DOI 10.12694/scpe.v15i2.955
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 2, pp. 187–200. http://www.scpe.org © 2014 SCPE

GENERALIZED MATRIX MULTIPLICATION AND ITS OBJECT ORIENTED MODEL

MARIA GANZHA[1], MARCIN PAPRZYCKI[2], AND STANISLAV G. SEDUKHIN[3]

Abstract.
Since the beginning of the 21st century, we observe rapid changes in the area of, broadly understood, computational sciences. One of interesting effects of these changes is the need for reevaluation of the role of dense matrix multiplication. The aim of this paper is two-fold. First, to summarize developments that point toward a need for reconsidering usefulness of matrix multiplication generalized on the basis of the theory of algebraic semirings. Second, to propose generalized matrix-matrix multiply-and-update (MMU) operation and its object oriented model.

Key words: matrix multiplication, algebraic semirings, algebraic path problem

AMS subject classifications. 65F30, 13A99

1. Introduction. Recently, a number of changes can be observed in computational sciences. They concern all levels of the computational stack. First, evolution of computer hardware, forced by limits imposed by physics, resulted in practical disappearance of processors with a single computational unit. As a matter of fact, today it is possible to have a quad-core processor in a cell phone (e.g. in the newest Samsung Galaxy 4) and even 8 cores (in the Motorola X8 Mobile Computing System [10]). Furthermore, it is already possible to have more than a thousand fused multiply and add (FMA) units in a single GPU processor [33]. Second, there is a constantly growing gap between the capacity of the processor to consume the data and hardware' ability to feed it. Third, rapidly decreasing cost of the FMA unit, combined with appearance of processors with thousands FMAs, lead to suggestions that a complete reevaluation of approach to computing is needed [34, 35]. Here, the basic assumption is that data access/movement is "expensive," while arithmetic operations are "cheap." Fourth, it is time to (re)consider complexity of codes that try to match (and effectively utilize) current computer hardware with as much as seven levels of data access latency. Finally, rapid proliferation of devices with matrix-like sensor input (e.g. digital cameras, medical imaging devices, radio telescopes, etc.) forcefully reminds us that, in multiple applications, actual data consists of 2D and/or 3D matrix-structures that are fed with high speed, and should not be stored but processed in-place as they elements are delivered to the processing units.

In this paper we will argue that the time has come for a meta-reflection and general change of approach to large-scale (primarily "scientific") computing. In particular, it is important to look into efficient solution of matrix-based problems, and this is precisely the scope of the current contribution. This paper modifies and extends our two conference papers [69, 30], and it is organized as follows. First, we discuss the interaction between progress in computer hardware and computational linear algebra in the early days of supercomputing. Second, we consider dense matrix multiplication, as one of key elements of large number of linear algebraic algorithms. Here, we also look into its generalization through the theory of algebraic semirings. Next, we reflect on the most current trends in hardware and software for computational linear algebra and combine these considerations to propose a generalized matrix multiply and update (MMU) operation. Finally, we use the discussion of the state-of-the-art in object oriented BLAS to propose an object oriented realization of the generalized MMU.

2. Computer hardware and computational linear algebra in the long gone past.

2.1. Single-processor computers. Let us start our discussion from late 1970's, when it became clear that many algorithms for matrix computations consist of similar building blocks (e.g. a *vector update*, or a *dot-product*). As a result, in the Cray-1 supercomputer, vector operation $\bar{y} \leftarrow \bar{y} + \alpha\bar{x}$ (where $\bar{x}$ and $\bar{y}$ are

[1]Systems Research Institute, Polish Academy of Sciences, Warsaw and Institute of Informatics, University of Gdańsk, Gdańsk, Poland (Maria.Ganzha@ibspan.waw.pl),
[2]Systems Research Institute, Polish Academy of Sciences, Warsaw and Department of Management and Technical Sciences, Warsaw Management Academy, Warsaw, Poland (Marcin Paprzycki@ibspan.waw.pl),
[3]Distributed Parallel Processing Laboratory, The University of Aizu, Aizuwakamatsu City, Fukushima 965-8580, Japan (sedukhin@u-aizu.ac.jp)

n-element vectors, while α is a scalar) has been efficiently implemented. Specifically, S. Cray proposed vector processors with *chaining* of multiply and update operations [66], where results of the multiply operations have been forwarded directly from the "multiplication unit" to the "addition unit." A few years later, the IBM build the 3090 series of vector computers with efficient implementation of the *dot-product* operation [40]. In the meantime, in 1979, the level 1 BLAS standard was proposed [44], which defined a set of core vector-vector operations. The assumption behind the level 1 BLAS was that the computer vendors would provide efficient hardware (and software) realizations of these operations. In this spirit, the Cray Inc. build computers with efficient *vector updates*, that became a part of the *scilib* library [39]; while the IBM developed the *ESSL* library [38], with highly optimized dot-products. This library was later ported to the IBM RS/6000 workstations; the first commercial computer to implement the fused multiply-and-add (FMA) operation [52]. Note that the FMA operation appears in both vector updates and dot products. Following this path, currently, processors from IBM, Intel, AMD, Nvidia, and others [18, 20, 24, 41, 53, 55, 56], include efficient hardware supported scalar floating-point fused multiply-add operation.

The FMA combines two basic floating-point operations (flops) into one (three-read-one-write) operation with only one rounding error, throughput of two flops per cycle, and a few cycles latency – depending on the depth of the FMA pipeline. There exist two FMA standards: FMA3 and FMA4. The difference is that the FMA4 has four operands ($d \leftarrow c + a * b$), while the FMA3 has three operands (it is an update $c \leftarrow c + a * b$). Since it is not our aim to discuss the philosophical differences between these two approaches, for the purpose of this paper, let us assume that the term "FMA" covers both standards. Besides the increased accuracy, the FMA minimizes operation latency, reduces hardware cost, and chip busing [52]. The standard floating-point add, or multiply, are performed by taking $a \leftarrow 1.0$ (or $b \leftarrow 1.0$) for addition, or $c \leftarrow 0.0$ for multiplication. Therefore, the two floating-point constants, 0.0 and 1.0, need to be available within the processor. As a result, in the current computer hardware, one cannot "practically distinguish" between (faster/simpler) addition and (slower/more complex) multiplication. This fact has important consequences for some classes of divide-and-conquer algorithms (see, below).

The development of software for computational linear algebra continued, with introduction of level 2 BLAS, standardizing matrix-vector operations [28]. Next, the level 3 BLAS was introduced in [27], defining matrix-matrix operations. Following, the supercomputer vendors developed highly optimized implementations of BLAS kernels for their hardware (e.g. the Cray Inc. provided Cray Assembly Language based implementations [39]). Furthermore, on the basis of the level 3 BLAS, the LAPACK library was proposed [14], defining templates for solving dense matrix problems.

It is also during these years, when the disparity between the speed of the processor and the memory access time became apparent (see, for instance [17]). In response to this trend, computers with hierarchical memory (introduced for data reuse) have been proposed. To match these architectures, block-oriented realizations of fundamental linear algebra algorithms for dense matrices have been introduced and experimented with (this is precisely the class of algorithms that constituted the core of the "LAPACK approach"). Here, blocking allowed data reuse and minimization of data movement. Since the process of finding perfect blocking was rather tedious, and only few programmers were ready to do this for the highly complex computer hardware (see, for instance, [31]), auto-tuners have been introduced. Among them, the ATLAS project [76] became the most popular.

2.2. Parallel computers. Let us now look into historical trends in *parallel computer hardware.* In the 1990's three main designs for the parallel computers were: (1) array processors, (2) shared memory parallel computers, and (3) distributed memory parallel computers. While quite popular initially, array processor supercomputers disappeared by approximately 1995. The main reason was an apparent lack of flexibility to deal with problems that do not natively appear in the "matrix-form." Furthermore, it is worthy observing that, regardless of the array organization of processors, no truly efficient implementation of matrix multiplication has been realized. Next, the shared memory parallel supercomputers started to fade away. Here, the main problem was the bottleneck caused by the connection between the memory and the processors. It turned out that, in computational practice, scaling such machines to more than 32 processors was extremely difficult and economically unfeasible. As the result, by the end of 20th century, the dominating supercomputer architecture became the distributed memory parallel computers.

As what concerns computational linear algebra, while the LAPACK library was focused on single-processor and shared-memory parallel computers, the ScaLAPACK [21] was to provide the same standardization for the distributed memory computers. The ScaLAPACK is based on the single program multiple data (SPMD) programming model. In addition to the computational kernels, parallel BLAS kernels [22], and communication routines (BLACS [5]) have been defined. Overall, the main assumption remained that the hardware/software vendors are going to provide efficient low-level realization of the needed functionality. Interestingly, while the LAPACK project became quite successful, the ScaLAPACK did not match its reach and popularity. There exist multiple of reasons for this fact. Among others: (1) programming distributed memory computers turned out to be more difficult than expected, (2) end of the Cold War cut funding for development of hardware and software on the large scale, (3) raise of cluster (COTS) computers moved the interest to low-cost solutions on the small scale (where highly optimized routines were not the biggest concern), (4) extending / updating Fortran 77 turned out to be a complete failure, (5) implementers of codes for computationally intensive problems turned to C and next to C++ and object oriented scientific computing (however, object orientation did not result in unification of the field; see, section 7).

In summary, there was a time when development of "high performance" hardware and software worked hand-in-hand. However, their pathways have diverged and software developers have been left to catch up with the computer hardware. Nevertheless, matrix multiplication remained the workhorse of a very large class of algorithms in computational linear algebra (in particular, their block-oriented implementations). Therefore, we will now focus on high performance matrix multiplication.

3. Dense matrix multiplication in scientific computing. While dense matrix-matrix multiplication appears in many computational problems, its most popular application is in the update step of block algorithms and has the form: $C \leftarrow C + A \times B$ (where A, B and C are appropriately dimensioned matrices; note that, in general, A and /or B can be in transposed form, which leads to four different variants of this operation). Despite its simplicity, the arithmetic complexity, and data dependencies, make it a challenging problem to reduce the *run-time complexity*. The two basic approaches to speeding matrix multiplication were, first, lowering the *arithmetic* complexity – achieved by reducing the number of scalar multiplications (complex/expensive), while increasing the number of scalar additions/subtractions (simple/cheap). Here, one could list Strassen [73], Pan [57], and Coppersmith-Winograd [23] algorithms (further discussion and references can be found in [65]). Second, by the *parallel implementation* [13, 75, 42, 29, 19, 47]. Of course, a combination of these two approaches is also possible (see, for instance, [72, 37, 32, 49]).

The recursive matrix multiplication "worked well" from the point of view of theoretical analysis of arithmetical complexity, and when implemented on early computers. However, its implementation started to became a problem on computers with hierarchical memory (e.g. to reach optimal performance of a Strassen-type algorithm, recursion had to be stopped when the size of the divided submatrices approximated the size of the cache memory – differing between machines; see, [15]). Furthermore, practical implementation of Strassen-type algorithms requires extra memory (e.g. the Cray's implementation of Strassen's algorithm required extra space of order $2.34n^2$ [25]). Here, recall two facts. First that in the modern FMA units, there is no distinction between time of addition and multiplication (these two operations cannot be "separated" into cheap and expensive). Second that the speed of memory access is one of the key factors limiting advances of high performance computing.

It is worthy recalling that recursive matrix multiplication has been successfully used within other algorithms, e.g. within blocked Gaussian Elimination [15, 61, 60, 58]. Unfortunately, recursive approaches result in problems with numerical stability. While, as proved by N. J. Higham in [36], these problems should not be extremely pronounced, in [61] it was shown that they may prevent solution of at least some practical problems. Specifically, for a class of PDE problems, substituting matrix multiplication by a fast one, in the block-oriented linear equation solver, resulted in a failure to reach the solution.

The second approach to speeding matrix multiplication is through the design of parallel algorithms, which have been implicitly or explicitly based on a time-space scheduling of FMAs in the 3D computational index space. This scheduling directly affects data reuse, by orchestrating data movement between different scalar operations. One of the key differences between this and the recursion-based approaches is that in the parallelized matrix multiplication, patterns of data movement remain relatively simple and well structured. Moreover, in the recent paper [70], it was shown that, actually, there exist only four basic classes of schedules and, therefore, classes of algorithms. In other words, **all** existing parallel matrix multiplication algorithms are extensions of these four basic schedulings, with respect to different matrix shapes, blocking or tiling, dimensionality of parallel implementation, underlined (assumed) computer architecture, etc. (see, also [46]). Furthermore, it was observed that all four schedulings can be used to implement the level 3 BLAS operation _GEMM [27] of the form $C \leftarrow C + A \times B$; the *matrix multiply-update* (*MMU*). Out of the four classes of parallel *MMU* algorithms defined in [70]: (i) Broadcast-Compute-Shift; (ii) All-Shift-Compute (or Systolic); (iii) Broadcast-Compute-Roll; and (iv) Compute-Roll-All (or Orbital), the last one is characterized by regularity and locality of data movement, maximal data reuse without data replication, recurrent ability to involve into computing all matrix data at once (focal-plane I/O), etc. This makes it well suited for the computer hardware that is likely to materialize in the near future (see, Section 5). Interestingly, the recently proposed 2.5D approach to matrix multiplication [71], represents a hybrid between a Broadcast-Broadcast-Compute (BBC) and a Compute-Roll-All (CRA). Specifically, when no extra memory is available the approach reduces to the CRA (Cannon Algorithm), while when N extra copies of appropriate matrices (where N is the size of the, square, matrices involved in multiplication) can be stored in the system, the matrix multiplication operation becomes a version of the Broadcast-Compute-Roll. However, an in-depth discussion of this point is out of scope of this contribution.

Obviously, it is possible to combine recursive and parallel approaches to dense matrix multiplication. Here, the situation becomes even more complex, as irregularity of data movement is exaggerated through the complexity of the underlying hardware (e.g. extra levels of latency of data access). However, with a lot of programmer's work, hybrid approaches outperformed the standard parallel matrix multiplication [72, 37, 32, 59]. Interestingly, the most recent research seems to contain two contradictory claims. First, results presented in [26] support the conclusion that Strassen-type approaches are not practically beneficial on current computer architectures. Second, in [49], benefits of the 2.5D Strassen multiplication are praised. However, careful study of results presented there (see, [49], Figure 1.f) indicates that the apparent performance gain (reported in, [49], Figure 1.e) is purely virtual, as it does not result in substantial reduction of the wall-clock time. This could be used as an argument supporting the conclusions reported in [26].

Let us leave the discussion concerning advantages and disadvantages of specific implementations of parallel dense matrix multiplication and observe that this operation appears also in a much broader context. Specifically, matrix multiplication can be generalized through the theory of algebraic semirings. Let us now look into this topic in more detail.

4. Algebraic semirings in scientific calculations.

Since 1970's, a large number of problems has been combined under a single umbrella, named the *Algebraic Path Problem* (*APP*; see [45, 16, 67]). Furthermore, it was established that the matrix "multiply-and-update" (MMU) operations, in different algebraic semirings, can be used as a centerpiece of various APP solvers.

Let us start from the needed definitions. A closed (scalar) semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ is an algebraic structure defined for a set S, with two binary operations: addition $\oplus : S \times S \to S$ and multiplication $\otimes : S \times S \to S$, a unary operation called *closure* $\circledast : S \to S$, and two constants $\bar{0}$ and $\bar{1}$ in S. Here, we are particularly interested in the set S consisting of matrices. Thus, following [45], we introduce a matrix semiring $(S^{n\times n}, \bigoplus, \bigotimes, \bigstar, \bar{O}, \bar{I})$ as a set of $n \times n$ matrices $S^{n\times n}$ over a closed scalar semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ with two binary operations, matrix addition $\bigoplus : S^{n\times n} \times S^{n\times n} \to S^{n\times n}$ and matrix multiplication $\bigotimes : S^{n\times n} \times S^{n\times n} \to S^{n\times n}$, a unary operation called *closure of a matrix* $\bigstar : S^{n\times n} \to S^{n\times n}$, the zero $n \times n$ matrix $\bar{O}$ whose all elements equal to $\bar{0}$, and the $n \times n$ identity matrix $\bar{I}$ whose all main diagonal elements equal to $\bar{1}$ and $\bar{0}$ otherwise. Here, matrix addition and multiplication are defined as usually in the linear algebra.

As stated, large number of matrix semirings appear in well-studied APPs. We summarize some of them in a table (similar to that presented in [12]). For simplicity of notation, in the Table 4.1, we represent them in a scalar form.

Note that the *Minimum reliability path* problem has not been encountered by the authors before. It was defined on the basis of systematically representing possible semirings—as a natural counterpart to the *Maximum reliability problem* (the only difference is the $\oplus$ operation: *min* instead of *max*). Since the maximum reliability path defines the *best* way to travel between two vertices of a graph; the *Minimum reliability problem* could be interpreted as: finding the *worst* pathway, one that should not be "stepped into").

Table 4.1
Semirings for various APP problems.

S	$\oplus$	$\otimes$	$\circledast$	$\bar{0}$	$\bar{1}$	Application
0,1	$\vee$	$\wedge$	1	0	1	Reflexive and transitive closure of binary relations
$\mathbb{R}$	$+$	$\times$	$1/(1-r)$	0	1	Matrix inversion
$\mathbb{R}_+ \cup +\infty$	min	$+$	0	∞	0	All-pairs shortest paths
$\mathbb{R}_+ \cup +\infty, -\infty$	max	$+$	0	-∞	0	Maximum cost (critical path)
$[0,1]$	max	$\times$	1	0	1	Maximum reliability paths
$[0,1]$	min	$\times$	1	0	1	Minimum reliability paths
$\mathbb{R}_+ \cup +\infty$	min	max	0	∞	0	Minimum spanning tree
$\mathbb{R}_+ \cup -\infty$	max	min	0	-∞	0	Maximum capacity paths

While Table 4.1 summarizes the *scalar* semirings, and *scalar* "multiply-and-add" operations, kernels of blocked algorithms for solving the APP, are based on (block) *matrix* "multiply-and-update" (MMU) operations [74, 51]. Therefore, let us present the relation between the *scalar* (fused) "multiply-and-update" (FMA) operation (ω), and the corresponding *matrix* "multipy-and-update" (MMU) kernel (α), for semirings in Table 4.1 (here, Nb is the size of a matrix block; see, also [68]):

- Matrix Inversion Problem (MIP):
  (α) $a(i,j) \leftarrow a(i,j) + \sum_{k=0}^{Nb-1} a(i,k) * a(k,j)$;
  (ω) $c \leftarrow a \times b + c$;
- Shortest Paths Problem (SPP):
  (α) $a(i,j) \leftarrow \min\{a(i,j), \min_{k=0}^{Nb-1}[a(i,k) + a(k,j)]\}$;
  (ω) $c \leftarrow \min(c, a+b)$;
- Critical Path Problem (CRP):
  (α) $a(i,j) \leftarrow \max\{a(i,j), \max_{k=0}^{Nb-1}[a(i,k) + a(k,j)]\}$;
  (ω) $c \leftarrow \max(c, a+b)$;
- Maximum Capacity Paths Problem (MCP):
  (α) $a(i,j) \leftarrow \max\{a(i,j), \max_{k=0}^{Nb-1} \min[a(i,k), a(k,j)]\}$;
  (ω) $c \leftarrow \max[c, \min(a,b)]$;
- Maximum Reliability Paths Problem (MaRP):
  (α) $a(i,j) \leftarrow \max\{a(i,j), \max_{k=0}^{Nb-1}[a(i,k) \times a(k,j)]\}$;
  (ω) $c \leftarrow \max(c, a \times b)$;
- Minimum Reliability Paths Problem (MiRP):
  (α) $a(i,j) \leftarrow \min\{a(i,j), \min_{k=0}^{Nb-1}[a(i,k) \times a(k,j)]\}$;
  (ω) $c \leftarrow \min(c, a \times b)$;
- Minimum Spanning Tree Problem (MST):
  (α) $a(i,j) \leftarrow \min\{a(i,j), \min_{k=0}^{Nb-1}[\max(a(i,k), a(k,j)]\}$;
  (ω) $c \leftarrow \min[c, \max(a,b)]$.

Summarizing, application of algebraic theory of semirings allows bringing together multiple problems, solution of which involves generalized matrix multiplication. Let us now look how this fits with current and predictable trends in computational sciences, in particular in computational linear algebra.

5. Current trends in high performance computing.

5. Current trends in high performance computing. Recently, high performance computer hardware found itself "in a box" imposed by physics. As the "floor" we can see the limits on miniaturization. It is physically impossible to continue reducing the size of the processor without having to deal with quantum effects. On the "ceiling" we have the heat dissipation problem. The current on-chip processor designs make it extremely difficult to cool the densely packed (miniaturized) transistors. Finally, the memory "walls" make it increasingly complex to feed the data-starving processors. Here, the use of hierarchical memory with up to three levels of cache memory provided only a temporary solution, with a price in workload of software implementers (see, also, below). An obvious solution to these problems is to combine multiple computational units. However, here the "speed of light problem" starts to materialize again. To keep the system synchronous, it is possible to send signal only for a certain distance (within a single clock cycle). This effect can be seen not only when considering design of systems with billions of processors that fill a room of a size of a football stadium. We can observe it also within multi-core chips. For instance, the frequency of Pentium processors reached 3.8 GHz (Pentium 4, Prescott), while the Nvidia Tesla processor runs at about 700 MHz. Here, through the reduction of frequency, it is possible to keep the multiple cores synchronous.

In this way, we have pointed to two recent trends in design of high performance hardware. First, today's multi-core processors can be seen as shared memory parallel computers on-chip. Second, GPU chips represent ideas originating from the SIMD supercomputers of the old. Both trends feature multiple FMA units (from a few FMAs in multi-core processors, up to more than a thousand in the newest GPUs). They also follow two different roadmaps to the exascale computing. One of them involves smaller number of more powerful (more complex) processors, while the other is based on larger number of less powerful (simpler) processing units. In a way, high performance computing has spiraled back to the 1990's. For instance, in 1993, the MassPar MP-2216 array processor with 2048 "cores" took 180's position at the TOP500 computer list [9]. It was capable of delivering 2.4 GFlops. Today's Tesla K20X GPU, produced by Nvidia, has 896 double precision FMAs (2688 single precision FMAs that work in triples to deliver double precision operations) and is capable of delivering 1.31 TFlops [33]. In other words, a TOP200 computer from 1993 was about 500 times slower than a single (multi)processor of today, while both had a comparable number of computational units.

As what concerns software, some researchers work on squeezing performance out of existing computer architectures (see, for instance, recent results from the team led by J. Demmel [49, 71]). Others see the future of computing in proposing complex kernels that are to replace the BLAS standard (see, [80]). However, it is also possible to see the problem from the perspective of William of Ockham, who suggested that "one should proceed to simpler theories." Taking into account what has been said so far, it should be clear that *simplicity and uniformity of data manipulation* should become the driving principle behind exa/zetta-scale computer system design. Here, the work of Sedukhin et.al. [70] focuses on development of a novel computer architecture capable, among others, of delivering efficient parallel matrix multiplication. In the same context, for sparse matrix operations, John Gustafson stated: "Go Ahead, Multiply by Zero!" [34]. His assumption is that in the hardware of the future, cost of arithmetic operations will be so low, in comparison with data movement (and indexing), that fundamental assumptions behind current approaches to computational sparse (and dense) linear algebra will have to be re-evaluated. Interestingly, the team of J. Dongarra has recently initiated a new project, in which they investigate possibility of reintroduction of systolic architectures to the computing mainstream [43]. Here, it is also worthy envisioning that this could mean that we may, some time in the future, forget about rectangular matrices (in computational practice). Instead we will pad them with zeros to make them square and in this way match the square arrangements of FMA's within processors. Since the price of an FMA is currently (June 2013) at about 22 US cents and dropping (see, [77]), such approach does not seem so unreasonable; especially, taking into account the cost of data manipulations involved in dealing with rectangular structures. This should be kept in mind when considering the fact that the APP algorithms (mentioned above) involve square matrices only.

Let us present a few more examples of a slow shift in view on programming modern (and future) high-performance computers (which are expected to be build of hundreds of thousands of processors with thousands of FMA units each). First, initial results concerning an implementation of the 2D FFT (2048×2048)-point, on a 24-wide FMA Cell/B.E.@PS3 processor shows only a moderate difference ($\sim$3 times) in performance over non-recursive matrix multiply and add-based 2D DFT, despite almost 30 times difference in the number of arithmetic operations [79]. Moreover, it was recently reported (in [48]) that, by using two quad-core Intel Nehalem CPUs, the direct convolution approach outperforms the FFT-based approach by at least five times, even when the associated arithmetic operation count is approximately two times higher. This "imbalance" towards non-recursive approach is directly related to efficient use of multicore processors achieved through efficient dense MMU implementation.

Second, the early work in sparse linear algebra was guided by the desperate need of saving memory. As a result, a number of "compressed matrix formats" has been proposed and experimented with. However, in the most recent work, instead of dealing with individual matrix entries, the basic element becomes a "dense block" (of size related to the processor cache; see, for instance [78, 50] and references collected there). Here, we observe the same general pattern as above, where simplicity of data access pattern(s) compensates for the higher operation count caused by multiplication by zero (note that some of such blocks may have only a few non-zero elements).

Separately, the work reported in [50] illustrates one more important aspect of using highly optimized, memory-preserving schema to deal with sparse matrix multiplication. The code generator that allows to deal with various forms of sparse matrix-vector multiplication, for various data types and matrix formats is approximately 6,000 lines long. However, the generated code is more than 100,000 lines long. This code can efficiently work on hierarchical memory multicore systems with up to 16 cores (see, [50]), but will require further tuning for larger number of FMAs (and it is neither GPU nor distributed memory computers oriented). This being the case, when thinking about fast matrix / matrix-vector multiplication, one needs to consider also the programming effort required to develop and later modify (re-optimize) codes based on complex data structures and movements. This observation provides also context for the question of long-term feasibility of approaches similar to the GOTOBLAS [31]. Furthermore, the original direction of the BLAS project needs to be considered. Part of the success of the BLAS was related to its simplicity (including relatively small number of basic routines). Therefore, it is not clear if proposing a standard with many complex kernels (as in [80]) is going to be successful in a long run.

Finally, let us look into relation between the APP and the recent trends in computing. While algebraic semirings can be seen as a simple "unification through generalization" of a large class of computational problems, they should be viewed in the context of, mentioned above, success of fused multiply-and-add (FMA) units. Note for instance that, the GPU processors from Nvidia and AMD combine large number of FMA units (e.g. the Nvidia's Tesla chip allows 2688 single-precision FMA operations completed in a single clock cycle [33]). In this context, recent experiments show that FMA-based kernels speed-up ($\sim 2\times$) solution of many scientific, engineering, and multimedia problems, which are based on matrix transforms [35]. However, other APPs "suffer" from lack of hardware support for the needed scalar FMA operations (e.g. those listed in Table 4.1). The need for min or/and max operations introduces one or two conditional branches, or comparison/selection instructions, which are highly undesirable for deeply pipelined processors. Recall that each of these operations is repeated Nb-times in the corresponding kernel (see, Section 4), while the kernel itself is called many times in the blocked APP algorithm. Here, note the recent results, reported in [68], which involve evaluation of an MMU operation in different semirings, on the Cell/B.E. processor. They showed that the "penalty" for lack of a *generalized* FMA unit may be up to 400%. This can be also interpreted as follows: having an FMA unit, capable of supporting operations and special elements from Table 4.1 could speed-up solution of APP problems by up to 4 times. Interestingly, we have just found that the AMD Cypress GPU processor supports the combined scalar (min, max)-operation through a single call with only 2 clock cycles per result. In this case, the Minimum Spanning Tree (MSP) problem (see, Table 4.1) could be solved more efficiently than previously realized. Furthermore, this could mean that the AMD hardware has $-\infty$ and ∞ constants already build-in. This, in turn, could constitute an important step towards hardware support of generalized FMA operations, needed to realize all kernels listed in Table 4.1.

6. Proposed generalized multiply-update operation. Let us now summarize the main points made thus far. First, future parallel computers will involve hundreds of millions (if not billions) of FMA units in a single (super)computing system. Such systems are likely to remain within the same (inflation adjusted) price range as today's largest supercomputers. As a result, price per FMA unit will be further substantially reduced (likely to reach less than 1 US cent per GFlop). Second, unless a great progress is going to be made in the area of quantum computing, the limitations imposed by physics will not go away, keeping the design of computer hardware in a "physics box." Furthermore, it is unclear if quantum computing will be applicable to all classes of computational problems. As a result, *simplicity and uniformity of data movement* has to become the guiding principle of design of both hardware and software for solving computationally intensive problems. This being the case, we predict a diminishing role of divide-and-conquer methods. Here, we mean approaches focused on replacing multiplications with additions/subtractions, and/or reduction of the total number of arithmetical operations, while introducing complex data movement and need for extra memory. Both of them are precisely the problems that stand in the way to development of exa/zetta-flop computers. Third, sometimes without realizing this, scientists solving large number of computational problems, have been working within algebraic semirings. Algebra of semirings involves not only standard linear algebra, but also a large class of other APPs.

Solutions to these problems involve execution of a large number of matrix "multiply-and-add" operations rooted in generalized scalar FMA operations. In this context, we have illustrated the positive effects of development of FMA processing units, and discussed potential benefits of "upgrading" such units to become *generalized FMA units*, capable of dealing with semiring-defined operations listed in Table 4.1. Finally, we have stressed that simplicity of data movement /access concerns also simplicity of code writing. In other words, it seems that the world of scientific computing has reached the era of Ptolomeic epicycles, investigating methods, which are effective in squeezing performance of existing supercomputers, but are going to be ineffective in a long run.

Based on these considerations, we can define a generic form of the matrix "multiply-and-update" (MMU) operation

$$\mathtt{C} \leftarrow \mathtt{GMMU}[\otimes,\oplus](\mathtt{A},\mathtt{B},\mathtt{C}) : \mathtt{C} \leftarrow \mathtt{C} \oplus \mathtt{A}^{\mathtt{T/N}} \otimes \mathtt{B}^{\mathtt{T/N}}, \tag{6.1}$$

where the $[\otimes,\oplus]$ operations originate from different matrix semirings, while $\mathtt{T/N}$ denotes the fact the either (or both) matrix(ces) A and/or B can be used in a transposed form. Note that, like in the scalar FMA operations, the generalized matrix *addition* or *multiplication*, can be implemented by making matrix A (or B) $= \bar{I}$ for addition, or matrix $C = \bar{0}$ for multiplication (where, the appropriate $\bar{0}$ and $\bar{I}$ matrices have been defined in Section 4).

Finally, observe that the proposed approach leads to new ways of development (design and implementation) of efficient codes solving variety of APPs. Upon reflection, it should also become clear that this approach can be seen as a generalization of the level 3 BLAS. In Section 2 we have discussed the development of the BLAS standard, in the context of development of computer hardware, and growing understanding of the nature of computational linear algebra. We have also mentioned that the BLAS standard did not move smoothly from Fortran to the object oriented languages. Let us therefore, summarize the state-of-the-art in the area of object oriented BLAS and object oriented libraries for computational linear algebra.

7. State-of-the-art in object oriented BLAS. While our work extends and generalizes matrix multiplication, taking into account changes in computing that took place within last 30 years, its current realization should be object oriented. Therefore, let us start from a summary of selected object oriented realizations of numerical linear algebra in general, and BLAS in particular: MTL [11], uBLAS [4], TNT [8], Armadillo [3] and Eigen [7]. Other object oriented projects that could be considered pertinent to the material presented here, have been summarized in [54].

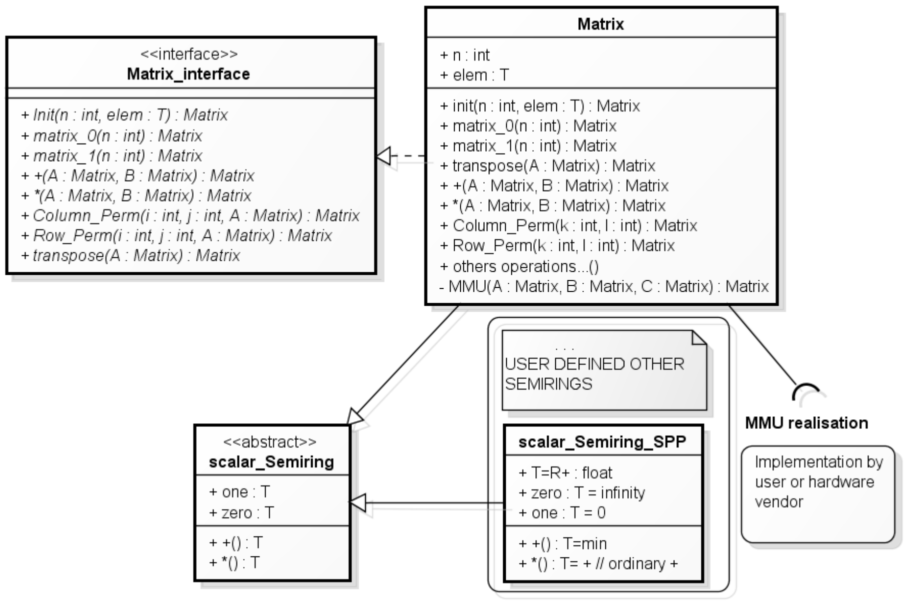
The uBLAS project [4] was focused on design of a C++ template class library for BLAS level 1, 2 and 3; for dense, packed, and sparse matrices. The primary goal of uBLAS was to provide usable software for the scientific computing community. However, its additional goal was to experimentally evaluate if the abstraction penalty, resulting from object orientation (use of matrix and vector classes), is acceptable. According to the information available at [4], the design of the uBLAS was guided by results originating from the following projects: (i) Blitz++ [1] by Todd Veldhuizen, (ii) POOMA [2] by Scott Haney et al., and MTL [11] by Jeremy Siek et al. Data found on the Web indicates that the project was completed around 2002 and later its results have been included in the BOOST [6] library of object oriented mathematical software. Comparing Web pages of the 2010 and 2012 releases of BOOST it is clear that the uBLAS is not developed further, but only maintained by the team of the BOOST project.

Year 2004, marks the end of the life cycle of the Template Numerical Toolkit (TNT) project from the NIST. The TNT is a collection of interfaces for sparse and dense matrices. In addition, a C++ reference implementations of these objects are provided. The library, while not updated since 2004, can still be downloaded from the project Web site and experimented with (the site is signed by Roldan Pozo, the last update took place in March 2004, and the project status is described as: active maintenance).

Interestingly, one of the projects that was taken into consideration while developing the uBLAS, the Matrix Template Library (MTL), outlived the explicitly traceable activity of the uBLAS project. Specifically, in 2005 the MTL project moved from US to Germany and changed the lead personnel. Furthermore, when designing the MTL 4, only the main ideas from the MTL 2 were used, but the code has been written from scratch. The MTL 4 team continued publishing research papers until, approximately, 2009. Since then the project was turned into a commercial endeavor through the SimuNova company, which provides open source, supercomputing, and GPU versions of the MTL 4.

Finally, there are two projects that are vigorously pursued today. These are the Armadillo (with the last release on February 20, 2013) and the Eigen (with the last release on November 5, 2012). Both of them provide support for an extensive set of matrix operations as well as other computational kernels (Eigen, in particular). While both libraries are open source, the Armadillo provides direct support for vendor optimized matrix libraries: MKL and ACML. The Eigen, on the other hand, has been optimized for vector processors (and specially optimized for small matrices), and accommodates large body of supplementary community implemented codes.

8. Initial object oriented model of generalized matrix multiplication. Following the ideas underlying the above summarized projects, as well as the main points discussed above, in Figure 8.1 we depict our proposed initial object oriented model for the generalized matrix multiplication. The proposed model is language independent, so that it will have to be appropriately adjusted if it is to be actually implemented, for instance, in C++ or Java. Here, and in the next section, we follow the lead of the creators of the BLAS standard, who in [44, 28, 27] have not only discussed the general ideas concerning BLAS, but also introduced some ideas concerning its implementation.



<<interface>>
Matrix_interface
+ Init(n : int, elem : T) : Matrix
+ matrix_0(n : int) : Matrix
+ matrix_1(n : int) : Matrix
+ +(A : Matrix, B : Matrix) : Matrix
+ *(A : Matrix, B : Matrix) : Matrix
+ Column_Perm(i : int, j : int, A : Matrix) : Matrix
+ Row_Perm(i : int, j : int, A : Matrix) : Matrix
+ transpose(A : Matrix) : Matrix
Matrix
+ n : int
+ elem : T
+ init(n : int, elem : T) : Matrix
+ matrix_0(n : int) : Matrix
+ matrix_1(n : int) : Matrix
+ transpose(A : Matrix) : Matrix
+ +(A : Matrix, B : Matrix) : Matrix
+ *(A : Matrix, B : Matrix) : Matrix
+ Column_Perm(k : int, l : int) : Matrix
+ Row_Perm(k : int, l : int) : Matrix
+ others operations...()
- MMU(A : Matrix, B : Matrix, C : Matrix) : Matrix
. . .
USER DEFINED OTHER SEMIRINGS
MMU realisation
Implementation by user or hardware vendor
<<abstract>>
scalar_Semiring
+ one : T
+ zero : T
+ +() : T
+ *() : T
scalar_Semiring_SPP
+ T=R+ : float
+ zero : T = infinity
+ one : T = 0
+ +() : T=min
+ *() : T= + // ordinary +

FIG. 8.1. *General schema of the proposed object oriented model of MMU matrix multiplication*

The main class of the proposed model is the *Matrix* class. This class is based on the user defined semirings, which are represented in the *scalar_Semiring* class. In Figure 8.1, we represent the fact that users can define different semirings. Specifically, defining a semiring involves defining the abstract (generalized) operations *addition* and *multiplication* (and *closure* for closed semirings), as well as special elements $\bar{0}$ and $\bar{1}$ (see, Section 4 and Table 4.1). In the case of the *Matrix* class, we can see that the matrix is defined by two generic parameters: matrix size N and type of its elements.

The *scalar_Semiring* class is the basis for the MMU operation (it defines scalar operations within the generalized MMU). Our assumption is that implementation of this operation will be vendor-supported (in a similar way that the BLAS operations have been implemented by the Cray or the IBM). For instance, if specific scalar FMA operations, corresponding to the user defined semiring, are available in the hardware, they will be utilized in the implementation of the MMU operation. Furthermore, if parallel implementation of the MMU is supported on a given computer hardware (e.g. on a GPU), it will be used here.

9. Sample realization. Let us now outline the main features of the proposed realization of the above described approach. Here, let us observe that one of our important goals is to simplify code development (in a way that the BLAS standard simplified it 30+ years ago). This means, that we want the code to be written by the programmer to be as simple as possible, with most details of the implementation hidden from her. In what follows, we distinguish between the *user*, who will use the proposed model to write codes to solve her problem and the *implementer* who will develop complete model of the generalized MMU. With this in mind, let us start from defining the interfaces. The first one of them is the interface `Matrix_interface`. This interface defines operations that are made available to the user to write his codes.

```
/* T - type of matrix element */

interface Matrix_interface {
  Init(n);/initialisation of square matrix nxn
  Matrix matrix0(n) {/*generalized zero matrix*/}
  Matrix matrix11(n) {/*generalized identity matrix*/}
  Matrix operator + /*generalized matrix addition A+B*/
  Matrix operator * /*generalized matrix product A*B*/
  Matrix transpose(A) {/*transposition of matrix A*/}
  /*generalized permutation of column i and j in matrix A*/
  Matrix Column_Permutation (A,i,j)
  /*generalized permutation of row i and j in matrix A*/
  Matrix Row_Permutation (A,i,j)
...
}
```

As we can see, the implementer is provided with ability to create matrix objects, as well and zero and identity matrices (for a given semiring). Furthermore, we define generalized multiply and add operators, as well as two permutation matrices. This interface can be extended to include other matrices / operations needed by the user.

The second interface is the `scalar_Semiring_interface`. This is where the scalar semiring is specified.

```
interface scalar_Semiring_interface{
  /*Operations:*/
    +,*
  ...
}

abstract class scalar_Semiring {
public:
  //T -- type of element;
  zero,one:T;
    +: c=a+b;
    *: c=a*b;
}
```

Here, the generalized scalar operations + and $*$, as well as scalar elements $\bar{0}$ and $\bar{1}$ are specified, in the abstract class `scalar_Semiring`. Specification of this class has to be provided by the user to define the semiring that she would like to work with in her code.

With these two interfaces in place we can now define the core class `Matrix`. This class is *not* made visible to the user (it is *internal* to the proposed realization of the generalized MMU). It inherits the `scalar_Semiring` interface and implements the interface `Matrix_interface`.

```
class Matrix inherit scalar_Semiring implement Matrix_interface {

  T: type of element;/*double, single,...*/
  n:int;
  // Methods
  Init(n);/initialisation of square matrix nxn
  Matrix matrix_0(n) {/*0 matrix*/}
  Matrix matrix_1(n) {/*identity matrix*/}
  Matrix matrix_1P(i,j,n){
  /*identity matrix with interchanged columns i and j*/
  }
  Matrix transpose(A:Matrix){/*MMU-based transposition of A*/}
  Matrix operator + {A,B:Matrix}
    {return MMU(A,matrix_1(n),B,a,b)}
  Matrix operator * {A,B:Matrix}
    {return MMU(A,B,matrix_0,a,b)}
  Matrix Column_Permutation (A,i,j){
    P=matrix_1P(i,j,n);
    O=matrix_0(n);
     return MMU(P,A,O)}
  Matrix Row_Permutation (A,i,j){
    P=matrix_1P(i,j,n);
    O=matrix_0(n);
     return MMU(A,P,O)}
  ...
  private MMU(A,B,C:Matrix(n)){
    return "vendor/implementer specific realization of
        MMU = C + A*B where
      + and * are from class scalar_Semiring"}
...
}
```

The most important part of this class is the private function *MMU*. This is the actual realization of the MMU operation (see, equation 6.1). It is implementer/vendor specific. In other words, user can perform matrix operations: $A \oplus B$, $A \otimes B$, or $C \oplus A \otimes B$, written in the code as $A + B$, $A * B$, or $C + A * B$ without any knowledge that they are actually realized through invocation of the *MMU* function. Furthermore, the *MMU* function can be implementer specified or hardware vendor provided, and be based on any of the existing matrix multiplication algorithms (see, section 3).

As far as the matrices in transposed form are concerned, we have to distinguish two senses in which the transpose can materialize. First, transpose may appear as an operation that has to be actually performed on a matrix. In this case, we will apply the $tanspose(A)$ operation (available within the *Matrix* class; see, also [64]). However, transpose may also appear within the context of the MMU operation. Here, as shown in [62], as long as the transpose concerns only one of the two matrices (A or B), it is possible to complete the MMU operation without actually transposing the matrix. Therefore we assume here, that the compiler confronted with code involving $C \leftarrow C + transpose(A) * B$ or $C \leftarrow C + A * transpose(B)$, will call the appropriate implementation of the MMU. In the case, when the user asks for $C \leftarrow C + transpose(A) * transpose(B)$, the actual transpose will be performed on one of the two matrices (e.g. $transpose(A)$) and then the appropriate MMU implementation will be called to complete the operation (e.g. $C \leftarrow C + A * transpose(B)$).

Observe also, that matrix column permutation and matrix row permutation have been defined as operations `Column_Permutation` and `Row_Permutation`, which are implemented through a call to the MMU function with appropriate matrices (see, also [69]).

In the next snippet we show the class `scalar_Semiring`, rewritten for the Shortest Path Problem (point 2, in Figure8.1). After defining this class the user can simply apply the MMU operation within his implementation of the solver. Obviously, this operation can be applied to the whole matrix or to appropriate blocks (in a blocked solver).

```

/*T = R+ PLUS infinity, general add = min,
general product = +, ZERO = infinity, ONE = 0; */

class scalar_Semiring {
  zero="infinity";
  one=0;
  scalar operator +(a,b:T){return min(a,b)};
  scalar operator *(a,b:T){return a+b};
}
```

Observe that after this definition, operation defined in the code as $C + A * B$ (with or without the transpose) will be performed within the needed semiring. Obviously, similar definitions can be easily instantiated on the basis of the remaining semirings listed in Table 4.1, as well as semirings materializing in other AAPs.

10. Concluding remarks. The aim of this paper was to summarize and extend recent research results concerning role of dense matrix multiplication in scientific computing. First, it was argued that with the decreasing price of arithmetical operations and abundance of memory, and the increasing cost of data access / move, the old approaches that focused on reducing number of arithmetical operations and memory use need to be reconsidered. Second, it was shown how the Algebraic Path Problem unifies different problems through the theory of algebraic semirings. This allowed to define the generalized matrix multiply and update operation. Finally, a language independent object oriented model of this operation was presented. In the near future we plan to combine the proposed approach with the, recently introduced, mesh-of-tori based systems (see, [63]) to show how it can be effectively applied to a larger class of matrix "manipulations" implemented through matrix multiplications.

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

REFERENCES

[1] http://blitz.sourceforge.net/.
[2] http://acts.nersc.gov/pooma/.
[3] Armadillo c++ linear algebra library. http://arma.sourceforge.net/.
[4] Basic linear algebra. http://www.boost.org/doc/libs/1_35_0/libs/numeric/ublas/doc/index.htm.
[5] Blacs.
[6] Boost c++ liberaries. http://www.boost.org/.
[7] Eigen wiki. http://eigen.tuxfamily.org/index.php?title=Main_Page.
[8] Template numerical toolkit. http://math.nist.gov/tnt/.
[9] Top500 list – june 1993. http://www.top500.org/list/1993/06/?page=2.
[10] Motorola x8 mobile computing system. http://www.motorola.com/us/X8-Mobile-Computing-System/x8-mobile-computing-system.html, 2013.
[11] Overview of mtl4. http://www.simunova.com/en/node/24, 01 2013.
[12] S. Kamal Abdali and B. David Saunders. Transitive closure and related semiring properties via eliminants. *Theor. Comput. Sci.*, 40:257–274, November 1985.
[13] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. of Res. and Develop.*, 38(6):673–681, 1994.
[14] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
[15] David H. Bailey, King Lee, and Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:357–371, 1991.
[16] V. Batagelj. Semirings for social network analysis. *Journal of Mathematical Sociology*, 19(1):53–68, 1994.
[17] K. Boland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, 1994.
[18] J.D. Bruguera and T. Lang. Floating-point fused multiply-add: reduced latency for floating-point addition. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 42–51, June 2005.
[19] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.
[20] Siddhartha Chatterjee, Leonardo R. Bachega, Peter Bergner, Kenneth A. Dockser, John A. Gunnels, Manish Gupta, Fred G. Gustavson, Christopher A. Lapkowski, Gary K. Liu, Mark P. Mendell, Rohini D. Nair, Charles D. Wait, T. J. Christopher Ward, and Peng Wu. Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L. *IBM Journal of Research and Development*, 49(2-3):377–392, 2005.
[21] Jaeyoung Choi, James Demmel, Inderjit S. Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *PARA*, pages 95–106, 1995.
[22] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *PARA*, pages 107–114, 1995.
[23] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[24] Marius Cornea, John Harrison, and Ping Tak Peter Tang. Intel Itanium Floating-point architecture. In *WCAE'03: Proceedings of the 2003 workshop on Computer Architecture Education*, page 3, New York, NY, USA, 2003. ACM.

[25] Cliff Cyphers and Marcin Paprzycki. Multiplying matrices on the cray-practical considerations. *CHPC Newsletter*, 6(6):77–82, June 1991.

[26] Paolo D'Alberto and Alexandru Nicolau. Using recursion to boost ATLAS's performance. In *ISHPC*, pages 142–151, 2005.

[27] J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[28] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Software*, 14:1–17, 1988.

[29] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[30] Maria Ganzha, Stanislav Sedukhin, and Marcin Paprzycki. Object oriented model of generalized matrix multipication. In *FedCSIS*, pages 439–442, 2011.

[31] Kazushige Goto and Robert van de Geijn. High Performance Implementation of the Level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1).

[32] Brian Grayson and Robert Van De Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, mar 1996.

[33] NVIDIA Group. Tesla K20X GPU Accelerator. Technical report, NVIDIA, November 2012.

[34] John L. Gustafson. Algorithm leadership. *HPCwire*, Tabor Communications, April 06, 2007.

[35] Fred G. Gustavson, José E. Moreira, and Robert F. Enenkel. The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to java and high-performance computing. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 4. IBM Press, 1999.

[36] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Softw.*, 16(4):352–368, December 1990.

[37] S. Hunold, T. Rauber, and G. Rünger. Combining building blocks for parallel multi-level matrix multiplication. *Parallel Comput.*, 34(6–8):411–426, 2008.

[38] IBM. Essl library, 2014.

[39] Cray Research Inc. Unicos math and scientific library reference manual sr-2081, 1990.

[40] A. Kamel, M. Kindelan, and P. Sguazzero. Seismic computations on the IBM 3090 vector multiprocessor. *IBM Systems journal*, 27(4):510–527, 1988.

[41] Vladik Kreinovich. Itanium's new basic operation of fused multiply-add: theoretical explanation and theoretical challenge. *SIGACT News*, 32(1):115–117, 2001.

[42] S.Y. Kung. *VLSI Array Processors*. Prentice Hall, 1988.

[43] Jakub Kurzak, Piotr Luszczek, Mark Gates, Ichitaro Yamazaki, and Jack Dongarra. Virtual systolic array for qr decomposition. Technical report, University of Tennessee, 2012.

[44] C. L. Lawson, R. J. Hanson, R. J. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[45] Daniel J. Lehmann. Algebraic structures for transitive closure. *Theor. Comput. Sci.*, 4(1):59–76, 1977.

[46] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.

[47] Jin Li, Anthony Skjellum, and Robert D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency - Practice and Experience*, 9(5):345–389, 1997.

[48] O. Lindtjorn, R. Clapp, O. Pell, Haohuan Fu, M. Flynn, and Haohuan Fu. Beyond traditional microprocessors for geoscience high-performance computing applications. *Micro, IEEE*, 31(2):41–49, March-April 2011.

[49] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. Communication-avoiding parallel Strassen: implementation and performance. In *SC*, page 101, 2012.

[50] Michele Martone, Salvatore Filippone, Pawel Gepner, Marcin Paprzycki, and Salvatore Tucci. Use of hybrid recursive csr/coo data structures in sparse matrices-vector multiplication. In *IMCSIT*, pages 327–335, 2010.

[51] Kazuya Matsumoto and Stanislav G. Sedukhin. A solution of the all-pairs shortest paths problem on the cell broadband engine processor. *IEICE Transactions*, 92-D(6):1225–1231, 2009.

[52] R. K. Montoye, E. Hokenek, and S. L. Runyon. Design of the IBM RISC system/6000 floating-point execution unit. *IBM J. Res. Dev.*, 34(1):59–70, 1990.

[53] Robert K. Montoye, Erdem Hokenek, and Stephen L. Runyon. Design of the IBM RISC System/6000 Floating-Point Execution Unit. *IBM Journal of Research and Development*, 34(1):59–70, 1990.

[54] Claire Mouton. A study of the existing linear algebra libraries that you can use from c++ (une étude des bibliothèques d'algèbre linéaire utilisables en c++). *CoRR*, abs/1103.3020, 2011.

[55] S.M. Mueller, C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S.H. Dhong. The vector floating-point unit in a synergistic processor element of a cell processor. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 59–67, June 2005.

[56] Frank P. O'Connell and Steven W. White. Power3: The next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6):873–884, 2000.

[57] V. Pan. Strassen's algorithm is not optimal: Trililnear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *FOCS*, pages 166–176, 1978.

[58] Marcin Paprzycki. Comparison of Gaussian elimination algorithms on a Cray Y-MP. *Linear Algebra and Its Applications*, 172:57–69, 1992.

[59] Marcin Paprzycki. Parallel matrix multiplication-can we learn anything new? *CHPC Newsletter*, 7(4):55–59, February 1992.

[60] Marcin Paprzycki. Parallel Gaussian elimination algorithms on a CRAY Y-MP. *Informatica*, 19(2):235–240, 1995.
[61] Marcin Paprzycki and Cliff Cyphers. Using strassen's matrix multiplication in high performance solution of linear systems. *Journal of Computers in Mathematics Applications*, 31(4/5):55–61, 1996.
[62] Abhijeet A. Ravankar. A new "mesh-of-tori" interconnection network and matrix based algorithms. Master's thesis, University of Aizu, September 2011.
[63] Abhijeet A. Ravankar and Stanislav G. Sedukhin. Mesh-of-tori: A novel interconnection network for frontal plane cellular processors. In *ICNC*, pages 281–284, 2010.
[64] Abhijeet A. Ravankar and Stanislav G. Sedukhin. An O(n) time-complexity matrix transpose on torus array processor. In *ICNC*, pages 242–247, 2011.
[65] Sara Robinson. Towards an optimal algorithm for matrix multiplication. *SIAM News*, 38(9), 2005.
[66] Richard M. Russell. The Cray-1 computer system. *Commun. ACM*, 21:63–72, January 1978.
[67] S. G. Sedukhin, T. Miyazaki, and K. Kuroda. Orbital systolic algorithms and array processors for solution of the algebraic path problem. *IEICE Transactions on Information and Systems*, E93.D(3):534–541, 2010.
[68] Stanislav G. Sedukhin and Toshiaki Miyazaki. Rapid*closure: Algebraic extensions of a scalar multiply-add operation. In *CATA*, pages 19–24, 2010.
[69] Stanislav G. Sedukhin and Marcin Paprzycki. Generalizing matrix multiplication for efficient computations on modern computers. In *PPAM (1)*, pages 225–234, 2011.
[70] Stanislav G. Sedukhin, Ahmed S. Zekri, and Toshiaki Myiazaki. Orbital algorithms and unified array processor for computing 2D separable transforms. *Parallel Processing Workshops, International Conference on*, 0:127–134, 2010.
[71] Edgar Solomonik, Bhinav Bhatele, and James Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of the ACM/IEEE Supercomputing Conference*. ACM, November 2011.
[72] Fengguang Song, Shirley Moore, and Jack Dongarra. Experiments with Strassen's algorithm: from sequential to parallel. In *International Conference on Parallel and Distributed Computing and Systems (PDCS06)*. ACTA Press, 2006.
[73] V. Strassen. Gaussian Elimination is Not Optimal. *Numerische Mathematik*, 14(3):354–356, 1969.
[74] Akihito Takahashi and Stanislav Sedukhin. Parallel blocked algorithm for solving the algebraic path problem on a matrix processor. In *HPCC*, pages 786–795, 2005.
[75] Robert van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. Technical Report TR-95-13, The University of Texas, apr 1995.
[76] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
[77] Wikipedia. Flops. `http://en.wikipedia.org/wiki/FLOPS`.
[78] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35:178–194, March 2009.
[79] Syoudai Yokoyama. Porting matrix inversion and 2D DFT algorithms to Cell/B.E. Master's thesis, The University of Aizu, Japan, 2011.
[80] Field G. Van Zee and Robert A. van de Geijn. Blis: A modern alternative to the BLAS. Technical report, University of Texas, 2012.

Edited by: Dana Petcu
Received: May 1, 2014
Accepted: Jun 24, 2014