Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 439–442

ISBN 978-83-60810-22-4

Developing an OO Model for Generalized Matrix Multiplication: Preliminary Considerations

Maria Ganzha, Marcin Paprzycki
Systems Research Laboratory
Polish Academy of Sciences
01-447 Warszawa, Poland
Email: Maria.Ganzha@ibspan.waw.pl

Stanislav G. Sedukhin
Distributed Parallel Processing Laboratory
The University of Aizu
Aizuwakamatsu City, Fukushima 965-8580, Japan
Email: sedukhin@u-aizu.ac.jp

***Abstract*—Recent changes in computational sciences force reevaluation of the role of dense matrix multiplication. Among others, this resulted in a proposal to consider generalized matrix multiplication, based on the theory of algebraic semirings. The aim of this note is to outline an initial object oriented model of the generalized matrix-multiply-add operation.**

I. Introduction

THE DENSE matrix multiplication appears in many computational problems. Its arithmetic complexity ($O(n^3)$) and inherent data dependencies pose a challenge for reducing its *run-time* complexity. There exist three basic approaches to decrease the execution time of dense matrix multiplication.

(1) Reducing the number of (time consuming) scalar multiplications, while increasing the number of (much faster) additions; see, discussion and references in [1]. These approaches had very good theoretical arithmetical complexity, and worked well when implemented on computers with a single processor and main memory. However, due to complex data access patterns they became difficult to efficiently implement on computers with hierarchical memory. Furthermore, recursive matrix multiplication requires extra memory; e.g. Cray's implementation of Strassen's algorithm required extra space of $2.34 * N^2$ for matrices of size $N \times N$.

(2) Parallelization of matrix multiplication, which is based one of four classes of schedules ([2]): (i) Broadcast-Compute-Shift; (ii) All-Shift-Compute (or Systolic); (iii) Broadcast-Compute-Roll; and (iv) Compute-Roll-All (or Orbital). The latter 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 (retina I/O), etc.

(3) Combination of these approaches, where irregularity of data movement is exaggerated through the complexity of the underlying hardware. Interestingly, work on recursive (and recursive-parallel) matrix multiplication seems to be subsiding, as the last known to us paper comes from 2006 [3].

Note that, in sparse matrix algebra the main goal was to save memory; achieved via indexing structures storing information about non-zero elements (resulting in complex data access patterns; [4]. However, nowadays the basic element becomes a dense block while regularity of data access compensates for the multiplications by zero [5].

Generalized matrix multiplication appears in the Algebraic Path Problem (APP), examples of which include: finding the most reliable path, finding the critical path, finding the maximum capacity path, etc. Here, a generalized is based on the algebraic theory of semirings (see [6] and references collected there). Note that, standard linear algebra (with its matrix multiplication) is an example of an algebraic (matrix) semiring. Application of algebraic semirings to "unify through generalization" a large class of computational problems, should be viewed in the context of recent changes in CPU architectures: (1) popularity of fused miltiply-add (FMA) units, which take three scalar operands and produce a result of $c \leftarrow a \cdot b + c$ in a single clock cycle, (2) increase of the number of cores-per-processor (e.g. recent announcement of 10-core processors from Intel), and (3) success of GPU processors (e.g. the Fermi architecture from Nvidia and the Cypress architecture from AMD) that combine multiple FMA units (e.g. the Fermi architecture delivers in a single cycle 512 single-precision, or 256 double-precision FMA results).

Finally, the work reported in [4] illustrates a important aspect of highly optimized codes that deal with complex matrix structures. While the code code generator, is approximately 6,000 lines long, the generated code is more than 100,000 lines. Therefore, when thinking about fast matrix multiplication, one needs to consider also the programming cost required to develop and later update codes based on complex data structures and movements.

II. Algebraic semirings in scientific calculations

Since 1970's, a number of problems have been combined into the *Algebraic Path Problem* (*APP*; see [7]). The APP includes problems from linear algebra, graph theory, optimization, etc. while their solution draws from theory of semirings.

A closed 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* $* : S \to S$, and two constants $\bar{0}$ and $\bar{1}$ in S. Here, we are particularly interested in the case when the elements of the set S are matrices. Thus, following [7], 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

1) Matrix Inversion Problem:
(α) $a(i,j) = a(i,j) + \sum_{k=0}^{N-1} a(i,k) \times a(k,j)$;
(ω) $c = a \times b + c$;
2) All-Pairs Shortest Paths Problem:
(α) $a(i,j) = \min\{a(i,j), \min_{k=0}^{N-1}[a(i,k) + a(k,j)]\}$;
(ω) $c = \min(c, a+b)$;
3) Minimum Spanning Tree Problem:
(α) $a(i,j) = \min\{a(i,j), \min_{k=0}^{N-1}\ [\max(a(i,k), a(k,j)]\}$;
(ω) $c = \min[c, \max(a,b)]$.

Fig. 1. Matrix and scalar semiring operations for sample APP problems

$\bigoplus : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$ and matrix multiplication $\bigotimes : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$, a unary operation called *closure of a matrix* $\bigstar : S^{n\times n} \rightarrow S^{n\times n}$, 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 linear algebra. Note that, special cases matrices that are non-square, symmetric, structural, etc., while not usually considered in the theory of semirings, are also handled by the above provided definition.

The existing blocked algorithms for solving the APP, are rich in generalized block MAA operations, which are their most compute intensive parts [8]. In the generalized block MMA, addition and multiplication originate from any semiring (possibly different than the standard numerical algebra). In Figure 1 we present the relation between the scalar multiply-add operation (ω), and the corresponding MMA kernel (α), for different semirings for three sample APP applications; here, N is the size of the matrix block (see, also [8]).

Overall, the generalized MMA is one of key operations for the APP problems, including MMA-based numerical linear algebra algorithms, which include block-formulations of linear algebraic problems (as conceptualized in the level 3 BLAS operations [9], and applied in the LAPACK library [10]).

III. Matrix operations and computer hardware

In 1970's it was realized that many matrix algorithms consist of similar building blocks (e.g. a vector update, or a dot-product). As a result, Cray computers provided optimized vector operations: $y \leftarrow y + \alpha x$, while IBM supercomputers featured optimized dot products. This resulted also in creation of libraries of routines for scientific computing (e.g. the $scilib$ library on the Cray's, and the $ESSL$ library on the IBM's). Separately, the first hardware implementation of the fused multiply-add (FMA) operation was delivered in the IBM RS/6000 workstations [11]. Following this path, most current processors from IBM, Intel, AMD, NVidia, and others, include scalar floating-point FMA [12]. Observe that, the basic arithmetic operations: add and multiply, are performed by the FMA unit by making $a = 1.0$ (or $b = 1.0$) for addition, or $c = 0.0$ for multiplication. Therefore, the two fundamental constants, 0.0 and 1.0, have to be available in the hardware. Therefore, processors that perform the FMA implement in hardware the scalar $(+, \times)$ semiring.

Obviously, the FMAs speed-up ($\sim 2\times$) the solution of scientific, engineering, and multimedia algorithms based on the linear algebra (matrix) transforms [13]. On the other hand, lack of hardware support penalizes APP solvers from other semirings. For instance, in [8] authors have showed that the "penalty" for lack of a generalized FMA unit in the Cell/B.E. processor may be up to 400%. Obviously, this can be seen from the "positive side." Having hardware unit fully supporting operations listed in Figure 1 would speed up solution of APP problems by up to 4 times. Interestingly, we have just found that the AMD Cypress GPU processor supports the $(\min, \max)$-operation through a single call with 2 clock cycles per result. Therefore, the Minimum Spanning Tree problem (see, Figure 1) could be solved substantially more efficiently than previously realized. Furthermore, this could mean that the AMD hardware has build-in elements corresponding to $-\infty$ and ∞. This, in turn, could constitute an important step towards hardware support of generalized scalar FMA operations needed to realize many APP kernels(see, also [8]).

In the 1990's three designs for parallel computers have been tried: (1) array processors, (2) shared memory parallel computers, and (3) distributed memory parallel computers. After a number of "trails-and-errors," combined with progress in miniaturization, we witness the first two approaches joined within a processor and such processors combined into large machines. Specifically, while vendors like IBM, Intel and AMD develop multi-core processors with slowly increasing number of cores, the Nvidia and the AMD develop array processors on the chip. However, all these designs lead to processors consisting of thousands of FMA units.

IV. Proposed generalized multlpy-and-add

Based on these considerations, in [14] we have defined a generic generalized matrix-multiply-add operation (MMA),

$$\mathtt{C} \leftarrow \mathtt{MMA}[\otimes, \oplus](\mathtt{A}, \mathtt{B}, \mathtt{C}) : \mathtt{C} \leftarrow \mathtt{A} \otimes \mathtt{B} \oplus \mathtt{C},$$

where the $[\otimes, \oplus]$ operations originate from different matrix semirings. Note that, like in the scalar FMAs, generalized matrix *addition* and *multiplication*, can be implemented by making an $n \times n$ matrix A (or B) $= \bar{O}$ for addition, or a matrix $C = \bar{I}$ for multiplication (see, Section II).

Obviously, the generalized MMA resemples the `_GEMM` operation from the level 3 BLAS. Therefore, in [14] it was shown that, except for the triangular solve, BLAS 3 operations can be expressed in terms of the generalized MMA (in the linear algebra semiring). Note also that the proposed approach supports the idea that, in future computer hardware, data manipulation will be easiest to complete through matrix multiplication. Specifically, since the MMA represents a linear transformation of a vector space, it can be used for reordering of matrix rows/columns, matrix rotation, transposition, etc. Furthermore, operations like global reduction and broadcast can be easily obtained via matrix multiplication (see, [2]).

In summary, the proposal put forward in [14] covers three important aspects: a) it subsumes the level 3 BLAS, b) it generalizes the MMA, to encompass most of APP kernels,

and c) allows for new way of writing APP kernels, optimized for computers consisting of large number of processors with thousands of generalized FMA cores each, and simplified to support code maintainability.

V. State-of-the-art in object oriented BLAS

While our work extends and generalizes dense matrix multiplication, its object oriented (OO) realization should be conceptually related to OO numerical linear algebra, including the OO BLAS. Here, we briefly introduce selected OO realizations of numerical linear algebra in general, and BLAS in particular: MTL, uBLAS, TNT, Armadillo, and Eigen.

The uBLAS project ([15]) was focused on design of a C++ library that provided BLAS functionality. An additional goal of uBLAS was to evaluate if the abstraction penalty, resulting from object orientation, is acceptable. The uBLAS was guided by: (i) Blitz++ [16], POOMA [17], and MTL [18]. Data found on the Web indicates that the project was completed around 2002 and later subsumed into the BOOST [19] library.

The TNT project ([20]) is a collection of interfaces and C++ reference implementations that includes, among others, operations on multidimensional arrays and sparse matrices. The library, while not updated since 2004, can still be downloaded from the project Web site.

The MTL project remained active until around 2008-09, when the last paper/presentation concerning the MTL 4 was reported. Interestingly, this project provided not only an open source library, but also a paid one (more optimized).

There are two projects that are vigorously pursued today: Armadillo ([21]; last release on June 29, 2011) and Eigen ([22]; last release on May 30, 2011). Both support an extensive set of matrix operations. While Eigen seems to be focused on vector processor optimizations, Armadillo "links" with vendor optimized matrix libraries: MKL and ACML.

VI. Initial object oriented model

Following the ideas described above, in Figure 2 we depict our proposed OO model for the generalized matrix multiplication. Here, we see the interface to be made available to the user. It will allow to instantiate needed matrices and develop code with generalized matrix operations: matrix addition, matrix multiplication, and the MMA. We also define the abstract class $scalar\_Semiring$ needed to define operations of the *scalar* semiring (operations on elements of matrices).

The main class is the *Matrix* class. It describes how the matrix operations defined in the interface class are realized. Among others, it contains the MMA function, which is used to actually realize the matrix operations. The implementation of the MMA function is to be provided by the user, or by the hardware vendor (in a way similar to the vendor-optimized implementation of BLAS kernels).

Finally, we depict a sample specialization of the $scalar\_Semiring$ class (for the Shortest Path Problem; see, Figure 1), with elements from the R_+, $\bar{0} = 1$, $\bar{1} = \infty$, $\oplus = min$, while $\otimes = +$. Based on this specific scalar semiring (and, possibly, a generalized FMA unit), the appropriate MMA operation is implemented in the $Matrix$ class.

VII. Sample realization

Let us now recall that one of our goals is to simplify code development (in a way that the BLAS simplified it 30+ years ago). Therefore, code should be as simple as possible, with details of the implementation hidden. With this in mind, let us start from defining the interfaces. The $Matrix_interface$ defines operations available to the user to write her codes (for simplicity, our description is limited to square matrices).

```
interface Matrix_interface {
        Init(n);//initialisation of square matrix nxn
        Matrix matrix0(n)
          {/*generalized 0 matrix*/}
        Matrix matrix11(n)
          {/*generalized identity matrix*/}
        Matrix operator +
          {/*generalized matrix addition A+B*/}
        Matrix operator *
          {/*generalized matrix product A*B*/}
        Matrix Column_Permutation (A,i,j)
          {/*generalized permutation of column i
            and j in matrix A*/}
        Matrix Row_Permutation (A,i,j)
          /*generalized permutation of column i and
            j in matrix A*/
...
}
```

Here, user can create matrix objects, zero and identity matrices (for a given semiring). Furthermore, we define generalized matrix operators and two permutation matrices. This interface can be extended to include other user-defined operations. Next, we define the $scalar\_Semiring$ class.

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

It specifies generalized operations $addition$ and $multiplication$ as well as elements $\bar{0}$ and $\bar{1}$. It has to be provided by the user, to "select" the semiring.

With the two interfaces in place, we can define the core class $Matrix$, which is **not** made visible to the user (is internal to the realization of the generalized MMA). It inherits the $scalar\_Semiring$ interface, and implements the $Matrix\_interface$ 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) {/*generalized 0 matrix*/}
        Matrix matrix1_1(n) {/*generalized identity matrix*/}
        Matrix matrix_1P(i,j,n)
            {/*generalized identity matrix with interchanged
              columns i and j*/}
        Matrix A+B {return MMA(A,M1:matrix_1(n),B)}
        Matrix A*B {A,B}{return MMA(A,B,M0:matrix_0(n))}
        Matrix Column_Permutation (A,i,j){
                P=matrix_1P(i,j,n);
                O=matrix_0(n);
                  return MMA(P,A,O)}
        Matrix Row_Permutation (A,i,j){
                P=matrix_1P(i,j,n);
                O=matrix_0(n);
                  return MMA(A,P,O}
        ...
        private MMA(A,B,C:Matrix(n)){
```

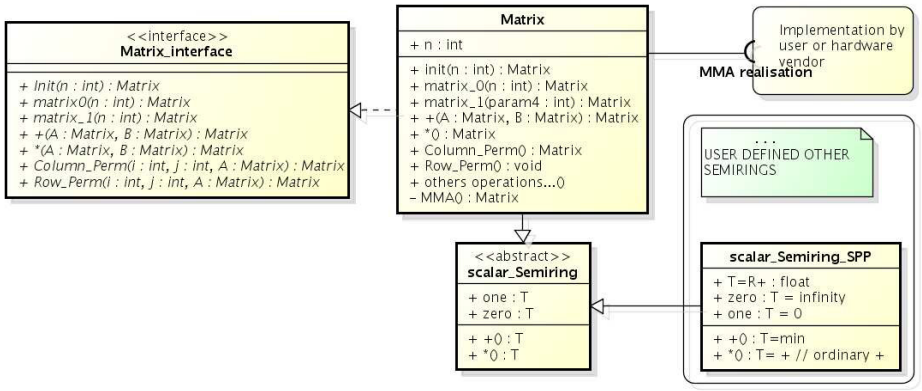

Fig. 2. General schema of the proposed object oriented model of generalized matrix multiplication

```
        return vendor/user specific realization of
          MMA = C+A*B where+ and * are from
          class scalar_Semiring}
}
```

The key part of this class is the private function MMA. This is the actual (user/vendor specific) realization of the MMA operation. As a result, user can perform 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 of their actual hardware/software realization. Observe also, that matrix column permutation, and row permutation have been defined in operations $Column\_Permutation$ and $Row\_Permutation$, which are implemented through a call to the MMA function with appropriate matrices (see, also [14]).

Finally, in the next snippet we show the class $scalar\_Semiring$ rewritten for the Shortest Path Problem (see, Figure1). After defining this class the user can simply apply the generalized MMA operation within the solver.

```
/*T = R+ PLUS infinity, + = min, * = +,
ZERO = infinity, ONE = 0; */

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

VIII. Concluding remarks

The aim of this paper was to propose the object model for the generalized matrix multiplication. The proposed approach is not language specific and presented at a very high level. Next, we will proceed with its more detailed realization in most important languages used in scientific computing.

References

[1] S. Robinson, "Towards an optimal algorithm for matrix multiplication." *SIAM News*, vol. 38, no. 9, 2005.

[2] S. G. Sedukhin, A. S. Zekri, and T. Myiazaki, "Orbital algorithms and unified array processor for computing 2D separable transforms," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 127–134, 2010.

[3] F. Song, S. Moore, and J. Dongarra, "Experiments with Strassen's Algorithm: from Sequential to Parallel," in *International Conference on Parallel and Distributed Computing and Systems (PDCS06)*. ACTA Press, November, 2006.

[4] M. Martone, S. Filippone, P. Gepner, M. Paprzycki, and S. Tucci, "Use of hybrid recursive csr/coo data structures in sparse matrices-vector multiplication," in *IMCSIT*, 2010, pp. 327–335.

[5] J. L. Gustafson, "Algorithm leadership," *HPCwire*, vol. Tabor Communications,, no. `http://www.hpcwire.com/features/17898659.html`, April 06, 2007.

[6] 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*, vol. E93.D, no. 3, pp. 534–541, 2010.

[7] D. J. Lehmann, "Algebraic structures for transitive closure," *Theor. Comput. Sci.*, vol. 4, no. 1, pp. 59–76, 1977.

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

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

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[11] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM J. Res. Dev.*, vol. 34, no. 1, pp. 59–70, 1990.

[12] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. 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*, pp. 59–67, June 2005.

[13] F. G. Gustavson, J. E. Moreira, and R. 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*. IBM Press, 1999, p. 4.

[14] S. Sedukhin and M. Paprzycki, "Generalizing matrix multiplication for efficient computations on modern computers," to appear.

[15] "uBLAS—Basic Linear Algebra Library," http://www.boost.org/doc/libs/1_46_1/libs/numeric/ublas/doc/index.htm.

[16] "BLITZ++," http://www.oonumerics.org/blitz/.

[17] "Parallel Object-Oriented Methods and Applications," http://acts.nersc.gov/pooma/.

[18] "Matrix Template Library," http://www.simunova.com/en/node/24.

[19] "Boost C++ library," http://www.boost.org/.

[20] "Template Numerical Toolkit," http://math.nist.gov/tnt/.

[21] "Armadillo: C++ linear algebra library," http://arma.sourceforge.net/.

[22] "Eigen," http://eigen.tuxfamily.org/index.php?title=Main_Page.