Library for Matrix Multiplication-based Data Manipulation on a "Mesh-of-Tori" Architecture

Maria Ganzha, Marcin Paprzycki
Systems Research Institute
Polish Academy of Sciences
Warsaw, Poland
Email: firstname.lastname@ibspan.waw.pl

Stanislav Sedukhin
University of Aizu
Aizu Wakamatsu, Japan
Email: sedukhin@u-aizu.ac.jp

***Abstract*—Recent developments in computational sciences, involving both hardware and software, allow reflection on the way that computers of the future will be assembled and software for them written. In this contribution we combine recent results concerning possible designs of future processors, ways they will be combined to build scalable (super)computers, and generalized matrix multiplication. As a result we propose a novel library of routines, based on generalized matrix multiplication that facilitates (matrix / image) manipulations.**

I. Introduction

SINCE the early 1990's one of the important factors limiting computer performance became the ability to feed data to the, increasingly faster, processors. Already, in 1994 authors of [1] discussed problems caused by the increasing gap between the speeds of memory and processors. Their work was followed, among others, by Burger and Goodman ([2]), who were concerned with the limitations imposed by the memory bandwidth on the development of computer systems. In 2002, P. Machanick presented an interesting survey ([3]) in which he considered the combined effects of doubling of processor speed (predicted by Moore's Law) and the 7% increase in memory speed, when compared in the same time scale.

The initial approach to address this problem was through introduction of memory hierarchy for data reuse (see, for instance, [4]). In addition to the registers, CPUs have been equipped with small fast cache memory. As a result systems with 4 layers of latency were developed. Data could be replicated and reside in (1) register, (2) cache, (3) main memory, (4) external memory. Later on, while the "speed gap" between processors and memory continued to widen, multi-processor computers gained popularity. As a result, systems with an increasing number of latencies have been built. On the large scale, data element could be replicated and reside in (and each subsequent layer means increasing / different latency of access): (1) register, (2) level 1 cache, (3) level 2 cache, (4) level 3 cache, (5) main memory of a (multi-core / multi-processor) computer, (6) memory of another networked computer (node in the system), (7) external device. Obviously, such complex structure of a computer system resulted in need for writing complex codes to efficiently use it. Data blocking and reuse became the method of choice for solution of large computational problems. This method was applied not only for multi-processor computers, but also computers with processors consisting of multiple computational units (e.g. cores, processors, etc.). In this context, let us note that as the number of computational units per processor is systematically increasing, the inflation adjusted price of a processor remains the same. As a result, the price per computational operation continues to decrease (see, also [5]).

While a number of approaches have been proposed to deal with the memory wall problem (e.g. see discussion of 3D memory stacking in [6]), they seem to only slow down the process, rather than introduce a radical solution. Note that, introduction of multicore processors resulted in (at least temporary) sustaining the Moore's Law and thus further pushing the performance gap (see, [3], [7]). Here, it is also worth mentioning recent approach to reduce memory contention via data encoding (see, [8]). The idea is to allow for hardware-based encoding and decoding of data to reduce its size. Since this proposal is brand new, time will tell how successful it will be. Note, however, that also this proposal is in line with the general observation that "computational hardware" (i.e. encoders and decoders) is *cheap*, and should be used to reduce volume of data *moved* between processor(s) and memory.

Let us now consider one of the important areas of scientific computing – computational linear algebra. Obviously, here the basic object is a matrix. While, one dimensional matrices (vectors) are indispensable, the fundamental object of majority of algorithms is a 2D, or a 3D, matrix. Upon reflection, it is easy to realize that there exists a conflict between the structure of a matrix and the way it is stored and processed in most computers. To make the point simple, 2D matrices are rectangular (while 3D matrices are cuboidal). However, they are *stored* in one-dimensional memory (as a long vector). Furthermore, in most cases, they are *processed* in a vector-oriented fashion (except for the SIMD-style array processors). Finally, they are sent back to be *stored* in the one-dimensional memory. In other words, data arrangement natural for the matrix is neither preserved, nor taken advantage of, which puts not only practical, but also theoretical limit on performance of linear algebra codes (for more details, see, [9]).

Interestingly, similar disregard to the natural arrangement of data concerns also many "sensor systems." Here, the input image, which is square or rectangular, is read out serially, pixel-by-pixel, and is send to the CPU for processing. This means that the transfer of pixels destroys the 2D integrity of data (an image, or a frame, start to exist **not** in their natural layout). Separately, such transfer introduces latency caused by serial communication. Here, the need to transfer large data streams to the processor may prohibit their use in applications, which require (near) real-time response [10]. Note that large data streams exist not only in scientific applications. For instance modern digital cameras capture images consisting of 22.3×10^6 pixels (Cannon EOS 5D Mark III [11]) or even 36.3×10^6 pixels (Nikon D800 [12]). What is even more amazing, recently introduced Nokia phone (Nokia 808 PureView [13]) has camera capturing 41×10^6 pixels.

For the scientific / commercial sensor arrays, the largest of them seems to be the 2-D pixel matrix detector installed in the Large Hadron Collider in CERN [14]. It has 10^9 sensor cells. Similar number of sensors would be required in a CT scanner array of size approximately $1m^2$, with about 50K pixels per $1cm^2$. In devices of this size, for the (near) real-time image and video processing, as well as a 3-D reconstruction, it would be natural to load data directly from the sensors to the processing elements (for immediate processing). Thus, a focal-plane I/O, which can map the pixels of an image (or a video frame) directly into the array of processors, allowing data processing to be carried out immediately, is highly desired. The computational elements could store the sensor information (e.g. a single pixel, or an array of pixels) directly in their registers (or local memory of a processing unit). Such an architecture has two potential advantages. First, cost can be reduced because there is no need for memory buses or a complicated layout. Second, speed can be improved as the integrity of input data is not destroyed by serial communication. As a result, processing can start as soon as the data is available (e.g. in the registers). Note that proposals for similar hardware architectures have been outlined in [15], [16], [17]. However, all previously proposed focal-plane array processors were envisioned as a mesh-based interconnect, which is good for the local data reuse (convolution-like simple algorithms), but is not proper to support the global data reuse (matrix-multiplication-based complex algorithms).

Separately, it has been established that computational linear algebra can be extended through the theory of algebraic semirings, to subsume large class of problems (e.g. including a number of well known graph algorithms). The theoretical mechanism is named Algebraic Path Problem (APP). As shown, for instance, in ([18]), there is an interesting link between the arithmetical fused multiply and add (FMA) operation, which is supported in modern hardware, and the FMAs originating from other semirings that are not. Specifically, if it was possible to modify the standard FMA to include operations from other semirings (in a way similar to the proposals of KALRAY; [19]), and thus develop a generalized FMA, it could be possible to speed-up large class of APP problems at least 2 times ([18]).

Let us now assume the existence of a computational unit that satisfies the above requirements: (1) accepts input from the sensor(s) and transfers it directly to its operational registers / local memory; (2) is capable of generalized FMA operations. The latter requirement means that such FMA should store (in its registers) constants needed to efficiently perform FMA operations originating from various semirings. Let us name it the *extended generalized FMA; EG FMA*. Recall, that the cost of computational units (of all types) is systematically decreasing ([5]). Therefore, cost of the *EG FMA* unit should not be much higher than that of a standard FMAs found in today's processors. Hence, it is easy to imagine m(b)illions of them "purchased" for a reasonable price. As stated above, such *EG FMAs* should be connected into a square array that will match the shape of the input data. Let us now describe how such system can be build.

II. Mesh-of-tori interconnection topology

Since early 1980's a number of topologies for supercomputer systems have been proposed. Let us omit the unscalable approaches, like a bus, a tree, or a star. The more interesting topologies (from the 1980's and 1990's) were:

- hypercube – scaled up to 64000+ processor in the Connection Machine CM-1,
- mesh – scaled up to 4000 processors in the Intel Paragon,
- processor array – scaled up to 16000+ processor in the MassPar computer,
- rings of rings – scaled up to 1000+ processors in the Kendall Square KSR-1 machines
- torus – scaled up to 2048 units in the Cray T3D

However, all of these topologies suffered from the fact that at least some of the elements were reachable with a different latency than the others. This means, that algorithms implemented on such machines would have to be asynchronous, which works well, for instance, for ising-model algorithms similar to [20], but is not acceptable for a large set of computational problems. Otherwise, extra latency had to be introduced by the need to wait for the information to be propagated across the system.

To overcome this problem, recently, a new (*mesh-of-tori*; *MoTor*) multiprocessor system topology has been proposed ([21], [22]). The fundamental (*indivisible*) unit of the *MoTor* system is a μ-Cell. The μ-Cell consists four computational units connected into a 2×2 doubly-folded torus (see, Figure 1). Logically, an individual μ-Cell is surrounded by so-called membranes that allow it to be combined into larger elements through the process of cell-fusion. Obviously, collections of μ-Cells can be split into smaller structures through cell division. In Figure 1, we see total of 9 μ-Cells logically fused into a single macro-μ-Cell consisting of 4 μ-Cells (combined into a 2×2 doubly folded torus), and 5 separate (individual) μ-Cells. Furthermore, in Figure 2 we observe all nine μ-Cells combined into a single system (a 3×3 doubly folded torus). Observe that, when the 2×2 (or 3×3) μ-Cells are logically fused (or divided), the newly formed structure remains a doubly folded torus. In this way, it can be postulated that the single μ-Cell represents the "image" of the whole system. While in earlier publications (e.g. [22], [23], [24], [25]

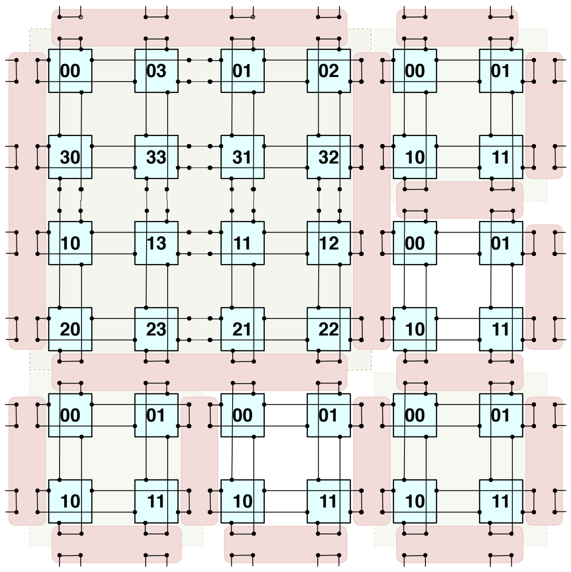

Figure 1. 9 μ-Cells fused into a single 2×2 "system," and 5 separate μ-Cells

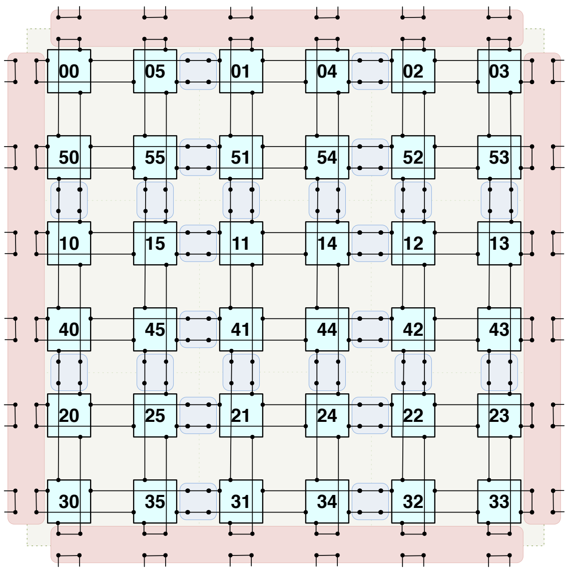

Figure 2. 9 μ-Cells fused into a single 6×6 EG FMA system

the computational units were mostly treated as "theoretical entities," in the context of this paper we assume that each one of them is the *EG FMA* described above. However, analysis of cell connectivity in Figure 1 shows that the model *EG FMA* proposed in the previous section has to be complemented by four interconnects that allow construction of the *MoTor* system. Therefore, from here on, we will understand the *EG FMA* in this way. Furthermore, we will keep in mind that the *MoTor* architecture is build from *indivisible* μ-Cells, each consisting of four, interconnected into a doubly folded torus *EG FAMs*.

Let us now observe that the proposed *MoTor* topology has similar restriction as the array processors from the early 1990's. To keep its favorable properties, the system must be square. While this was considered an important negative (flexibility limiting) factor in the past, this is no longer the case. When the first array processors were built and used, arithmetical operations and memory were "expensive." Therefore, it was necessary to avoid performing "unnecessary" operations (and maximally reduce the memory usage). Today, when GFlops costs about 50 cents (see, [5]) and this price is systematically dropping, and when laptops come with 8 Gbytes of RAM (while some cell phones come with as much as 64 Gbytes of flash memory on a card), it is data movement / access / copying that is "expensive" (see, also [26]). Therefore, when matrices (images) are rectangular (rather than square), it is reasonable to assume that one could just pad them up, and treat them as square. Obviously, since the μ-Cell is a single indivisible element of the $MoTor$ system, if the matrix is of size $N \times N$ then N has to be even.

Observe that there are two sources of inspiration for the $MoTor$ system: (i) matrix computations, and (ii) processing data from, broadly understood, sensor arrays (e.g. images). Furthermore, we have stated that the extended generalized FMA can contain a certain number of data registers to store (a) the needed scalar elements originating from various semirings, (b) elements of special matrices needed for matrix transformations (see, below), as well as (c) data that the FMA is to operate on. However, we also consider the possibility that each FMA may have a "local memory" to allow it to process "blocks of data." This idea is based on the following insights. First, if we define a pixel as " the smallest single component of a digital image" (see, [27]), then the data related to a single pixel is very likely to be not larger than a single 24 bit number. Second, in early 2013 the largest number of FMA units combined in a single computer system was 5.2×10^6. This means that, if there was a one-to-one correspondence between the number of FMA units and the number of "sensed pixels" then the system could process stream of data from a 5.2 Megapixel input device (or could process a matrix of size $N \simeq 2200$).

Let us now consider, development of the *MoTor*-based system. In the initial works, e.g. in [22], links between cells have been conceptualized as programmable abstract links (μ-Cells were surrounded by logical membranes that could be fused or divided as needed, to match the size of the problem). Obviously, in an actual system, the abstract links and membranes could be realized logically, while the whole system would have to be hard-wired to form an actual $MoTor$ system of a specific size. Therefore to build a large system with M^2 μ-Cells (recall the assumption that the mesh will constitute a square array), it can be expected that their groups will be combined into separate "processors," similarly to multicore / multi-FMA processors of today. As what concerns cell fusion and division, it will be possible to assemble sub-system(s) of a needed size, by logically splitting and/or fusing an appropriate number of cells within the $MoTor$ system. However, it is worthy to stress that, while the theoretical communication latency across the mesh-of-tori system is uniform, this may not be the case when the system will be assembled from processors constituting logical (and in some sense also physical) macro-μ-Cells. In this case it may be possible that the communication within the processor (physical macro-μ-Cell) will be slightly faster than between processors. Therefore, the most natural split would be such that would involve complete macro-μ-Cells (processors). However, let us stress that, the design of the mesh-of-tori topology does not distinguish between the connections that are "within a chip" and "between the chips." Therefore, the communication model used in the algorithms described in [22], [23], and considered in subsequent sections, is *independent* of the hardware configurations.

Finally, let us consider the input from the sensor array (or sending a matrix) into the mesh-of-tori type system. As shown in [22], any input that is in the canonical (square matrix) arrangement, is not organized in a way that is needed for the matrix processing in a (doubly folded) torus. However, adjusting the data organization (e.g. to complete a 2D $N \times N$ DFT), requires 2 matrix multiplications (left and right multiplication by appropriate transformation matrices, see below). These two multiplications require $2N$ time steps on a *MoTor* architecture. Next, after the processing is completed, the canonical arrangement can be restored, by reversing the original transformation. Here, again, two multiplications are needed and their cost is $2N$ time steps. For the details about the needed transformations and their realizations as a triple matrix multiplication, see [22].

III. DATA MANIPULATIONS IN A $MoTor$ SYSTEM

Let us summarize points made thus far. First, we have refreshed arguments that there is an unfulfilled need for computer systems that (1) have focal-plane I/O that, among others, can feed data from sensors directly to the operand registers / memory of extended FMA units (generalized to be capable of performing arithmetical operations originating from different semirings), (2) operate on matrices treating them as square (or cuboidal, e.g. tensor) objects, (3) are developed in such a way that (i) minimizes data movement / access / copying / replication, (ii) maximizes data reuse, and (iii) is aware of the fact that arithmetical operations are cheap in comparison with any form of data "movement." Such systems are needed not only to process data originating from the Big Hadron Collider, but also for everyday electronics. Here, it is worth mentioning that virtual reality and 3D media (part of the new enterprises, so called *creative industries*) are in the latter category, illustrating where the computational power is going to be needed in an increasing rate, beyond the classic domains of scientific computing.

Second, we have briefly outlined key features of the, recently proposed, mesh-of-tori topology, which has some favorable features and naturally fits with the proposed *EG FMAs*. Furthermore, we have pointed to some issues that are likely to be encountered in the development of (large-scale) $MoTor$ based systems. Let us now assume, that the, just proposed, $MoTor$ computer systems have been built. In ([22], [23], [25]) it was shown that a large number of matrix operations / manipulations can be unified through the use of a generalized matrix multiply-and-update (MMU) operation. However, in this context it is important to realize that one more problem we are facing today is the increasing complication of codes that are being developed to take advantage of current computer architectures (see, for instance [28], [26]). This being the case, a return to simplicity is needed. In the remaining sections of this paper we will illustrate how a MATLAB / MATHEMATICA style (meta-level) approach can be used to build a library of matrix manipulation operations that, among others, can be implemented on the proposed $MoTor$ computer architecture. This library will be uniformly based on the "fused" matrix multiply-and-update (MMU) operation.

A. Basic operations

To proceed, we will use the generalized matrix multiply and update operation in the form (as elaborated in [28]):

$$\mathtt{C} \leftarrow \mathtt{MMU}[\otimes,\oplus](\mathtt{A},\mathtt{B},\mathtt{C}) : \mathtt{C} \leftarrow \mathtt{C} \oplus \mathtt{A}^{\mathtt{N/T}} \otimes \mathtt{B}^{\mathtt{N/T}}. \qquad (1)$$

Here, A, B and C are square matrices of (even) size N (recall the, above presented, reasons for restricting the class of matrices); while the $\otimes, \oplus$ operations originate from a scalar semiring; and `N/T` specify if a given matrix is to be treated as being in a canonical (`N`) or in a transposed (`T`) form, respectively.

In what follows, we present a collection of matrix / image manipulations that can also be achieved through matrix multiplication. While they can be implemented using *any* matrix multiplication algorithm, we use this as a springboard to further elaborate the idea of *MoTor* system, and a library of routines that can complement it. Note that, for simplicity of discussion (and due to the lack of space), in what follows we only discuss the special case when a single *EG FMA* stores scalar data elements. However, as discussed in [22], [23], [25], [24] all matrix manipulations can be naturally extended to blocked algorithms. Therefore, we actually do not contradict our earlier assumption that each *EG FMA* holds a block of data (e.g. pixel array, or a block of a matrix).

1) Reordering for the mesh-of-tori processing: Let us start from the above mentioned fact that the canonical form of the matrix (image) fed to the $MoTor$ system through the focal-plane I/O is not correct for further (parallel) processing on a doubly folded torus. As shown in [23], the proper format can be obtained by corresponding linear transform through two matrix-matrix multiplications. Specifically, matrix product in the form $M \leftarrow R \times A \times R^T$, where A is the original / input $(N \times N)$ matrix that is to be transformed, M is the matrix in the format necessary for further processing on the mesh-of-tori system, and R is the format rearranging matrix (for details of the structure of the R matrix, consult [23]). Taking into account the implementation of the generalized MMU, proposed in [28], the needed transformation is:

$$M = R \cdot A \cdot transpose(R). \tag{2}$$

Note that on the $MoTor$ system: (a) operation $R \cdot A$ is performed in place and requires N time steps, (b) operation $A \cdot transpose(R)$ is performed in place, requires N time steps and is implemented as a parallel matrix mutiplication with a different data movement (operation scheduling) that the standard multiplication. In other words, the matrix arrangement remains unchanged and well-known problems related to row vs. column matrix storage (see, for instance, [29]) do not materialize (for more details, see [30]).

Observe that when instantiating the $MoTor$ system, it is assumed that appropriate matrix R will be pre-loaded into the macro-μ-Cell, upon its creation. In earlier work, see for instance [28] and references to earlier work of S. Sedukhin collected there, it was assumed that the generalized FMA will store in its operand registers the (scalar) constants originating from implemented semiring(s) and representing elements $\bar{0}$ and $\bar{1}$. Here, we assume that, in addition to the separate operand registers dedicated to each $\bar{0}, \bar{1}$ element originating from the semiring implemented in the hardware, in a separate operand register an appropriate element of a transformation matrix needed to perform operations summarized in this paper will be pre-loaded. It is in this way, that the matrix R will be pre-loaded into the $MoToR$ system.

Observe that in the $MoTor$ system, the size of the "logical" system (macro-μ-Cell) can vary with time and be changed through cell fusion and division. This means that, after a group of μ-Cells is fused (split), some of the "transformation matrices" will have to be re-instantiated. However, this will not concern matrices like `ONES` (see, below) that preserve format during cell fusion / division. Nevertheless, matrix R will have to be re-initialized each time cells are fused / divided. We summarize these considerations, for the "special matrices" identified in this paper, in Table I.

When considering the implementation of the transformation, observe that the information about the R matrix does not need to be known to the user (only to the implementer). This being the case, we can define the $Canonical\_to\_MoTor$ function that will have as its input the matrix A in the canonical form, and as its output matrix M in the form ready to use in the $MoTor$ system. This function will perform operations from equation 2, while hiding matrix R from the user. Obviously, an inverse function $MoTor\_to\_Canonical$, that will perform operation $A \leftarrow R^T \cdot M \cdot R$ (with the same matrix R), will restore the matrix to its original (canonical) format in $2N$ time steps. Obviously, these two functions make sense only in the context of the $MoTor$ system. Specifically, such transformations can be performed on any computer system, but they are useless if that system has a different topology.

2) Row and column permutations: It is a well known fact, that row and column permutations can be represented as matrix-matrix multiplications. Specifically, permutation of rows of matrix A is achieved through left hand side multiplication by an appropriate matrix P ($A' \leftarrow P \cdot A$), while column permutation is achieved through the right hand side multiplication by an appropriate matrix Q ($A' \leftarrow A \cdot Q$). Both matrices P and Q are identity matrices, with two elements (corresponding to appropriate rows or columns) modified. While the matrix multiplication is typically treated as a convenient notation used in theoretical linear algebra, in computational practice row and column permutations are usually implemented as vector operations. However, in the $MoTor$ approach, row and column permutations will be achieved through actual $N \times N$ matrix multiplication, performed in place, in $O(N)$ time steps.

As far as the implementation is concerned, matrix P will be initially stored in the meta-μ-Cell as a copy of the identity matrix(I). Note that this means that this matrix will be actually represented in separate operand registers of appropriate *EG FMAs*. Let us now introduce function $Row\_permute(A, i, j)$. This function, when called, will send information to appropriate four *EG FMAs* ((i, i), (j, j), (i, j), (j, i)) to flip their values from $\bar{0}$ to $\bar{1}$ and vice-versa. Depending on the implementation, this should be achieved in no more than 4 time steps. Next, the actual matrix multiplication will be performed. Finally, the four processing units (belonging to matrix P) that changed their values, will revert to the original ones (again, in no more than 4 time steps). The same approach will be used in the case of column permutation (function $Column\_permute(A, i, j)$). Observe that, while there exist algorithms that ask for ($A' \leftarrow P \cdot A \cdot Q$), these two operations (left and right side multiplication) do not have to be performed simultaneously (in this paper we do not consider a more general case of performing concurrently triple matrix multiplications). This means that actually, we do not need to store two separate matrices P and Q. All that is needed is a single `PERMUT` matrix than can be used by the implementer to support both operations (this matrix is not being made available to the user). Potential use of the actual identity matrix (in place of the separate `PERMUT` matrix) needs to be further considered, before such decision could be made. As what concerns the effect of cell fusion and splitting, it should be obvious from Figures 1 and 2 that it is during this process it is necessary to reinitialize both the identity and the `PERMUT` matrices.

3) Scalar data replication (broadcast): Let us now consider replication of a data element across all processors in the system (operation that in MPI [31] is known as *MPI_BCAST*). As shown in [23], this can be achieved through two matrix multiplications. The appropriate triple has the form $B \leftarrow$ `LBCAST` $* D *$ `RBCAST`, where B is the resulting matrix with all of its elements equal to the replicated one; D is a zero matrix with the element to be replicated in position $d(i, j)$; `LBCAST` is a matrix with all zeros except of the ones in column i; and `RBCAST` is a zero matrix with ones in row j. Obviously, on the $MoTor$ system (since the broadcast involves 2 matrix multiplications) it will be completed in place, in $2N$ time steps.

Based on the material presented thus far, we propose the following implementation of broadcast of a selected element across all processors of a $MoTor$ system. Assume that matrices LBCAST and RBCAST are zero matrices with ones in column 1 and row 1 respectively. Furthermore, matrix D is a copy of the $\bar{0}$ matrix, which is going to be used in both multiplications. In the first step, the selected element is send to the *EG FMA* located in position $(1,1)$ of the $MoTor$ system and stored in the operand register corresponding to $D(1,1)$. Next, the MMU operation is invoked twice ($B \leftarrow \texttt{LBCAST} * D * \texttt{RBCAST}$) within a $ElBcast$ function, which has the form: $ElBcast(element)$, where the $element$ specifies element that should be replicated. As a result, in $2N$ time steps, the selected element is replicated to all elements of matrix B and thus made available across the $MoTor$ system. Finally, the $D(1,1)$ element is zeroed. Note that, while matrices LBCAST and RBCAST have to be reinstantiated, the matrix D, being a copy of the zero matrix remains unchanged during cell fusion / division. Possibility of use of the actual zero matrix, instead of matrix D has to be further evaluated. Obviously, matrices LBCAST and RBCAST are available only to the implementer, while being hidden from the user.

4) Global reduction and broadcast: The next matrix operation that can be formulated in terms of matrix multiplications, is the *global reduction and broadcast*. As seen in [23], when the standard arithmetic is applied, and matrix A is multiplied from both sides by a matrix of ones (matrix with all elements equal to one, let us name it ONES), then the resulting matrix will have its elements equal to the sum of all elements of A. On a mesh-of-tori system, this can be implemented in place, in $2N$ time steps, when the matrix ONES is available (preloaded) in all *EG FMAs* of the system.

However, recall that our approach is based on use of the generalized MMU. Thanks to this, we can apply operations originating from different semirings. Here, particularly interesting would be semirings, in which the addition and multiplication operations are defined as $(\times, max)$ or $(\times, min)$. In this case, the "generalized reduction and broadcast" operation is going to consist of two generalized MMUs (represented in notation from the equation 1):

$$\texttt{MMU}[\otimes,\oplus](\texttt{A},\texttt{ONES},\texttt{TEMP}):\texttt{TEMP}\leftarrow\texttt{TEMP}\oplus\texttt{A}\otimes\texttt{ONES};$$
$$\texttt{MMU}[\otimes,\oplus](\texttt{ONES},\texttt{TEMP},\texttt{RESULT}):$$
$$\texttt{RESULT}\leftarrow\texttt{RESULT}\oplus\texttt{ONES}\otimes\texttt{TEMP}.$$

Here, operations $[\otimes, \oplus]$ are defined in an appropriate semiring, while matrices TEMP and RESULT are initialized as copies of the $\bar{0}$ matrix (zero matrix for a given semiring). Finally, ONES is a matrix of all ones, where the "one" element originates from a given scalar semiring (its element $\bar{1}$).

Under these assumptions it is easy to see that we can define at least three functions that will have the same general form, while being based on different semirings: *AddBcast* – realizing summation of all elements in a matrix and broadcasting the result to all processors (function based on the standard arithmetic); $MaxBcast$ finding the largest element in a matrix and broadcasting it to all processors (based on the $(\times, max)$ semiring); and $MinBcast$ finding the smallest element in the matrix and broadcasting it to all processors (based on the $(\times, min)$ semiring). Each of these functions will be completed in place, in $2N$ time steps, through 2 generalized matrix multiplications. Observe that, for all practical purposes, matrix ONES does not have to be instantiated. It consists of $\bar{1}$ elements that, according to our assumptions, are already stored in operand registers of the *EG FMAs*. Finally, note that (regardless of the way it will finally be instantiated in the *MoTor* system) matrix ONES is independent of the size of the macro-μ-Cell and remains unchanged and available after cell fusion / division operation. As previously, all information about very "existence" of matrices ONES and TEMP, and initialization of matrices TEMP and RESULT is going to be hidden from the user.

B. Matrix (image) manipulations

Let us now consider three simple matrix *manipulations* that can be achieved with help of matrix multiplication. While they are presented as matrix operations, their actual value can be seen when the underlying matrices represent images (e.g. each matrix element represents a pixel, or a block of pixels).

1) Upside-down swap: Image (matrix) upside down swap can be achieved by multiplying the matrix from the left hand side by the $SWAP$ matrix, which has the following form:

$$SWAP = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix}$$

Obviously, we assume that on the $MoTor$ system, matrix SWAP will be instantiated when macro-μ-Cell(s) will be created (appropriate elements will be stored in separate operand registers of the *EG FMAs*). However, it will not be made available to the user. This means that the upside-down swap will be achieved by calling a $UDswap(A)$ function, and completed in place, in N time steps. Matrix SWAP will have to be re-initialized after each μ-Cell fusion or division.

2) Left-right swap: The left-right image (matrix) swap can be achieved the same way as the upside-down swap, with the only difference being that the image matrix A is going to be multiplied by the SWAP matrix from the right hand side. Therefore, on the $MoTor$ system, the left-right swap will be completed in place, in N time steps, by calling the $LRswap(A)$ function. All the remaining comments, concerning the SWAP matrix, presented above, remain unchanged.

3) Rotation: Interestingly, combining the two swaps into a single operation (multiplication of a given image / matrix A from left and right by the matrix SWAP) results in rotation of the matrix / image A by 180°. Obviously, from the above follows that on the $MoTor$ system, this operation can be completed in place, in $2N$ steps using two matrix multiplications, by calling an appropriately defined $Rotate(A)$ function.

IV. Towards library of matrix multiplications based data manipulations

Let us now summarize the above considerations from the point of view of development of a library of operations that can be performed on matrices / images though generalized matrix

Table I
SUMMARY OF FUNCTIONS PROPOSED FOR THE LIBRARY

Functionality	Function	Matrices	Re-instantiate
Reorder Canonical to $MoTor$	$Canonical\_to\_MoTor(A)$	R	yes
Reorder $MoTor$ to Canonical	$MoTor\_to\_Canonical(A)$	R	yes
Row permutation	$Row\_permute(A,i,j)$	PERMUT	yes
Column permutation	$Column\_permute(A,i,j)$	PERMUT	yes
Replication	$ElBcast(element)$	LBCAST, RBCAST, D	yes & no
Addition and broadcast	$AddBcast(A)$	ONES	no
Max and broadcast	$MaxBcast(A)$	ONES	no
Min and broadcast	$MinBcast(A)$	ONES	no
Upside-down swap	$UDswap(A)$	SWAP	yes
Left-right swap	$LRswap(A)$	SWAP	yes
Rotation by 180°	$Rotate(A)$	SWAP	yes

multiplication. In Table I we combine proposals presented thus far. There, we present the functionality, the proposed function name, the "special matrices" that have to be instantiated within the *MoTor* system to complete the operations, and information if these matrices have to be reinitialized after μ-cell fusion / splitting operation. Observe that, while the first two functions are directly connected with the *MoTor* architecture, the remaining ones can be seen as "system independent." In other words, they can be implemented for any computer architecture, taking full advantage of the underlying architecture.

This latter observation deserves further attention, and some points have to be made explicit. Only the transformations from the canonical to the $MoTor$ format and back are $MoTor$ architecture specific. The remaining functions are system independent. While the above considerations have in mind the $MoTor$ architecture, the proposed functions use only matrix multiplication and thus can be implemented to run on any computer architecture, using its best of breed matrix multiplication algorithm. This being the case, and taking into account discussion presented in Section I, it may be desirable to implement functions from Table I on existing computers, using state-of-the-art matrix multiplication algorithms and consider their efficiency.

A. Object oriented realization

Let us now recall that our main goal is to consider functions from Table I in the context of the *MoTor* architecture. However, we also see them as a method of simplifying code writing (by introducing matrix operations represented in the style similar to that found in MATLAB / MATHEMATICA). This being the case we assume that there may be multiple ways of implementing these routines, and that they are likely to be vendor / hardware specific. Nevertheless, at the time of writing of this paper, object oriented programming is one of more popular ways of writing codes in scientific computing and image processing. Furthermore, this means the possible trial implementations, suggested above, are likely to be tried using this paradigm. This being the case, we have decided to conceptualize the top-level object-oriented representation of the library of routines from Table I. Since different OO languages have slightly different syntax (and semantics), we use a generic notation, distinguishing information that needs to be made available in the interface and in the main class. We start from the interface (see, also [28]).

```
/* T — type of matrix element */

interface Matrix_interface {
        public Matrix 0(n) {/*generalized zero matrix*/}
        public Matrix I(n) {/*generalized identity matrix*/}
        public Matrix operator + /*generalized  A+B*/
        public Matrix operator * /*generalized A*B*/
        public Matrix Canonical_to_Motor(A);
        /*reordering for themesh—of—tori processing*/
        public Matrix Motor_to_Canonical (A);/*inverse of
        the reordering for the mesh—of—tori processing*/
        public Matrix transpose(A);
        {/*transposition of matrix A*/}
        /*generalized permutation of column / row
                i and j in matrix A*/
        public Matrix Column_Permut (A,i,j);
        public Matrix Row_Permut (A,i,j);
        /*generalized element broadcast*/
        public Matrix ElBcast(element);
        public Matrix AddBcast (A);/*generalized summation
        of all elements of A and broadcast*/
        /*broadcast the largest element of A*/
        public Matrix MaxBcast (A);
        /*broadcast the smallest element of A*/
        public Matrix MinBcast (A);
        /*Matrix (Image) Manipulation*/
        public Matrix UDswap (A);/*upside—down swap*/
        public Matrix LRswap (A);/*left—right swap*/
        /*image vertical rotation*/
        public Matrix Rotate (A);...
}
```

Just defined interface is to be used with the following class *Matrix*. This class summarizes the proposals outlined above.

```
class Matrix inherit scalar_Semiring
                implement Matrix_interface {
   T: type of element;/*double, single,...*/
   private Matrix R (n);/*matrix for MoTor
         transformation*/
   private Matrix ONES (n) /* matrix of ones*/
   private Matrix PERMUT(i,j,n)/*identity matrix
        with interchanged columns i and j*/
   //anti—diagonal matrix of ones
   private Matrix SWAP (n)
        // Methods
   public Matrix 0(n) {/*0 matrix*/}
   public Matrix I(n) {/*identity matrix*/}
   public Matrix transpose(A:Matrix){
                /*MMU—based transposition of A*/}
   public Matrix operator + {A,B:Matrix}
                {return MMU(A,I(n),B,a,b)}
   public Matrix operator * {A,B:Matrix}
                {return MMU(A,B,matrix_0,a,b)}
   public Matrix Column_Permut (A,i,j){
                 return MMU(PERMUT(i,j,n),A,O(n))}
   public Matrix Row_Permut(A,i,j){
                return MMU(A,PERMUT(i,j,n),O(n))}
```

```
    public Matrix Canonical_to_Motor (A){
                M = R * A * transpose(R);
                return M}
    public Matrix UDswap (A);/*upside-down swap*/
                {return MMU(O(n),SWAP,A)}
    public Matrix LRswap (A);/*left-right swap*/
                {return MMU(O(n), A,SWAP)}
        /*image vertical rotation*/
        public Matrix Rotate (A);
                {A=MMU(O(n),SWAP,A);
                 return MMU(O(n),A,SWAP)}
        ...
    private MMU(A,B,C:Matrix(n)){
      return "vendor/implementer specific
          realization of MMU = C + A*B where
       + / * are from class scalar_Semiring"}
...
}
```

Obviously, just defined class and interface allow us to write codes in the suggested manner. Here, the matrix operations (image manipulations) can be performed by calling very simple functions, and hiding all implementation details from the user.

V. Concluding remarks

The aim of this paper was to reflect on current trends in computational sciences. We have focused our attention on selected trends in hardware and software design, to visualize possible designs of future processors and ways they will be combined to build scalable (super)computers. In this context, the mesh-of-tori architecture is one of the more promising concepts for design of large-scale computer systems. This provided us with background, against which we have considered the role of generalized matrix multiplication. As a result we proposed a novel library of routines, based on generalized matrix multiplication that allows for data (matrix / image) manipulations. In the future we plan to implement the proposed library on the virtual *MoTor* system.

Acknowledgment

Work of Marcin Paprzycki was completed while visiting the University of Aizu.

References

[1] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, vol. 14, no. 4, pp. 59–67, 1994.

[2] D. Burger, J. R. Goodman, and A. Kagi, "Memory bandwidth limitations of future microprocessors," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1996, pp. 78–89, doi:http://doi.acm.org/10.1145/232973.232983.

[3] P. Machanick, "Approaches to addressing the memory wall," http://homes.cs.ru.ac.za/philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf, 2002.

[4] R. van der Pas, "Memory hierarchy in cache-based systems," http://www.sun.com/blueprints/1102/817-0742.pdf, Sun Microsystems, Tech. Rep., 2002.

[5] Wikipedia, "Flops," http://en.wikipedia.org/wiki/FLOPS.

[6] P. Jacob, A. Zia, O. Erdogan, P. M. Belemjian, J.-W. Kim, M. Chu, R. P. Kraft, J. F. McDonald, and K. Bernstein, "3D memory stacking; mitigating memory wall effects in high-clock-rate and multicore CMOS 3-D processor memory stacks," *Proceedings of the IEEE*, vol. 97, no. 1, January 2009.

[7] F. Alted, "Why modern CPUs are starving and what can be done about it," *Computing in Science and Engineering*, vol. 12, pp. 68–71, 2010, doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.51.

[8] A. Wegener, "Numerical encoding shatters exascale's memory wall," http://www.hpcadvisorycouncil.com/events/2013/Stanford-Workshop/pdf/Presentations/Day2013.

[9] F. G. Gustavson, "Cache blocking for linear algebra algorithms," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Springer Berlin Heidelberg, 2012, vol. 7203, pp. 122–132.

[10] D. Fey and D. Schmidt, "Marching-pixels: a new organic computing paradigm for smart sensor processor arrays," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 1–9, doi:http://doi.acm.org/10.1145/1062261.1062264.

[11] "Canon EOS 5D," http://www.usa.canon.com/cusa/consumer/products/cameras/slr_cameras/eos_5d_mark_iii, 2013.

[12] "Nikon D800," http://www.nikonusa.com/en/Nikon-Products/Product/Digital-SLR-Cameras/25480/D800.html, 2013.

[13] "Nokia 808 pureview," http://reviews.cnet.com/smartphones/nokia-808-pureview-unlocked/4505-6452_7-35151907.html, 2013.

[14] E. H. M. Heijne, "Gigasensors for an attoscope: Catching quanta in CMOS," *IEEE Solid State Circuits Newsletter*, vol. 13, no. 4, pp. 28–34, 2008.

[15] S. Chai and D. Wills, "Systolic opportunities for multidimensional data streams," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 4, pp. 388–398, 2002.

[16] S. Kyo, S. Okazaki, and T. Arai, "An integrated memory array processor architecture for embedded image recognition systems," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, June 2005, pp. 134–145.

[17] Á. Zarándy, *Focal-Plane Sensor-Processor Chips*. Springer, 2011. [Online]. Available: http://books.google.co.jp/books?id=wpCsjwEACAAJ

[18] S. G. Sedukhin and T. Miyazaki, "Rapid*closure: Algebraic extensions of a scalar multiply-add operation," in *CATA*, 2010, pp. 19–24.

[19] "Kalray multi-core processors," http://www.kalray.eu/.

[20] P. Altevogt and A. Linke, "Parallelization of the two-dimensional ising model on a cluster of ibm risc system/6000 workstations," *Parallel Computing*, vol. 19, no. 9, pp. 1041–1052, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819193900964

[21] S. Sedukhin, T. Miyazaki, K. Kuroda, H. Oi, and Y. Okuyama, "Arithmetic Processing Unit, Patent Application," Filled on September 2007. [Online]. Available: {http://worldwide.espacenet.com/publicationDetails/biblio?adjacent=true&locale=en_EP&FT=D&date=20070927&CC=JP&NR=2007249744A&KC=A}

[22] A. A. Ravankar and S. G. Sedukhin, "Mesh-of-tori: A novel interconnection network for frontal plane cellular processors," *2013 International Conference on Computing, Networking and Communications (ICNC)*, pp. 281–284, 2010.

[23] A. A. Ravankar, "A new "mesh-of-tori" interconnection network and matrix based algorithms," Master's thesis, University of Aizu, September 2011.

[24] A. Ravankar and S. Sedukhin, "Image scrambling based on a new linear transform," in *Multimedia Technology (ICMT), 2011 International Conference on*, 2011, pp. 3105–3108.

[25] A. A. Ravankar and S. G. Sedukhin, "An O(n) time-complexity matrix transpose on torus array processor," in *ICNC*, 2011, pp. 242–247.

[26] J. L. Gustafson, "Algorithm leadership," *HPCwire*, vol. Tabor Communications, April 06, 2007. [Online]. Available: http://www.hpcwire.com/features/17898659.html

[27] "Wikipedia pixel," http://en.wikipedia.org/wiki/Pixel, March 2013.

[28] S. G. Sedukhin and M. Paprzycki, "Generalizing matrix multiplication for efficient computations on modern computers," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Springer Berlin Heidelberg, 2012, vol. 7203, pp. 225–234.

[29] M. Paprzycki, "Parallel Gaussian elimination algorithms on a Cray Y-MP," *Informatica*, vol. 19, no. 2, pp. 235–240, 1995.

[30] S. G. Sedukhin, A. S. Zekri, and T. Myiazaki, "Orbital algorithms and unified array processor for computing 2D separable transforms," in *Parallel Processing Workshops, International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 127–134.

[31] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.