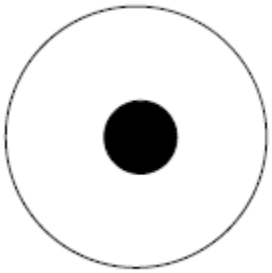
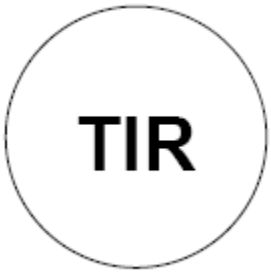




The IPSI BgD Transactions on Internet Research

Multi-, Inter-, and Trans-disciplinary Issues in Computer Science and Engineering

A publication of

IPSI Bgd Internet Research Society
New York, Frankfurt, Tokyo, Belgrade
July 2013 Volume 9 Number 2 (ISSN 1820-4503)

Guest Editor: Jakob Salom

Table of Contents:

Application of Maxeler Dataflow Supercomputing to Spherical Code Design
Stanojevic, Ivan; Senk, Vojin; Milutinovic, Veljko 1

Bitonic Merge Sort Implementation on the Maxeler Dataflow Supercomputing System
Rankovic, Vukasin; Kos, Anton; Milutinovic, Veljko 5

Implementation of the RSA Algorithm on a Dataflow Architecture
Bezanic, Nikola; Popovic-Bozovic, Jelena; Milutinovic, Veljko; Popovic, Ivan 11

Solving Gross Pitaevskii Equation Using Dataflow Paradigm
Stojanovic, Sasa; Bojic, Dragan; Milutinovic, Veljko 17

Feasibility Study on the SAT Solver on DataFlow Architecture
Sustran, Zivojin; Ognjanovic, Zoran; Todorovic, Milan; Milutinovic, Veljko 23

The Source-Sink Model
Ivkovic, Sinisa; Ilic, Luka; Stankovic, Marko; Radojicic, Radomir; Babovic Zoran 28

Accelerating Lattice-Boltzman Method Using the Maxeler DataFlow Approach
Korolija, Nenad; Djukic, Tijana; Milutinovic, Veljko; Filipovic Nenad 34

The IPSI BgD Internet Research Society

The Internet Research Society is an association of people with professional interest in the field of the Internet. All members will receive this TRANSACTIONS upon payment of the annual Society membership fee of €100 plus an annual subscription fee of €1000 (air mail printed matters delivery).

Member copies of Transactions are for personal use only

IPSI BGD TRANSACTIONS ON ADVANCED RESEARCH

www.internetjournals.net

STAFF		
Veljko Milutinovic, **Co-Editor-in-Chief**	**Jakob Salom,** **Co-Editor-in-Chief**	**Nemanja Trbovic,** **Journal Manager**
Department of Computer Engineering ETF University of Belgrade POB 35-54 Belgrade, Serbia Tel: (381) 64-1389281	Department of Computer Science Mathematical Institute of SANU University of Belgrade POB 367 Belgrade, Serbia Tel: (381) 64-8183030	Department of Computer Science IPSI Internet Research Society University of Belgrade POB 35-54 Belgrade, Serbia Tel: (381) 60-3191011
vm@eft.rs	**jakob.salom@yahoo.com**	**ntrbovic@gmail.com**

EDITORIAL BOARD		
Lipkovski, Aleksandar	**Gonzalez, Victor**	**Milligan, Charles**
The Faculty of Mathematics, Belgrade,	University of Oviedo, Gijon,	Sun Microsystems, Colorado
Serbia	Spain	USA
Blaisten-Barojas, Estela	**Janicic, Predrag**	**Kovacevic, Milos**
George Mason University, Fairfax, Virginia	The Faculty of Mathematics, Belgrade	School of Civil Engineering, Belgrade
USA	Serbia	Serbia
Crisp, Bob	**Jutla, Dawn**	**Neuhold, Erich**
University of Arkansas, Fayetteville, Arkansas	Sant Marry's University, Halifax	Research Studios Austria, Vienna
USA	Canada	Austria
Domenici, Andrea	**Karabeg, Dino**	**Piccardi, Massimo**
University of Pisa, Pisa	Oslo University, Oslo	Sydney University of Technology, Sydney
Italy	Norway	Australia
Flynn, Michael	**Kiong, Tan Kok**	**Radenkovic, Bozidar**
Stanford University, Palo Alto, California	National University of Singapore	Faculty of Organizational Sciences, Belgrade
USA	Singapore	Serbia
Fujii, Hironori	**Kovacevic, Branko**	**Rutledge, Chip**
Fujii Labs, M.I.T., Tokyo	School of Electrical Engineering, Belgrade	Purdue Discovery Park, Indiana
Japan	Serbia	USA
Ganascia, Jean-Luc	**Patricelli, Frederic**	**Mester, Gyula**
Paris University, Paris	ICTEK Worldwide L'Aquila	University of Szeged, Szeged
France	Italy	Hungary

Application of Maxeler Dataflow Supercomputing to Spherical Code Design

Ivan Stanojević, Vojin Šenk, and Veljko Milutinović

***Abstract*—An algorithm for spherical code design, based on the variable repulsion force method is presented. The iterative nature of the algorithm and the large number of operations it performs make it suitable for implementation on dataflow supercomputing devices. Gains in computation speed and power consumption of such an implementation are given. Achieved minimum distances of obtained codes are presented.**

I. Introduction

A spherical code is a set of N D-dimensional real vectors on the unit sphere. Two standard optimization problems are associated with spherical codes:

- given N and D, find a spherical code such that the minimum Euclidean distance between any two code vectors is maximized over the set of all such codes (packing problem);
- given N and D, find a spherical code such that the radius of equally sized balls centered at code vectors and whose union contains the unit sphere, is minimized over the set of all such codes (covering problem).

Here, only the first problem will be addressed. The design of spherical codes is both an interesting theoretical and practical problem. There are only a few cases in which the exact solution is known [5], [6]. Good spherical codes can be used both for source coding [7] and channel coding [8], [9].

Dataflow supercomputing implemented on Maxeler systems is a new computing paradigm which allows more time and energy efficient implementations of algorithms that are inherently parallelizable. The main idea is translation of code that performs an algorithm into a dataflow structure in which streams of data are exchanged between simple processing blocks of programmable logic. If the number of operations that can be performed at the same time is high, acceleration can be achieved by using space multiplexing in an FPGA circuit, for which the host machine only provides input data and collects its output.

II. Optimization Methods

A. Direct Optimization

The optimization problem corresponding to optimal spherical code design can be formulated as follows: find D-dimensional vectors $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N\}$ lying on the unit sphere, $|\mathbf{r}_i| = 1$, such that the cost function

$$U = -\min_{i \neq j} |\mathbf{r}_j - \mathbf{r}_i| \tag{1}$$

attains its minimum value. A usual procedure for finding the minimum of a function like this one is to create an initial set of vectors and iteratively adjust every one of them until a minimum is reached.

Although this optimization would produce a desired code, algorithms that perform it suffer from serious practical problems [10]. Since U only depends on the smallest distance between any two vectors, it locally depends only on the code vectors which have their nearest neighbours at the current minimum distance. If vectors are moved in the direction away from their nearest neighbours in every iteration, it is difficult to choose the length of those moves. Since the minimum distance should be increased at every move, when a vector has two or more neighbours at approximately the same minimum distance, the length of its move may be only very small, leading to extremely slow convergence.

Another problem is that U is not differentiable with respect to vector coordinates, so the gradient descent method or Newton's method cannot be applied directly to its minimization.

B. Variable Repulsion Force Method

A different approach, successfully used in the literature [1], [2], [3], [4], [5], [11], and in this work, is to choose the differentiable cost function (assuming no vectors overlap)

$$V = \sum_{i<j} \frac{C}{|\mathbf{r}_j - \mathbf{r}_i|^{\beta-2}}, \tag{2}$$

where $C > 0$ is an arbitrary constant and $\beta > 2$ is an adjustable parameter. V can be thought of as the potential of N particles repelling by the conservative central force (from the ith to the jth)

$$\mathbf{F}_{i \to j} \triangleq -\nabla_{\mathbf{r}_j} \left(\frac{C}{|\mathbf{r}_j - \mathbf{r}_i|^{\beta-2}} \right) = C(\beta - 2) \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^{\beta}}. \tag{3}$$

As β is increased, the force other particles exert to one particle starts to be dominated by the force of its nearest neighbours. If $\beta \to \infty$, the force equilibrium (the local stationary point of V) is attained at a position where any particle has two or more nearest neighbours at an equal distance which is as large as possible. The force exerted to the jth particle only has the component normal to the surface of the unit sphere, i.e. collinear with $\mathbf{r}_j$.

Iterative minimization of V usually produces only a local minimum, and there is no guarantee that it is also a global minimum. In order to obtain results as good as possible, the procedure must be repeated multiple times with different initial vectors, which can be chosen at random. Since it is desirable that all the randomly chosen points on the unit sphere be equally probable, i.e. that no directions be privileged, a simple two step procedure is used [10]:

Ivan Stanojević (cet_ivan@uns.ac.rs) and Vojin Šenk (vojin_senk@uns.ac.rs) are on the Faculty of Engineering (Technical Sciences), University of Novi Sad, Serbia.

Veljko Milutinović (vm@etf.rs) is on the Faculty of Electrical Engineering, University of Belgrade, Serbia.

1) choose coordinates $g^*_{i,1}, \ldots, g^*_{i,D}$ as independent samples of a normalized Gaussian random variable ($\mathcal{N}(0,1)$);
2) normalize the obtained vector,

$$\mathbf{r}_i = \frac{\mathbf{g}^*_i}{|\mathbf{g}^*_i|}. \tag{4}$$

For a fixed value of β, V can be minimized using the gradient descent method or Newton's method. Even though Newton's method has a faster rate of convergence, the gradient descent method is chosen here for the following reasons:

- The overall solution of the problem is not the minimum of V for a particular value of β, but for β as high as possible, so β must be gradually increased after every iteration. The number of iterations which produce the minimum for a fixed β is irrelevant, since after β is increased the minimization is restarted (with a better initial position).
- Gradient descent is much simpler and a single iteration is much faster.

The negative partial gradient of V with respect to the jth vector is

$$-\nabla_{\mathbf{r}_j} V = \sum_{i \in \{1,\ldots,N\} \setminus \{j\}} \mathbf{F}_{i \to j} \stackrel{\triangle}{=} \mathbf{F}_{\to j}, \tag{5}$$

which is the total force exerted to the jth particle. In order to preserve the constraint that all vectors are on the unit sphere, their new values can be calculated by moving them along the directions of the corresponding total forces and normalizing them,

$$\mathbf{r}^*_j = \mathbf{r}_j + \alpha \mathbf{F}_{\to j}, \tag{6}$$

$$\mathbf{r}_j := \frac{\mathbf{r}^*_j}{|\mathbf{r}^*_j|}. \tag{7}$$

The constant $\alpha > 0$ determines the speed of the procedure and should be chosen so that the procedure is stable (vectors converge) and the error (the difference between the current and the final values) diminishes as fast as possible.

Although α could be optimized to achieve the previous goals, when β is high enough the iteration in (6) and (7) suffers from numerical difficulties in a finite precision implementation:

- if for some j, $|\mathbf{r}_j - \mathbf{r}_i| > 1$ for all $i \neq j$, $|\mathbf{r}_j - \mathbf{r}_i|^{-\beta}$ can be calculated as 0 in (3) due to numerical underflow and $\mathbf{F}_{\to j}$ can be calculated as 0 as well;
- if for some i and j, $|\mathbf{r}_j - \mathbf{r}_i| < 1$, the calculation of $|\mathbf{r}_j - \mathbf{r}_i|^{-\beta}$ in (3) can cause numerical overflow.

The likelihood of these conditions increases as β increases. In order to avoid these problems, a slight modification of the procedure will be used. Since every vector is moved along the direction of the corresponding force, that direction can be calculated by first scaling the force and then normalizing it. A convenient way to scale the force is

$$\mathbf{F}^*_{\to j} = \sum_{i \in \{1,\ldots,N\} \setminus \{j\}} \left(\frac{\delta_j}{|\mathbf{r}_j - \mathbf{r}_i|^2} \right)^{\beta/2} (\mathbf{r}_j - \mathbf{r}_i), \tag{8}$$

where

$$\delta_j = \min_{i \in \{1,\ldots,N\} \setminus \{j\}} |\mathbf{r}_j - \mathbf{r}_i|^2. \tag{9}$$

The value of the subexpression in the first parentheses in (8) is in $(0, 1]$ for all i, and is equal to 1 for at least one term, so neither underflow nor overflow can occur in the calculation of (8).

The normalized force

$$\mathbf{F}_{\to j} = \frac{\mathbf{F}^*_{\to j}}{|\mathbf{F}^*_{\to j}|} \tag{10}$$

is used in a modified version of (6),

$$\mathbf{r}^*_j = \mathbf{r}_j + \alpha_j \mathbf{F}_{\to j}, \tag{11}$$

where

$$\alpha_j = \frac{\delta_j}{2\beta}. \tag{12}$$

It can be shown that this value of the step size, α_j, ensures the stability of the process and fast convergence.

C. Force loosening

In order to maximize the minimum distance, β should be as high as possible. If vectors are moved with a high β immediately from their random initial position, the convergence is slow and the obtained local minimum of V usually produces lower minimum distance than if β is gradually increased (the force gradually loosened). Given $\mathbf{r}_1, \ldots, \mathbf{r}_N$ and β, the total move norm of one iteration,

$$S = \sum_j |\mathbf{r}'_j - \mathbf{r}_j|^2, \tag{13}$$

can be calculated ($\mathbf{r}'_1, \ldots, \mathbf{r}'_N$ are new vectors). It is observed in numerical results that for a continuous chain of iterations from a starting set of vectors and for fixed β, S decreases asymptotically exponentially with the number of iterations. It is also observed that if β is increased by the same amount after each iteration (linearly), after a transient period S starts decreasing as $S \sim \beta^{-1}$. In order to reach high values of β quickly, yet without forcing the vectors into a position of numerical deadlock (which happens whenever β is increased too rapidly), the following strategy for increasing it is used:

1) Set $n := 1$ and $l := 2$.
2) Starting from $\mathbf{r}_1, \ldots, \mathbf{r}_N$ and using $\beta = 2^l$, calculate new vectors $\mathbf{r}'_1, \ldots, \mathbf{r}'_N$ and the corresponding total move norm S'.
3) Starting from $\mathbf{r}_1, \ldots, \mathbf{r}_N$ and using $\beta = 2^{l(1+1/n)}$, calculate new vectors $\mathbf{r}''_1, \ldots, \mathbf{r}''_N$ and the corresponding total move norm S''.
4) If $S' > S''$, set $\mathbf{r}_j := \mathbf{r}'_j$ for all j.
   Otherwise, set $\mathbf{r}_j := \mathbf{r}''_j$ for all j and set $l := l(1+1/n)$.
5) If more iterations are necessary, set $n := n + 1$ and go to step 2.

III. Implementation

The majority of operations in the iterative procedure is performed in the loop for calculating new vectors. This is the part in which a number of operations can be done at the same time and can thus benefit from hardware acceleration.

For convenience, an overview of steps performed for a single vector is given here:

1) $\delta_j = \min_{i \in \{1,\ldots,N\} \setminus \{j\}} |\mathbf{r}_j - \mathbf{r}_i|^2$.
2) $\mathbf{F}^*_{\to j} = \sum_{i \in \{1,\ldots,N\} \setminus \{j\}} \left(\frac{\delta_j}{|\mathbf{r}_j - \mathbf{r}_i|^2} \right)^{\beta/2} (\mathbf{r}_j - \mathbf{r}_i)$.
3) $\mathbf{F}_{\to j} = \frac{\mathbf{F}^*_{\to j}}{|\mathbf{F}^*_{\to j}|}$.
4) $\mathbf{r}^*_j = \mathbf{r}_j + \frac{\delta_j}{2\beta} \mathbf{F}_{\to j}$.

5) $\mathbf{r}_j := \frac{\mathbf{r}_j^*}{|\mathbf{r}_j^*|}$.

The new vector, calculated in the last step can be stored:

- in the same memory location as the old one, which is more convenient for a software implementation, since only one block of memory is used;
- in the appropriate location of a different block of memory or another data stream, which is more convenient for a streamed hardware implementation, since it simplifies dependencies on input data.

Although the new values obtained in these ways differ, they can both be successfully used for finding good spherical codes.

A. Software Implementation

In the software implementation, vector coordinates are stored as floating-point numbers. Operations with them are performed by the floating-point unit (FPU) of the central processor (CPU) and by the library functions, which are highly optimized. Their data type is the standard IEEE 754 double precision type [12], with 53 bits of mantissa and 11 bits of exponent.

Since vector coordinates and vectors themselves are accessed sequentially in all the operations performed on them, the algorithm has good cache locality. Typically, for codes used in practice, values N and D are such that the total amount of iterated data is small enough to fit completely in the cache of modern processors.

B. Hardware Implementation

In the hardware implementation, vector coordinates are stored as signed fixed-point numbers with 3 integer bits and 44 fractional bits. Maxeler dataflow engines support highly configurable number formats and this choice is the result of the following limitations:

- Vector coordinates are in the interval $[-1, 1]$, their differences are in $[-2, 2]$ and their squares are in $[0, 4]$, so at least 3 integer bits are necessary.
- Some library functions (e.g. for calculating square roots) have an upper bound on the total number of bits of fixed-point numbers they can operate on. Since other fixed-point types with a higher number of integer bits are used for storing intermediate results, a total of 47 bits is obtained as the highest value for which all the calculations can be performed using only library functions.

Although floating-point numbers can be used as well, fixed point numbers occupy much less hardware resources.

Operations on vectors, such as addition, subtraction, multiplication by a scalar or norm calculation are performed in parallel on all their coordinates.

In the algorithm above, δ_j, which is calculated in step 1, is used in steps 2-5. In the software implementation two separate inner loops over i are necessary in steps 1 and 2. Whereas this is also the case in the hardware implementation, another level of parallelism is possible: while δ_j is being used in steps 2-5, δ_{j+1} can be calculated, as shown in Figure 1. Since one inner loop (for calculating a single value of δ_j or $\mathbf{r}_j$) lasts N cycles, one complete iteration lasts $N(N+1)$ cycles.

The maximum number of vectors, $N_{\max}$, and their maximum dimension, $D_{\max}$, must be hard coded in the dataflow structure. In order to provide a flexible solution, the actual number and dimension of vectors, N and D, are configured as run-time parameters (scalar inputs). On a Maxeler MAX2 device,

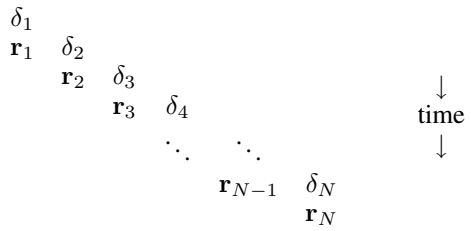


Fig. 1. Order of calculation: dependent values are in the same columns, and independent values calculated at the same time are in the same rows

TABLE I
MAX2 FPGA RESOURCE USAGE

	Used	Maximum	Percentage
LUTs	157716	207360	76.06%
FFs	193082	207360	93.11%
BRAMs	225	324	69.44%
DSPs	141	192	73.44%

possible values are $N_{\max} = 16384$ and $D_{\max} = 6$. These values are obtained by fine tuning the design, since FPGA circuits have a limited number of programmable logic elements: look-up tables (LUTs), flip-flops (FFs), block RAM modules (BRAMs) and digital signal processing modules (DSPs). The actual resource usage is shown in Table I.

The architecture of the system is shown in Figure 2. The host application is responsible for creating the initial vectors, sending the current vectors in each iteration to the dataflow device and collecting their updated values. Vector coordinates are transferred serially from the host and back to the host. On the dataflow device, they are internally converted to the parallel form in the serial-to-parallel (S/P) converter and after the update back to the serial form in the parallel-to-serial (P/S) converter. Both conversions last ND cycles, but since outputs of each block are available before its execution ends, the total execution time of one iteration is less than $2ND + N(N+1)$ cycles.

C. Performance

Measured execution times of a single iteration are shown in Figure 3. The host CPU is Intel i7-3770K@3.5GHz and the software implementation is single-threaded. During the measurements, all functions of the CPU which change its clock frequency depending on the load are disabled.

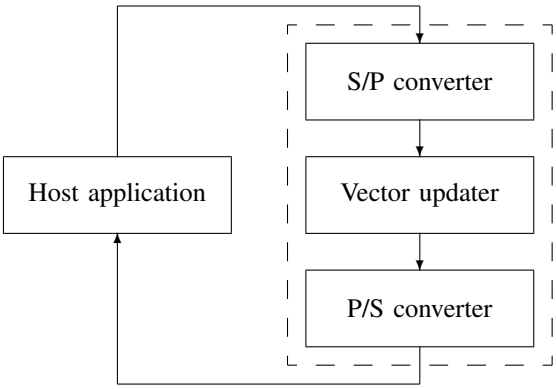


Fig. 2. Architecture of the system: blocks inside the dashed box are on the Maxeler MAX2 dataflow device

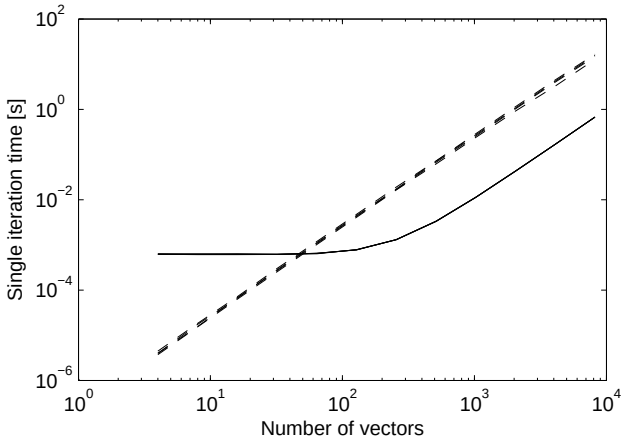


Fig. 3. Execution times, $N \in [4, 8192]$ and $D \in [3, 6]$: results for the software implementation are shown in dashed lines and for the hardware implementation in solid lines

TABLE II
HOST POWER CONSUMPTION

Mode	Power [W]
Idle	68
Software execution	88
Hardware execution	74

For low values of N, the software implementation is considerably faster than the hardware one, since in the latter case the majority of time is spent on the communication between the host and the dataflow device.

For $N \gtrsim 50$, the hardware implementation becomes faster than the software one despite all the overhead of communication and control the host needs to perform.

For $N \gtrsim 500$, the execution time of the hardware implementation starts to show similar asymptotic behaviour as that of the software one, since it is dominated by the actual calculations. Both procedures are essentially the same and quadratic in N.

It is interesting to note that the time of execution in software increases as the vector dimension, D, increases, but is not proportional to it. This can be explained by the time needed for other operations apart from vector operations. In hardware, the times for different values of D are practically the same (their curves in Figure 3 almost overlap) since vector operations are done in parallel. Their only small differences are caused by different amounts of data exchanged between the host and the dataflow engine, but those are within measurement errors.

The asymptotic speed gains of hardware acceleration are approximately $(18 \div 24)\times$, for $D \in [3, 6]$.

Power consumption of the host in different execution modes is shown in Table II. The measurements are performed on the mains cable of the host, with estimated relative error of 5%.

Although the majority of power is used for idle operation of the host, which includes power for its auxiliary devices such as hard disks and fans, additional consumption for hardware execution is lower than for software execution, even when only a single thread is active.

IV. RESULTS

Minimum distances of spherical codes obtained by taking the best of 32 optimizer runs for different N and D are shown in Figure 4. These results are not the absolute optimum values (the reader is referred to online tables compiled by Sloane [13] for a detailed list of the best known and other interesting spherical codes), but can serve as a starting point for calculating possible coding gains in systems which employ them.

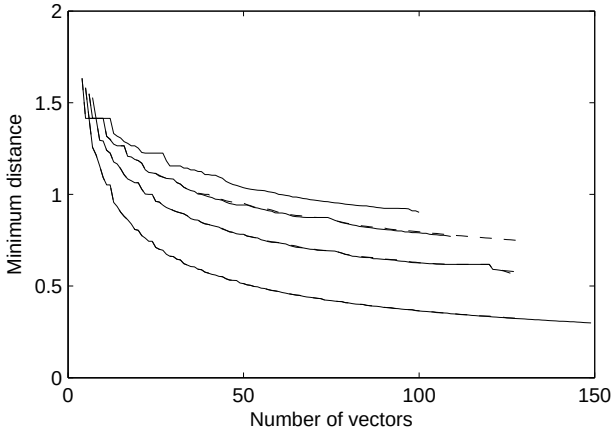


Fig. 4. Minimum distances of spherical codes: achieved minimum distances of obtained codes for $D = 3, 4, 5, 6$ (from bottom to top) are shown in solid lines and minimum distances of the best known spherical codes for $D = 3, 4, 5$ (from Sloane's tables) are shown in dashed lines

ACKNOWLEDGEMENT

This work was supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, grant 451-03-00605/2012-16/198, project "Cloud Services for Applications with High Performance Requirements."

REFERENCES

[1] D. E. Lazić, "Class of Block Codes for the Gaussian Channel," *IET Electronics Letters*, volume 16, issue 5, pp. 185-186, February 1980.
[2] D. E. Lazić, D. P. Drajić, V. Šenk, "A Table of Some Small-Size Three-Dimensional Best Spherical Codes," *Proceedings of the IEEE International Symposium on Information Theory*, Les Arcs, France, 1982.
[3] D. E. Lazić, V. Šenk, I. Šeškar, "Arrangements of Points on a Sphere Which Maximize the Least Distance," *Bulletins for Applied Mathematics* (Technical University of Budapest), volume 47, No. 479/87, pp. 7-21, 1987.
[4] D. E. Lazić, V. Šenk, "A New Packing of 19 Equal Circles on a Sphere," *Bulletins for Applied Mathematics* (Technical University of Budapest), volume 57, No. 709/91, pp. 93-97, 1991.
[5] B. W. Clare, D. L. Kepert, "The Optimal Packing of Circles on a Sphere," *Journal of Mathematical Chemistry*, volume 6, pp. 325-349, 1991.
[6] P. G. Adams, "A Numerical Approach to Tamme's Problem in Euclidean n-space," M.Sc. thesis, Oregon State University, June 1997.
[7] J. Hamkins, K. Zeger, "Gaussian Source Coding With Spherical Codes," *IEEE Transactions on Information Theory*, volume 48, No. 11, pp. 2980-2989, November 2002.
[8] N. J. A. Sloane, "Tables of Sphere Packings and Spherical Codes," *IEEE Transactions on Information Theory*, volume 27, No. 3, pp. 327-338, May 1981.
[9] J. Gao, L. Rudolph, C. Hartmann, "Iteratively Maximum Likelihood Decodable Spherical Codes and a Method for Their Construction," *IEEE Transactions on Information Theory*, volume 34, No. 3, pp. 480-485, May 1988.
[10] K. J. Nurmela, "Constructing Spherical Codes by Global Optimization Methods," Helsinki University of Technology, Digital Systems Laboratory, Technical Report No. 32, February 1995.
[11] D. A. Kottwitz, "The Densest Packing of Equal Circles on a Sphere," *Acta Crystallographica*, section A47 pp. 158-165, 1991.
[12] http://en.wikipedia.org/wiki/Binary64
[13] http://neilsloane.com/packings/index.html

Bitonic Merge Sort Implementation on the Maxeler Dataflow Supercomputing System

Ranković, Vukašin; Kos, Anton; and Milutinović, Veljko

Abstract — ***Sorting is extensively used in many applications. Predominantly, comparison based sequential sorting algorithms are used. High speed computing is also driving the quest for ever faster sorting. Sorting networks executing parallel sorting and dataflow computational paradigm are offered as a possible solution. Sorting networks are briefly explained, Details are given for the Bitonic merge sort algorithm, which we used in our experiments. Sorting was implemented on an entry model of the Maxeler dataflow supercomputing systems. We employed different testing scenarios. Results show, that for a small size array of 128 numbers, the speedup, comparing to the fastest sorting algorithm on a CPU, achieves factor of 16. Moving to more advanced Maxeler systems, we expect to be able to sort larger arrays and achieve much greater speedups.***

Index Terms — ***bitonic merge sort, dataflow computing, number sort, parallel sorting, sorting algorithm acceleration, sorting network.***

1. INTRODUCTION

SORTING is one of the most important algorithms in computer systems. According to [1] it is estimated that sorting accounts to about 25 percent of the running time on computers, and in some systems even to about 50 percent. From these data we can conclude that either there are many applications that use sorting, or sorting is used even when not necessary, or inefficient sorting algorithms are in common use, or most probably all of the above. Therefore the constant quest for better sorting algorithms, their proper use, and their practical implementations is necessary.

Many times sorting is inherent to applications and at the same time concealed from users, who are not even aware that their actions use sorting algorithms to produce results. One of many such examples is searching for information on the Internet. Most common, search algorithms work with sorted data and the natural way of presenting search results is an ordered list of items matching the search criteria.

Manuscript received April 29, 2013.

Vukašin Ranković is with the School of Electrical Engineering, University of Belgrade, Serbia (e-mail: rvukasin@gmail.com).

Anton Kos is with the Faculty of Electrical Engineering, University of Ljubljana, Slovenia (e-mail: anton.kos@fe.uni-lj.si).

Veljko Milutinović is with the School of Electrical Engineering, University of Belgrade, Serbia (e-mail: vm@etf.rs).

The majority of applications and systems use comparison based sequential sorting. It has been proven [1] that such sorting algorithms require at least the time proportional to $O(N \cdot \log_2 N)$, where *N* is the number of items to be sorted. Speedups are possible with parallel processing and the use of parallel sorting algorithms. Parallelization is commonly achieved through the use of *multi-core* or *many-core* systems that can produce speedups roughly proportional to the number of cores. Recently a new paradigm called *Dataflow Computing* re-emerged and it is being successfully used in many computationally intensive fields. Dataflow computing offers immense parallelism by utilizing thousands of tiny simple computational cores, improving the performance by orders of magnitude.

Our work focuses on implementing parallel sorting algorithms on Maxeler Dataflow Supercomputing Systems. For large number of items to be sorted, we expect speedups in order of magnitudes.

This paper is organized as follows. In section 2 the motivation for re-emerged interest for the sorting networks is presented. Followed by sections 3 and 4 that give a brief explanation of sorting networks, in particular the chosen Bitonic merge sorting network. Section 5 illustrates the implementation of the above on a Maxeler dataflow supercomputing systems. Sections 6 and 7 explain testing scenarios and results. We conclude with section8.

2. MOTIVATION

The need for ever faster communication networks, exponentially growing volume of data being produced, high speed computing, and other developments are driving the search for more efficient, faster and if possible parallelized computer algorithms that would be able to satisfy these demands.

Many of the sequential sorting algorithms also have their parallel version. An efficient parallelization of a sorting algorithm by using many-core (up to a few tens) and multi-core (up to a few hundred) processors and systems is not really possible. For a true parallel sorting, such a system would need the number of cores in the order of number of items to be sorted, and this number can grow into thousands or millions.

Sorting algorithms are computationally undemanding in a sense that the computational operations are simple, mostly just comparisons between two items. Therefore dataflow computing should be a perfect match for parallel sorting algorithms by executing many thousands of operations in parallel, each of them inside of a simple computational core provided by the Maxeler Dataflow Supercomputing Systems.

Power consumption is also worth considering. Maxeler Dataflow Supercomputing Systems are using most of the energy for the processing of data, while traditional control flow processors use most of the energy for the process control.

By implementing sorting algorithms on a Maxeler Dataflow Supercomputing System, we expect considerable acceleration in time needed to sort the sequence of numbers, and at the same time save considerable amount of energy.

3. Sorting Networks

Maxeler Dataflow Supercomputing Systems are essentially programmable hardware accelerators. They are typically programmed once and the "burned-in code" is executed many times. The execution of algorithms running on them must be highly independent of data or intermediate computational results.

Therefore all comparison based parallel sorting algorithms running on such systems should be non-data-dependent and non-adaptive. This means that the sequence of comparisons executed by the sorting algorithm does not depend on or adapt to data, but only on the number of inputs [3].

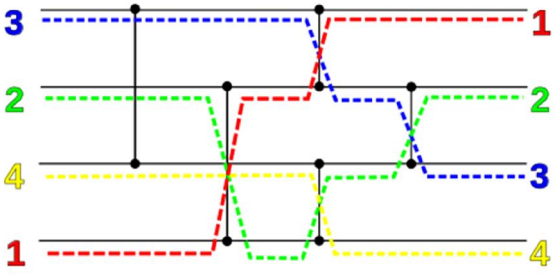


Figure 1: An example of a simple sorting network with four wires and five comparators. Each comparator connects two wires and emits higher value to the bottom wire and lower value to the top wire. Two comparators on the left hand side and two comparators in the middle can work in parallel. Parallel operation of this sorting network sorts the input numbers in three steps.
(Source: http://en.wikipedia.org/wiki/Sorting_network)

All of the above demands are met by sorting networks, a structure of wires and comparators, connected in a way to perform the function of sorting the values on their inputs. Figure 1 shows a simple sorting network and its operation. Reader can find more on the sorting networks in [1]-[4].

Efficiency of a sorting network is defined by two properties: depth and number of comparators. The depth is defined as the maximum number of comparators along any path from an input to an output. Efficient and simple sorting networks exist that have a depth proportional to $O((\log_2 N)^2)$ and the number of comparators proportional to $O(N\,(\log_2 N)^2)$ [3].

Assuming that all the comparisons on each level of the sorting network are done in parallel, the depth of the network defines the number of steps needed to sort all the N numbers on the inputs and thus defines the complexity of a sorting network. This is for the factor of $N/\log_2 N$ better from any of the comparison based sequential sort algorithms.

4. Bitonic Merge Sort

Bitonic sorting network is one of the fastest comparison sorting networks. It has been shown in [1] that its depth is

$$D(N) = \frac{\log_2 N \cdot (\log_2 N + 1)}{2} \qquad (1)$$

and the number of comparators that it employs is

$$C(N) = \frac{N \cdot \log_2 N \cdot (\log_2 N + 1)}{4} \qquad (2)$$

this conforms to the values given in the previous section.

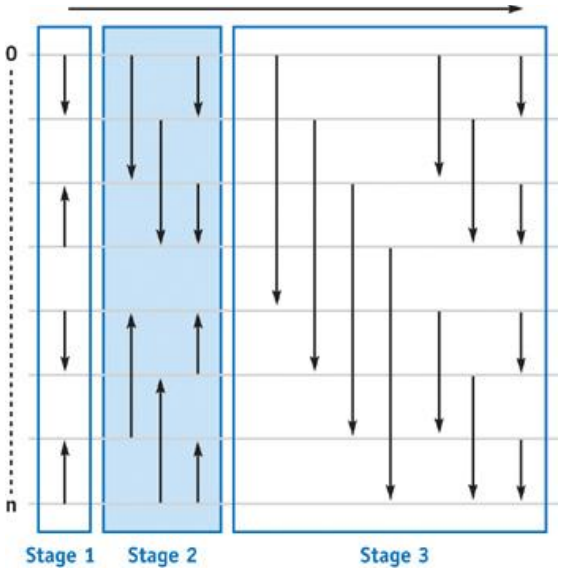


Figure 2: A Bitonic merge sort network with eight inputs (N=8). It operates in 3 stages, it has a depth of 6 steps and employs 24 comparators - see the equations (1) and (2). Arrows represent comparators that sort inputs in the arrow direction.
(Source: http://http.developer.nvidia.com)

Bitonic sorting network works on a *divide and conquer* principle. First it divides all the inputs into pairs, and sorts each pair into a bitonic sequence. A bitonic sequence has the property:

$x_1 \leq \cdots \leq x_k \geq \cdots \geq x_N$ for some k within $1 \leq k < N$. Next it merges adjacent bitonic sequences and repeats the process through all stages until the entire sequence is sorted. Figure 2 below depicts an implementation of a simple bitonic merge sorter for$N = 8$. Details on this process and merg bitonic merge sort algorithm can be found in [1]-[6].

5. IMPLEMENTATION

We implemented a Bitonic Merge Sorting Network on a simple MAX2 PCI Express card version of a Maxeler Dataflow Supercomputing System. The MAX2 card is inserted into an Intel based workstation with an Intel Core2 Quad processor running at a clock rate of 2.66 GHz. The card is connected to the PCI Express x16 slot on the workstation's motherboard. MAX2 card is equipped with a XILINX Virtex-5 FPGA device running at a clock rate of 200 MHz.

As follows from the sections 3 and 4, Bitonic Merge Sorting Network is an acyclic graph. It fits perfectly to the Dataflow computing paradigm and it is suitable for the implementation on FPGA hardware accelerators such as our MAX2 card.

To keep figure sizes inside the margins of this paper, we present the implementation of a Bitonic Merge Sorting Network through an example with four inputs ($N = 4$).

The application development process for the MAX2 card and other Maxeler Dataflow Supercomputing Systems is explained in [10] and [11]; here we only give explanation of the kernel. The kernel is a part of the code that is compiled and burned onto an FPGA and it actually executes the sorting on a MAX2 card. The kernel code was generated using a dedicated C++ script. It takes N as an input parameter and

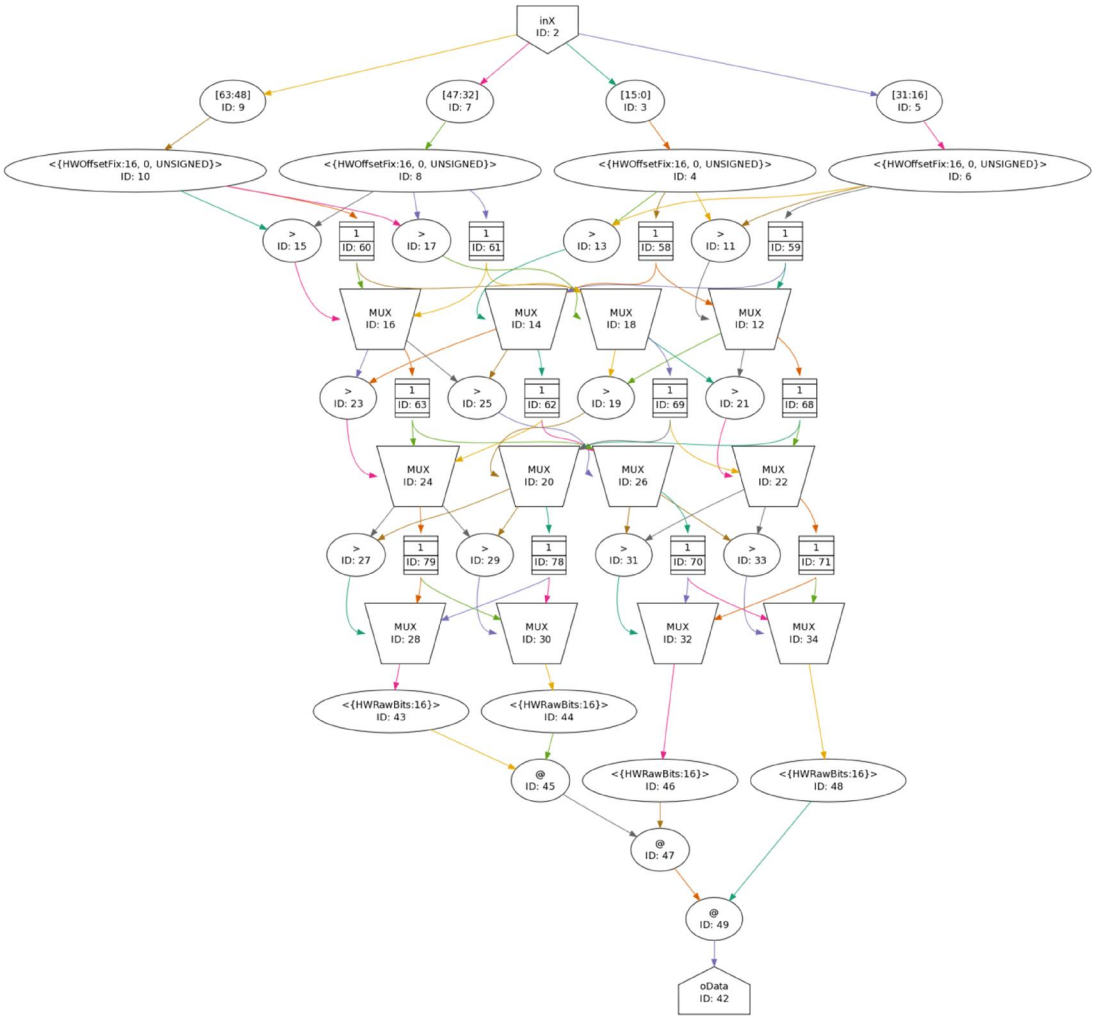


Figure 3: Maxeler graph of the Bitonic Merge Sorting Network at $N = 4$. An array of numbers from the host to be sorted (inX) is represented by elements with IDs: 4, 6, 8, and 10. The array enters the sorting network at the top of the graph. Two pairs of array elements on the first stage are compared and sorted by two comparators. One comparator consists of two comparison elements and two multiplexers. For instance, one of the comparators is represented by elements with IDs 14 to 17. Sorting is done by three layers of comparators. All elements on the same horizontal level are executed in parallel. At the bottom of the graph the sorted array (oData) is sent back to the host.

generates the code representing the appropriate network structure that is later compiled for Maxeler systems. Java source code defining our sorting network structure is listed below. MaxCompiler ternary-if operators (?:) are employed for comparisons.

```
for (int j = 0; j <4 /2; j++){
  for (int i = 1; i <= 2 / 2; i++){
    y[i+2*j-1] = x[i+2*j-1] > x[2*(j+1) + 1 - i-1] ?
    x[i+2*j-1] : x[2*(j+1) + 1 - i-1];
    y[2*(j+1) + 1 - i-1] = x[i+2*j-1] >
    x[2*(j+1) + 1 - i-1] ? x[2*(j+1) + 1 - i-1]:
    x[i+2*j-1];
  }
}
for (int j = 0; j <4 /4; j++){
  for (int i = 1; i <= 4 / 2; i++){
    x[i+4*j-1] = y[i+4*j-1] > y[4*(j+1) + 1 - i-1] ?
    y[i+4*j-1] : y[4*(j+1) + 1 - i-1];
    x[4*(j+1) + 1 - i-1] = y[i+4*j-1] >
    y[4*(j+1) + 1 - i-1] ? y[4*(j+1) + 1 - i-1 :
    y[i+4*j-1];
  }
}
for (int j = 0; j <4 /2; j++){
  for (int k = 1; k <= 2 / 2; k++){
    y[k+2*j-1] = x[k+2*j-1] > x[1+k + j*2-1] ?
    x[k+2*j-1] : x[1+k + j*2-1];
    y[1+k + j*2-1] = x[k+2*j-1] > x[1+k + j*2-1] ?
    x[1+k + j*2-1] :x[k+2*j-1];
  }
}
```

In general Bitonic Merge Sorting Network can be divided into $D(N)$ layers (1), in our case the number of layers is three. In the above code each layer is defined by two nested *for* loops. MaxCompiler is able to make a graph out of that code and to understand that each layer can be executed in one FPGA clock cycle. Generated graph of our sorting network is depicted and explained in **Error! Reference source not found.**. It can be seen that it performs $N/2$ independent comparisons per layer that are executed in parallel. Given the complexity of the best sequential sorting algorithm is $O(N \cdot \log_2 N)$ and knowing the complexity of the Bitonic sorting network working in parallel is the same as its depth $D(N)$ (1) gives us the theoretical speedup in the order of $N/\log_2 N$.

For sending data to Maxeler card we used MaxJava's build-in type called KArray which is a structure that has information about every element of the array. This structure is the most optimized for our problem because we can physically connect input array to the Sorting Network.

There are also some constraints that have to be addressed. Problem with sorting network implementation is that it takes a lot of comparators to build and because of that, the size of the FPGA chip can became too small. Another constraint is that some amount of time is needed to send whole number array to be sorted to the MAX card. In most cases sorting is just a part of a bigger application and we may assume that numbers are already on the MAX card, what lessens this problem. To save space on the FPGA and consequently being able to construct larger sorting networks for longer arrays, we limited the values of numbers to be sorted to 16 bits.

6. SETTING THE EXPERIMENT

Applications typically use sorting as a part of a broader process. Sorting can be used once in a while, regularly or continuously. To cover some of the above uses we have tried two scenarios. First scenario is to sort array of numbers one by one by starting the Maxeler card based sorting every time we need to sort an array. The second scenario is to start the Maxeler card and then continuously send many arrays for sorting.

The goal of the experiment is to establish possible acceleration for sorting on a Maxeler system comparing to the sorting on a CPU. We define the following variables and functions:

- N is the size of array to be sorted,
- M is the number of arrays to sort,
- $T_H(N)$ is the time needed to sort one array on the host,
- T_0 is the time needed to start a Maxeler card,
- $T_C(N)$ is the sum of times needed to send one array to the Maxeler card and receive back the sorted result,
- $T_S(N)$ is the time a sorting network on a Maxeler card needs to sort one array,
- $T_M(N)$ is the total time needed to sort one array on the Maxeler system.

The acceleration or speedup S is given by the ratio between the time needed for sorting on a CPU and time needed for sorting on a Maxeler system

$$S = \frac{T_H(N)}{T_M(N)}$$

In the first scenario we must start the Maxeler card, send one array, sort it and then receive the results back to the host. The total sorting time is defined as $T_M(N) = T_0 + T_C(N) + T_S(N)$ and the speedup is given as:

$$S_1 = \frac{T_H(N)}{T_0 + T_C(N) + T_S(N)} \quad (3)$$

In the second scenario we consecutively sort many arrays. We start the Maxeler card, send I arrays one after another, sort them, and receive the results back to the host. The total sorting time is defined as $T_M(N) = T_0 + M \cdot T_C(N) + M \cdot T_S(N)$ and the speedup is given as:

$$S_2 = \frac{M \cdot T_H(N)}{T_0 + M \cdot \max(T_C, T_S) + \min(T_C, T_S)} \qquad (4)$$

In the first scenario the speed-up is highly dependent on each of the denominator factors. Given that for small to medium values of N, the starting time T_0 for the Maxeler card could be relatively high comparing to the times $T_C(N)$ and $T_S(N)$, we can not expect important speedups.

In the second scenario the speedup depends mostly on M and the ratios between $T_H(N)$, $T_C(N)$ and $T_S(N)$. With the groving M the time T_0 becomes negligible. Given that the theoretical ratio between $T_H(N)$ and $T_S(N)$ is in the order of $N/\log_2 N$, the second scenario is expected to yield significant speedups.

7. RESULTS

Because of constraint stated before (size of FPGA chip) we were only able to implement Sorting Network up to the size of $N = 128$, what is conformant to the other works on the Sorting Networks on similar FPGA chips [8].

Speedup results for the first scenario, sending only one array for sorting at a time, are given in Figure 4. They are not particularly favorable as the CPU performs better than Maxeler. The solution would be in increasing array size beyond $N = 128$ what would be achievable on a better Maxeler system which was not at our reach at the time of research.

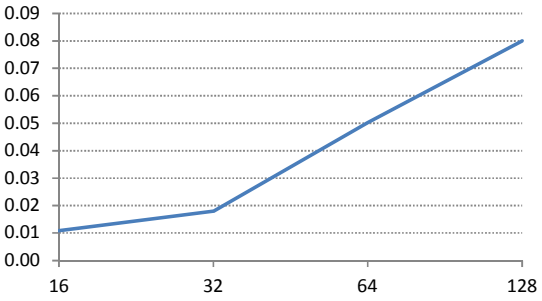


Figure 4: The achieved speedup S_1 (3) in the first scenario for array sizes from $N = 16$ to $N = 128$. We can observe that S_1 grows exponentially with N. Note that S_1 is much smaller than 1, meaning that sorting one array at a time on a Maxeler using bitonic merge sorting is slower than sorting it on a CPU using quicksort.

Results for the second scenario are given in Figure 5 shows how the sorting time depends on the number of 128 number arrays ($N = 128$) that are consecutively sent to be sorted. It demonstrates quite strong influence of the Maxeler card staring time T_0. Only if we send more than 20 arrays, we achieve a moderate speedup.

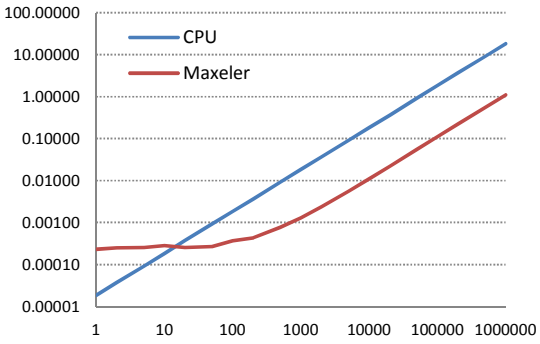


Figure 5: Sorting times in seconds for sorting of the different number of consecutive arrays of 128 numbers ($N = 128$) from $M = 1$ to $M = 1000000$. For smaller M CPU performs better because of the constant time T_0 needed to start the Maxeler card. With the rising M Maxeler starts to perform better and above certain M the ratio between CPU and Maxeler sorting times S_2 (speedup) becomes constant.

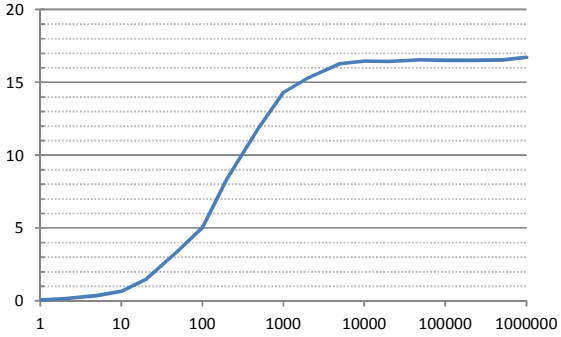


Figure 6: The speedup S_2 for arrays with $N = 128$ in dependence from the number of consecutive arrays M sent for sorting. With the growing M the speedup grows to 16.

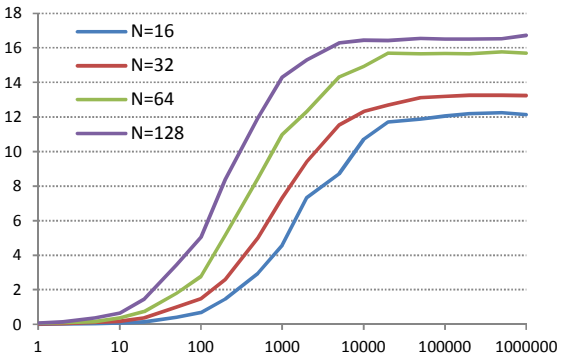


Figure 7: The speedup S_2 for arrays with $N = \{16, 32, 64, 128\}$ in dependence from the number of consecutive arrays M sent for sorting. The peak speedup increases with the array size N and is achieved at different M. The smaller the array size N, the bigger M yields the peak spedup.

The speedup S_2 for arrays with $N = 128$ is shown in Figure 6. We can observe that the peak speedup of around 16 is achieved at approximately $M > 1000$. The lower speedups at smaller M are caused by Maxeler card staring time T_0, with growing M this influence gets smaller and becomes negligible at higher values

of M where the speedup curve flattens. Here only the ratio between $T_H(N)$ and $T_C(N)$, or between $T_H(N)$ and $T_S(N)$ is important, see equation (4).

The comparison of speedup curves for different array sizes is given in Figure 7. We can observe that bigger array sizes N yield higher peak speedups that are also achieved at smaller values of M.

The above results clearly show the great potential for achieving high speedups by implementing sorting on Maxeler dataflow supercomputer systems. Even though the sorting network that we manage to implement on our MAX2 system is really small we were able to achieve speed-up of 16 compared to the quick sort algorithm on the CPU. The speed-up was achieved by sending multiple consecutive arrays, what can be useful in application where we operate with the need for many small size sortings in a short period of time.

8. CONCLUSION

Our research has demonstrated once again that almost forgotten algorithms and solutions can re-emerge with the advance of technology that enable their use, which was before impossible or impractical. Sorting networks implemented on FPGA are a natural match, where parallel execution of comparisons yields high sorting speedups that were also demonstrated through our research and the above results. The restriction here is the number of comparators one can implement on a FPGA. It limits the array size that can be sorted at a time. We managed to sort the array size of 128 and get the speedups of about 16. Another limitation is also the communication between the host and the Maxeler card which slows down the time of execution of the Maxeler part of the application code. While those results seem unimpressive and restrictions quite serious, it should be stressed that all work was done on the least capable Maxeler system. With the use of more advanced Maxeler system, we plan to increase the array size, shorten the data transfer time, and consequently improve the achieved speedup considerably. Another possible improvement is also the sorting of data already present in the Maxeler system, for instance data used by other application processes or algorithms running on the Maxeler system that must be sorted. By achieving the above, serious improvements to many applications that need sorting, would be possible.

REFERENCES

[1] Donald E. Knuth, “The art of computer programming. Vol. 3, Sorting and searching”, *Addison-Wesley*, 2002

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms, Second Edition”, Cambridge (Massachusetts), London, *The MIT Press*, cop. 2009

[3] Robert Sedgewick, “Algorithms in Java, Third Edition, Parts 1-4”, *Addison-Wesley*, 2010

[4] “*Sorting Network*”, http://en.wikipedia.org/wiki/Sorting_network, accessed 27.4.2013

[5] “*Bitonic Sorter*”, http://en.wikipedia.org/wiki/Bitonic_sort, accessed 27.4.2013

[6] H. W. Lang, “*Bitonic Sort*”, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, accessed 27.4.2013

[7] Nadathur Satish, Mark Harris, Michael Garland, “Designing Efficient Sorting Algorithms for Manycore GPUs”, *IEEE, International Symposium on Parallel & Distributed Processing*, 2009

[8] R Mueller, J Teubner, G Alonso, “Sorting networks on FPGAs”, *The International Journal on Very Large Data Bases archive*, Volume 21 Issue 1, February 2012

[9] http://www.maxeler.com/technology/dataflow-computing, accessed 27.4.2013

[10] Maxeler Technologies, “*MaxCompiler - Manager Compiler Tutorial*”, Version 2012.1, London, 2012

[11] Maxeler Technologies, “*Dataflow Programming with MaxCompiler*”, Version 2012.1, London, 2012

Implementation of the RSA Algorithm on a DataFlow Architecture

Bežanić, Nikola; Popović-Božović, Jelena; Milutinović, Veljko; and Popović, Ivan

Abstract — ***This paper presents a dataflow-based implementation of the RSA, a popular algorithm used in the public-key cryptography. The RSA implementation relies on Montgomery reduction method that is used to make the encryption/decryption process more efficient. Montgomery method improves a modular exponentiation process which constitutes the algorithm computational core. An appropriate scheduling of the Montgomery word-based operations is chosen among several options from the open literature and accordingly implemented in C. Common CPU architectures suffer from the fixed word granularity constrained by the register width. On the other side, reconfigurable platforms often lack compiler support to provide high level system design. In this paper, we rely on Maxeler dataflow platform that offers the best from both worlds. Maxeler is a FPGA platform that offers a Java extension for hardware description. Algorithm modifications necessary for offloading the word-based multiplication to a dataflow engine are proposed. Our results show that multiplication accounts for about 40% of the total CPU encryption time. We have shown that multiplication part can be improved by approximately 70%, leading total improvement to 28%, according to the Amdahl's law. Directions for further potential improvements are also discussed.***

Index Terms — ***RSA, public key cryptography, dataflow architecture, Maxeler, Montgomery reduction, encryption***

1. INTRODUCTION

RSA is a widely used algorithm in public-key cryptography [1]. It was invented at MIT as a block-based algorithm in 1977 and named according to initial letters of authors names: Rivest, Shamir and Adleman. Its security relies on the fact that it is difficult to factorize a public encryption key obtained by a multiplication of two big prime integer numbers that are kept secret. Decryption key is also obtained from these two prime numbers, it is kept secret and it is used for restoring original data, hence, providing necessary mathematical linkage.

A dataflow execution analysis for the presented algorithm is performed on Maxeler HPC platform. The platform's core named Dataflow Engine (DFE) runs on a FPGA chip and is based on a deeply pipelined hardware structure for processing streams of data. This way, inherently parallel algorithms can be accelerated through a reconfigurable logic. Current configuration state represents an implementation of a previously defined dataflow graph [2]. For a mentioned case and a big input data load, the platform offers more time and energy efficient execution in comparison with existing multi-core and many-core architectures for the same initial investment. On the opposite, this solution is not intended for the latency-sensitive, control flow intensive and small input load applications. Dataflow execution is host-controlled via appropriate API and compilation is performed within the customized development environment provided by Maxeler.

2. MONTGOMERY METHODS

If the data bit sequence to be encrypted is presented as an integer number M and the public key is $\{e, n\}$, core equation for getting an encrypted value is

$$C = M^e \bmod n \qquad (1).$$

For the equation (1) to be used in a computer data encryption, M and n are considered big integers. These integers can be implemented as array variables of the programming language supported types. Number of digits denoted s, needed to represent a big number, depends on a particular architecture and a chosen "stem" type of a program. "Stem" type provides w bits to store a digit in memory. If the modulo n is k bits long, then number of digits for a big integer is

$$s = k / w \qquad (2).$$

Conditions to be met for choosing the correct values e and n are not in the scope of this paper. Next relations illustrate decomposition of a number M to s digits, and number e to p bits:

Manuscript received March 31, 2013.

Bežanić Nikola is with the School of Electrical Engineering, Department of Electronics, University of Belgrade, Serbia (e-mail: nbezanic@gmail.com).

Popović-Božović Jelena is with the School of Electrical Engineering, Department of Electronics, University of Belgrade, Serbia (e-mail: jelena@etf.rs).

Milutinović Veljko is with the School of Electrical Engineering, Department of Computer Engineering, University of Belgrade, Serbia (e-mail: vm@etf.rs).

Popović Ivan is with the School of Electrical Engineering, Department of Electronics, University of Belgrade, Serbia (e-mail: popovici@el.etf.rs).

$$M=\sum_{i=0}^{s-1} m_i 2^{wi},\ e=\sum_{j=0}^{j=p-1} e_j 2^j \qquad (3),$$

where

$$m_i \in \{0,\ 1,\ \ldots,\ 2^w-1\},\ e_j \in \{0,1\} \qquad (4).$$

Whenever M is to be processed, it is done by iterating through its digits, what will be demonstrated later in the paper. The same is valid for modulo n. This way, one can balance between parallel and sequential processing by modifying platform dependent parameter w. Iterating through bits of e and calculating partial results of (1) are known as modular exponentiation operation which is the most popular method used in the public-key cryptography.

2.1 Montgomery Exponentiation

For the modular exponentiation process we rely on Montgomery reduction method that is used to improve modular arithmetic [3]. Key advantage of the method is replacing n modulo with r modulo, which is a power of 2 giving an efficient CPU execution. It is necessary to define r as

$$r = 2^{sw} \qquad (5),$$

and to find value n' so that condition

$$rr^{-1} - nn' = 1 \qquad (6)$$

is satisfied [3]. Computing (1) using the Montgomery method is done as shown in pseudo-code in Figure 1.

function *ModExp(M, e, n) { n is odd }*
Step 1. *Compute n'.*
Step 2. $M_m := M \cdot r \bmod n$
Step 3. $x_m := 1 \cdot r \bmod n$
Step 4. ***for*** $i = k - 1$ ***down to*** 0 ***do***
Step 5. $x_m := MonPro(x_m, x_m)$
Step 6. ***if*** $e_i = 1$ ***then*** $x_m := MonPro(M_m, x_m)$
Step 7. $x := MonPro(x_m, 1)$
Step 8. ***return*** x

Figure 1 RSA pseudo-code: Montgomery method

In steps 2 and 3 value M and 1 are converted to Montgomery's domain, while exponentiation is done in steps 4-6. In step 7, result is converted back from Montgomery's domain and encrypted data are obtained. Function *MonPro* should implement a Montgomery reduction (also known as Montgomery multiplication) and has a central role in the modular exponentiation. This is the operation where modular arithmetic with modulo r takes effect which is shown on Fig. 2. Numbers a and b are s digits long and all steps assume a digit based processing.

function *MonPro(a, b)*
Step 1. $t := a \cdot b$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (t + m \cdot n) / r$
Step 4. ***if*** $u \geq n$ ***then return*** $u - n$
else return u

Figure 2 Pseudo-code of Montgomery reduction

2.2 Montgomery Reduction Implementations

Different ways exist to implement a digit-based processing from Fig. 2. Five ways are presented in [4], namely:

- Separated Operand Scanning (SOS),
- Coarsely Integrated Operand Scanning (CIOS),
- Finely Integrated Operand Scanning (FIOS),
- Finely Integrated Product Scanning (FIPS),
- Coarsely Integrated Hybrid Scanning (CIHS).

Authors conclude that CIOS is the most efficient one considering implementations in C language and assembler. Our reference host implementation is based on the SOS method. That way it is easier to transform the code for Maxeler platform, and also, this method does not differ much from the best performed CIOS.

In [5], a multi-core implementation of CIOS is presented. Advantage of multi-core architecture is shown using OpenMP support. Partial results of a multiplication-dedicated part of the code are obtained on different cores and then accumulated on a designated core. Number of products that can be obtained in parallel is much higher on a Maxeler system, since number of multipliers is limited only with the FPGA hardware resources. For 4096-bit key size execution reported in [5] is 4.87 times faster than the original CPU Montogmery version.

In [6], customized Montgomery multipliers are designed for Altera Stratix family of chips. Smaller adders are used in this design to simplify the place and route process. Multiplication is performed on digit-by-digit basis. Encryption time is 79.95ms for 1024-bit key size and 128-bit digit width (w=128).

In [7], five versions of Montgomery method are implemented in assembly language for both Intel Pentium and Sun Sparc family of processors. Switching between different algorithms does not affect the execution time significantly, while switching between the processor technology features does. Some of the reasons for that are out-of-order execution, different pipeline depth, additional special instructions, and higher cache memory capacity.

3. RSA: HOST CODE

3.1 Implementation

As a reference CPU code we use our own C implementation of RSA which follows the optimized version of pseudo-codes from [8]. Tight coupling the optimized algorithms presented in pseudo-codes from [8] with our version of C code ensures having a relevant host reference program for performance comparison. All necessary algorithms for a complete RSA encryption process are implemented. Figure 3 shows a pseudo-code realization of crucial steps from Figure 2.

```
for i = 0 to s-1
    C := 0
    for j = 0 to s-1
        (C, S) := t[i + j] +  a[j]·b[i] + C
        t[i + j] := S
    t[i + S] := C
```

a)

```
for i = 0 to s-1
    C := 0
    m := t[i]·n'[0] mod w
    for j = 0 to s-1
        (C, S) := t[i + j] + m·n[i] + C
        t[i + j] := S
    ADD ( t[i + S], C )
```

b)

Figure 3 Realization of Montgomery reduction in pseudo-code: a) Step 1; b) Interleaved steps 2 and 3

Interleaving steps 2 and 3 allow us to calculate only the first digit instead of an entire value n' from (6). It is important to notice that the structures of inner loops in a) and b) are the same. Order of operations for pseudo-code from Figure 3 a) is presented on Figure 4.

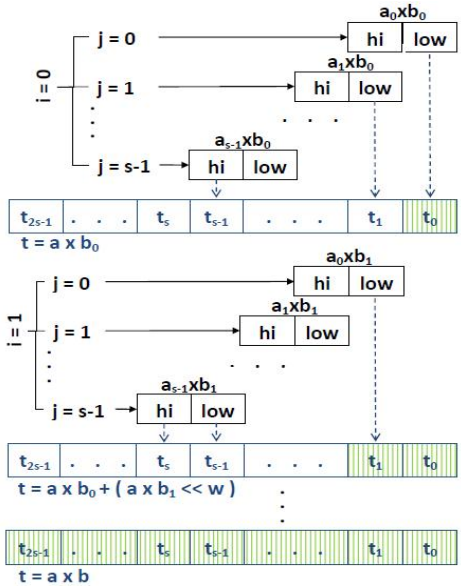


Figure 4 Digit-by-digit processing: step 1 of Montgomery reduction

It assumed that digit width is $w=32$. Every multiplication in the inner loop takes two 32-bit integers and produces two 32-bit outputs as a result. Upper half of the product can be accumulated together with the output carry value. Iteration for each j value is presented by a row, it takes input carry into account, updates appropriate digit of a partial result t, and finally, produces the output carry. After leaving the inner loop, ready digits of the final results are shaded with green color.

3.2 Verification

Debugging the RSA code is challenging due to operations in a number system basis that is equal to 2^w. In the first approach, code is tested for a simple case where $w=8$ and a key size is equal to 32 bits. Intermediate results of Montgomery exponentiation are checked per exponent bit using online simulator at the University of Massachusetts, Amherst [9]. After the first functional version was prepared, code was tested for $w=32$ and a key size of 1024 bits. Test vectors from [10] were used.

4. RSA: DATAFLOW CODE

4.1 Host Code Profiling

For program acceleration with Maxeler, it is necessary to find the critical section of the original host code. It is obvious that function *MonPro* from Figure 1 is a critical section. Otherwise, using Montgomery method would make no sense, since conversion to Montgomery's domain and back would constitute the most significant portion of execution time. However, profiling is performed using *gprof* on a Linux platform to discover how much time takes to execute the multiplication part. Results for $w=32$ are shown in Table 1. First column shows the percentage of total running time of the program used by a function. Second column shows a running sum of number of seconds accounted for by the function and those listed above. Function *mult32x32* implements digit-by-digit multiplication from Figure 4.

TABLE 1: PROFILING RESULTS FOR MULTIPLE RUNS.

Time [%]	*Cumulative seconds*	*Name*
56.25	12.47	monPro
43.08	22.02	mult32x32
0.32	22.09	calcRem
0.14	22.12	product
0.05	22.16	RDTSC
0.05	22.17	setVectors

4.2 Multiplication Speedup

In this subsection, we analyze the multiplier implementation on a DFE. Code that will be moved to DFE is code of function *mult32x32*, implemented according to [11]. First, this function is prepared by embedding existing macros and modifying statements that eventually lead to

building the cyclic dataflow graphs. Then, kernel is written using the Maxeler extension for Java called MaxJ. By executing this Java code, dataflow graph is created in the host memory and result is shown in Figure 5. Inputs to the graph are two 32-bit integer numbers delivered to hardware variables *d1* and *d2*. Upper part of the graph is producing four partial products *pp1-pp4*. Lower part does the carry transfer between these products. Signals named *hi* and *low* correspond to the values with the same name presented on Fig. 4. Graph elements represent the pipelined hardware units and it is possible to send new pair of 32-bit digits every cycle. The first result latency depends on a pipeline depth, but each following result can be obtained in the following clock cycle. Carry propagation is performed on a host in order to avoid acyclic graphs and pipeline stalls.

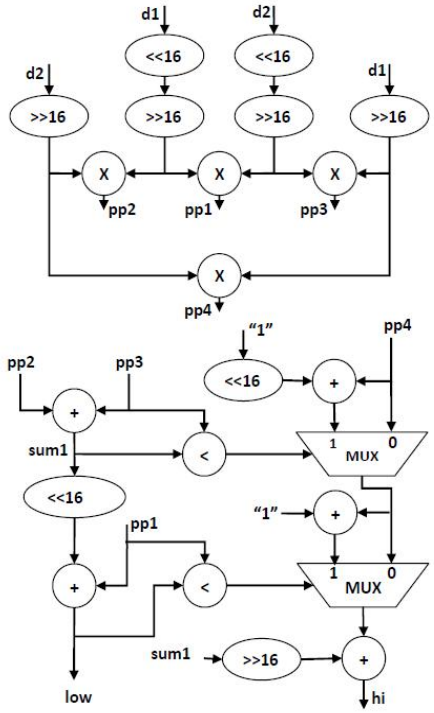


Figure 5 Dataflow graph of a 32x32 multiplier with decoupled lower and upper part of the product

In order to get an uninterrupted stream of data at the graph input, different arrangement is proposed comparing to the execution flow shown on Figure 4. Instead of updating the carry variable after multiplication of each digit pair, we perform all multiplications and deal with a carry at the end of this process. This idea is described on Figure 6. For $i=0$, all inner loop multiplications are done in parallel. This setup makes the DFE from Figure 5 usable in producing multiplication results. Now, it is possible to send b_0 as a constant value to input *d2* of the graph, and stream $a_{s-1}...a_0$ to input *d1*. Upon completion, arrays x and y are collected on the host side and carry transfer is performed.

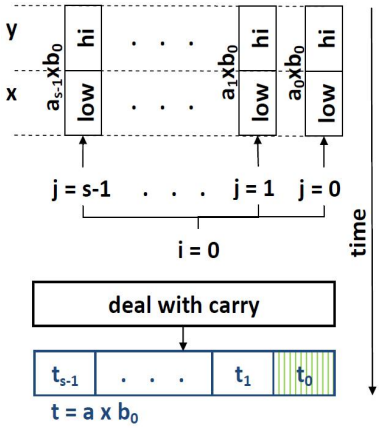


Figure 6 Algorithm adjustments toward dataflow architecture

Switching to another value of *i* can be done by setting a new constant (b_1) and sending the same stream of digits (value a) to DFE. This requires restarting entire process through appropriate API. According to (2) stream a holds only 32 digits for a key size of 1024 bits. This fact leads to very poor performance due to very short streams of data relative to the pipeline depth of the graph. To demonstrate that this is true, we use n value from [10] and set $r=2^{1024}$. Next, we calculate constant value X_m according to the step 3 in Fig. 1

$$X_m = 1 \cdot r \bmod n = const \qquad (7).$$

This way we get a convenient test vector, since it holds that

$$X_m = X_m * X_m \qquad (8),$$

where * denotes a Montgomery product. Then, we generate test vectors for different steps of the Montgomery product. Further on, we set the following values

$$b_0 = X_{m0}, \; a = X_m \qquad (9),$$

where X_{m0} is a digit of X_m with index 0. Then we start 32 runs, each one to multiply a constant and a stream, as it was described before. Measurements show that host processor finishes all multiplications in 24ns, while DFE does it in 41ms.

To eliminate the overhead of filling the pipeline with data, we rely on the fact that RSA is a block-based algorithm. In a next attempt, it was necessary to prepare a continuous stream of blocked data to form an appropriate DFE load. Figure 7 illustrates the block-based step 1 of Montgomery product for $i=0$. The index that was used before for digits of a(index j) and for digits of b(index i) is moved to the second place and has the same meaning, while the first index denotes a block that is a number "owner". To switch between blocks, restarting DFE run is still needed in order to setup the appropriate constant. This means that there is more data to send to DFE, but also, on each restart there is a new pipeline fill-up overhead.

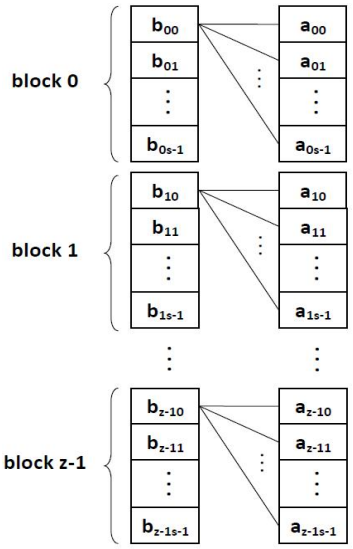


Figure 7 Block-based RSA and dataflow architecture

To eliminate the startup overhead per block we propose data choreography from Figure 8. Instead of sending a constant value to DFE, we rather send the same repeated value as a part of the stream. This results in the same product result and in the same number of execution cycles. Since the constant is replaced with a stream, there is no need to restart the DFE execution between the blocks any more. There are only s runs now.

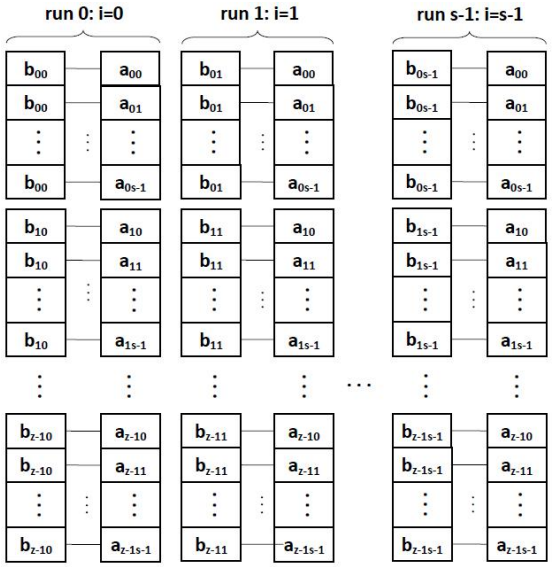


Figure 8 Block-based RSA: replacing constants with the streams of redundant data

Results for this setup are presented in Table 2. Overhead of preparing redundant streams shown on Figure 8 by a host program has not been taken into account. However, we believe that this overhead can be reduced by exploiting the control flow support provided by Maxeler in a form of dynamic offsets, counters and control streams [12]-[14]. DFE can keep on referencing the same digit over and over again using a variable dynamic offset, so that effectively a constant-vector multiplication is performed. This means that the original stream can be forwarded without creating a redundant structure.

TABLE 2: SPEEDUP TABLE FOR THE STEP 1 OF MONPRO: MULTIPLICATION.

Input data size [MB]	*Number of blocks*	*Time CPU [s]*	*Time DFE [s]*	*Speedup [%]*
1	8192	0.13	0.13	0
4	32768	0.51	0.43	16
40	327680	5.11	3.86	25
400	3276800	53.73	37.68	30

So far we have used only one hardware multiplier (one lane). It is possible to spread presented streams over the multiple lanes to improve performance. Results are presented in Table 3.

TABLE 3: SPEEDUP TABLE FOR THE DIFFERENT NUMBER OF LANES, RELATIVE TO CPU TIME.

Input data size [MB]	*Speedup [%]*	*Number of multipliers (lanes)*
1	31	2
4	55	2
40	56	2
400	61	2
1	31	4
4	57	4
40	64	4
400	69	4

Speed up of 69% is significant, but function transferred to DFE accounts for 40% of the total execution time. Referring to the Amdahl's law this gives about 28% of speedup in total. However, we have shown how blocks can be used to improve performance. Further on, block-based execution opens an opportunity for a carry transfer to be performed on DFE instead on a host. Rearranging data in stream a from Figure 8, makes possible to create acyclic graphs that work properly during a carry transfer. While a carry propagates in one block during execution, it is possible to perform multiplication in another block overlapping these two operations. For more information about how this can be done with a correct timing synchronization, reader is directed to examples from [13]. For further improvements, it is also possible to use the parallel programming to overlap existing DFE multiplication with host carry transfer. The last improvement is applicable to the step 1 of Montgomery reduction. Step 2 needs a separate analysis due to calculation of m.

5. CONCLUSION

RSA algorithm is frequently used in public-key cryptography and Montgomery method makes it inherently time efficient. Existing realizations of Montgomery reduction algorithm make a good starting point for tuning cryptography calculations toward different computer architectures. This work analyzed potentials of transferring

Montgomery realization to a dataflow architecture provided by Maxeler.

As expected, Maxeler solution shows the improvement only for a big input data set. This leads to applications that require encryption of large data files. Example would be a picture encryption where input data stream should be broken in many blocks having lengths equal to the key size. Our tests showed an improvement of 28% compared to a host RSA version and potential future improvements have been discussed.

REFERENCES

[1] Jovanović, Zoran, Lecture Notes in "Computer Security", Department of Computer Engineering and Information Theory, School of Electrical Engineering, University of Belgrade, Spring 2013.

[2] Maxeler, "MaxCompiler," White Paper, 2011.

[3] Montgomery, Peter, "Modular Multiplication Without Trial Division", American Mathematical Society, Mathematics of Computation, Vol. 44, No. 170, pp. 519-521, 1985.

[4] Kaya Koc C., Acar T., Kaliski B.S Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms", IEEE, Micro, Vol. 16, No. 3, pp. 26-33, 1996.

[5] Selcuk Baktir, Arkay Savas, "Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors", AESOP, International Symposium on Computer and Information Sciences, pp. 467-476, 2012.

[6] Fry John, Langhammer Martin, "RSA & Public Key Cryptography in FPGAs", Altera, Technical Report, 2005.

[7] Celebi Emre, Gozutok Mesut, Ertaul Levent, "Implementations of Montgomery Multiplication Algorithms in Machine Languagues", International Conference on Security & Management (SAM), pp. 491-497, 2008.

[8] Kaya Koc Cetin, "High-Speed RSA Implementation", RSA Data Security, Inc, RSA Laboratories, Technical Report, 1994.

[9] http://www.ecs.umass.edu/ece/koren/arith/simulator/Montg/mm_sim_simulator.html, February, 2013.

[10] http://www.dimgt.com.au/rsa_alg.html#realexample, February, 2013.

[11] MIPS Technologies, Inc, "64-Bit Architecture Speeds RSA by 4x", White Paper, 2002.

[12] Maxeler, "Acceleration Tutorial, Loops and Pipelining", Tutorial, 2012.

[13] Maxeler, "Dataflow Programming with MaxCompiler", Tutorial, 2012.

[14] http://home.etf.rs/~vm/os/vlsi/predavanja/maxeler.html, February, 2013.

Solving Gross Pitaevskii Equation Using Dataflow Paradigm

Saša Stojanović, Dragan Bojić, and Veljko Milutinović

Abstract – This paper presents a dataflow approach to Gross Pitaevskii equation solving by using hardware dataflow accelerator. A small part of the sequential code written in C language, that is the most time consuming, is adapted and executed on a Maxeler MAX2 dataflow card. Presented approach is compared to the approach where the same amount of computations is conducted on CPU. For the purpose of comparison, execution time and consumed energy are measured. Results show that a single dataflow implementation is faster 8 times over compared implementation on the single core of CPU. Also, computation on the dataflow accelerator card has saved over 70% of electricity compared to the same computation on the CPU.

I. Introduction

Gross Pitaevskii equation is partial differential equation that is often used in physics simulations [1]. For the purpose of simulation, the equation is numerically solved in 3D space [2]. This method requires a huge amount of computations that are time and energy consuming.

During research, physic scientist iteratively repeats simulation and analysis of results, running new simulation with new beginning conditions based on analysis of previous results. Long execution time slows down research process and decreases efficiency of researchers. Therefore, reducing execution time of simulation is one of main goals.

According with contemporary concerns about ecology, energy consumption is the other one, no less important problem. Beside of ecology, large amount of consumed energy leads to large electricity bill. Therefore, alongside of reducing execution time, reducing energy consumed on calculations is highly desirable additional goal.

Further advances in science require increase of precision and resolution of simulations. As the mentioned problem is solved in 3D space, execution time depends on third power of the problem size. Therefore, the problem of decreasing execution time and energy consumption is highly important for the future of science, and it is expected to be more and more important with further advances in science.

Sasa Stojanovic, Dragan Bojic, Veljko Milutinovic are with faculty of Electrical Engineering, University of Belgrade. Email: {stojsasa, bojic, vm}@etf.bg.ac.rs

II. Problem Statement

For the purpose of numerical solving of Gross Pitaevskii equation, an existing sequential implementation written in C language is used [4]. Sequential implementation has several phases, depicted in Figure 1. At the beginning of algorithm, input data and parameters are prepared. After preparation, main loop begins that repeats five steps

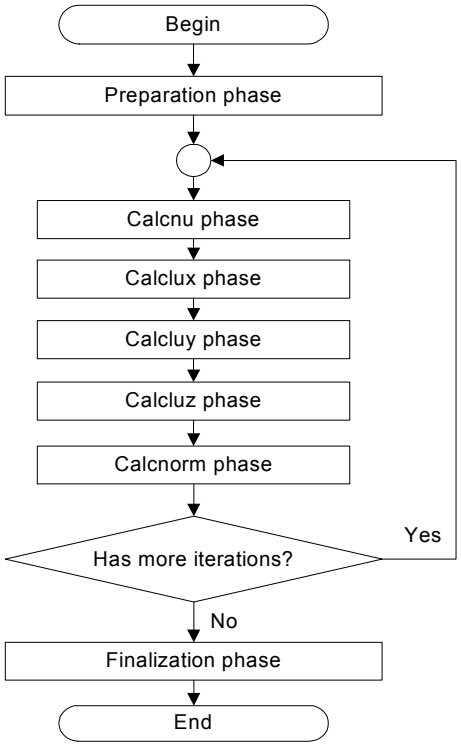


Figure 1 Phases of sequential implementation of Gross Pitaevskii equation solving.

for a given number of times. The number of iterations starts from 1000 and can be significantly higher. When this loop finishes, some final processing is needed.

The most of time is spent in the main loop, executing one of five steps in the loop. Each of steps takes one big 3D matrix on input, and based on values from this input, calculates new values for this 3D matrix. For that purpose, whole matrix has to be fetched from a memory, and after calculation, to be written back to the memory. Output matrix from one phase is input matrix to the next phase. Output matrix of the last phase in main loop is input matrix to the first phase of the next iteration of the main loop. Therefore, aside with a huge amount of calculations, needed memory bandwidth is huge because each data has to be read out and written into the memory six times per iteration (last phase in main loop requires two passes through matrix).

Depending on a problem size, size of used matrix easily can be more than several gigabytes, and hence, it cannot fit in on chip memory (block RAM of FPGA has only several MBs [5]). Interchanging order (mixing) of operations from different phases is not always possible because of dependencies that exist in some phases. In the rest of section, phases of main loop will be explained from viewpoint of dependencies that exists in those phases.

The first phase of main loop is Calcnu phase. Inside of this phase, there are three nested loops that process each element of input 3D matrix in the same way, using only already existing values. There are no dependencies, so given enough processing power, all elements of 3D matrix can be calculated

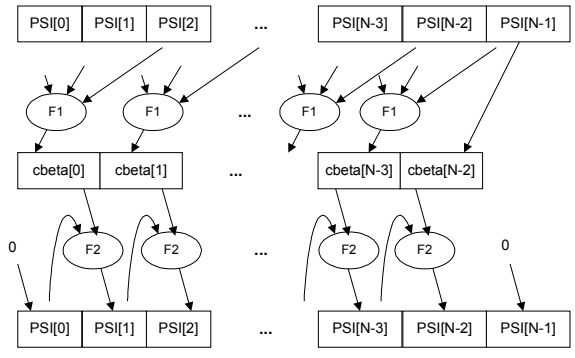


Figure 2 Diagram of data dependencies in Calclux, Calcluy and Calcluz phases. Array PSI is one of array in matrix of arrays. Processing starts from the last element. The last element of new temporary data cbeta are calculated. Then, previously calculated cbeta is used in calculation of next cbeta. After whole cbeta array is calculated, calculation of new values for PSI array is started from beginning.

in parallel. This is ideal kind of parallelism to speedup with the most of accelerators.

The second phase of the main loop is Calclux phase. This phase can be thought of as processing of 2D matrix of arrays. Arrays are taken from input 3D matrix along x-axe. Dependencies inside of one array are depicted on Figure 2. Between different arrays, there are no dependencies, so all arrays can be processed in parallel.

Next two phases, Calcluy and Calcluz, are similar to the second phase. The only difference is the direction of processing. In Calcluy phase, arrays are processed along y-axis, while in Calcluz phase, arrays are processed along z-axis. This difference puts limits on possibility of concatenating phases Calclux, Calcluy, and Calcluz. It is not possible to fetch one element of input 3D matrix, process it by all three phases and then, write it back to the memory (or it will be needed to keep whole 3D matrix in on chip memory).

The last phase, Calcnorm is composed of two passes through input 3D matrix. In the first pass, an accumulated value is calculated based on values of all elements from input 3D matrix. Calculated value is used in the second pass to divide all elements of input 3D matrix. This process is well known as normalization of data. One good property of last phase is that it does not require any particular order for the processing of elements.

III. Existing Solutions

There are a lot of solutions for numerical solving Gross Pitaevskii equation. The most of them are implemented for servers based on standard processors [4]. There are also solutions that use GPU to speedup execution [6, 7]. Some solutions try to break execution and solve the problem using cluster [8].

Mentioned solutions demonstrate significant performance improvement over compared base version. However, no one used hardware dataflow implementation. Also, the most of them lack of an energy consumption analysis that is becoming highly important in contemporary world.

IV. The Proposed Solution

Solution proposed in this paper is to use Maxeler hardware implementation of dataflow engine to speedup solving of Gross Pitaevskii equation. As

accelerator, MAX2 card is used. Card is intended for PCIe x16 bus that can be split into two ports, x8 each. The card is used in system that does not support division of PCIe port, implying that only a half of card is available.

On the dataflow system, duration of one phase is composed of the time needed to read data from memory and write results to the memory and time needed to fill pipeline. In the case of huge amount of data, as is the case, time to read from memory and write to memory is several orders of magnitude higher than time needed to fill pipeline, leading to conclusion that processing time in dataflow system will be approximately same for all phases except for last, that requires two passes through 3D matrix and it is two times longer (five phases contains six passes). As the execution time on dataflow system depends linearly on number of passes, some passes of original six passes will be merged to total of three passes per an iteration of the main loop. Whenever it is possible, overhead processing will be overlapped with calculations. In this way, the time needed for reordering of the data from memory, and the time needed for reordering results will be completely hidden, except at beginning of pass, and at the end of the pass. However, these times are small compared to whole processing time of one pass and can be ignored.

In this paragraph, it will be explained how some passes can be merged. As described in section II, it is possible to interchange order of calculations in Calcnu phase to match order needed for Calclux. In this way, Calcnu and Calclux phases can be merged. Calcluy cannot be easily merged with any other phase. Calcnorm phase can be split into two sub-phases. The first sub-phase calculates weighted sum of all elements for normalization purpose, while the second sub-phase divides input data by the sum calculated in the first sub-phase (normalize data). The order of calculations of the sum in the first sub-phase can be interchanged to match the order of the results produced by the Calcluz phase. In this way, Calcluz can be merged with first sub-phase of Calcnorm sub-phase. The second sub-phase of Calcnorm can be moved to the beginning of the next iteration, by dividing input data of Calcnu phase with sum calculated in the first sub-phase of Calcnorm phase from previous iteration of the main loop. In the first iteration of main loop, there is no previous iteration, so initial value for the sum will be 1 (this gives the same result as there is no division at the beginning of first iteration of the main loop). In order to keep correct result, after the main loop is finished, one additional division of values in 3D matrix by last calculated value of sum is needed.

V. DETAILS

In order to efficiently utilize pipeline and all data from burst fetched from memory (burst is unit of data transferred to/from memory; size of burst is 96B [9, 10]), groups composed of 36 independent arrays will be processed in parallel. Because of the same reason, instead of some round number, size of input 3D matrix will be set to 432x432x432. Type of each element is double.

These data are packed in memory along Z axes, meaning that one burst contains data with same X and Y index, while Z index is in range that encompasses 12 succeeding elements. Special care must be taken to fetch data in right order for processing of data in phases Calclux, Calcluy, and Calcluz. Required order is from end of array to the beginning. From the group of 36 arrays, last element of each array is the first needed. In the case of Calclux and Calcluy phases, it is possible to fetch these 36 elements in three bursts, then previous 36 elements in other three bursts and so on. In the case of Calcluz, fetching arrays starting with last elements of arrays requires more attention. Last 12 elements are fetched from memory for all 36 arrays in group, but in an inappropriate order. Elements are buffered in BRAM memory on FPGA chip. During the reading from BRAM memory, data are reordered and passed to processing kernel in the appropriate order. This process does not have drawbacks onto execution time due to overlapping of this process and processing of data. Only latency to the first result will be visible.

Because of dependencies in phases Calclux, Calcluy, and Calcluz, additional latency is introduced. This latency is due to two passes through array in mentioned three phases, where the second pass cannot begin before the first result is produced. Described latency will happen after processing of each group of arrays, and it will be 86 clock periods long. This latency is hidden during execution and it will influence only latency to the first result.

The worst case is in Calcnu phase, where it is required to read 16 bytes from memory and to write 8 bytes to memory. These requirements are small enough compared to available bandwidth to/from memory and cannot slowdown calculations. In these

circumstances, hardware dataflow is capable to start processing one new element of input 3D matrix per clock period. In order to start processing of n elements of input 3D matrix, n clock period is needed. Because of pipelined implementation, the first result will appear on the output after d clock periods, and rest of n-1 results will appear in next n-1 clock periods. Therefore, total execution time of one pass is n+d-1. For each new pass whole process has to be repeated.

VI. Axioms, Conditions, and Assumptions of the Analysis to Follow

Due to limitation of available mainboard, only half of used Maxeler accelerator card is available for calculation. In addition, used card is one of the oldest and of the smallest Maxeler cards, several generations older than used processor. In order to mitigate these differences, original existing code for CPU will be used, that is intended for single thread execution.

VII. Mathematical Analysis

As described in sections IV, V, and VI, instead of original 6 passes through input 3D matrix, dataflow implementation will do only 3 passes. Let us suppose working frequency of streams to be 100MHZ and dimension of a problem to be 432^3. As shown in section IV, execution time is approximately same as time needed to flow data from memory through dataflow engine (one new data per clock is inserted in the pipeline) to memory. Now, total execution time of an iteration of main loop can be calculated as number of passes per iteration multiplied by number of elements that will be processed and divided by the working frequency of streams. According with previous expression and supposed values of parameters of the system, expected execution time per iteration is 2.42s. Comparing this value to measured value of existing CPU implementation of 19.85s, one can expect a speedup of approximately 8 times.

VIII. Implementation Analysis

During the implementation, new problems raised. Half of used card was not enough to fit solution in its original form. Resource usage report shows that the critical resource is the number of available DSP blocks. In the half of the used MAX2 [11] card has 196 DSP blocks, while original design requires 380 DSP blocks. Further analysis shows that DSP blocks are mainly used for implementation of multiplication of floating point numbers. Original design uses double precision. Implementation of one multiplication of two double numbers requires 10 DSP blocks, while for multiplication of float numbers, only 2 DSP blocks are required. Unfortunately, using float numbers is not possible because some numbers during the calculation are out of range of float numbers. Next tried solution was using combination of float and double format. Size of exponent field is used from double, while mantissa is used from float numbers. This combination also requires only 2 DSP blocks. Results are completely different from expected ones, so this combination is not usable too.

The last one regular optimization is to try to use double numbers with slightly reduced size of mantissa. The first smaller size of mantissa, for which the number of used DSPs is decreased, is 41. In this case, 5 DSP blocks are required, that can fit in available space on the half of card. Because of precision reduction, it is expected to get results that differ from results produced with original code. As differences appear only after first 8 significant decimal digits, this case is satisfactory.

Execution times are given in Table 1. Streams are clocked with 100MHz clock. Achieved speedup over the CPU is 8.2 times. It is interesting to compare actual execution time per iteration and same execution time predicted in section VII. Actual execution time is only slightly higher than predicated one. This difference can be explained by neglected latency of pipeline that results in latency to the first result, and latency introduced by process of time measurement.

Table 1 Execution times and achieved speedups

	Without card	With card
Execution time	19980s	2425s
Number of iterations	1000	
Execution time per iteration	19.98s	2.425s
Speedup	1	8.2

Energy consumed during calculation is shown in Table 2. At the first glance, average power of a system with Maxeler card is higher than system without the card. But, for end user it is much more important how much it will spend if runs system with or without the card. Because the problem will be solved much faster, despite of higher power of

system with the card, this system will consume 5.2 times less energy for the same amount of calculations. Of course, this implies that the system with the card is turned off when job is finished. From the other viewpoint, system with the card can replace 8.2 systems without the card, and consume slightly more energy than one of replaced systems, while its energy consumption is significantly less than total energy consumed by all 8.2 systems.

Table 2 Energy consumption during the processing

	Without card	With card
Idle	61W	86W
Under workload	76W	118W
Execution time per iteration	20s	2.43s
Number of iterations	1000	
Spent Energy	422.22Wh	79.98Wh
Relative energy	100%	18.9%

IX. CONCLUSION

Work presented in this paper shows that hardware dataflow has a potential to alleviate problems of time and energy consumption during the Gross Pitaevskii equation solving. The problem demonstrated enough parallelism, so the dataflow approach achieved speedup of execution of 8.2 times, while saves over 80% of energy spent for the same amount of processing on a system without the card. Note that more than half of consumed energy in system with dataflow is spent on a part of system that is idle. Increasing number of cards in system can possibly decrease influence of energy spent on idle part of system. Other influences of additional cards have to be examined.

Using more recent Maxeler cards with larger capacity will open new ways for speedup of Gross Pitaevskii equation solver. Provided enough hardware resources and large enough bandwidth to the memory, one can implement more than one dataflow engine, exploiting more parallelism, just like multicore systems do. Also, one new way for further improvements in the light of new available resources is decreasing number of passes by splitting Calcluy phase into two sub-phases that will be merged to Calclux and Calcluz. In this way, instead of standard data from input 3D matrix, some intermediate result will be written to the memory.

ACKNOWLEDGMENTS

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (Cloud services for applications with high performance requirements. Number: 451-03-00605/2012-16/198, and projects III44009, TR32047, and III44006). Special thanks to our colleague Mr. Zeljko Djurisic who provided us with equipment for measurement of consumed electricity

REFERENCES

[1] L. P. Pitaevskii, S. Stringari, "Bose-Einstein Condensation," Oxford University Press, isbn 9780198507192, 2003.

[2] J. Rogel-Salazar, "The Gross–Pitaevskii equation and Bose–Einstein condensates,"European Journal of Physics, ISSN 0143-0807, vol. 34, no 2, pp 247, 2013.

[3] W. Bao, D. Jaksch, P. A. Markowich, "Numerical solution of the Gross–Pitaevskii equation for Bose–Einstein condensation," Journal of Computational Physics, vol 187, no 1, ISSN 0021-9991, pp 318-342, May 2003.

[4] D. Vudragovic, I. Vidanovic, A. Balaz, P. Muruganandam, S. K. Adhikari, "C programs for solving the time-dependent Gross–Pitaevskii equation in a fully anisotropic trap," Computer Physics Communications, vol 183, no 9, ISSN 0010-4655, pp 2021-2025, September 2012.

[5] Xilinx, "Virtex 5 Family Overview", web: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, accessed on April 19, 2013.

[6] A. Gothandaraman, S. Sadatian, M. Faryniarz, O. L. Berman, G. V. Kolmakov, "Application of Graphics Processing Units (GPUs) to the Study of Non-linear Dynamics of the Exciton Bose-Einstein Condensate in a Semiconductor Quantum Well," Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp.68-71, 19-21 July 2011.

[7] T. Dziubak, J. Matulewski, "An object-oriented implementation of a solver of the time-dependent Schrödinger equation using the CUDA technology," Computer Physics Communications, vol 183, no 3, ISSN 0010-4655, pp 800-812, March 2012.

[8] A. Gorecki, P. Bala, B. Lesyng, "TI - Parallelization of the quantum dynamics code for cluster architecture and its applications to the Gross–Pitaevskii equation," International Journal of Quantum

Chemistry, vol. 106, no. 3, Wiley Subscription Services, Inc., A Wiley Company, pp 664-669, 2006.

[9] Maxeler, "MaxCompiler, Kernel Compiler Tutorial", 2011.

[10] Maxeler, "MaxCompiler, Manager Compiler Tutorial", 2011.

[11] F. Haohuan, "Accelerating Scientific Computing Through GPUs and FPGAs," Stanford Center for Computational Earth & Environmental Science (CEES), web: http://cees.stanford.edu/docs/CEESWorkshop8-HFu.pdf, accessed on April 20, 2013.

Feasibility Study on the SAT Solver on DataFlow Architecture

Zivojin Sustran, Zoran Ognjanovic, Milan Todorović and Veljko Milutinovic

Abstract — The SAT problem is the first known example of NP- complete problems. This means that there is no known efficient algorithm for solving this problem, and it is mostly believed, yet still not proven, that such algorithm doesn't exist.

This research tries to explore possibility for implementing a SAT solver on the DataFlow architecture. High degree of independency between calculations, used in solving SAT problems, is a reason for possible speed-up on the DataFlow architecture.

Index Terms — SAT problem, DPLL procedure, SAT solvers, DataFlow

I. Introduction

Boolean satisfiability, or SAT for short, is a problem of determining whether there is an assignment for the variables (valuation) for a given propositional formula, that makes the formula true. This problem is one of the most famous NP-complete problems, which was independently shown by Cook [1] and Levin [3]. The consequence of this is that, unless $P = NP$, all complete algorithms for solving SAT problem require exponential time in the worst-case scenario. Despite this fact, modern SAT solvers are capable of handling large problem instances due to improvements made to them in the last 15 years. This allows their application to a wide range of practical problems [4], such as hardware and software verification, constraint satisfaction, planning, etc. These problems can be solved by translating into SAT, and although translating a problem to SAT usually produces larger representation than the original, it is often more efficient to do that and to use modern SAT solvers, than to use solvers specialized for the problem's initial representation [5].

SAT solvers can be divided into two groups, complete and stochastic. For a given propositional formula, complete SAT solvers guarantee to either find valuation that satisfies the formula or to determine that such valuation does not exist. On the other hand, stochastic solvers can prove that formula is satisfiable, often faster than complete solvers, but they can't prove that formula is unsatisfiable [7].

II. DPLL procedure

Most modern complete SAT solvers are based on DPLL (Davis-Putnam-Logemann-Loveland) procedure [2]. The procedure uses backtrack search and some additional rules for increasing efficiency. As input, the procedure expects formula in conjunctive normal form, and as output gives YES if formula is satisfiable and NO otherwise. As conjunction and disjunction are commutative and associative, the order of clauses in an input formula is not important and formulas can be represented as a set of clauses which can be, similarly, represented as sets of literals (variable or negation of variable). The procedure assumes that empty set of clauses (empty formula) is satisfiable, and that empty set of literals (i.e., empty clause) is unsatisfiable, which makes the formula that contains such clause unsatisfiable.

The main step of the procedure - branching on literals- consists of selecting a literal l that occurs in the formula F. Then l is replaced with $\top$. After that, the new formula is simplified, and it's satisfiability is examined recursively. If it is proven to be satisfiable, the starting formula F is also satisfiable. Otherwise, l is replaced with $\bot$ in F, and the process is repeated. If this formula is proven to be satisfiable, F is also satisfiable, but if it is proven to be unsatisfiable, then the original formula F is also unsatisfiable. Although the strategy for selection of literals has no impact on procedure's correctness, it has great impact on its efficiency.

Two additional rules, that are not required for completeness of procedure, are used to increase efficiency of this search. The first rule is called unit clause, and it is based on the fact that if a clause in the formula consists of only one literal, that literal must be true if the formula is to be satisfied. The second rule, pure literal, is based on the fact that if literal occurs in a formula, and its negation doesn't, then all clauses that contain the literal can be erased without affecting the affect satisfiability of the formula. Both of these rules eliminate the need for branching on those literals. The original DPLL procedure is shown in Fig. 1.

Most of the modern, state-of-the-art, SAT solvers, like PICOSAT, MiniSAT, SATO, CHAFF and others, use DPLL procedure with many additional features that are introduced to increase performance. In the original DPLL procedure, argument is a formula that is modified through recursive calls. This is inefficient for larger instances, so the first feature would be to maintain the current partial valuation, to keep the formula in the original form and only check its value against the current valuation. Also, the procedure is usually implemented in an iterative way rather than recursive, which reduces memory usage [5].

III. Problem Statement

The main goal of this research is to create a prototype SAT solver using the Maxeler DataFlow Engine MPC-N40 [10].

This work was supported by the Serbian Ministry of Sciences, under the project III 44006.

Z. Sustran is with University of Belgrade, Serbia (e-mail: zika@etf.bg.ac.rs).
M. Todorović is with the Mathematical Institute is with Serbian Academy of Sciences and Arts, Serbia.
V. Milutinovic is with University of Belgrade, Serbia (e-mail: vm@etf.bg.ac.rs).

```
function DPLL (F :formula)
begin
  if F is empty then return YES;
  replace all ¬⊥ with ⊤ and ¬⊤ with ⊥ in F;
  remove all ⊥ from F;
  if F contains empty clause then return NO;
  if ∃ clause c s.t. ⊤ ∈ c or l, ¬l ∈ c
    then return DPLL(F \ {c});
  if ∃ unit clause {l} in F
    then return DPLL(F [l → ⊤]);
  if ∃ pure literal l in F
    then return DPLL(F [l → ⊤]);
  begin
    select a literal l from F;
    if DPLL(F [l → ⊤]) = YES then return YES;
    else return DPLL(F [l → ⊥]);
  end
end
```

Fig. 1. DPLL procedure.

Existing SAT solvers are implemented for general purpose processors with ControlFlow architecture. ControlFlow processors are usually optimized for integer instructions. Calculating Boolean expressions using integer instructions is slow. The idea is to transfer Boolean expressions to the MPC-N40 and accelerate their evaluation.

The prototype solver will show how much acceleration can be accomplished using DataFlow architecture for certain set of SAT problems. The set of SAT problems solvable by the prototype solver contains SAT problems with limited number of variables. Upper bound of the number of variables is determined by hardware limits.

Determining of how much speed-up is possible with DataFlow architecture is important because it will give us insight whether the prototype solution is possible path toward creation of a useful industry SAT solver on DataFlow architecture. If speed-up is big, it is probably possible to make the useful industry SAT solver by reducing some constraints used in the prototype solver. Reducing some constraints will reduce speed-up of the prototype solver, but if that reduction is smaller than initial speed-up feasible speed-up solution is possible.

In the near future DataFlow architecture will be developed faster than ControlFlow architecture [12] and its hardware will have much more capabilities. Currently clock speed used in hardware is 200 MHz. Hardware with much higher clock speed was made available at the market recently [11]. Use of this hardware for implementing DataFlow architecture can additionally accelerate SAT solver.

IV. The Proposed Solution

The proposed solution tries to solve a SAT problem using a simple algorithm. The simple algorithm for every possible assignment of input variables evaluates the given propositional formula. When evaluates the given propositional formula to true, it returns the variable assignments. Number of calculation steps exponentially depend on the number of input variables. This is also true for all existing complete solvers in the worst case.

Because there is no true data dependency in calculations for one possible assignment of input variables on DataFlow architecture, they can be successfully pipelined and results can be output each clock cycle [13]. Calculation of one possible assignment of input variables on the ControlFlow architecture requires several machine instructions. Accounting for latency generated by cache misses, required time for one calculation can be measured in thousands of clock cycles in the worst case scenario. This is relatively big advantage for a SAT solver on the DataFlow architecture. Since calculations are independent, both architectures can speed them up by means of parallel execution. On ControlFlow architecture solver can be parallelized by the number of executions determined by processor threading capabilities. On DataFlow architecture solver can be parallelized by the number of executions determined by hardware used for implementation. The number of execution on DataFlow architecture is the same or greater than the number of execution on ControlFlow architecture, if hardware used for both architectures is in the same price range.

Best way to determine whether proposed solution is better than the existing solutions is to implement them and empirically compare them. Comparison is done on different hardware used for both architectures, keeping hardware in the same price range. Comparison is done for a set of SAT problems that can be solved on proposed solutions. Since the set of SAT problems is relatively large, only the worst case scenario is considered. This will give the upper bound for time of execution on both architectures.

V. The Details of Proposed Solution

The proposed solution is made for a simple SAT solving algorithm. The simple algorithm can be divided into two phases: generation of input variables and calculation of a Boolean propositional formula. Those two phases are directly implemented on DataFlow architecture. In order to make execution as fast as possible, multiple calculation units are inserted into second phases. Generation phase distributes input variables to all parallel calculation units. Since every calculation unit generates a result every clock cycle, the third phase is incorporated into a simple algorithm for processing of results. Transfer of results from DataFlow engine to host requires data in specific format, which is different from format used for calculation. This conversion makes the forth and final phase incorporated into a simple algorithm. The complete visualization of algorithm is presented in Fig. 2. Detailed explanation of each phase is in the following subsections.

A. Generation Phase

Generation phase is used to deliver input variables to the calculation phase. Since all input data are generated on DataFlow engine, there is no slow transfer operation of data from host. This means that calculations can begin as soon as DataFlow engine is initialized.

Every possible variables assignment has to be generated. For this purpose a simple unsigned integer counter is used. The counter is initialized to zero and on each clock cycle it is incremented. Every bit of the counter value represents one

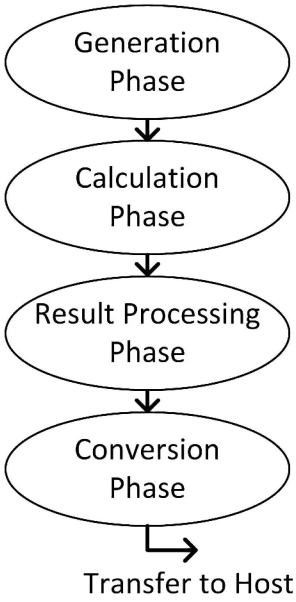


Fig. 2. The simple algorithm for solving SAT problem.

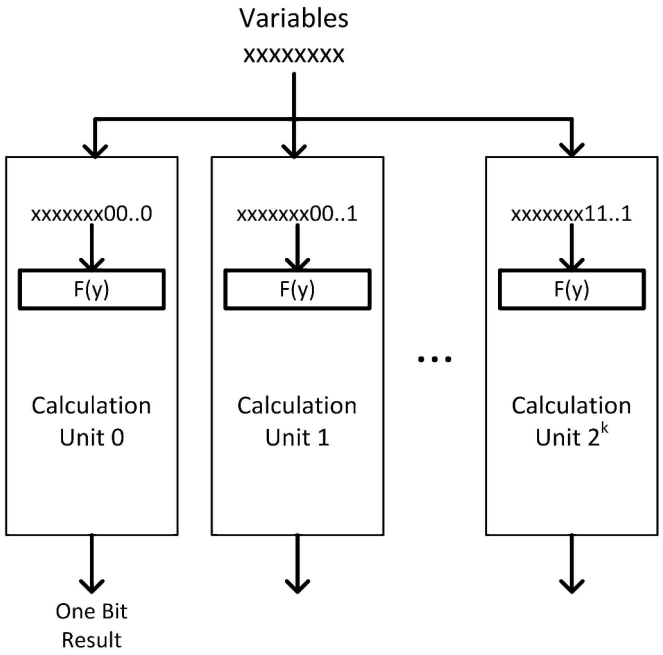


Fig. 3. Calculation phase.

variable. When counter reaches maximum value, every possible combination of input variables is generated. The width of counter value is lower than the number of values in a Boolean propositional formula. Since there are multiple calculation units, each unit has its own private set of assignments for several variables that are not delivered by the generation phase. The delivery of counter value to multiple calculation units can become a problem, if number of calculation units becomes large. The problem is in complicated routing of wires needed to deliver the counter value to each calculation unit. One solution to this problem is to use multiple counters in generation phase. This reduces routing problem with use of redundant counters. The redundant counters do not represent a problem, if relatively small number of counters is used. In this solution the number of counters is kept small, by creating counter for a set of calculation units for which routing is not a problem.

B. Calculation Phase

The calculation phase is used to calculate a given Boolean propositional formula. The formula is directly implemented on DataFlow engine. For different formula recompilation source code is necessary. Compilation can take a lot of time. Calculation of the formula is pipelined so result is produced each clock cycle, independent of the complexity of the formula. There is no true data dependency between different calculations, so the pipeline, after the initial fill phase, is always kept full, till the end of all calculations. This means that, there is no need for stall and there will not be bubbles in the pipeline. The pipeline is maximally utilized all the time.

Calculation phase is divided into multiple calculation units, as illustrated in Fig. 3. Every calculation unit in parallel calculates the given formula and outputs the result. The result is one bit representing whether the formula is satisfied for the set of input variables. The number of calculation units is power of two. That makes easier job distribution over different units. If the number of variable is n and the number of units is 2^k, the generation phase generates m - k global variables and each calculation unit has k private variables. Global variables are the same for each calculation unit. Private variables for each calculation units are unique. The calculation unit output result using both global and private variables. The result is sent to the result processing phase.

C. Result Processing Phase

The Result Processing Phase receives each clock cycle 2^k results. The phase's job is to determine whether there is one result with value one (not important which one). Results processing can be presented as a binary tree of logical OR gates, as depicted in Fig. 4. The results are inserted into tree leaves and the tree root gives answer whether the formula is satisfiable for the input variables assignment. If the formula is satisfiable, the variable assignment is output to the conversion phase.

Result processing phase determines maximum level of parallelization. Routing of wires needed to implement phase function is the most complex task that determines whether the solver is possible to implement on DataFlow engine. Routing complexity is determined by the number of results entering the phase each clock cycle. If routing complexity can be circumvented, the maximum level of parallelization will be determined by the maximum number of the calculation units that can be placed on DataFlow engine.

D. Conversion Phase

The conversion phase receives variables from the result processing phase, that satisfies the formula. Variables are needed to be converted to different format, so they can be transferred to the host. Each variable is transformed to the

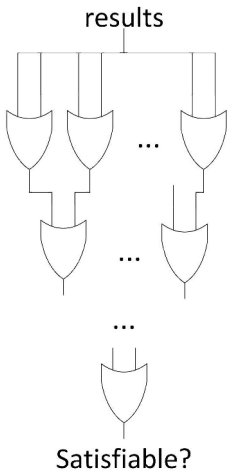


Fig. 4. Result processing phase.

size of one machine word. Machine word is much bigger than one bit thus unnecessary data is being transferred to the host. Time to transfer data linearly depend on the number of variables in the formula, so this latency in not dominant one and further optimization of transfer in not necessary. If there is no assignment that satisfies the formula, nothing is being transferred to the host. When this situation is detected on the host, the formula is declared unsatisfiable.

VI. Analysis of Proposed Solution

Proposed solution is implemented for specified DataFlow engine and existing solution is implemented on ControlFlow processor. Both versions are compared for worst case scenario. Details of comparison are in following subsections.

A. Conditions and Assumptions

Existing solution is implemented on two high-end server processors with use of 16 threads. Each thread has hardware support for execution and does not share processing resources with other processors. The algorithm is implemented to the best of author knowledge.

Proposed solution is implemented on the Maxeler DataFlow Engine MPC-N40. The number of computation units is 2^{11}. There is space on the DataFlow engine for more computation units, but it is impossible to connect them due to routing problem discussed in previous sections. The set of solvable SAT problems is limited by maximum number of variables, 42. Raising the number of variables will slow down calculations by factor of two for each new variable.

Time needed for execution on DataFlow engine can be predicted mathematically. Each computation unit will get 2^{31} different inputs variables assignment, so it will take at least same number of clock cycles to output result in the worst case scenario. Time to execute will be at least 11s. Time to start DataFlow engine and time to transfer data to the host is not included in the previous analysis because it can be considered constant. Time needed for execution on ControlFlow processor will not be predicted here.

B. Result of the Analysis

Execution of the existing SAT solver on ControlFlow architecture it timed to be in the range of 40000s. Execution of the prototype SAT solver on DataFlow architecture is in the range of 30s. The speed-up accomplished by using DataFlow architecture is over one thousand.

VII. Conclusion

SAT problem is well known, and one of the most important NP-complete problem. In the last two decades, considerable progress, both practical and theoretical, has been made in the study of this problem. Many SAT solvers have been developed that can solve challenging satisfiability problems [6]. This allows encoding of different problems from various domains into SAT: hardware verification and fault diagnostic, planning in artificial intelligence, machine-shop scheduling, haplotyping in bioinformatics and graph coloring [4], [9]. On the theoretical side, techniques developed for SAT solvers can be adapted and extended for usage in solvers for other, similar, problems.

The implemented prototype SAT solver greatly overcomes the current implementations of SAT solvers on ControlFlow architecture, for certain set of SAT problems. This leads us to say that research in this direction might be fruitful and successful.

There are two problems that arise from previous discussion. First is programmability of the prototype solver. The prototype solver is made only for one Boolean propositional formula. Creation of such prototype requires time not included into this analysis. There is a possibility to send configuration data to DataFlow engine during initialization to modify formula being calculated. This will additionally complicate the implementation and will slow down the solver. If this slowdown is less than thousand times, better SAT solver will be created than on the existing solutions. Second problem is the number of maximum variables in the formula. The maximum number is too small for use in any practical situations. There are no indications whether the maximum number of variables can be higher. If there is no possibility for that, maybe some hybrid solution can be made, utilizing the best from both worlds.

References

[1] Stephen Cook, "The Complexity of Theorem-Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151-158 (1971)

[2] Martin Davis, George Logemann, Donald Loveland, "A machine program for theorem-proving," *Communications of the ACM*, 1962, Vol.9, Num.7, pp. 394-397

[3] Leonid Levin, "Universal Sequential Search Problems," *Problems of Information Transmission*, 1973, Vol.9, Iss 3, pp. 256-266 (1973)

[4] Marques-Silva, J., "Practical Applications of Boolean Satisfiability," *International Workshop on Discrete Event Systems*, WODES 2008, pp. 74-80

[5] Carla Gomes, Henry Kautz, Ashish Sabharwal, Bart Selman, "Satisfiability Solvers," *Handbook of Knowledge Representation*, Elsevier B.V., pp. 89-122

[6] Holger H. Hoos, Thomas Sttzle, "SATLIB: An Online Resource for Research on SAT" *In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pp.283-292, IOS Press, 2000.

[7] Mladen Nikolić, "Metodologija izbora pogodnih vrednosti parametara SAT rešavača," *Magistarska teza*, Matematički Fakultet, Univerzitet u Beogradu, 2008.

[8] Filip Marić, Predrag Janičic, "Formalization of abstract state transition systems for SAT," *Logical Methods in Computer Science* Vol. 7 (3:19) 2011, pp. 1-37

[9] Ian Gent, Toby Walsh, "The search for satisfaction," *Department of Computer Science University of Strathclyde*, Scotland (1999).

[10] Maxeler Techologies. *http://www.maxeler.com/products/mpc-nseries/,* April 2013

[11] Achronix Semiconductor Corporation. "Speedster22iHD FPGA Family." *http://www.achronix.com/wp-content/uploads/docs/* Speedster22iHD FPGA Family DS004.pdf, April 2013

[12] Flynn, M., Mencer, O., Milutinovic, V., Rakocevic, G., Stenstrom, P., Trobec, R., Valero, M., "Moving from petaflops to petadata", *Communications of the ACM*, ACM, New York, NY, USA, Volume 56, Issue 5, May 2013, pp. 39-42.

[13] -, "Multiscale Dataflow Programming," *Maxeler Techologies*, Version 2012.2, December 2012

The Source-Sink Model

Siniša Ivković, Luka Ilić, Marko Stanković, Radomir Radojičić, and Zoran Babović

Abstract – **Weather forecasting is based on the use of numerical weather prediction (NWP) models that are able to perform necessary calculations that describe/predict major atmospheric processes. Numerical weather prediction advances have been always very closely related with advances in computing infrastructure. Consequently, new computational technologies and facilities are exploited to boost the development of numerical weather prediction applications on different computing platforms. This paper describes the Source-sink model implementation that is significantly accelerated using accelerator based on DataFlow architecture. In our implementation, the algorithm execution time exploiting Maxeler MaxCard is 50 times shorter than the execution time on the Intel i3 based PC.**

I. Introduction

Atmospheric computer model is a computer representation of processes in the atmosphere. These processes are mathematically described by ordinary and partially differential equations which are in computer models typically approximated with finite-difference method [[4]]. There are number of sequential Finite-difference method implementations and their algorithm complexity is most often $O(n^2)$.

The Eta Model is a state-of-the-art atmospheric model used for research and operational purposes [[2]]. The model is a descendent of the earlier HIBU (Hydro-meteorological Institute and Belgrade University) model, developed in the seventies in the former Yugoslavia. In the eighties, the code has been upgraded to the Arakawa-style horizontal advection scheme. Arakawa-style grid is a rectangular grid used for Earth system models, especially for meteorology and oceanography. After Arakawa style, the code has been rewritten to use the eta vertical coordinate [[11]], and subsequently, at NCEP (National Centers for Environmental Prediction), supplied with an advanced physics package [[9]]. It became officially operational at NCEP on June 8, 1993 [[7]], and was operational until 2003. In its various versions, the model has been and/or is widely used in numerous countries of Europe, Asia and North America.

The Source-sink model is a shallow-water equations model defined on Arakawa E-grid that represents the core of Eta model. It models the process of geostrophic adjustment present in the Eta model.

Manuscript received May 31, 2013.
S. Ivkovic, School of Electrical Engineering, University of Belgrade, Serbia (e-mail: sinisa.ivkovic@gmail.com).
L. Ilic, Faculty of Physics, University of Belgrade, Serbia (e-mail: luka.ilic@sewa-weather.com).
M. Stankovic, School of Electrical Engineering, University of Belgrade, Serbia (e-mail: stankovic.marko@gmail.com).
R. Radojicic, School of Electrical Engineering, University of Belgrade, Serbia (e-mail: bataradojicic@gmail.com).
Z. Babovic, School of Electrical Engineering, University of Belgrade, Serbia (e-mail: z.babovic@gmail.com).

Simulation of atmospheric models requires processing large amounts of data. It is always hard to work with big data problems because they require much processing power. The increase of processors clock frequency causes the increase of power consumption and the heat. These problems are hard to solve, so implementations are now turning to parallel processing to improve performance. Parallel processing is simply the ability to execute multiple operations or tasks simultaneously. One form of parallel processing is a simple use of more than one CPU to execute the program. In practice, it is difficult to divide program in a way that every CPU executes its part of the code without interfering with each other. Recently, the hardware designers have been investigated other technologies to overcome these limitations of standard microprocessor technology. A very efficient paradigm is the dataflow computing which is relying on FPGA hardware accelerators. The basic idea is to consider an application as dataflow graph of executable actions. When operands become valid, the action is executed and result is forwarded to the next node in graph. Maxeler Technologies is one of the leading vendors in this domain and it provides hardware platform and tools for easy development of applications utilizing dataflow architecture [[1]], [[5]].

Because of atmospheric models computation nature, these models are very suitable for parallelization. This paper presents implementation of parallelization of the Source-sink model utilizing dataflow based accelerators provided by Maxeler Technologies.

In Problem Statement part, we presented more details about Eta model, Source-sink model as the core of the Eta model and Maxeler FPGA accelerator cards as a technology that is used for implementation. Section Implementation describes details of implementation of Source-sink model on a MaxCard. Analysis of Results shows an overview of performance and accuracy measurements that were performed.

II. Problem Statement

A. Eta Model

The name of the model derives from the Greek letter η (eta) which denotes the vertical coordinate, one of the model features, defined as:

$$\eta_s = (p_{ref}(z_s)-p_t) / (p_r(z=0)-p_t) \quad (1)$$

where p is the atmospheric pressure. The indices s and t refer to the surface and the top of the model atmosphere, respectively. The index ref refers to a prescribed reference atmosphere, and z_s is the surface height. The model orography is formed of steps. The steps can have slopes in the current version. Model variables are distributed on an Arakawa E-grid. Major features of the Eta dynamical core are: the eta vertical coordinate, resulting in quasi-horizontal

coordinate surfaces, and thus prevention of pressure-gradient force errors due to steep topography which can occur with terrain-following coordinates; forward-backward scheme for time differencing of the gravity-wave terms; the Arakawa approach in space differencing, with conservation of enstrophy and energy; energy conservation in transformations between the potential and the kinetic energy in space differencing; and option to run the model in a non-hydrostatic mode. The model physics package Comprises of: convection schemes; cloud microphysics; radiation scheme; land surface scheme with 12 types of vegetation and 7 types of soil texture, 4 soil layers; turbulence and PBL: Mellor-Yamada 2.5, and Monin-Obukhov similarity theory in the surface layer, with Paulson stability functions. While the primary use of the model has been for regional weather prediction and NWP type applications, the model has, also, been very successful in regional climate and seasonal prediction applications.

B. SOURCE-SINK MODEL

Source-sink model (SSM) is a shallow-water equations model defined on Arakawa E-grid (*Figure 1*) [[1]]. It models the process of geostrophic adjustment present in the Eta model. E-grid is a superposition of two C-grids. Numerical solving of geostrophic adjustment produces a noise in numerical solution as a separation of solutions on two C-sub-grids of the E-grid occurs. In SSM and Eta model numerical techniques are used to filter out the noise from the numerical solution. Forward-backward scheme for time differencing of the gravity-wave terms is modified so as to suppress separation of solutions on two C-sub-grids of the Eta model's E-grid [[10]][[8]].

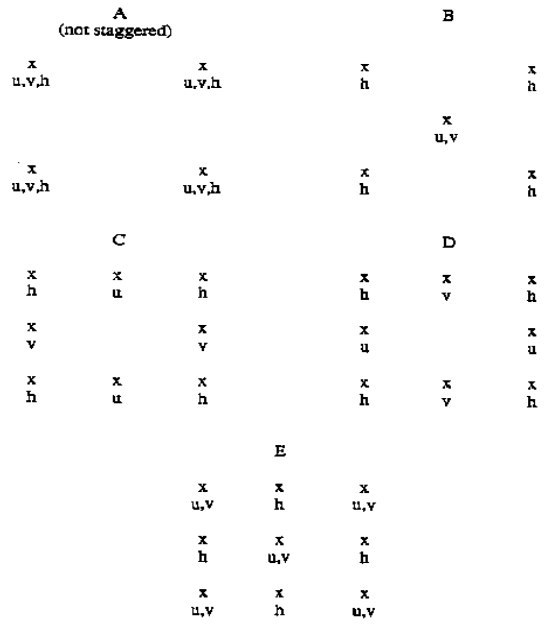


Figure 1. Variables arrangement on the Arakawa grids

SSM models two parameters: height (mass, pressure) and speed of fluid. Algorithm (*Figure 2*) has four main components: continuity equation, updating the height, equation of motion, and updating the speed. Every component's output is the next component's input and the latest component's output is the first component's input. One iteration of the algorithm corresponds to one step in time [[3]].

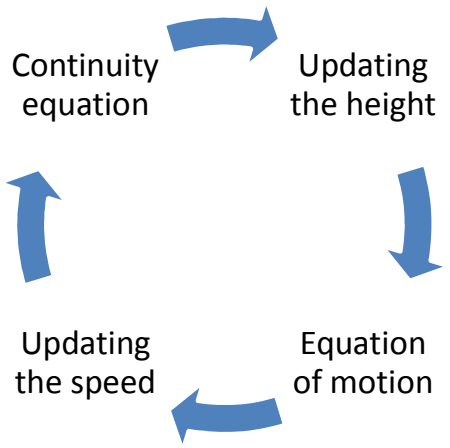


Figure 2: Source-sink model algorithm

Continuity equation takes the speed for its input and gives the speed gradient which is used to update the height. These new height values are then used as input to the motion equation, which gives the height gradient. As the last step, new speed values are calculated from the height gradient and the Coriolis effect.

C. Technology

Dataflow machines showed great efficiency and speed up for big data problems, and weather forecast analysis certainly is one of them. Control flow is a model of computation in which program's source code is transformed into a list of instructions for particular processor. These instructions then move through processor and execute. This model is inherently sequential and performance depends of CPU clock cycle and memory latency. Modern processors contain cache memory, forwarding and predicting logic for improving efficiency. In dataflow paradigm we describe the operations and data choreography for a particular algorithm. Data is read from memory and then passed from one arithmetic unit to another until the processing is complete. Dataflow cores compute only a single type arithmetic operation which enables us to fit thousands of cores on one dataflow engine. Data dependencies are resolved statically at compile time. There are no instructions, so there is no need for instruction decode logic, instruction caches, branch prediction etc., and because of this, full resources of the chip can be used for performing computation (Error! Reference source not found.) [[5]].Significant acceleration can be gained using dataflow machines but in return but we need to invest more time in programming them. We used MaxCards because they are easier to program then other FPGAs. MaxCompiler is the application that generates a dataflow implementation of given problem programmed in MaxJ, which is a version of Java that adds operator overloading semantics to the basic Java language [**[5]**].

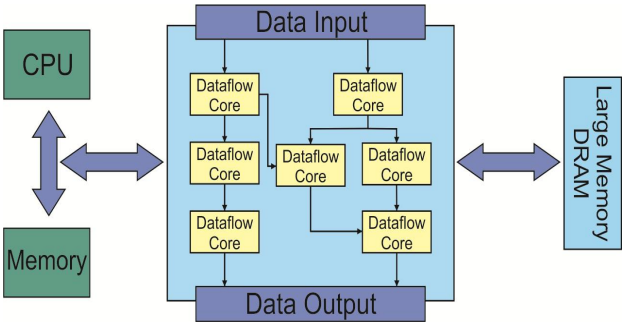


Figure 3: Architecture of MaxCard connected to Host PC

III. Implementation

Original implementation of source-sink model that we ported to a dataflow engine was written in FORTRAN. FORTRAN is most frequently used for atmospheric research, but its dominance is mainly for historical reasons. The first models were coded in FORTRAN and parts of those models are commonly used in new models. For analysis we translated FORTRAN code into C programming language. Results of this paper are performance and accuracy comparison between algorithm in C and its dataflow implementation ported on MaxCard. Actual model is working on three sets of a half a million data points each: height (h) and two speed components (u and v). These sets are shown in matrices dimensions IMxJM, where IM is number of rows, and JM is number of columns. The computing is evenly distributed among all the loops, so the only way to achieve a performance gain is to implement the whole model in a Dataflow Engine (DFE).First task was to implement a function that converts matrix into an array by taking columns one by one and concatenating them. Matrices h, u and v converted into arrays represent input streams for the MaxCard. In each chip clock cycle, one new value from each input stream enters the MaxCard. The algorithm of continuity equation is shown in *Figure 4*.

```
FOR j:=2 step 2 UNTIL JM-2
BEGIN
    FOR i:=1 step 1 UNTIL IM-1
    BEGIN
        uxvy[i,j] := function of up and vp
        Lputa[i,j] := function of h
        Lplus[i,j] := function of h
    END
END

FOR j:=1 step 2 UNTIL JM-1
BEGIN
    FOR i:=1 step 1 UNTIL IM-1
    BEGIN
        uxvy[i,j] := function of up and vp
        Lputa[i,j] := function of h
    END
END

FOR j:=1 step 2 UNTIL JM-1
BEGIN
    uxvy[0,j] := function of up and vp
    Lputa[0,j] := function of h
END

FOR j:=3 step 2 UNTIL JM-3
BEGIN
    FOR i:=1 step 1 UNTIL IM-1
        Lplus[i,j] := function of h
    Lplus[0,j] := function of h
END

FOR i:=1 step 1 UNTIL IM-1
BEGIN
    Lplus[i,1] := function of h
    Lplus[i,JM-2] := function of h
END
Lplus[0,1] := function of h
Lplus[0,JM-2] := function of h
```

Figure 4: Continuity equation

The new value of element on position [i ,j] in result matrices uxvy, Lputa, and Lplus do not depend only on elements in the same position in matrices up, vp and h, but on their neighbor elements as well. Therefore it was necessary to modify the algorithm to locate elements which are needed for computation in array instead in matrix. For example, for computing element uxvy[i, j], where i is number between 1 and IM-1, and j is even number between 2 and JM-2, we will need elements of matrices up, and vp on positions [i-1, j+1], [i-1, j], [i, j], and [i-1, j-1]. If we want to do the same thing with arrays, the element uxvy[i, j] would be uxvy[i+j*IM] and the elements for computation would be [i-1+j*IM], [i-1], [i+j*IM], and [i-1+(j-1)*IM]. The order of elements after converting matrix into an array is shown in *Figure 5*. Matrix elements which are needed to compute value marked with red ellipse are marked with blue octagons.

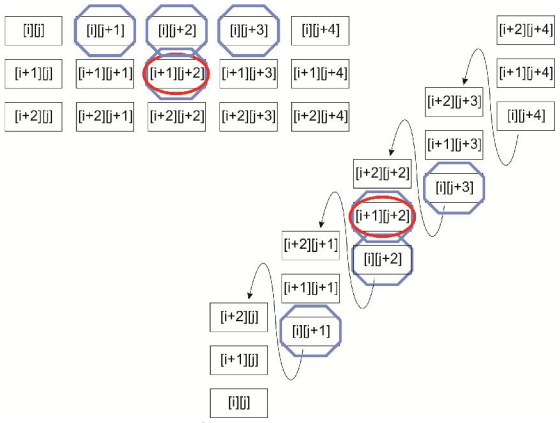


Figure 5: Converting matrices to arrays

Maxeler Kernel is hardware implementation of computations needed within an algorithm. To implement this algorithm on Maxeler Kernel, we used Stream offsets. They allow us to access past and future elements of data stream. Window of data is distance from largest to smallest offset that is held in the dataflow engine [[5]]. A Maxeler Kernel element up[i-1+j*IM] would be accessed using function stream.offset(up, -1+IM). The main difference is that we do not need i and j values because we address other elements relative to the current element in stream. But we need to know current element position in a stream so we can decide which formula to use. That is why we have a counter which increments every time new value enters the Kernel. Depending on output of combination logic whose inputs are values of counters, Kernel decide which formula to use to compute a result.

A result of one pass of data through the Kernel is the same as the result of one iteration of an algorithm. MaxCard is connected to the PC via PCI Express slot. This is often a bottleneck because of data transfer between MaxCard and PC. For a 24 hours computation with a step of 112 seconds, we would need to transfer data between MaxCard and PC 771 times, which is very time-consuming. Solution for this problem lies in the use of MaxCard DRAM. DRAM is a large external resource of memory, which enables us to keep data local to the board and iterate over data by using FPGA (*Figure 6*) [[6]]. Data can be stored in DRAM from both PC and MaxCard. This way, before starting the Kernel, we can store data in DRAM from PC. The Kernel reads data from DRAM and after every execution write them back into MaxCard DRAM. After the last run of the Kernel, the results should be transferred from DRAM back to PC. With this approach we achieved acceleration of 24-50 times.

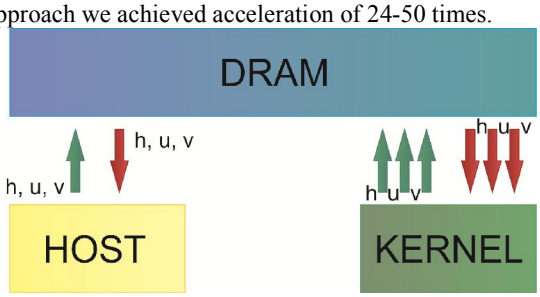


Figure 6: Connection between Host, DRAM and Kernel

The model works with floating point numbers. In our dataflow implementation we used 32 bits floating point numbers, 8 bits for mantissa and 24 bits for exponent. The accuracy was 10^{-4} so we increased number of bits for mantissa to 11 and for exponent to 37 which gave us an accuracy of 10^{-6}. This was the maximal accuracy we could achieve because of the lack of hardware resources that did not allow further increase of number of bits in the variables.

IV. Analysis of Results

The obtained results can be observed from two aspects acceleration and accuracy. Measurements were performed executing algorithm written in C programming language on PC, and on MaxCard. Our test environment consists of a standard PC configuration with microprocessor Intel i3-540 3.09 GHz and 4GB of RAM. MaxCard model is MAX2336B which belongs to the second generation of Maxeler acceleration cards. MaxCard is connected to the PC via PCI Express slot. Maxeler Max2 Acceleration Card resource summary and its comparison with Max3 Acceleration Card are given in *Figure 7*.

LUT – LookUp table – a small combinational units that can be configured to implement any logical function
FF – Flip-flops – used as registers
BRAM – Block RAM – low latency on chip memory cell
DSP – fast multiplication elements
DRAM – onboard memory

MaxCard	LUTs	FFs	DSPs	BRAMs	DRAM	DRAM bandwidth
Max2	207360	207360	196	324	24 GB	28 GB/s
Max3	297600	595200	2016	1064	48 GB	38 GB/s

Figure 7: Max 2 and Max 3 resource summary

First solution was to gather data from the host machine and, after computation, return all the data to host (in every iteration). This solution didn't use MaxCards DRAM, and it gave us speed up of approximately 4.

Use of DRAM significantly increased speed of execution. Depending on accuracy, speed up was between 24 and 50 times. That is why one of the major improvements for this application that Max3 card brings is DRAM bandwidth. Accuracy has strong influence on execution speed of C code. We can see the difference between execution speed on both MaxCard and CPU with 32, and 48 bits floating point numbers on MaxCard and single and double precision numbers on CPU (*Figure 8*)

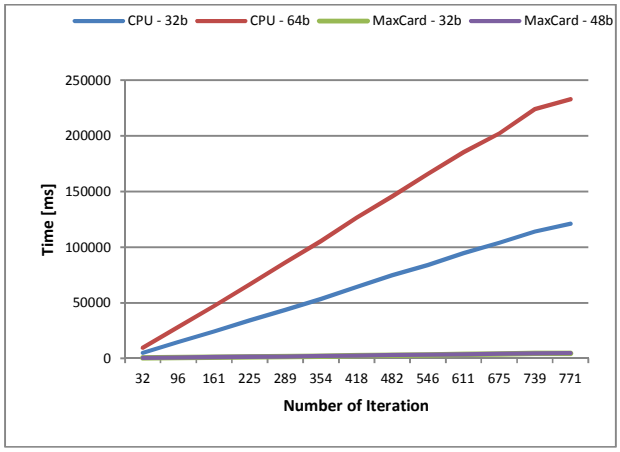


Figure 8: MaxCard and CPU time of execution

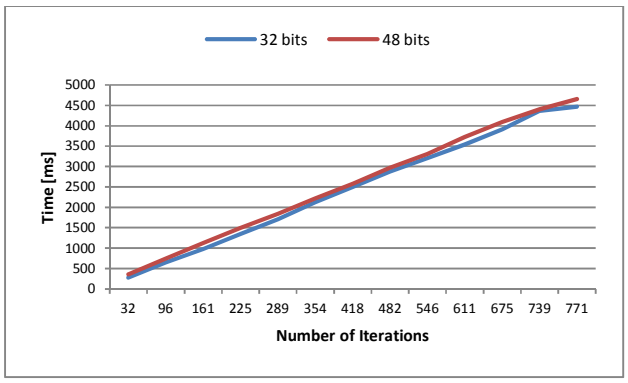


Figure 9: MaxCard time of execution with 32 bits and 48 bits floating point numbers

As we can see on Figure 9: MaxCard time of execution with 32 bits and 48 bits floating point *numbers*, the difference in speed between the two implementations on MaxCard is insignificant. Opposite to that, in C program, using 32 bits precision is almost two times faster than 64 bits precision. Values of execution times are presented in the table in *Figure 10*. Number of iterations is selected to represent number of iteration needed for 3, 7, 11, 15, 19, and 23 hours simulation. Because of a so small difference between execution times on MaxCard in further analysis we will use only execution times for 48 bits precision on MaxCard.

Number of Iterations	96	225	354	482	611	739
CPU - 32b	14784	34226	53440	74706	94599	113999
CPU – 64b	28310	66358	105291	145654	185297	224048
MaxCard – 32b	646	1351	2127	2878	3543	4364
MaxCard - 48b	**742**	**1497**	**2221**	**2975**	**3728**	**4404**

Figure 10: Execution speed on CPU and on MaxCard [ms]

The resulting acceleration when using MaxCard is between 24 and 50 depending on data types used in host code, and it is shown on *Figure **11***.

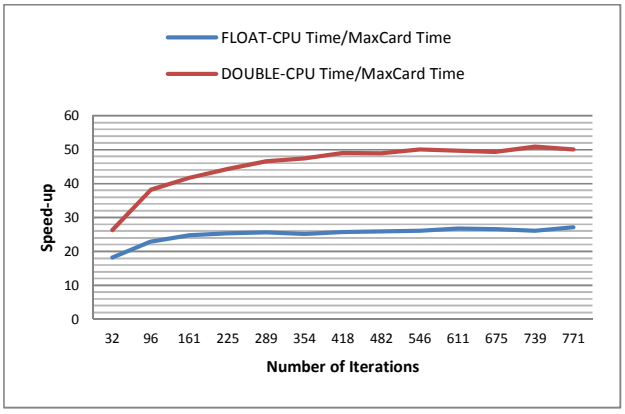


Figure 11: MaxCard and CPU execution time difference

Algorithm works nearly 25 times faster on MaxCard comparing to CPU which works with 32 bits precision and around 50 times faster comparing to CPU with 64 bits precision.

By comparing the results of executions of C code that works with 64 bits floating point numbers, and a MaxCard that works with 48 bits floating point numbers, we get 10^{-6} accuracy. Critical resource that prevents further increase of accuracy are FFs. Use of newer Max3 card would enable us to further increase the accuracy.

V. CONCLUSION

The research in this paper presents efficient approach to solving big data problems using FPGA accelerators based on dataflow architecture provided by Maxeler technology. As a demonstration example we choose Source-sink model, core of the weather forecast Eta model. Previous works and experiences with Maxeler show that accelerators based on dataflow architecture are very efficient for problems that have the following characteristics: they are big data they are latency tolerant and they have large percent of execution time in loops.

Increasing complexity of weather forecast simulations increases their accuracy, but also makes them more computational demanding and requires more execution time. By using MaxCard we gain acceleration of 50 times with 10^{-6} accuracy. It is important to emphasize that these results are achieved using second generation of Maxeler accelerator cards, and that nowadays Maxeler offers the fourth generation of accelerator cards. The most important improvements in newer MaxCards are increased resources and increased bandwidth to MaxCard DRAM. In our application, critical resource for accuracy are flip-flops, and the newer generation of MaxCard with more resources of this type would allow us further increase of precision. This could be achieved by increasing number of bits we use for floating point numbers. Higher bandwidth to MaxCard DRAM would result in further acceleration, because it would speed up the transfer of data between the Kernel and the DRAM.

Most of the problems that have large amount of matrix computations, such as Source-sink model, can be accelerated in this way. Acceleration depends mainly on data dependencies. The amount of data dependencies dictates how

much we will be able to adapt the algorithm to the dataflow architecture, in other words how much we can parallelize it. The more we parallelize the algorithm the more speed up we will achieve.

Accelerators of this type could be very useful for Meteorological labs by enabling them to perform simulations much faster than using only CPUs. This approach also leads us to the idea of using accelerator as a service, so anybody could use it by connecting to it via internet connection. This could be very useful for smaller labs and researchers who cannot afford to buy their own one.

VI. REFERENCES

[1] Kotroni, V., Floros, E., Lagouvardos, K., Pejanovic, G., Ilic, L., Zivkovic, M., "Multi-model multi-analysis ensemble weather forecasting on the grid for the South Eastern Mediterranean Region," *Earth Science Informatics*, December 2010, Volume 3, Issue 4, pp. 209-218.

[2] Radojicic, Z., R., Zivkovic, T., M., Avramovic, Z., Z., "Eta2000 weather forcast modeling system and railway applications," *Južel*, 345-348, Vrnjačka Banja, 2000.

[3] Djurdjevic, V., "Geostrophic Adjustment Process in Shallow Water Equation system," *Graduation Thesis*, Institute of Meteorology, Belgrade, 1998.

[4] Jacobson, M.. Z,, "Fundamentals of Atmospheric Modeling," Second Edition, *Cambridge University Press*, USA, New York, 2005.

[5] -, "Multiscale Dataflow Programming," Maxeler Technologies, 2012.

[6] -, "MaxCompiler Manager Compiler Tutorial," Maxeler Technologies, 2012.

[7] Black, T., L., "The new NMC mesoscale Eta Model: description and forecast examples," Weather Forecast 9, 1994, pp. 265–278.

[8] Janjic, Z., I,. "Forward-backward scheme modified to prevent two-grid-interval noise and its application in sigma coordinate models," ContribAtmosPhys 52, 1979, pp. 69–84.

[9] Janjic, Z.. I,, "The step-mountain coordinate: physical package," Mon Wea Rev 118, 1990, pp. 1429–1443.

[10] Mesinger, F., Janjic, Z., I., "Noise due to time-dependent boundary conditions in limited area models," The GARP Programme on Numerical Experimentation, Rep. No. 4, WMO, Geneva, 1974, pp. 31-32.

[11] Mesinger, F., Janjic, Z., I., Nickovic, S., Gavrilov, D., Deaven, D., G., "The step-mountain coordinate: model description and performance for cases of Alpine lee cyclogenesis and for a case of an Appalachian redevelopment," Mon Wea Rev 116, 1988, pp. 1493–1518.

Siniša Ivković graduated from University of Belgrade, School of Electrical Engineering, Department of Computer Science. After graduation he was employed by Innovation Center of School of Electrical Engineering as a researcher on a project of implementing the Source-Sink model (Weather Forecast Analysis) on a dataflow architecture. He is currently employed by Seven Bridges Genomics company, where he works on integration and development of bioinformatics tools for a Cloud based platform.

Luka Ilic is a PhD student of meteorology at Institute of Meteorology, Faculty of Physics, University of Belgrade. He has more than five years of experience in the area of numerical weather prediction (NWP). During this period he actively participated in implementing and maintaining NWP systems in the region. He took part in numerous research projects regarding NWP application in renewable energy and environmental studies. One of the major projects was installation, implementation, and testing of an NWP ensemble system on the SEE-GRID infrastructure.

Radomir Radojicic is a PhD student at the School of Electrical Engineering, University of Belgrade, Serbia. Author and coauthor of about 10 conference and journal papers. Extensive experience of more than 15 years in IT project management founded by Royal Norwegian Ministry of Foreign Affairs, FIVB, Serbian ministry, as well as industry leaders (Meteos GMBH, REC, FK Partizan, HE Visegrad, JP Sluzbeni glasnik…)
2010-2013 Working on Setting up Environmental Management Center (EMC) in Serbia for efficient emission monitoring and reporting within Serbian Agency for Environmental Protection. 1997-2013 Leads a team of 15 members for the statistical analysis of international FIVB volleyball matches in Serbia.

Zoran Babovic is a Ph.D. student at the School of Electrical Engineering, University of Belgrade, Serbia, where he also received his M.Sc. in electrical engineering in 2004. Since 2006 he is working as a research associate at the Innovation Center of the School of Electrical Engineering, University of Belgrade, Serbia. He participated in four EU FP6 and FP7 research projects in the domain of sensor networks, data mining, and computer architecture, as well as in four innovation and research projects funded by Serbian Government. He published several conference and journal papers, and gave numerous talks at conferences in Europe.

Accelerating Lattice-Boltzman Method Using Maxeler DataFlow Approach

Korolija, Nenad; Djukic, Tijana; Milutinovic, Veljko; and Filipovic, Nenad

Abstract — ***This paper describes one approach to solving Lattice-Boltzmann algorithm. The algorithm is used for simulating real systems by calculating flow speed in each of elementary particles of total volume. Unlike traditional programming models, we tried to solve the algorithm by using dedicated hardware for executing instructions. Results indicate that this approach brings not only better performances in terms of time needed to execute the simulation, but it also reduces the electrical power needed for the simulation. This problem is important, since simulations often last for many hours, and are repeated many times. Since computational power of computers increases each year, it is expected that this simulation will be even more important in the future.***

Index Terms — ***Lattice-Boltzmann, FPGA, simulation, accelerating, power consumption***

I. INTRODUCTION

As computer processing capabilities increase, simulation of relatively big systems becomes more and more prevalent. Since technological capabilities limit the maximum speed of processors, the technology of modern processors becomes oriented towards parallel processing. Most of modern processors have many cores. The same paradigm is used on many supercomputers. There are also some different approaches. Scheduled data-flow paradigm is focusing on solving CPU-intensive problems by introducing relatively big number of small threads that communicate between each other, as if data is continuously flowing between them. One implementation of scheduled data-flow paradigm is Decoupled threaded architecture explained in the paper by Roberto Giorgi at al. [1]. Another way of solving CPU-intensive applications is to use dedicated hardware for faster instructions execution [2]. Some implementations of this paradigm are explained in the Maxeler Technologies document [3]. Our paper describes one possible solution to solving Lattice-Boltzmann problem using a personal computer with one Maxeler card. This card is equipped with FPGA which can be set-up (changed) by programming in a programming language similar to Java thus reducing necessity to know details of FPGA programming model in order to program the Maxeler card.

Manuscript received on April 23, 2013.

Authors 1 and 3 are with the Department of Computer Science, University of Belgrade, Bulevar Kralja Aleksandra 73, 11000 Belgrade, Serbia (nenadko@etf.rs, vm@etf.rs). Authors 2 and 4 are with the University of Kragujevac, Serbia (tijana@kg.ac.rs, fica@kg.ac.rs).

Work presented in this paper was supported by the project “Cloud servisi za aplikacije sa zahtevima za visokim performansama”.

II. ABOUT THE MAXELER DATAFLOW COMPUTING

The DataFlow computers, according to many, are the most attractive solution for ExaScale applications based on BigData. Note, however, that any presentation about DataFlow architectures also should be a presentation about algorithmic changes that make the existing algorithm for a given application better suited for execution on DataFlow architecture.

The above mentioned is due to the fact that DataFlow architectures represent a new paradigm for computing. For the same production price, compared to conventional ControlFlow architectures, DataFlow architectures could generate the following advantages: (a) speed-ups that, depending on application, may go up to 20, or even up to 200, or even more; (b) the power consumption gets reduced about 20 times, which means that the monthly electricity bills for computer centers become about 20 times smaller; (c) the size of the system also becomes about 20 times smaller. All the above holds for the cases when the engineering issues, like design cost and design complexity, are the only factors determining the price of the system. Of course, in the real world economy, factors like production volume and market demand, do count, as well.

Assuming that, sooner or later, production volumes and market demands of DataFlow machines will reach those of ControlFlow machines, it is realistically to expect that the purchasing prices of DataFlow machines will be even lower compared to ControlFlow machines, mostly due to the fact that design costs and design complexity of DataFlow machines are much lower.

However, for DataFlow architectures to be superior over ControlFlow architectures, some conditions must hold. We see six major relevant conditions.

(1) **BigData**. The higher the data volume, the more are DataFlow architectures expected to be better. This is due to the "flow though" nature of data processing in DataFlow architectures.

(2) **WORM**. The application has to be of the WORM type (WORM = write once, run many). This is due to the fact that the programming paradigm for DataFlow architectures is more complex and programming makes sense only if the program will be run a large enough number of times.

(3) **Tolerance to latency**. This is due to the fact that the initial latency, i.e. the latency till the first result, is hither in DataFlow architectures, compared to ControlFlow architectures.

(4) **Most of the run-time is related to the execution of programming loops**. This is due to the fact that DataFlow architectures are mostly meant to speed-up the execution time of loops. From the Amdahl's low it stems directly that speed-up of 20 can be achieved, ideally, only if 95% of runtime is related to the execution of loops. In this context, ideally means that the architecture is able to squeeze the loop execution time down to zero. Consequently, this means that, ideally, the above mentioned speed-ups of 200 could be achieved with the code that spends 99.5 percent of time executing loops.

(5) **Reusability of data**. This is due to the fact that DataFlow architectures typically serve as system accelerators (they are almost never used as stand-alone machines). Consequently, once a data item is brought through some slow interconnection (e.g., a PCIe bus), the more it is used inside the DataFlow system, the more are compensated the negative influences of the slow interconnection bus. For example, performance of computing $y = x + x^2 + x^3 + x^4 + \ldots$ (which can be found in AstroPhysics applications) is expected to be much better than for $y = x + 1$.

(6) **Programmer sophistications**. This is due to the fact that direct programming of existing algorithms would bring only a single digit speed-up, and sometimes no speed-up at all: for a good two-digit speed-up to be achieved, the programmer has to dedicate lots of time, effort, and knowledge into the following activities: (a) algorithmic changes; (b) input data recoreography for data to fit better into the algorithmic changes and the underlying implementations of the DataFlow computer hardware; (c) better utilisation of the internal pipelines; and (d) if the application permits, lowering of the precision of fixed-point and/or floating-point computations needed.

Anecdotally speaking, the above can be phrased as "do not use a Ferrari car to drive over a plowed field; use a tractor for plowed fields and Ferrari at paved highways."

In the case of the Maxeler DataFlow computer, the system was generated through a kind of Euro-American synergistic interaction. While the basic concepts and ideas were born in the USA, the hardware and software were designed and are produced in Europe, using the components generated in the USA (Xilinx and Java), using the vision and mission of Europeans.

Two notes are now in place, one about the history of DataFlow computing, and the other one about ranking of DataFlow computers, in comparison with ControlFlow computers.

The first ideas about the DataFlow computing were born about 50 years ago. Famous is the work of professors Danning (static DataFlow) and Arvind (dynamic DataFlow). Both performed their research at MIT, Cambridge, Massachusetts, USA. Known are also some industrial efforts to create a commercial DataFlow machines (e.g., HEP, or: Heterogeneous Element Processors). None of the commercial efforts made it through, mostly due to the following two facts: (a) the underlying reconfigurable hardware was expensive; and (b) the technology for creation of system software was not yet ready for the challenge. It was only very recently that Maxeler (related to Stanford in Palo Alto and Imperial in London) was able to make a successful commercial machine. The HEP was an example on a coarse-grained approach to DataFlow, while the Maxeler is an example of a fine-grained approach to DataFlow. This basically tells that DataFlow machines are divided into two categories: (a) coarse-grained; and (b) fine-grained. Each one of two categories can be further subdivided into "explicit DataFlow" and "implicit DataFlow" (see Stojanovic2013 for a more detailed explanation) [4].

The best supercomputers of the world are ranked using the top500 lists [5]. Until recently, the Japanese "Computer K" was the number one [6]. Until some weeks ago, the IBM Sequoia was the number one [7]. Currently, Cray Titan item is the number one [8]. It is notable that no DataFlow machine is listed among Top500. This is because the Top500 ranking is based on a very simple program, Linpack [9]. In a new Top500 list, based on some BigData benchmarks, a DataFlow machine would probably dominate, as indicated in the paper from the Maxeler team [10].

The basic difference between ControlFlow and DataFlow systems stems from the fact that ControlFlow machines compile down to the

machine code level, while the DataFlow machines compile down to a lower level (the level of gates, transistors, and wires). This fact brings speed-ups power reductions, size reductions, and cost reductions. In the rest of the text, the system cost is assumed to be the same for the classical technology (ControlFlow) and the emerging technology (DataFlow), which creates the basis for a fair comparison of the two technologies.

As indicated below, the price to pay for the above advantages is a more complex programming paradigm. Consequently, DataFlow technologies in general are better suited for the so-called WORM (Write-Once-Run-Many) applications; it is not a technology to be used by students to use it in order to complete their semester homework, to run it, and never to re-run it again.

Examples of a WORM applications can be found in geophysics (typical speed-ups from 20 to 40), banking (maximal speed-ups up to 200 or even 1000, which explains why JP Morgan decided to own Maxeler with 20%), monitoring and control (e.g., ideal for image mining from thousands of cameras in big cities like New York), and DataMining (e.g., Google can data-mine from the social networks it hosts, for better future of the mankind).

In essence, a Maxeler system serves as an accelerator for the host CPU. This means the following: (a) a Maxeler board or a Maxeler box is connected to the host CPU via the PCI express (PCIe) bus (smaller systems use only the PCIe bus; larger systems use both the PCIe bus and the InfiniBand bus); and (b) two layers of the Maxeler system software are installed at the CPU host, MaxCompiler RT (to help in the compilation process) and Maxeler OS (to integrate the host and the accelerator). The host CPU can be of any type: (a) it can be a small laptop or it can be a large supercomputer worth millions of dollars. The host CPU can run any operating system: (a) it can be a Unix/Linux or Windows, or (b) it can be MPI or OpenMP.

Often times, labs run by uninformed leaders claim that they cannot use DataFlow accelerator because they use ready-to-use software packages for a given applications. This statement is simply not correct. A Maxeler accelerator can work with any of the above; one only has to be willing to dedicate some time and effort to small but appropriate code changes. Namely, conditionally speaking, all loops from the host applications have to be removed, and instead of each loop, two specific instructions have to be inserted into the host applications: **PUT** data into the Maxeler accelerator and **GET** results from the Maxeler accelerator. In addition, for each migrated loop, a kernel program in MaxJava has to be written (MaxJava is an extension of classical Java). It defines the details of the path to be used by the data corresponding to the migrated loop. Of course, for each kernel, one also has to write a kernel test program. Figure 1 implies that two loops are migrated from the host application onto the Maxeler accelerator. In general, if the number of the migrated loops is **n**, the number of additional MaxJava programs to write is **2n**. In addition, one has to write a manager program in MaxJava, which integrates all the kernels together with the input data stream and the output result stream. As it will be seen later, two more MaxJava programs have to be written, as well: The simulator descriptor program and the real hardware descriptor program. Consequently, the total number of MaxJava programs to write is **2n+3**. Please note that **n** is the number of loops to migrate; not the number of loops in the host application. It is better for some of the applications loops to be executed on the host (more about what to migrate to Maxeler and what to stay on CPU, later in this text).

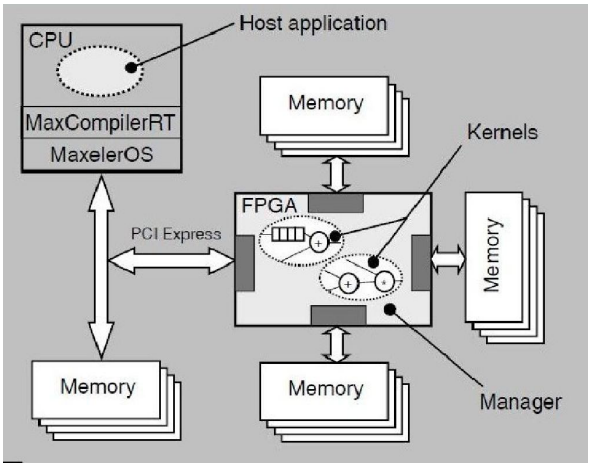


Figure 1: Generic acceleration architecture

One way to look at differences between ControlFlow and DataFlow is presented in Figure 2. For ControlFlow, a program, originality written in C, C++, Fortran, etc., is compiled down to the executable level and then executed on a machine of the Von Neumann architecture, of the single core, multi core, or many core type. In DataFlow, as indicated before, the original coding in C, C++, Fortran, etc., is modified (selected loops substituted with PUT and GET) and then an extension of the Java language is used to write kernels for DataFlow. After the Java kernels are compiled (by MaxCompiler), a configuration file is formed (MaxFile), and it performs the needed reconfiguration. The right-hand-side of Figure 2 refers to a case with eight loops migrated. That is why we have eight kernels in the figure (eight

DataFlow cores blocks). The manager file describes the DataFlow between kernels, plus the flow into the kernels, and the flow out of the kernels.

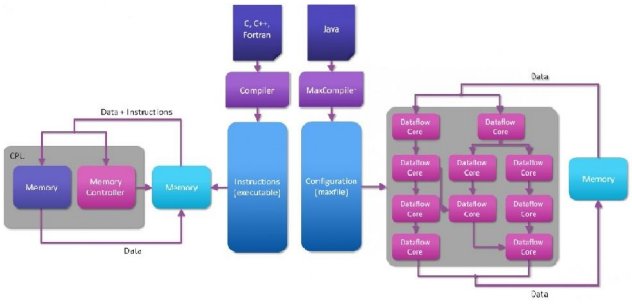
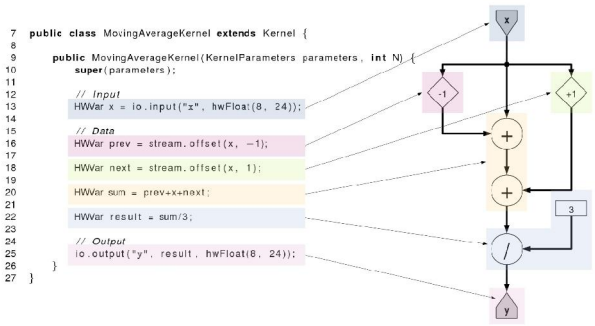


Figure 2: ControlFlow vs. DataFlow

As indicated before, the host code in C has to be modified, plus additional **2n+3** Java programs have to be written. The most sophisticated one of the Java programs is, as mentioned before, the kernel. Figure 3 shows an example of the kernel for moving average, using the average of the width three. This structure is typical for mathematical models using convolutions. This is a public class that defines a number of new hardware variables (HWVar). In the MaxJ (Maxeler Java), we have two types of variables: (a) The traditional Java variables that control the compilation process; and (b) The newly developed hardware developers that control the runtime signal flow. In this figure, **x** is a hardware variable. Before it reaches the Maxeler system, it is denoted as **"x"**. After it enters the Maxeler system, it is denoted as **x**. The Maxeler Java includes a contract that defines the format of variables: **hwFloat(8, 24)** refers to floating point with an exponent of 8 bits and a mantissa of 24 bits. The rest of the figure is self-explanatory.



```
7  public class MovingAverageKernel extends Kernel {
8
9      public MovingAverageKernel(KernelParameters parameters, int N) {
10         super(parameters);
11
12         // Input
13         HWVar x = io.input("x", hwFloat(8, 24));
14
15         // Data
16         HWVar prev = stream.offset(x, -1);
17
18         HWVar next = stream.offset(x, 1);
19
20         HWVar sum = prev+x+next;
21
22         HWVar result = sum/3;
23
24         // Output
25         io.output("y", result, hwFloat(8, 24));
26     }
27 }
```


Figure 3: DataFlow programming

One possible question is why Java. The answer is related to minimal Kolmogorov complexity.

Figure 4 presents an example. The host application written in C includes a "**for**" loop. As it can be seen from the figure, the loop is deleted and instead of it, a **stream_data** function is included, which is nothing else but the previously mentioned **PUT** and **GET**. The parameter **device** is a handle pointing to the structure that defines the communication mechanism to be used for data transfer, which is the PCIe bus. Parameter "**a**" defines an array in the host PC.

The manager program, written in MaxJava, consists of three parts. The first part is used of initializing the **Manager** object with a name and a type of Maxeler card to be used. The second part tells compiler where to find the kernel and also specifies that the memory structure **A** will be streamed to the Maxeler via the PCIe bus, while the results array **B** will be stored into the memory block on the linearly organized DRAM memory of the MAX2 card. The third part builds the software system configuration with previously initialized contents, connecting the manager and the kernel. It tells the compiler that the manager program will be responsible for looping on a MAX2 card.

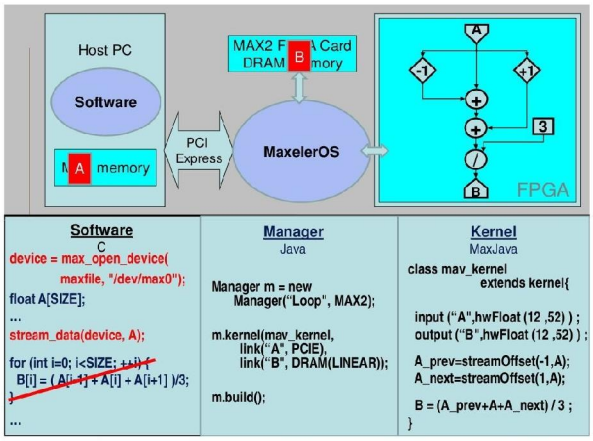


Figure 4: Connecting a host, a manager, and a kernel

The kernel program, also written in MaxJava, defines the format of data in structures **a** and **b**, which is hardware float **hwFloat(12, 52)**. It also defines computation to be performed by the Maxeler card during the DataFlow.

The MaxCompiler assumes the following input: (a) One or more Kernel programs (.java); (b) One Manager program (.java); and (c) One modified Application program (.c). Of course, the modified application program means a program with loops substituted by the **stream_data** structure described previously. The Kernel(s) and the Manager are compiled with the help of **MaxelerOS** construct used to help build hardware. However, the actual building of hardware is done with Xilinx

tools that do: (a) Synthesis; (b) Simulation; (c) Mapping, placement, and routing; and (d) Resource estimation. This means, it is not enough only to have the Maxeler system software provided by the MAXUP initiative (MAXeler University Program) free of charge (if the specific conditions hold). One also has to have the appropriate Xilinx tool, which is also obtainable free of charge. The Xilinx hardware building tool generates: (a) Simulation reports; (b) Resource usage reports; and the most importantly (c) The HW Accelerator code (.max). Finally, the Maxeler **Compiler**/**Linker** combines together the HW Accelerator code (.max) and the host application code (.c) and produces the application executable. Once this executable is applied to the Maxeler card, it checks if the card is already configured. If not, the acceleration has to take place, which takes several hours. If yes, the execution can start in a matter of seconds (the time to download zeros and ones of the application executables to the memory). What gets downloaded to the Maxeler card is .max file.

Figure 5, using a simple mathematical model, tries to explain the difference or differences between multi-core (like Intel), many-core (like nVidia), and DataFlow (like Maxeler). In all three cases, the following assumption holds: (a) Software includes enough parallelism to keep all cores busy; and (b) The only limiting factor for the exploitation of parallelism is the number of cores.

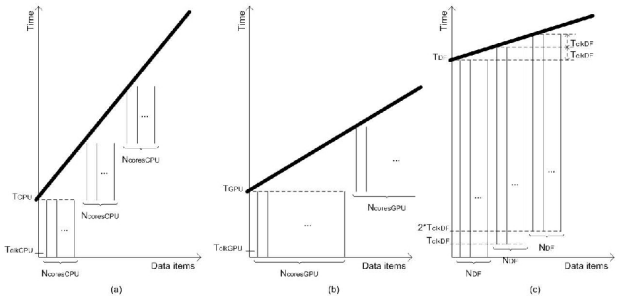


Figure 5: Acceleration graphs

The curve (a) refers to a multi-core with **n** cores. In figure (b), the number of cores is assumed to be 1000 **n**. In Figure (c), we assume a Maxeler system with one million logic blocks. Please note that the clock of the multi-core case (**TclkCPU**) is a smaller compared to the clock of the many- core (**TclkGPU**), while both are much smaller than the clock of the DataFlow (**TclkDF**). Also note that multi-core line is the steepest, while the DataFlow line is the least steep. The explanation to follow implies that we have a loop with **N** iterations and NOPS operation per iteration. The value of **N** is the same for all three cases. The value of **NOPS** differs for a multi -core (the smallest, since operations are relatively complex), for many-core (larger, since operations are reduced in complexity), and DataFlow (the smallest, since logic blocks are tiny). Parameter **CCPU** refers to number of clock periods for one operation in multi-core. Parameter **CGPU** refers to the number of clock periods for one operation in many-core (**CGPU** is higher than **CCPU,** since the GPU operations are simpler than CPU operations). Parameter **CDF** has analogous meaning for DataFlow. Parameter **TclkCPU** refers to the clock durations of multi-core. Parameter **TclkGPU** refers to the clock durations of many-core, and is larger, so that 1000 cores dissipate less. Parameter **TclkDF** refers to the clock durations of DataFlow and is the largest of the three, so DataFlow dissipates the least.

Parameter **NcoresCPU** refers to the number of cores in multi-core (assume it is equal to **n**). Parameter **NcoresGPU** refers to the number of cores in many-core (assume it is equal to 1000 **n**). To calculate the execution time of multi-core and many-core (TCPU and **TGPU**), in the first case we divide by n, and in the second case we divide by 1000 n. Consequently, at the first glance, **TGPU** will be much smaller than **TCPU**, which is not the case, because numerator of the **TCPU** formula is much smaller than the numerator of the **TGPU**. Therefore, the speed of the two technologies depends on the applications. For some applications, **TCPU** > **TGPU**, and for some **TGPU** > **TCPU**.

The loop execution time on the DataFlow (**TDF**) has two parts: (a) The part until the first partial result comes out (**TDF**); and (b) The part for the remaining **N-1** partial results; until the final result. It is noticeable that the time until the first partial result comes out is the largest for a DataFlow computer. This fact may restrict DataFlow architectures from a relatively small set of time-critical applications (like radar transponders in military aviation, etc.). Parameter **NDF** refers to the number of blocks in the DataFlow hardware (assume it is equal to million **n**).

Anecdotally speaking, processing using a dual-core can be treated as plowing with two horses. The direction of the moving of the horses is controlled by the plow driver. We put our wishes into the brain of the driver (we write a program to control the flow of processing). For the driver to be able to understand our wishes, his brain has to be equipped with appropriate Von Neumann

constructs (like control unit, arithmetic and logic units, program counters), and for him to be able to understand all of that fast enough, his brain has to be equipped with appropriate acceleration constructs (like cache memories, predictors, on-chip hardware accelerators etc.).

The many-core case can be compared with plowing using 2000 chicken instead of two horses. The question is what is better. The answer depends on the application.

Note that plowing with chickens includes multiple drivers (if the CUDA programming model is used) and also a dispatcher (if the rCUDA programming model is used).

In the case of DataFlow we plow with what? With two million ants, each ant carrying on its shoulders a backpack full of data.

In order to execute an algorithm on a DataFlow machine, we first have to download the big data into the backpacks of the ants' colony. Then, a programmer is asked to write a bunch of kernels. Finally, we now have somehow to motivate the ants to move into the desired direction through the field that programmers have configured for their progress (the kernels are mapped onto an FPGA structure – in field programmable data array).

How do we motivate them to move into the desired direction? We have to put a bowl of marmalade on the other side of the field. They smell the marmalade, and then they move. Of course, the marmalade here symbolizes the voltage difference between the input and the output of the FPGA structure. You remember, here we do not write a program to control the DataFlow. Here we write a program to configure the hardware. Therefore we can control the DataFlow only with the voltage difference.

III. ABOUT THE LATTICE-BOLTZMANN ALGORITHM

Simulation of fluid dynamics can be done using conventional CFD methods. In this case, fluid equations must be constructed (e.g. Navier-Stokes equations - PDE). These equations are usually being solved by using discrete approximation of PDE.

N. Korolija at al. explained in his paper [11] one approach do divide volume into finite elements, and simulate air flow through the wind-tunnel by calculating approximation of PDE in around 200 iterations.

After this, numerical integration is used for solving the equations on a given mesh. And then, PDE boundary conditions are applied.

Lattice based equations are already in discrete form and require no further approximation. Numerical integration is done by solving on lattices and applying kinetic based BC [12].

Lattice-Boltzmann (abbreviated LB) method is a discrete method that is used to simulate fluid flow. This method observes the fluid as a set of fictitious particles and by studying the dynamics of these particles; the fluid flow is modeled on the macroscopic level. A single distribution function is defined that describes the propagation of particles through the domain. This function depends on the state of neighboring cells and it has an identical form for all the cells. The state of all cells is updated synchronously, through a series of iterations, in discrete time steps.

LB method is successfully applied in simulating flows through micro and nano channels [13], modeling blood flow, including simulation of cancer invasion and tumor cell migration [14], analysis of leukocytes-erythrocytes interaction [15] as well as in many other areas in biomedicine.

In this paper Bhatnagar-Gross-Krook (BGK) model (using the single relaxation time approximation) is considered [16]. The equation that represents LB numerical scheme and that is used in all solvers based on LB method is given by:

$$f_i(\mathbf{x}+\boldsymbol{\xi}_i,t+1)-f_i(\mathbf{x},t)=-\frac{1}{\bar{\tau}}\left(f_i(\mathbf{x},t)-f_i^{eq}(\mathbf{x},t)\right)+\left(1-\frac{1}{2\bar{\tau}}\right)F_i$$

where $\bar{\tau}$ is the relaxation time (the average time period between two collisions), f is the distribution function, f^{eq} is the equilibrium distribution function and F_i is the external force term.

IV. THE MAXELER IMPLEMENTATION OF THE LATTICE-BOLTZMANN ALGORITHM

Figure 6 depicts The Lattice-Boltzmann algorithm diagram. Functions stream, apply_BCs, and collide are repeated for a number of iterations times. In this work, number of iterations is set to 10000. Function apply_BCs is not time critical as stream and collide are. The host sends the data to kernels that perform functions of stream and collide. The apply_BCs is done at the host.

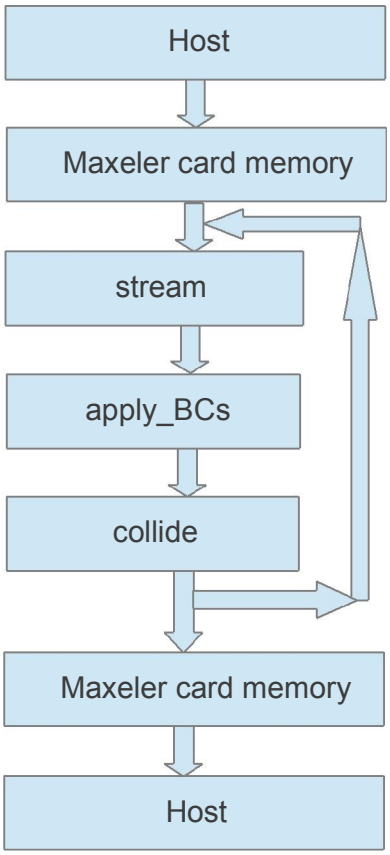


Figure 6: The Lattice-Boltzmann algorithm diagram

Function stream needs to calculate values of all elements of eight matrices depending on neighboring elements. If the dimensions of the matrix are both n, the complexity of this function will be $O(n^2)$. Function apply_BCs needs to calculate values of elements only from edges of matrices. If the dimensions of the matrix are both n, the complexity of this function will be O(n). Function collide needs to calculate values of all elements of eight matrices. These elements are calculated by determining components of speeds in all points. It is the most time-demanding function. If the dimensions of the matrix are both n, the complexity of this function will be $O(n^2)$. Since implementation on a specific architecture was not the main goal of this paper, only part of the kernel Kernel6 code for the function collide is given in the Appendix 8.2.

In order for the algorithm to be able to function as described, a modification to the implemented kernels had to be added. The limitation of using a maximum of eight streams from the DRAM memory to the Maxeler card and from the Maxeler card to the DRAM memory made it necessary to implement additional kernels enabling splitting four streams into eight streams, and vice versa. For example, Kernel6 that implements the function collide receives streams from the kernel Kernel6Output. The code for the Kernel6Output is given in the Appendix 8.1.

V. RESULTS

5.1. Main results

The main goal of this project was to accelerate execution of the Lattice-Boltzmann method using special hardware. Figure 7 depicts execution time of the algorithm using processor Intel I5 650 (3.2GHz) and execution time using the Maxeler card. When using Maxeler card, the processor was idle during the execution of the algorithm. It can be seen that the Maxeler card performs better for any number of iterations between 1 and 100000.

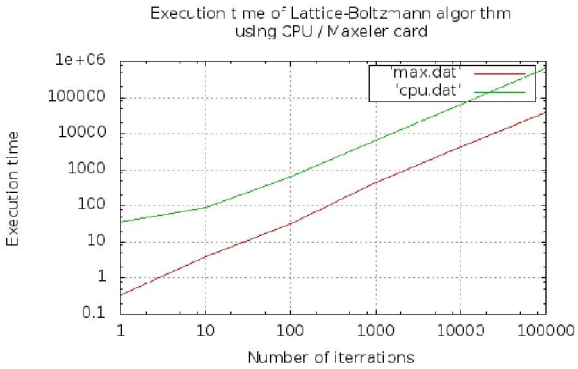


Figure 7: Compared Lattice-Boltzmann algorithm execution time to the CPU execution time

Figure 8 represents acceleration of the Maxeler card comparing to Intel I5 650 processor. From this figure, it can be seen that the acceleration is much higher when the algorithm is run for only one iteration.

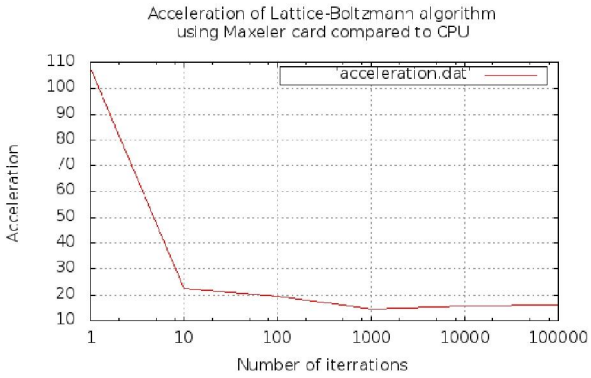


Figure 8: The Lattice-Boltzmann algorithm acceleration using Maxeler card

5.2. Discussion of results

Although the execution time of the Maxeler card increases linearly with increasing number of iterations, the acceleration is not a constant. The reason for this is that CPU uses cache, making it execute e.g. 10 iterations only around 2.8 times slower than one iteration. This result suggests that the acceleration of the Maxeler card comparing to running the algorithm on a processor could be around 100, rather than around 15.

VI. CONCLUSIONS

This paper describes an implementation of a Lattice-Boltzmann algorithm using Maxeler technologies. It includes results of implementation and optimization of the code for achieving greater speed, as well as lower power consumption. Results show the improvement in performances in both fields, comparing to implementation of the same algorithm for running on CPU. Execution time in average reduces by the factor of around 15. When CPU is not using cache, the acceleration can be even more than 100.

Having in mind the above mentioned, it can be concluded that using the scheduled DataFlow paradigm, simulations of fluid flow can be done faster than before, reducing the energy consumption at the same time.

By implementing the algorithm using the Maxeler technologies, a new field of programming problems occurred. These include running more than one kernel on the Maxeler card in parallel, accelerating execution of time-critical code. Another issue includes dividing the problem onto a cluster of Maxeler technologies nodes.

VII. ACKNOWLEDGEMENT

We thank Oskar Mencer for providing us full support in solving emerged issues in programming Maxeler card, as well as providing us with an extensive set of materials, describing the programming model that Maxeler card uses.

REFERENCES

[1] R. Giorgi, Z. Popovic, N. Puzovic, “DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems,” in Proc. IEEE SBAC-PAD, ISBN:0-7695-23014-1, Gramado, Brasil, Oct. 2007, pp. 263-270.

[2] MJ Flynn, O Pell, O Mencer, "DataFlow supercomputing," 22nd International Conference on Field Programmable Logic and Applications (FPL), Aug. 2012, pp. 1-3.

[3] “DataFlow Programming with Maxcompiler,” Maxeler Technologies, Version 2012.1, 2012.

[4] Mencer, O., “DataFlow Programming with Maxcompiler,” Maxeler Technologies, London, UK, Version 2012.1, 2012

[5] D.G. Feitelson, "On the Interpretation of Top500 Data," Intl. J. High Performance Computing Applications, vol. 13, no. 2, 1999, pp. 146-153.

[6] Yokokawa, M., Shoji, F.; Uno, A.; Kurokawa, M.; Watanabe, T., "The K computer: Japanese next-generation supercomputer development project," IEEE Xplore International Symposium on Low Power Electronics and Design (ISLPED) 2011, Page(s): 371 – 372, Aug 2011.

[7] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, P. Hanrahan, "Sequoia: programming the memory hierarchy," Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Article No. 83, ACM, New York, ISBN:0-7695-2700-0, doi 10.1145/1188455.1188543.

[8] Diede, T., Hagenmaier, C.F.; Miranker, G.S.; Rubinstein, J.J.; Worley, W.S., Jr., “The Titan Graphics Supercomputer architecture,” Computer, Volume: 21, issue 9, Page(s): 13 – 30, ISSN: 0018-9162, Sep 1988.

[9] J. J. Dongarra, P. Luszczek1, A. Petitet2, "The LINPACK Benchmark: past, present and future," Concurrency and Computation: Practice and Experience, Vol 15, Issue 9, DOI: 10.1002/cpe.728.

[10] Flynn, M., Mencer, O., Milutinovic, V., Rakocevic, G., Stenstrom, P., Trobec, R., and Valero, M., “Moving from petaflops (on simple benchmarks) to petadata per unit of time and power (on sophisticated benchmarks),” Communications of the ACM, May 2013.

[11] O Mencer, "Maximum Performance Computing for Exascale Applications," Embedded Computer Systems (SAMOS), 2012.

[12] NG Korolija, V Milutinovic, S Milosevic, "Accelerating Conjugate Gradient Solver: Temporal Versus Spatial Data," Transactions on Advanced Research, January 2007, Vol 3. No 1, ISSN 1820 – 4511.

[13] S. Chen, GD Doolen, “Lattice Boltzmann Method for Fluid Flows,” Annual Review of Fluid Mechanics, Vol. 30: 329-364, January 1998.

[14] V. Milutinovic, “Surviving the Design of Microprocessor and Multiprocessor Systems: Lessons Learned,” Wiley-Blackwell, ISBN-10: 0471357286, ISBN-13: 978-0471357285, August 2000.

[15] Chen, S., and Tian, Z., (2009) Simulation of microchannel flow by lattice Boltzmann method, *Physica A*, 388(23), pp. 4803-4810.

[16] Sequeira, A., Artoli, A. M.,, Silva-Herdade, A. S., and Saldanha, C. (2009) Leukocytes dynamics in microcirculation under shear-thinning blood flow, *Computers and Mathematics with Applications*, 58(5), pp. 1035-1044.

VIII. APPENDIX: CODE

8.1. Kernel Kernel6Output

```
public class Kernel6Output extends
Kernel { public Kernel6Output(
    KernelParameters parameters)
{ super(parameters);
HWVar cnt =
control.count.simpleCounter(1); HWVar
paran = cnt < 1;

HWVar f1 = io.input("f1k6u", hwFloat(8, 24));
HWVar f2 = io.input("f2k6u", hwFloat(8, 24));
HWVar f3 = io.input("f3k6u", hwFloat(8, 24));
HWVar f4 = io.input("f4k6u", hwFloat(8, 24));
HWVar f5 = io.input("f5k6u", hwFloat(8, 24));
HWVar f6 = io.input("f6k6u", hwFloat(8, 24));
HWVar f7 = io.input("f7k6u", hwFloat(8, 24));
HWVar f8 = io.input("f8k6u", hwFloat(8, 24));

io.output("f1k6i", (paran?f1:f5), hwFloat(8,
24)); io.output("f2k6i", (paran?f2:f6),
hwFloat(8, 24)); io.output("f3k6i",
(paran?f3:f7), hwFloat(8, 24));
io.output("f4k6i", (paran?f4:f8), hwFloat(8,
24));
 }
}
```

8.2. Part of the kernel Kernel6

```
HWVar v2x = vx * vx;
HWVar v2y = vy * vy;
HWVar v_sq_term = 1.5f*(v2x + v2y);
// Evaluate the local equilibrium f values in all directions
HWVar vxmvy = vx - vy;
HWVar vxpvy = vx + vy;
HWVar rortau = ro * rtau;
HWVar rortaufaceq2 = rortau * faceq2;
HWVar rortaufaceq3 = rortau * faceq3;
HWVar vxpvyp3 = 3.f*vxpvy; HWVar
vxmvyp3 = 3.f*vxmvy; HWVar vxp3
= 3.f*vx;
HWVar vyp3 = 3.f*vy;
HWVar v2xp45 =
4.5f*v2x; HWVar v2yp45
= 4.5f*v2y;
HWVar mv_sq_term = 1.f - v_sq_term;
HWVar mv_sq_termpv2xp45 = mv_sq_term +
v2xp45; HWVar mv_sq_termpv2yp45 = mv_sq_term
+ v2yp45; HWVar vxpvyp45vxpvy =
4.5f*vxpvy*vxpvy; HWVar vxmvyp45vxmvy =
4.5f*vxmvy*vxmvy; HWVar
mv_sq_termpvxpvyp45vxpvy =
  mv_sq_term + vxpvyp45vxpvy;
HWVar mv_sq_termpvxmvyp45vxmvy
= mv_sq_term - vxmvyp45vxmvy;
HWVar f1eq = rortaufaceq2 *
(mv_sq_termpv2xp45 + vxp3);
HWVar f2eq = rortaufaceq2 *
(mv_sq_termpv2yp45 + vyp3);
HWVar f3eq = rortaufaceq2 *
(mv_sq_termpv2xp45 - vxp3);
HWVar f4eq = rortaufaceq2 *
(mv_sq_termpv2yp45 - vyp3);
HWVar f5eq = rortaufaceq3 *
 (mv_sq_termpvxpvyp45vxpvy + vxpvyp3);
HWVar f6eq = rortaufaceq3 *
(mv_sq_termpvxmvyp45vxmvy - vxmvyp3);
HWVar f7eq = rortaufaceq3 *
(mv_sq_termpvxpvyp45vxpvy - vxpvyp3);
HWVar f8eq = rortaufaceq3 *
(mv_sq_termpvxmvyp45vxmvy + vxmvyp3);
// Simulate collisions by "relaxing"
 //toward the local
equilibrium f1 = rtau1 * f1 +
f1eq;
f2 = rtau1 * f2 + f2eq;
f3 = rtau1 * f3 + f3eq;
f4 = rtau1 * f4 + f4eq;
f5 = rtau1 * f5 + f5eq;
f6 = rtau1 * f6 + f6eq;
f7 = rtau1 * f7 + f7eq;
f8 = rtau1 * f8 + f8eq;
```

Reviewers:

Australia
Abramov, Vyacheslav; Monash University
Begg, Rezaul; Victoria University
Bem, Derek; University of Western Sydney
Betts, Christopher; Pegacat Computing Pty. Ltd.
Buyya, Rajkumar; The University of Melbourne
Chapman, Judith; Australian University Limited
Chen, Yi-Ping Phoebe; Deakin University
Hammond, Mark; Flinders University
Henman, Paul; University of Queensland
Palmisano, Stephen; University of Wollongong
Ristic, Branko; Science and Technology Organisation
Sajjanhar, Atul; Deakin University
Sidhu, Amandeep; University of Technology, Sydney
Sudweeks, Fay; Murdoch University

Austria
Derntl, Michael; University of Vienna
Hug, Theo; University of Innsbruck
Loidl, Susanne; Johannes Kepler University Linz
Stockinger, Heinz; University of Vienna
Sutter, Matthias; University of Innsbruck

Brazil
Parracho, Annibal; Universidade Federal Fluminense
Traina, Agma; University of Sao Paulo
Traina, Caetano; University of Sao Paulo
Vicari, Rosa; Federal University of Rio Grande

Belgium
Huang, Ping; European Commission

Canada
Fung, Benjamin; Simon Fraser University
Grayson, Paul; York University
Gray, Bette; Alberta Education
Memmi, Daniel; UQAM
Neti, Sangeeta; University of Victoria
Nickull, Duane; Adobe Systems, Inc.
Ollivier-Gooch, Carl; The University of British Columbia
Paulin, Michele; Concordia University
Plaisent, Michel; University of Quebec
Reid, Keith; Ontario Ministry og Agriculture
Shewchenko, Nicholas; Biokinetics and Associates
Steffan, Gregory; University of Toronto
Vandenberghe, Christian; HEC Montreal

Czech Republic
Kala, Zdenek; Brno University of Technology
Korab, Vojtech; Brno University of technology
Lhotska, Lenka; Czech Technical University

Finland
Lahdelma, Risto; University of Turku
Salminen, Pekka; University of Jyvaskyla

France
Cardey, Sylviane; University of Franche-Comte
Klinger, Evelyne; LTCI – ENST, Paris
Roche, Christophe; University of Savoie
Valette, Robert; LAAS - CNRS

Germany
Accorsi, Rafael; University of Freiburg
Glatzer, Wolfgang; Goethe-University
Gradmann, Stefan; Universitat Hamburg
Groll, Andre; University of Siegen
Klamma, Ralf; RWTH Aachen University
Wurtz, Rolf P.; Ruhr-Universitat Bochum

India
Pareek, Deepak; Technology4Development
Scaria, Vinod; Institute of Integrative Biology
Shah, Mugdha; Mansukhlal Svayam

Ireland
Eisenberg, Jacob; University College Dublin

Israel
Feintuch, Uri; Hadassah-Hebrew University

Italy
Badia, Leonardo; IMT Institute for Advanced Studies
Berrittella, Maria; University of Palermo
Carpaneto, Enrico; Politecnico di Torino

Japan
Hattori, Yasunao; Shimane University
Livingston, Paisley; Linghan University
Srinivas, Hari; Global Development Research Center
Obayashi, Shigeru; Institute of Fluid Science, Tohoku University

Netherlands
Mills, Melinda C.; University of Groningen
Pires, Luís Ferreira; University of Twente

New Zealand
Anderson, Tim; Van Der Veer Institute

Portugal
Cardoso, Jorge; University of Madeira
Natividade, Eduardo; Polytechnic Institute of Coimbra
Oliveira, Eugenio; University of Porto

Singapore
Tan, Fock-Lai; Nanyang Technological University

South Korea
Kwon, Wook Hyun; Seoul National University

Spain
Barrera, Juan Pablo Soto; University of Castilla
Gonzalez, Evelio J.; University of La Laguna
Perez, Juan Mendez; Universidad de La Laguna
Royuela, Vicente; Universidad de Barcelona
Vizcaino, Aurora; University of Castilla-La Mancha
Vilarrasa, Clelia Colombo; Open University of Catalonia

Sweden
Johansson, Mats; Royal Institute of Technology

Switzerland
Niinimaki, Marko; Helsinki Institute of Physics
Pletka, Roman; AdNovum Informatik AG
Rizzotti, Sven; University of Basel
Specht, Matthias; University of Zurich

Taiwan
Lin, Hsiung Cheng; Chienkuo Technology University
Shyu, Yuh-Huei; Tamkang University
Sue, Chuan-Ching; National Cheng Kung University

United Kingdom
Ariwa, Ezendu; London Metropolitan University
Biggam, John; Glasgow Caledonian University
Coleman, Shirley; University of Newcastle
Conole, Grainne; University of Southampton
Dorfler, Viktor; Strathclyde University
Engelmann, Dirk; University of London
Eze, Emmanuel; University of Hull
Forrester, John; Stockholm Environment Institute
Jensen, Jens; STFC Rutherford Appleton Laboratory
Kolovos, Dimitrios S.; The University of York
McBurney, Peter; University of Liverpool
Vetta, Atam; Oxford Brookes University
WHYTE, William Stewart; University of Leeds
Xie, Changwen; Wicks and Wilson Limited

USA
Bach, Eric; University of Wisconsin
Bolzendahl, Catherine; University of California
Bussler, Christoph; Cisco Systems, Inc.
Charpentier, Michel; University of New Hampshire
Chong, Stephen; Cornell University
Collison, George; The Concord Consortium
DeWeaver, Eric; University of Wisconsin - Madison
Gans, Eric; University of California
Gill, Sam; San Francisco State University
Hunter, Lynette; University of California Davis
Iceland, John; University of Maryland
Kaplan, Samantha W.; University of Wisconsin
Langou, Julien; The University of Tennessee
Liu, Yuliang; Southern Illinois University Edwardsville
Lok, Benjamin; University of Florida
Minh, Chi Cao; Stanford University
Morrissey, Robert; The University of Chicago
Mui, Lik; Google, Inc
Rizzo, Albert ; University of Southern California
Rosenberg, Jonathan M. ; University of Maryland
Shaffer, Cliff ; Virginia Tech
Sherman, Elaine; Hofstra University
Snyder, David F.; Texas State University
Song, Zhe; University of Iowa
Wei, Chen; Intelligent Automation, Inc.
Yu, Zhiyi; University of California

Authors of papers are responsible for the contents and layout of their papers.

Welcome to IPSI BgD Conferences and Journals!

http://www.internetconferences.net

http://www.internetjournals.net

CIP – Katalogizacija u publikaciji
Narodna biblioteka Srbije, Beograd
ISSN 1820 – 4511
The IPSI BGD Transactions on Internet Research
COBISS.SR - ID 119128844