table of the group G is the Kronecker product of m character tables of its cyclic subgroup C_p.

Example 2 For the group C_3^2, the character table can be computed as the Kronecker product of two character tables of its cyclic subgroup C_3. In this way, only the character table of C_3 is computed through (1) and the character table for C_3^2 is generated as:

$$\left[C_3^2\right] = \left[C_3\right] \otimes \left[C_3\right] = \begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} \otimes \begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} =$$

$$\begin{bmatrix} 1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & 1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & 1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} \\ 1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & e_1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & e_2\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} \\ 1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & e_2\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} & e_1\cdot\begin{bmatrix}1 & 1 & 1\\ 1 & e_1 & e_2\\ 1 & e_2 & e_1\end{bmatrix} \end{bmatrix}. \quad (3)$$

This property of the character table will be exploited in the mapping of the computation of the character table to the GPU.

III. GPU CONSTRUCTION METHOD

A. GPU Computing

The technique of performing general-purpose algorithms on graphics processors, known as GPGPU (*general-purpose computing on GPUs*) or *GPU computing*, has recently become a subject of a fast growing research interest and practical application [1, 13].

This interest is mainly the result of two factors. First is the evolution of the GPU hardware towards a scalable, programmable, and highly parallel computing platform [1, 13], and the second is the development of the *Nvidia CUDA* [13] and *OpenCL* (*Open Computing Language*) [10] programming frameworks, based on the C/C++ language, which made the immense GPU computational resources more accessible. For the implementation purposes, we use OpenCL, since it allows the development of the code that is both accelerated and portable across heterogeneous processing platforms (GPUs, FPGAs, DSPs) [8, 10].

B. Algorithm Mapping

The key task in porting algorithms to the GPU is their efficient mapping to the *SPMD* (*single program*, *multiple data*) *processing model* and the *multi-level memory hierarchy* of GPUs [1, 2, 8, 12, 13]. In the GPU SPMD model, a single data parallel function called a *kernel* is executed over a stream of data by many threads in parallel. A *thread* is the smallest execution entity and represents a single instance of the kernel. The execution of the kernel is controlled by the *host program* processed by the CPU.

The mapping of the algorithm for the construction of group character tables to the GPU is explained using Example 2.

The matrix $\left[C_3^2\right]$ in (3) has the following block structure:

$$\left[C_3^2\right] = \begin{bmatrix} B_{00} & B_{01} & B_{02}\\ B_{10} & B_{11} & B_{12}\\ B_{20} & B_{21} & B_{22}\end{bmatrix}. \quad (4)$$

Blocks $B_{x,y}$ (x, y = 0, 1, 2) are the character tables for C_3 multiplied by the elements of the matrix $[C_3]$. Therefore, each block can be represented as:

$$B_{x,y} = c_{x,y}\cdot[C_3] = c_{x,y}\cdot\begin{bmatrix} a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{bmatrix} = c_{x,y}\cdot\left[a_{i,j}\right], \quad (5)$$

where $c_{x,y}, a_{i,j} \in \{1, e_1, e_2\}$, $x, y, i, j = 0,1,2$.

To each block we assign a thread $t = (x, y, a_{i,j})$, x, y, i, j = 0, 1, 2. Each thread performs a multiplication of $[C_3]$ by a scalar, as in (5). Threads are organized into a two-dimensional (x, y) array corresponding to the matrices to be computed. Fig. 1 represents the mapping of the character table computations to the GPU threads. Each thread processes a single block, which is indicated by a different color in Fig. 1.

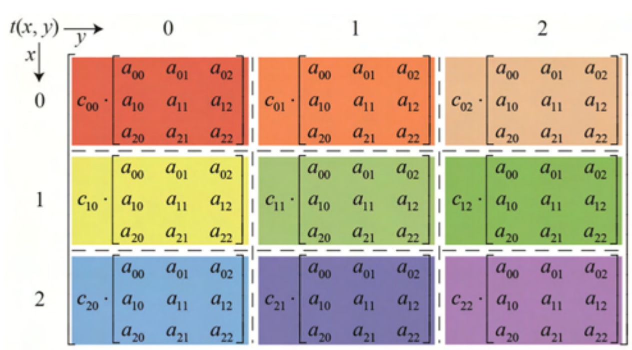

Figure 1. Mapping of the computations to the GPU threads for Example 2.

For the group C_3^2 in Example 2, we have nine threads in the first and only step of the algorithm (since this example involves only one Kronecker product), each performing the operation from (5) in parallel. In this case, indices of memory locations, where a thread $t(x, y, a_{i,j})$ stores the first element ($c_{x,y}\cdot a_{0,0}$) of the block, are computed as:

$$startElement \leftarrow x\cdot 3^3 + y\cdot 3\,. \quad (6)$$

Indices of memory locations for the rest of the elements ($c_{x,y} \cdot a_{i,j}$, x, y, i, j = 0, 1, 2, except for the case $i = j = 0$) in a computed block are determined as:

$$nextElement \leftarrow startElement + i \cdot 3^2 + j \ . \qquad (7)$$

The results of the computations are stored in the GPU global memory which has a linear layout. Formulas for the computation of the memory location indices ((6) and (7)) lead to the GPU global memory access pattern which is, for Example 2, depicted in Fig. 2. Coloring of the blocks and the memory locations in this figure corresponds to the thread coloring in Fig. 1.

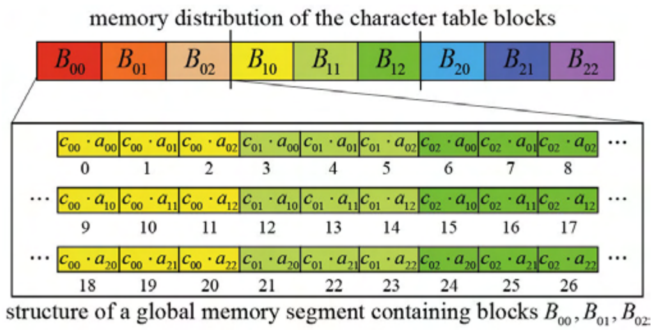

Figure 2. GPU global memory access pattern for Example 2.

In the general case, in the k^{th} step of the algorithm, we perform the Kronecker product of a ($p^k \times p^k$) matrix by the ($p \times p$) matrix, and the result is a ($p^{k+1} \times p^{k+1}$) matrix. Therefore, there are p^2 active threads in the first step of the algorithm, while in the k^{th} step, there are p^{2k} active threads. The index of the GPU memory location for the first entry ($c_{x,y} \cdot a_{0,0}$) of the block is determined as:

$$startElement \leftarrow x \cdot p^{k+2} + y \cdot p, \qquad (8)$$

The indices of the memory locations for the other elements ($c_{x,y} \cdot a_{i,j}$, i, j = 0, 1,…, p-1, except for the case $i = j = 0$)) in a block are:

$$nextElement \leftarrow startElement + i \cdot p^{k+1} + j \cdot p \ . \qquad (9)$$

C. *Features of the Mapping*

The proposed method for computing the character tables has the following features:

1. The character table is stored as a vector of length p^{2m} obtained by the concatenation of rows of [C_p^m]. This allows reading the values of characters directly without any reordering.
2. Elements of [C_p^m] computed by threads with the same first index and the successive second index are stored in neighboring memory locations. This automatically allows memory coalescing, due to which multiple data accesses to the GPU global memory are performed as a single memory transaction [2, 12].

D. *Algorithm Implementation*

A GPGPU program consists of two parts:

1. *Host program*, which executes on the CPU and creates and controls the context for the execution of kernels as well as allocates and transfers data to the GPU memory.
2. *Device program*, which is processed on the GPU and implements the SPMD kernels.

In the presented OpenCL implementation, the host program determines the character table for the cyclic subgroup C_p through (1). Notice that not all of the characters of C_p need to be computed by using (1), since, e.g., $e_{p-i} = e_i^*$, for i = 1, 2, …, $\lceil p/2 \rceil - 1$. Thus, we compute half of the rows of the character table for C_p, while other rows are determined by using this property.

The host allocates GPU global memory space for two ($p^m \times p^m$) matrices that are used as buffers to store the results of the application of the Kronecker product. This minimizes the communication between the host and the device, which is a bottleneck in the GPU computing [8, 12, 13]. Note that we have to reserve the space for ($p^m \times p^m$) matrices at the beginning of the computation, since the size of the GPU buffers cannot be changed after their creation, otherwise, we would have to create buffers and transfer data between the host and the device for each step of the algorithm, as the resulting intermediate matrices increase in size. To minimize the memory bandwidth occupation on the GPU itself, we use the technique of buffer swapping [7]. For odd-numbered steps, the first matrix is used as the input to the kernel and the second matrix as the output. For even-numbered steps, the order is reversed.

The character table for C_p is stored in a ($p \times p$) matrix and it is used as the second operand in the Kronecker product operation in each step. Since it is of a small size, we keep it in the constant GPU memory, which is cached. This allows much faster access and leads to improved program performance [12].

The Algorithm 1 presents a pseudo-code for the device program. Code in lines 2 and 6 implements (8) and (9), respectively. Since the characters of finite Abelian groups are complex numbers, elements of $\left[C_p^k\right]$, $\left[C_p\right]$, and $\left[C_p^{k+1}\right]$ are stored in the GPU buffers using the *float2* OpenCL vector data type [10]. The first component in the vector variable stores the real part and the second component the imaginary part of the complex number.

Algorithm 1 Pseudo-code for the device program

1: $x, y \leftarrow$ acquire thread indices in the two-dimensional grid
2: $startElement \leftarrow x \cdot p^{k+2} + y \cdot p$
3: $adr1 \leftarrow x \cdot p^k + y$
4: **for** i = 0 to p-1 do
5: **for** j = 0 to p-1 do
6: $nextElement \leftarrow startElement + i \cdot p^{k+1} + j$
7: $adr2 \leftarrow i \cdot p + j$
8: $\left[C_p^{k+1}\right](nextElement).\mathrm{re} \leftarrow \left[C_p^k\right](adr1).\mathrm{re} \cdot \left[C_p\right](adr2).\mathrm{re} - \left[C_p^k\right](adr1).\mathrm{im} \cdot \left[C_p\right](adr2).\mathrm{im}$
9: $\left[C_p^{k+1}\right](nextElement).\mathrm{im} \leftarrow \left[C_p^k\right](adr1).\mathrm{re} \cdot \left[C_p\right](adr2).\mathrm{im} + \left[C_p^k\right](adr1).\mathrm{im} \cdot \left[C_p\right](adr2).\mathrm{re}$

TABLE II SPECIFICATION OF TEST PLATFORMS

Platform	A	B
CPU	*AMD Phenom II N830 triple-core (2.1GHz)*	*Intel Core i7-920 quad-core (2.66GHz)*
RAM	*4GB DDR3 1066MHz*	*12GB DDR3-2000*
OS	*Windows 7 Ultimate (64-bit)*	
IDE	*MS Visual Studio 2010 Ultimate*	
SDK	*AMD APP 2.6*	*Nvidia GPU Computing 4.0*
GPU	*ATI Radeon 5650*	*Nvidia GTX 650 Ti*
engine speed	*650 MHz*	*900 MHz*
memory	*1 GB GDDR3 800 MHz*	*1 GB GDDR5 4.2 GHz*
processors	*80*	*384*

I. EXPERIMENTAL RESULTS

The experiments reported in this section are performed using two hardware platforms, labeled **A** and **B**, respectively, and specified in Table II. The GPU kernel performance analysis is done through the application of AMD APP Profiler 2.4 (for **A**) and Nvidia Parallel Nsigth 2.1 (for **B**), in accordance with instructions provided in [2, 12].

The referent C/C++ implementation uses the *complex* data type from the Standard Template Library (STL) for the representation of the values of group characters. This data structure best corresponds to the *float2* OpenCL vector data type [10] used for the same purpose in the GPU implementation. The referent C/C++ implementation is compiled for the *x64* platform using the MS C++ compiler set to the maximum level of performance-oriented optimizations.

The results of the experiments performed on both test platforms for the construction of the character table for the groups C_3^m , $m = 1, 2,\ldots, 8$, are presented in Fig. 3. Notice that for $p = 3$ and $m = 8$, the size of the character table is $p^m \times p^m = 3^8 \times 3^8 = 6561 \times 6561$, and, therefore, to complete the task, we have to compute and store 43 046 721 complex numbers. The OpenCL implementation processed on the GPUs outperforms the referent CPU C/C++ implementation on both platforms and for all values of m used in the experiments. The speed-up is almost constant throughout the range for m, and it goes up to a factor of $7.8\times$, on the test platform **A**, and up to a factor of $8.2\times$ on the platform **B**.

II. CONCLUSIONS

In this paper, we propose a method for the construction of characters of finite Abelian groups of the form $G = C_p^m = (\{0,1,\ldots,p-1\}^m, \oplus_p)$, using the graphics processing unit (GPU) as the computational platform. We identify the sources of the parallelism available in the algorithm for construction of the character table for G formulated in terms of the Kronecker product. Based on this analysis, we devise a mapping of the computations to the SPMD processing model of the GPU and develop an OpenCL implementation of the algorithm. The experimental results obtained through the comparison of the proposed solution and the referent C/C++ implementation of the same algorithm show speed-ups of up to $7.8\times$ and $8.2\times$, depending on the platform, when using the GPU and, thus, confirm the validity of the proposed approach.

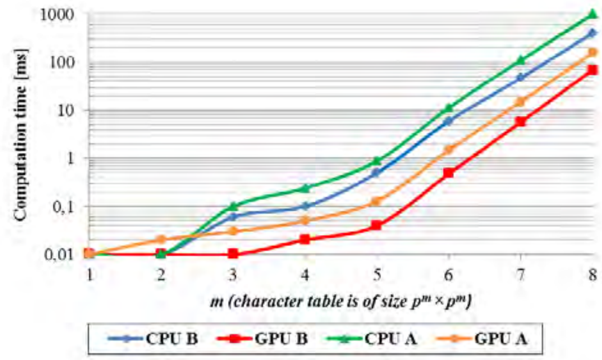

Figure 3. Computation times for the groups C_3^m , $m = 1, 2,\ldots, 8$, on CPUs and GPUs for the test platforms specified in Table II.

REFERENCES

[1] T. M. Aamodt, “Architecting graphics processors for non-graphics compute acceleration”, in *Proc. 2009 IEEE PacRim Conf. Comm., Comp. & Sig. Proc.*, Victoria, BC, Canada, 2009.

[2] AMD, “AMD Accelerated Parallel Processing OpenCL Programming Guide”, available from: *http://developer.amd.com /sdks/AMDAPPSDK*, [accessed 1 April 2012].

[3] T. Apostol, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, USA, 1976.

[4] M. Clausen, “Fast generalized Fourier transforms”, *Theoretical Computer Science*, No. 67, 1989, pp. 55-63.

[5] J. W. Cooley and J. W. Tukey, ”An algorithm for the machine calculation of complex Fourier series”, *Mathematics of Computation*, No. 90, 1965, pp. 297-301.

[6] D. S. Dummit and R. M. Foote, *Abstract Algebra*, John Wiley & Sons, 2003.

[7] D. B. Gajić, R. S. Stanković, "GPU accelerated computation of fast spectral transforms", *Facta Universitatis - Series: Electronics and Energetics*, Vol. 24, No. 3, University of Niš, Niš, Serbia, 2011, pp. 483-499.

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

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

[10] Khronos,”OpenCL Specification 1.2”, Khronos OpenCL Working Group, 2011.

[11] D. K. Maslen and D. N. Rockmore, “Generalized FFTs – A survey of some recent results”, in *DIMACS Workshop in Groups and Computation*, 1998, pp. 183-238.

[12] Nvidia, “OpenCL Best Practices Guide”, available from: *http://developer.nvidia.com/nvidia-gpu-computing-documentation*, [accessed 1 April 2012].

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

[14] C. C. Pinter, *A Book of Abstract Algebra*, Dover, 2010.

[15] W. Rudin, *Fourier Analysis on Groups*, Wiley, 1990.

[16] R. S. Stanković, and J. T. Astola, *Spectral Interpretation of Decision Diagrams*, Springer, New York City, USA, 2003.

[17] M. R. Stojić, M. S. Stanković, and R. S. Stanković, *Diskretne transformacije u primeni*, Nauka, Beograd, 1993, (in Serbian).

[18] M. A. Thornton, “Spectral transforms of mixed-radix MVL functions“, in *Proc. IEEE Int. Symp. on Multiple-Valued Logic (ISMVL)*, Tokyo, Japan, May, 2003, pp. 329-333.

[19] M. A. Thornton, R. Drechsler and D. M. Miller, *Spectral Techniques in VLSI CAD*, Kluwer Academic Publishers, 2001.