Acta Electrotechnica et Informatica, Vol. 18, No. 3, 2018, 47–56, DOI: 10.15546/aeei-2018-0025

EXPERIMENTAL COMPARISON OF MATRIX ALGORITHMS FOR DATAFLOW COMPUTER ARCHITECTURE

Jurij MIHELIČ, Uroš ČIBEJ
Laboratory of algorithmics, Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, Ljubljana, Slovenia, E-mail: Jurij.Mihelic@fri.uni-lj.si, UroS.Cibej@fri.uni-lj.si

ABSTRACT

In this paper we draw our attention to several algorithms for the dataflow computer paradigm, where the dataflow computation is used to augment the classical control-flow computation and, hence, strives to obtain an accelerated algorithm. Our main goal is to experimentally explore various dataflow techniques and features, which enable such an acceleration. Our focus is to resolve one of the most important challenges when designing a dataflow algorithm, which is to determine the best possible data choreography in the given context. In order to mitigate this challenge, we systematically enumerate and present possible techniques of various data choreographies. In particular, we focus our interest on the algorithms that use matrices and vectors as the underlaying data structure. We begin with simple algorithms such as matrix and vector multiplication, evaluation of polynomials as well as more advanced ones such as the simplex algorithm for solving linear programs. To evaluate the algorithms we compare their running-times as well as the dataflow resource consumption.

Keywords: *dataflow, choreography, matrix, algorithm, experiment, evaluation*

1. INTRODUCTION

The von Neumann architecture is pervasive in modern day computers, even though several alternatives exist [1]. One of them is the dataflow architecture [2], which was once viewed as a competitor to the von Neumann architecture, but is now considered more as a complementary one as it is now more often used to augment the classical control-flow paradigm in order to obtain algorithmic accelerations.

The dataflow computers were consider dead until recently, when technological advances, driven mainly by Maxeler [3], brought the dataflow paradigm back to life by making it not only competitive with the control-flow processors, but overtaking them in many aspects [4].

For example, in the era of BigData, the possibility to manipulate huge amounts of data while at the same time consuming significantly less energy than comparable solutions based on control-flow processors [5, 6] seems very lucrative.

Unfortunately, the speedups offered by the dataflow approach are not straightforward and the algorithms have to be carefully re-engineered since the majority of the current algorithms are tailored specifically for the control-flow architecture. Many successful examples of such engineering already exist in various application domains (mostly numerical computation). However, some successes were also achieved in semi-numerical applications such as sorting and simplex algorithms [7–9].

With this paper we try to shift the focus to computational problems and application domains where dataflow computation may not be deemed so successful due to several reasons such as low data reuse and short loop bodies, which basically results in less computation per input element. We strive to explore various dataflow techniques using the approaches from algorithm engineering [10], experimental algorithmics [11], and good practices [12] in order to show their practical applicability. Many of the results already presented in [13] are discussed and evaluated here in a much broader perspective.

In particular, our subject of interest are the algorithms that use matrices and vectors as the underlaying data structure (for example, for storing data and representing higher concepts such as polynomials or graphs). We look at these problems from a practical point of view, where our main method of investigation is based on experimental analysis, comparison and evaluation of various indicators obtained from the results of thorough experiments. In particular, we evaluate a running-time performance of the algorithms and we also give a discussion on the dataflow resource consumption, which is deemed important from the feasibility perspective.

Additionally, an important focus of this paper is to investigate various data representation techniques, i.e., how the data is stored in the main computer memory, as well as data streaming techniques, i.e., how the data is input into the dataflow engine. Our approach is systematic in the sense that we also consider approaches that seem to be unpromising or inefficient from a theoretical point of view in order to better pinpoint the differences between various techniques.

The paper is structured as follows. In the next section, we present the dataflow paradigm and computer architecture. We briefly compare it to the classical control-flow architecture and give advantages of augmenting the latter with the former. We also discuss the dataflow paradigm from the programmer's point of view. In Section 3 we first discuss the data representation in the computer main-memory, followed by a presentation of various data streaming options and techniques. Our main result is in Section 4 which contains experimental evaluation of the presented choreography techniques. We base our evaluation on several algorithms such as matrix multiplication and the simplex algorithm. Finally, Section 5 concludes the paper and gives some further directions.

2. DATAFLOW ARCHITECTURE

In this section we briefly describe the dataflow paradigm and architecture as implemented by Maxeler. As opposed to the control-flow architecture (see also Fig. 1), where the operations to be executed by the processor are delegated by the sequence of instructions, in the data-flow architecture (see also Fig. 2), the operation is executed when its operands are available. Hence, one of the main dataflow programming challenges is to organize the data in such a way that it is readily available and processed by the dataflow processor.

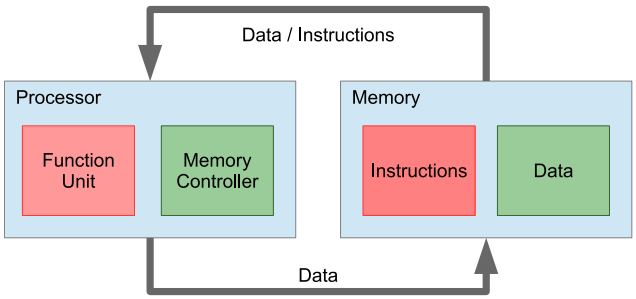


Fig. 1 A conceptual representation of the control-flow computer architecture

The dataflow architecture is based on the stream processing paradigm [2, 14]. In particular, the dataflow process has one or more inputs as well as outputs, where each is a stream of (primitive) values, i.e., characters, fixed- or floating-point numbers, etc. Such change in a way of data availability requires a change in thinking about an algorithm design.

In general, Maxeler's dataflow architecture consists of a set of dataflow engines. Each engine executes one or more *kernels*, i.e., self-contained computing modules with specified input and output. The execution of kernels is controlled from the control-flow computer, and a dataflow engines may be viewed at as a co-processors.

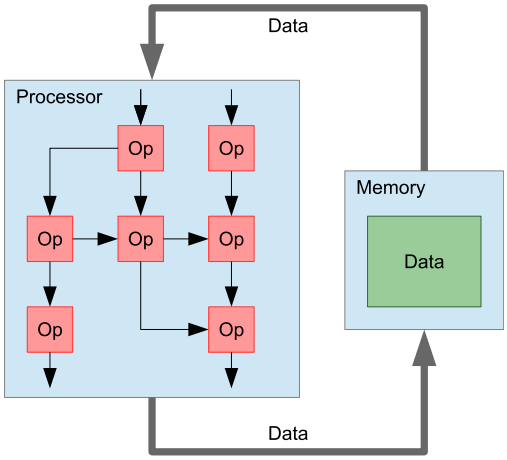


Fig. 2 A conceptual representation of the data-flow computer architecture. The representation of dataflow computation is greatly simplified.

Each dataflow computation can be viewed at as a directed graph, where nodes represent operations and edges represent paths for the data. An example of such a dataflow graph can be seen in Fig. 2.

In order to develop an efficient algorithm for the dataflow computer, the algorithm designer must carefully think about the distribution of work between the control-flow and the dataflow part. For the control-flow part, her job is to implement the code (usually in the C programming language) which controls the whole computational process. The most common implemented scenario is to obtain the input data, then pre-process it and send it to the dataflow engine. When the dataflow engine finishes the computation the result needs to be transfered back to the memory of the control-flow part, and finally, present the result to the user. Additionally, the control-flow part may also include a partial implementation of the algorithm.

Considering the dataflow part, her job is use the MaxJ programming language (a superset of Java) to implement each dataflow kernel. This can be viewed as a construction of the dataflow graph, where nodes correspond to operations and edges connect inputs and outputs of particular operations. The dataflow computations can be traced following the paths from the input to the output nodes.

Additionally, the implementor must provide an implementation of the so-called *manager* part, which configures the dataflow engine, i.e., specifies the interface to interact with the control-flow part, connects the data streams and interconnects the implemented kernels.

There are two main parallelization mechanisms available in the dataflow engines. The first is implicit in every dataflow operation as each operation is automatically *pipelined* by a compiler. Consequently, after initial delay to fill in the pipeline, one may be able to obtain a new result in each clock period.

The second must be explicitly considered by the implementor as there is a possibility to replicate the computation similarly to the thread-based control-flow parallelization. Each such replication is called a *pipe*. It can be seen as executing several kernels of the same type at once, each with its own element from the input data stream. We explain pipes in more details in Section 3.3.

Another acceleration technique is to put the data closer to the dataflow engine. Namely, instead of the main memory, one may use the large memory available for each dataflow unit and thus exploit its greater data throughput. Additionally, one may also consider using the fast memory, which is actually a part of each kernel and provides the fastest access times.

3. DATA REPRESENTATION AND STREAMING

In this section, we describe the representation of data, which is suitable for stream processing as used in the dataflow architecture. We focus on vectors and matrices and the way they are stored in the main memory of the computer. Additionally, we also give a discussion of various techniques of streaming the data to the dataflow engine. We base our techniques mostly on the matrix-vector and matrix-matrix multiplication problems.

3.1. Main-memory Storage

From a mathematical point of view the representation of vectors and matrices is not important, as they are considered conceptual objects. To refer to its elements, one simply uses subscript indices, i.e., v_i refers to the i-the element of the vector v while $m_{i,j}$ refers to the element in the i-th row and j-th column of the matrix m. However, in the computer science, the representation is of uttermost importance since it may have a profound effect on the algorithm performance due to the specifics of a particular computer architecture [10, 11].

Vectors are most often represented as a sequence (i.e., an array) of elements. In particular, if A denotes an array, then the i-th element of A is denoted simply by $a_i = A[i]$. (In all the examples we consider that indices start from 0.)

Another mathematical object that can be efficiently represented by a vector are polynomials: the coefficients (including zeros) are listed in a vector in the increasing order of term degrees. When there are many zero coefficients, such polynomials are called *sparse* polynomials, and both degrees and coefficients of the terms are listed either using two vectors or a vector of pairs.

Two main representations exist for the matrices: the *row-major* and *column-major* order [15]. In both, the elements are listed consecutively as they appear in the matrix, but in the former they follow the rows from top to bottom as well as from left to right inside each row, while in the latter they follow the columns from left to right as well as from top to bottom inside each row.

Let A denote the (zero-indexed) array to store the matrix; thus, we have $a_{i,j} = A[i \cdot n + j]$ in the row-major representation, and $a_{i,j} = A[i + j \cdot n]$ in the column-major representation.

It is straightforward to convert a matrix between the two representations using the matrix transposition operation. However, the in-place transposition for non-square matrices may be much more elaborate. If A is a matrix then A^T denotes a transposition of A. If A is represented in row-major order then A^T is its column-major representation and vice versa.

3.2. Rowwise Streaming

Assuming row-major representation, the easiest (from a programmer's point of view) is to stream the matrix elements as stored in the memory. This results in the data choreography as depicted in Fig. 4 a).

A disadvantage of such choreography is that it often causes a *static* loop, i.e., a loop depending only on the computation, in the resulting dataflow graph. Unfortunately, such loop results in an immediate slowdown (proportional to the length of the loop) of the whole kernel.

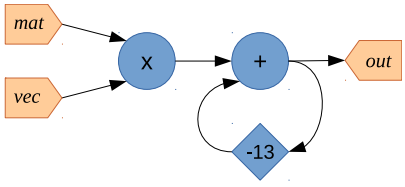


Fig. 3 A loop in the dataflow graph

For example, when multiplying a matrix with a vector, several dot products of row-vectors with a given vector are computed. Here, each addition operation in the dot product has to wait for its result to become available as it is reused as an input for the next operation. See also Fig. 3 for a simplified representation of the dataflow graph corresponding to a calculation of a dot product. Here, one element from each of the two input streams *mat* and *vec* is first multiplied and then result is added to the accumulated sum. However, the summation operation has a specific latency, thus its result is not immediately available to be used for the succeeding addition operation.

3.3. Replication of Computation

A common technique for creating dataflow implementations is, similarly to the thread-based control flow parallelization, to replicate a single stream computation (also called a *pipe*) within a kernel in order to process several elements at a time: p pipes have a p-fold potential speedup. The replication may mitigate a slowdown caused by a loop in the dataflow graph. Other factors, such as maximum bandwidth of PCIe (Peripheral Component Interconnect Express, [16]) bus may also effect the performance and potentially slow-down the computation even further.

Let us now briefly discuss two options for parallelization of rowwise processing, namely we discuss piped based replication in the *rowwise* and *columnwise* direction. Both options are graphically presented in Fig. 4 b) and c), respectively.

Notice that, using p pipes, the general processing order is still rowwise, i.e., the next p elements are taken in the rowwise direction; however, either p elements in the corresponding row (in the former) or column (in the latter) of the matrix are processed at a time. The rowwise direction of replication is straightforward to stream, i.e., no rearrangement of the matrix in the main memory is needed, while a small addition of code is needed in the dataflow kernel to add the final p accumulated sums. The columnwise direction requires a rearrangement of the matrix elements as they are stored in row-major order in the main memory, while the kernels are straightforward to implement.

3.4. Columnwise Streaming

Often a loop in the dataflow graph, caused by a dependence of consecutive operations, can be completely mitigated, e.g. in matrix-vector multiplication, using a columnwise processing, where the elements are accessed sequentially from the top to the bottom by columns, starting in the top-left corner and ending in the bottom-right corner. Assuming row-major representation in the main computer memory, the input matrix must be transposed before it is fed to the kernel. The data choreography is depicted in Fig. 4 d).

Obviously, the change in the choreography requires the change in the dataflow kernel. In particular, algorithms such as matrix-vector multiplication need to store the partial results of processing the previous column (i.e., accumulated sums), when processing the current one. Fortunately, in

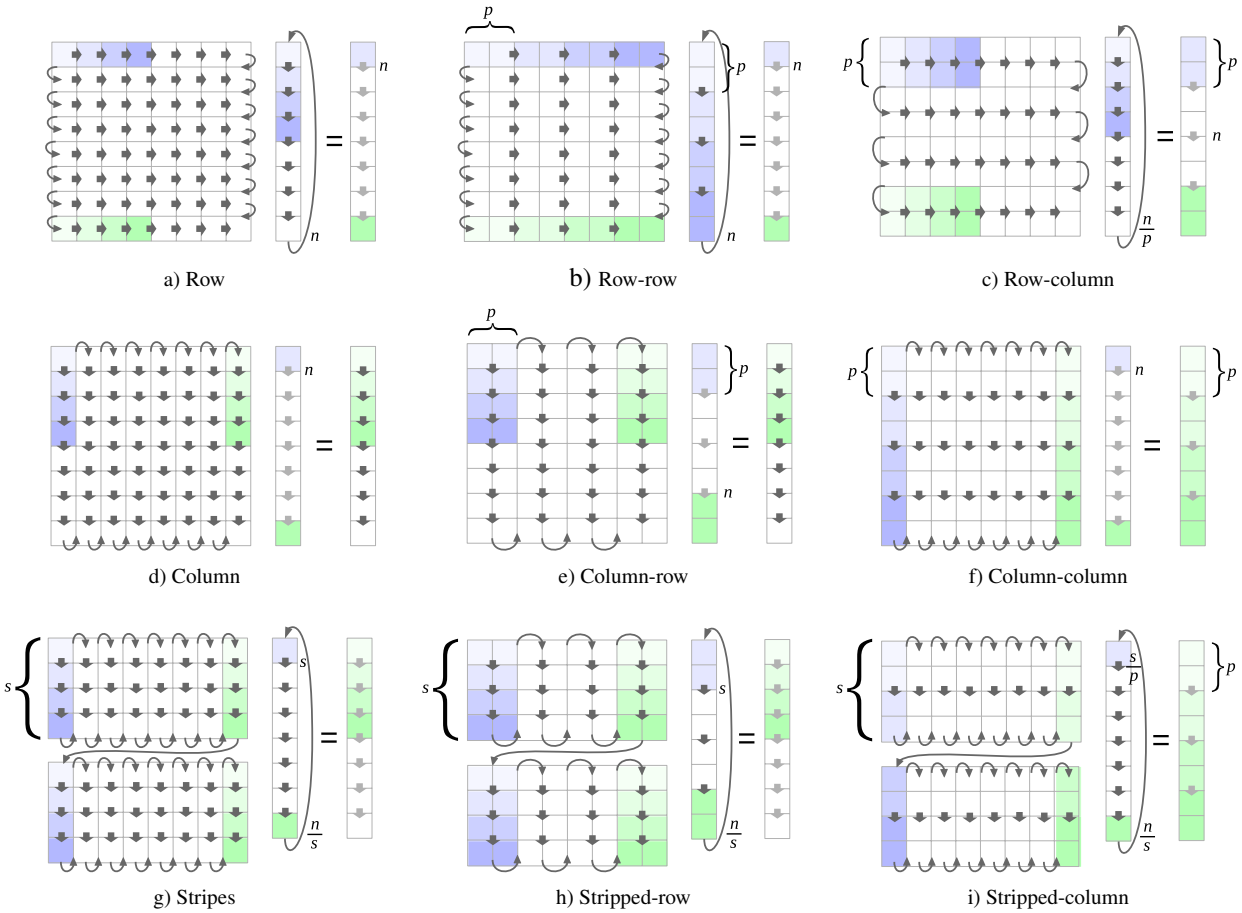


Fig. 4 Various data choreographies: rowwise (top row), columnwise (middle row), and stripes (bottom row), and replication of stream computation based od pipe parallelism: no replication (left column), rowwise replication (middle column), columnwise replication (right column).

columnwise processing, these accumulated values are already available when needed since the latency of the required operations is (usually) lower then the column size. Such dependence of data is called a *dynamic* loop in the dataflow terminology because it depends on the input data, e.g., the matrix size.

Again we explore the rowwise and columnwise possibility of computation replication with pipes; see also Fig. 4 e) and f) for the respective data choreographies. Here, the former processes the p elements at the time form the current row (the matrix has to be preprocessed to be streamed in the corresponding order) while the latter takes p elements at a time from the current column.

3.5. Stripped Streaming

Both techniques presented in the previous two sections have some drawbacks. The rowwise choreography contains a static loop to overcome the latency of the addition operation, while the columnwise choreography produces a long dynamic loop to store intermediate results. Hence, the former introduces a multi-fold kernel slowdown, and the latter consumes significant amounts of FPGA resources. To mitigate these two issues, we present another technique which is a combination of both techniques.

The main idea is to split a matrix into horizontal *stripes*, which are furthermore processed in a columnwise fashion. See Fig. 4 g) for a graphical representation of the choreography. Thus, the speed of the columnwise technique is retained, while the resource consumption is significantly reduced. Observe also, that analogous choreography with vertical stripes is also an option; see also Fig. 4 d) for an example.

Denote with s the stripe width, i.e., the number of elements from a particular column. Since the stripe length is n, we have sn elements in total for each stripe. There are n/s stripes and for every stripe the whole vector must be streamed. For the efficient re-streaming of the vector the fast memory of DFE unit may be used.

Processing of each stripe may be additionally parallelized using pipes. See Fig. 4 h) and i) for the two common techniques, both of which are based on the same ideas as already discussed in the previous two sections. Obviously, the parallelization technique is applied to each stripe separately. Finally, notice that, in rowwise parallelization $s \leq n$ and $p \leq n$ while in columnwise parallelization $p \leq s \leq n$ must hold.

3.6. Block-based Streaming

Now we present a choreography which is suitable for problems such as matrix multiplication. First let us explain what a block matrix is. It is a matrix that is interpreted as having been composed of (non-overlapping) submatrices, called *blocks*. One can also visualize such matrix with a collection of vertical and horizontal lines that partition it into a collection of smaller submatrices. In general, blocks can be of different sizes, and, thus, there are several possible ways to interpret a particular matrix as a block matrix.

Consider matrices $A = [a_{i,j}]$ of dimension $m \times l$ and $B = [b_{i,j}]$ of dimension $l \times n$. Now, a block matrix A with q row partitions and s column partitions, and a block matrix B with s row partitions and p column partitions are

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,s} \\ A_{1,1} & A_{1,2} & \cdots & A_{1,s} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q,1} & A_{q,2} & \cdots & A_{q,s} \end{bmatrix}$$

and

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,p} \\ B_{1,1} & B_{1,2} & \cdots & B_{1,p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{s,1} & B_{s,2} & \cdots & B_{s,p} \end{bmatrix},$$

respectively. The matrix product $C = AB$ can be formed blockwise, yielding a $m \times n$ matrix C with q row partitions and r column partitions, where

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} \cdot B_{k,j}$$

are block products and $1 \leq i \leq q$ and $1 \leq j \leq r$.

Block product can only be calculated if blocks of matrices A and B are compatible, i.e., when the number of columns of the block $A_{i,k}$ equals to the number of rows of the block $B_{k,j}$, for each $1 \leq i \leq q$, $1 \leq j \leq r$, and $1 \leq k \leq p$. In what follows we consider $m = n = l$ and $p = q = r$ as well as that p divides n; consequently, such blocks are always compatible.

4. EXPERIMENTAL COMPARISON

In this section we discuss several dataflow algorithms for various problems, all of which use matrices and/or vectors for their storage of data. Our main focus is on the experimental evaluation of different dataflow techniques and data choreographies. First, we give a brief description of settings we used for performing the experiments, followed by the results and their evaluation.

4.1. Experimental Background

For the experiments we used Maxeler's Vectis MAX3424A PCI-express extension card, which contains dataflow unit based on Xilinx Virtex 6 SXT475 field-programmable gate array. The control-flow part of the computer contained Intel i7-6700K processor with 8 MB and 64 GB of cache and main memory, respectively.

Test programs are written in the C programming language and compiled using the highest optimization level, while the dataflow kernels were programmed in the MaxJ programming language. Benchmarking and results visualization was automated using scripts. The source code is publicly available in several GitHub repositories [17] or upon a request to the authors. For the indicator of algorithm efficiency we used wall-clock time.

4.2. Matrix-Vector Multiplication

First we discuss the problem of multiplying a matrix with a vector from the dataflow perspective. Let $A = [a_{i,j}]$ be a $m \times n$ matrix and $B = [b_i]$ a vector of size n. The result of multiplying the matrix A with the vector B is a vector $C = [c_i]$ of dimension m, where

$$c_i = \sum_{j=1}^{n} a_{i,j} c_j.$$

Either rowwise, columnwise, or stripewise processing of the matrix may be used when considering dataflow algorithm for the problem. In Fig. 5 we give plots of running times versus matrix sizes for several variants of the stripe-based processing using rowwise replication of computation. The used stripe width is $s = 128$ while the number of pipes is 1 (no replication, denoted with Stripe128), 2 (StripeRowP2), 4 (StripeRowP4), and 8 (StripeRowP8).

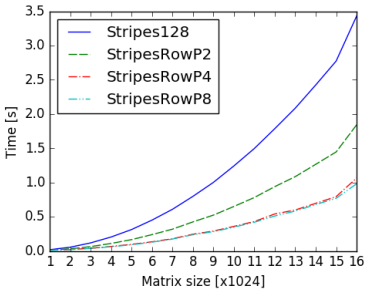


Fig. 5 Running-time performance for matrix-vector multiplication using stripe-based streaming

Observe that, doubling the number of pipes effects in halving the running-time, except for the 8 pipes (or more), where no improvement is observable due to the maximized PCIe bus throughput. In the comparison we excluded the control-flow algorithm, which outperformed all dataflow variants. Nevertheless, this is to be expected as there are not enough operations per a streamed element [12].

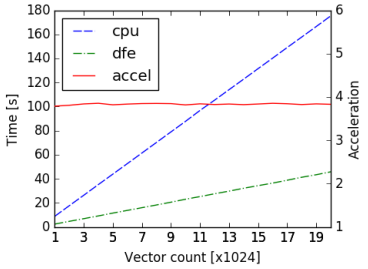


Fig. 6 Running-time performance for multiplication of a matrix with many vectors

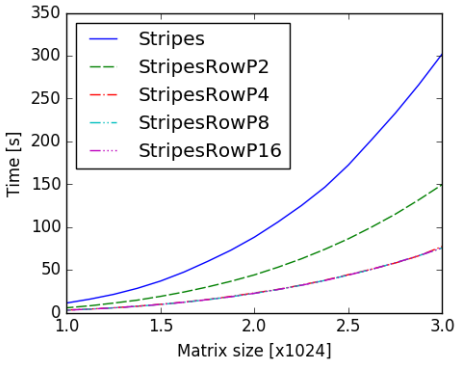
Despite this, a use case to consider dataflow approach can be found in the setting where many vectors are to be multiplied with a given matrix. Here, the matrix is stored in the large memory of the dataflow engine, which has significantly better throughput than PCIe bus.

In this case, the results are in favor of the dataflow architecture. See Fig. 6 for a graphical comparison: the left-hand side y-axis gives a running time in seconds while the right-hand side y-axis gives the acceleration factor obtained (about 4). In the experiment we run both algorithms (dataflow and control-flow), where the number of vectors varies from 1024 to 20480 in steps of 1024.

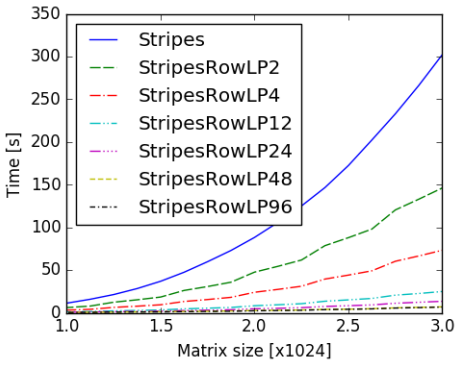
4.3. Matrix Multiplication

In this subsection we present the results of experimental evaluation of several matrix multiplication algorithms. We evaluated various algorithms using the above presented techniques (Section 3) for data choreography. In our experiments we vary the matrix dimension $n \times n$ from $n = 1024$ to $n = 3072$ in steps of 128 for slower algorithms as well as from $n = 1024$ to $n = 10240$ in steps of 512 for faster ones.

As expected, out of basic (i.e., no block matrices) approaches the stripe-based streaming gives the best results. Here, we discuss stripe-based matrix access with rowwise pipe replication of computation, which is employed for streaming the left matrix of the multiplication problem while the right matrix is streamed in the columnwise fashion. Hence, we basically obtain n matrix-vector multiplication problems which are computed by the dataflow engine.



a) Streaming from the main memory (via PCIe)



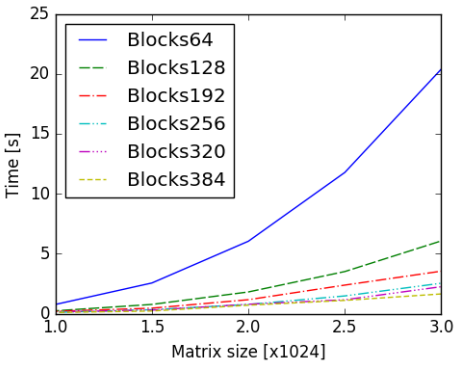
b) Streaming from the large memory

Fig. 7 Running-time comparison of stripped matrix-access depending to the number of pipes

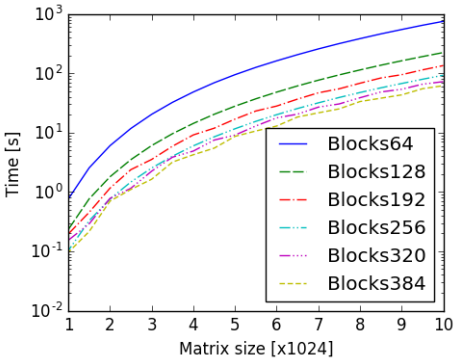
In Fig. 7 a) we show the results of the experiment where the data is streamed from the main memory to the dataflow engine using the PCIe bus, and in Fig. 7 b) where instead of the main memory the dataflow large memory is used. In the latter case, the matrix is only transfered once from the main memory to the large memory.

We can observe that the number of pipes has a desired effect on the performance. However, in the former technique up to 16 pipes are used, but 4 or more pipes do not exhibit any significant improvement because the throughput of the PCIe bus is already maximized. In the latter case up to 48 pipes are used, each increase in pipe count causing an observable improvement in the performance. Here, the 48 pipes case is the last to give a performance improvement.

Now let us focus on the block-based matrix multiplication, which also has the greatest potential. As can be observed in Fig. 8 a) this group of algorithms was much better. To show the scalability and practical usefulness of the algorithm when used with larger matrices we also show a graph of the running-time performance up to matrices of size 10240. Obviously, the larger the block size the better the performance achieved. See Fig. 8 b) for the corresponding graphical comparison, where the ordinate axis uses logarithmic scale.



a) Small matrices



b) Large matrices (log scale)

Fig. 8 Running-time comparison of block-based matrix multiplication depending on the block size

In Fig. 9 we give a comparison of a representative data-choreography techniques. Namely, we include the best from each group: rowwise replication with columnwise (ColRowLP48) and stripe (StripesRowLP48) access, block-multiplication with block sizes of 128 (Blocks128) and 384

(Blocks384) as well as the control-flow implementation of the algorithm to give a better overview on the comparison. Notice that, the control-flow implementation is not highly optimized; however, we employed the classical technique of transposing the second matrix before multiplication, in order to get better performance of the cache memory due to the decrease of cache misses.

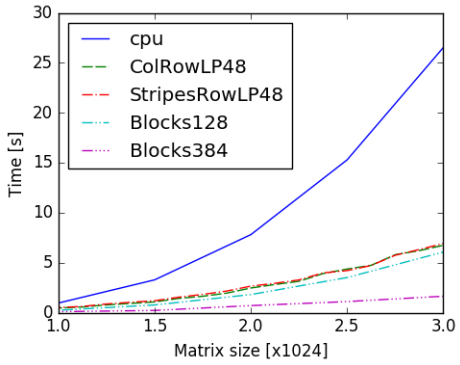


Fig. 9 Running-time comparison of various data choreographies for the matrix multiplication problem

All the (selected) dataflow algorithms are better in running-time performance than the control-flow algorithm. The columnwise and stripped-based techniques perform very similarly with the same number of pipes while block multiplication with the block size at least 128×128 outperforms all other algorithms. Observe also that, one of the slowest block-based algorithm Block128 runs for about 6 seconds while the fastest stripped-based algorithms, StripesRowLP96, runs for about 7 seconds, when the matrix size is 3072×3072.

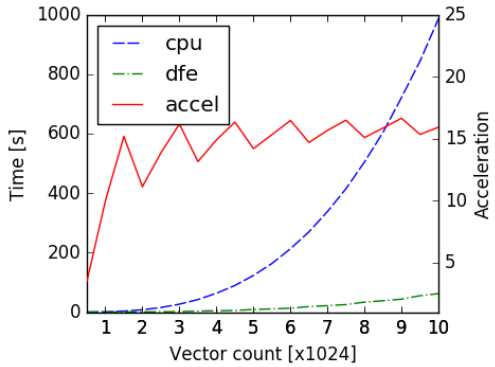


Fig. 10 Acceleration of the dataflow-based approach for the matrix multiplication algorithm

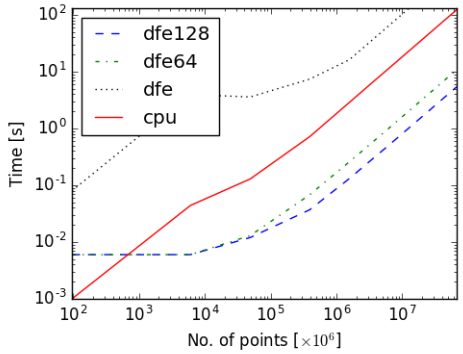
To finish with the comparison of the running time performance, let us have a look on the potential acceleration, which can be obtained with the dataflow-based algorithm. See Fig. 10 for the plot of the running time (left-hand side axis) and acceleration (right-hand side axis): the control-flow algorithm (cpu) and the best dataflow algorithm (dfe) are compared. Observe that, the acceleration of the dataflow over control-flow algorithm is about 15-fold.

4.4. Evaluation of Polynomials

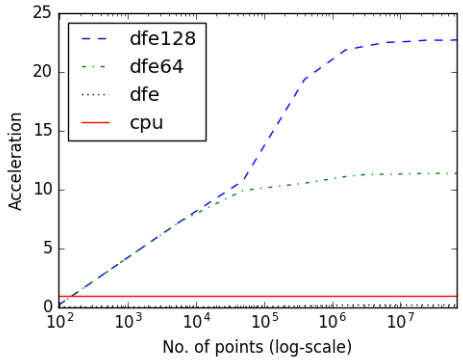
Let us now focus on the multi-point polynomial evaluation problem. Here the input is a polynomial (i.e., either a stream of coefficients if the polynomial is dense, or a stream of exponents and coefficients if the polynomial is sparse), and a stream of points in which the polynomials is to be evaluated.

Control-flow solutions are usually based on the well-known Horner algorithm [18]. Dataflow algorithms explore the data choreography ideas similar to the ones presented in Section 3. Here we leave the technical details out, and present only the results of the experiments.

First, let us focus on the experiments with dense polynomials, where we used polynomials with 1024 coefficients. See Fig. 11 a) for the running-time plots, and Fig. 11 b) for the acceleration plots. Three dataflow algorithms are included: no pipes (dfe), 64 pipes (dfe64), and 128 pipes (dfe128). Observe, that without pipe-based replication the dataflow would be slower than the control-flow algorithm. Nevertheless, using 64 or 128 pipes gives about 11- or 22-fold acceleration, respectively.



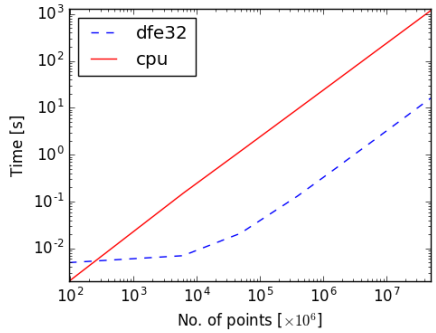
a) Running time (log-log scale)



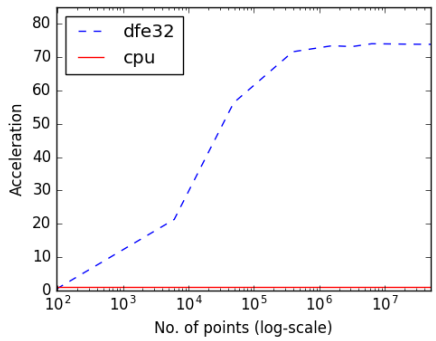
b) Acceleration (log-x scale)

Fig. 11 Running time and acceleration of dense polynomial evaluation algorithms

Now we switch to sparse polynomials, where we performed similar experiments. See Fig. 12 a) and Fig. 12 b) for running-time and acceleration plots, respectively. Again, the dataflow approach with 32 pipes outperformed the control-flow. In particular, the acceleration is about 70-fold.



a) Running time (log-log scale)



b) Acceleration (log-x scale)

Fig. 12 Running time and acceleration of the dataflow algorithm for sparse polynomial evaluation algorithms

4.5. Simplex Pivoting

In this subsection we focus on a different problem, which is the main part of the classical simplex algorithm for finding the optimal solutions of linear programs. Here, a linear program in a canonical form is to optimize cx considering the constraints $Ax \leq b$, where c is an n-dimensional vector of the coefficients of a linear objective function, and A is the matrix of the coefficients of linear constraints; similarly b is a vector representing the coefficients of the right-hand side of the constraints.

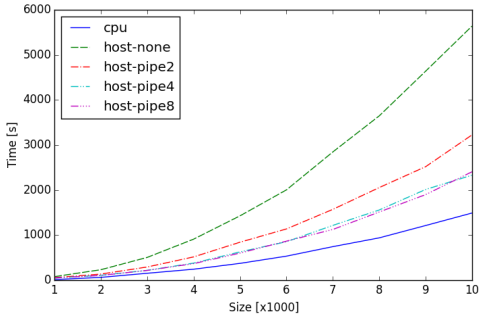


Fig. 13 Running-time comparison of various implementations of the simplex pivoting

In order to solve such a linear system, the simplex algorithm repeatedly transforms the matrix until the optimal solution is found. Such transformations are based on the pivoting operation, which recalculates the matrix based on the selected (pivot) row and column.

We have implemented several versions of the pivoting operations: both variants of streaming from the main memory and from the large memory as well as variations on the number of pipes [9]. The results of the experiments are shown in Fig. 13 for streaming from the main memory via PCIe bus, and Fig. 14 for streaming from the large memory of the dataflow engine.

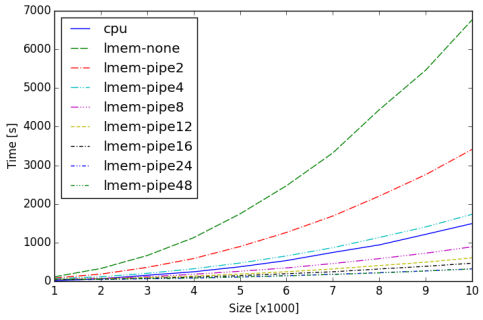


Fig. 14 Running-time comparison of various implementations of the simplex pivoting, where dataflow algorithms use the large memory

As expected, the large-memory variant outperforms the main-memory variant. However, the goal of our experiment is to characterize this difference and determine the maximum number of pipes that still achieve the acceleration.

While the large-memory variant is able to achieve much greater accelerations (by using up to 24 pipes) it is also more complex to implement (consists of several kernels since the selection of the pivot column is done by the dataflow engine).

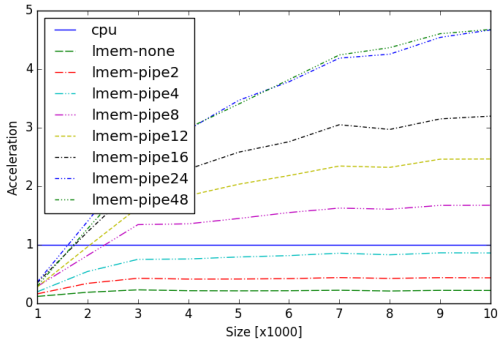


Fig. 15 Acceleration of various implementations of simplex pivoting

Finally, see Fig. 15 for accelerations (over the control-flow algorithm) of the dataflow algorithms streaming from the large memory. Observe that 4 pipes are inadequate for the dataflow to outperform the control-flow algorithm. Hence, 8 or more are suggested while 24 already hit the transfer rate bottleneck of the large memory.

In Fig. 16 we also give resource consumption comparison for all the implementations. Observe that large memory variants require much more resources than the ones where the data is streamed from the host. Additionally, increasing the number of pipes also increases the needed resources.

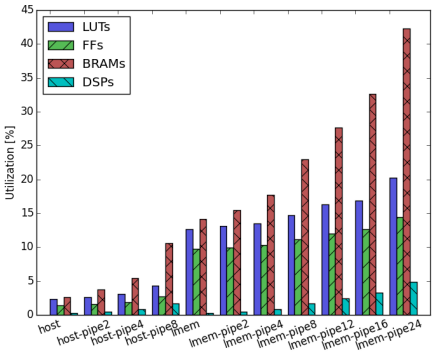


Fig. 16 Resource consumption of various implementations of the simplex pivoting

5. CONCLUSION

For this paper we made a plethora of experiments with various dataflow algorithms and their variations. Doing this we focused on the algorithms and problems which theoretically seem unsuitable for the dataflow architecture. Despite this we were able to show some of the advantages of the dataflow-based algorithms.

Additionally, our goal was to determine the data choreography techniques which work well with particular algorithms and problems. We performed extensive experiments in order to compare the algorithmic performance and resource consumption. In the paper we present the selected set of the results, whereas full experimental setting as well as the corresponding code is publicly available in several GitHub repositories [17] or upon a request to the authors.

In our further work we would like to deal with more dataflow algorithms and problems in order to put the dataflow algorithm engineering and experimental algorithmics into a broader perspective. We believe that such an overview of engineering, which we already initiated in [12], would greatly benefit the dataflow scientific community.

ACKNOWLEDGEMENT

This work was partially supported by the Slovenian Research Agency and the projects "P2-0095 Parallel and distributed systems" and "N2-0053 Graph Optimisation and Big Data".

Our gratitude goes to prof. Veljko Milutinovć who introduced us to dataflow computing as well as Nemanja Trifunović for allowing us to use the dataflow computer and Ivan Milanković for helping us handling the computer. Many of the tests were performed by Matej Žniderič (matrix and vector multiplications) and Anže Sodja (polynomials).

REFERENCES

[1] ŠILC, J. – ROBIČ, B. – UNGERER, T.: *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag Berlin Heidelberg, 1999.

[2] HURSON, A. R. – MILUTINOVIĆ, V.: *Dataflow Processing*. In Advances in Computers, 96. Elsevier, 2015.

[3] Maxeler Technologies Inc. Maximum performance computing. http://www.maxeler.com. Accessed: 2018-03-09.

[4] KOS, A. – TOMAŽIČ, S. – SALOM, J. – MILUTINOVIĆ, V.: *New benchmarking methodology and programming model for big data processing*. International Journal of Distributed Sensor Networks, 2015.

[5] FLYNN, M. J. – MENCER, O. – MILUTINOVIĆ, V. – RAKOČEVIĆ, G. – STENSTROM, P. – TROBEC, R. – VALERO, M.: *Moving from petaflops to petadata*. Communications of the ACM, 56(5):39–42, 2013.

[6] TRIFUNOVIĆ, N. – MILUTINOVIĆ, V. – SALOM, J. – KOS, A.: *Paradigm shift in big data supercomputing: Dataflow vs. controlflow*. Journal of Big Data, 2(1):1–9, 2015.

[7] RANKOVIĆ, V. – KOS, A. – MILUTINOVIĆ, V.: *Bitonic merge sort implementation on the maxeler dataflow supercomputing system*. The IPSI BgD Transactions on Internet Research, 9(2):5–10, 2013.

[8] KOS, A. – RANKOVIĆ, V. – TOMAŽIČ, S.: *Sorting networks on maxeler dataflow supercomputing systems*. In Advances in Computers, 96:139–186, 2015.

[9] ĆIBEJ, U. – MIHELIČ, J.: *Adaptation and Evaluation of the Simplex Algorithm for a Data-Flow Architecture*. In Advances in Computers, 106:63–105, Elsevier, 2017.

[10] MÜLLER-HANNEMANN, M. – SCHIRRA, S.: *Algorithm engineering: bridging the gap between algorithm theory and practice*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.

[11] McGEOCH, C. C.: *A guide to experimental algorithmics*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[12] MIHELIČ, J. – ČIBEJ, U.: *Experimental algorithmics for the dataflow architecture: Guidelines and issues*. IPSI BgD Transactions on Advanced Research, 13(1):1–8, 2017.

[13] MIHELIČ, J. – ČIBEJ, U.: *Dataflow Processing of Matrices and Vectors: Experimental Analysis.* Proceedings of the IEEE 14th International Scientific Conference on Informatics, Poprad, Slovakia, 2017.

[14] MILUTINOVIĆ, V. – SALOM, J. – TRIFUNOVIĆ, N. – GIORGI, R.: *Guide to DataFlow Supercomputing: Basic Concepts, Case Studies, and a Detailed Example*. Computer Communications and Networks. Springer International Publishing, 2015.

[15] KNUTH, D. E.: *The Art of Computer Programming Volume 1: Fundamental Algorithms*, 3rd edition, section 2.2.6. Addison-Wesley: New York, 1997.

[16] PCI-SIG: Peripheral component interconnect special interest group. http://pcisig.com/, 2015. Accessed: 2018-03-09.

[17] Jurij MIHELIČ's GitHub Profile and Repositories. https://github.com/jurem, 2018. Accessed: 2018-03-09.

[18] CORMEN, T. H. – STEIN,C. – RIVEST,R. L. – LEISERSON, CH. E.: *Introduction to Algorithms*, 2nd edition. McGraw-Hill Higher Education, 2001.

Received March 24, 2018, accepted July 23, 2018

BIOGRAPHIES

Jurij Mihelič received his doctoral degree in Computer Science from the University of Ljubljana in 2006. Currently, he is with the Laboratory of Algorithmics, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, as an assistant professor. His research interests include algorithm engineering, combinatorial optimization, heuristics, approximation algorithms, and uncertainty in optimization problems as well as system software and operating systems.

Uroš Čibej received his doctoral degree in Computer Science from the University of Ljubljana in 2007. Currently, he is with the Laboratory of Algorithmics. His research interests include location problems, distributed systems, computational models, halting probability, graph algorithms, and computational complexity.