Regular Paper

Level-3 BLAS and LU Factorization on a Matrix Processor

Ahmed S. Zekri[†1] and Stanislav G. Sedukhin[†1]

As increasing clock frequency approaches its physical limits, a good approach to enhance performance is to increase parallelism by integrating more cores as coprocessors to general-purpose processors in order to handle the different workloads in scientific, engineering, and signal processing applications. In this paper, we propose a many-core matrix processor model consisting of a scalar unit augmented with b×b simple cores tightly connected in a 2D torus matrix unit to accelerate matrix-based kernels. Data load/store is overlapped with computing using a decoupled data access unit that moves b×b blocks of data between memory and the two scalar and matrix processing units. The operation of the matrix unit is mainly processing fine-grained b×b matrix multiply-add (MMA) operations. We formulate the data alignment operations including matrix transposition and skewing as MMA operations in order to overlap them with data load/store. Two fundamental linear algebra algorithms are designed and analytically evaluated on the proposed matrix processor: the Level-3 BLAS kernel, GEMM, and the LU factorization with partial pivoting, the main step in solving linear systems of equations. For the GEMM kernel, the maximum speed of computing measured in FLOPs/cycle is approached for different matrix sizes, n, and block sizes, b. The speed of the LU factorization for relatively large values of n ranges from around 50–90% of the maximum speed depending on the model parameters. Overall, the analytical results show the merits of using the matrix unit for accelerating the matrix-based applications.

1. Introduction

The idea of attaching accelerators or co-processors to general-purpose processors for enhancing the compute-intensive parts of applications has been used and is now getting more attention due to approaching the physical limits of VLSI technology. One recent example is the STI Cell/BE processor which has eight special-purpose Synergistic Processing Elements (SPEs) augmented to a PowerPC general-purpose microprocessor to enhance vector operations in graphics and scientific applications [1]. Other example accelerators include: vector co-processors [2),3)] that enhance the floating-point workloads found in scientific applications, and Graphics Processing Units (GPUs) that can efficiently deal with graphics and streaming applications.

The basic linear algebra subprograms (BLAS) were introduced to make the performance of dense linear algebra algorithms portable on high performance computers [4)–6)]. There are three levels of BLAS according to the complexity of computations. The most important of these levels is Level-3 BLAS, and the most important of Level-3 BLAS is the GEneral Matrix Multiply operation (GEMM) since it is possible to express all Level-3 BLAS kernels in terms of the highly optimized GEMM and a small percentage of Level-1 and Level-2 of the BLAS [7)–9)].

Solving linear systems of equations is one of the most important computations in scientific computing. The most compute-intensive part of the solution process is the factorization phase. The standard factorization algorithm using Gaussian elimination can be expressed using levels 1 and 2 of the BLAS. However, these levels have a low degree of data reuse and hence couldn't reach the potential speed on high performance processors [10)]. To use the GEMM-based Level-3 BLAS, the matrices and vectors are segmented into blocks that can reside in the upper levels of the storage hierarchy (registers, cache, local memory) and be maximally reused before moving to lower levels of storage (off-chip memories and disks) [11),12)]. In so doing the speed gap between processing and memory can be hidden and consequently the performance of the LU factorization increases.

Our motivation in this research is that matrix multiplication is a pervasive operation, and many scientific and signal processing applications can be formulated in terms of matrix-matrix multiplication [11),13)]. Moreover, the trend in chip-manufacturing is to put more processing units or cores together to handle the diverse demands of current and future ap-

†1 Department of Information Systems, The University of Aizu

plications. However, many problems remain unsolved such as the interconnection topology that guarantees scalability, the applications that could benefit from all of these parallel resources, the memory organization that keeps pace with the powerful processing speeds, etc.

In this paper, we propose a model of a many-core matrix processor consisting of a scalar unit augmented with b×b simple cores tightly connected into a 2D torus SIMD matrix unit to accelerate matrix-based applications. The SIMD matrix unit uses local communications between PEs and avoids global connections (the latency of the long wrap-around connections can be overcome by folding the torus [14]), and hence it is scalable. The matrix unit is optimized for executing fine-grained b×b MMA operations in the minimal computing time. A key point that distinguishes our proposed accelerator from others such as ClearSpeed CSX-600 processor, Cell processor and GPUs is that local communications between PEs are merged and overlapped with computing. Specifically, each PE performs a scalar 'multiply-add-roll' as one operation. In this operation the scalar multiply-add $c \leftarrow c+a \times b$, is executed while both scalars a and b, or a (or b) and c are rolled or circularly shifted to the appropriate neighbor PEs. The rolling of a or b is done simultaneously with computing c while in the case of rolling the result c, it must be done at the end of the multiply-add-roll operation due to data dependency. Using software pipelining and loop unrolling optimization techniques, the throughput of the multiply-add-roll operation can reach one multiply-add-roll operation every clock cycle. Consequently, the theoretical peak performance of the proposed b×b matrix unit can reach $2b^2$ FLOPs/cycle.

We present three optimal ways to perform the b×b MMA on the matrix unit [15),16)]. To implement these ways, we propose using three forms of the multiply-add-roll operation differing in the directions of data rolling (see Section 2). The three optimal ways that perform the b×b MMA have different data distributions and require alignment overhead before and may be after computing. One simple solution to the alignment problem is to store data in the aligned form. Indeed, this is not practical since alignment is also needed during all the processing time. Our approach is to hide this alignment overhead by formulating the alignment operations, including the matrix transpose, as MMA operations executed on the matrix unit. Using a decoupled load/store unit, we could overlap the alignment overhead with the load/store operations.

To evaluate the merits of the proposed matrix processor, we designed our algorithms to implement the Level-3 BLAS, represented by the GEMM kernel, and the LU factorization with partial pivoting. It was recently recognized that storing matrices by square blocks enhances the performance of memory hierarchy and consequently on dense linear algebra applications [17),18)]. Therefore, the idea behind our algorithm design is that the matrices are stored as b×b blocks rather than row- or column-major formats. This required the load/store unit to move contiguous b×b blocks to both the scalar and matrix units. Therefore, any block loaded into the matrix registers and/or data cache must be maximally reused in order to reduce the cost of the load/store operations. Our analytical evaluation of the GEMM kernel showed that the maximum speed of computing measured in FLOPs/cycle for different matrix sizes is approached.

The LU factorization with partial pivoting is an example of an application that includes scalar, vector and matrix operations. Based on storing the coefficient matrix as square b×b blocks, we designed a fine-grained blocked right-looking algorithm to run on both the matrix and scalar processing units. We did not consider execution overlapping between the two processing units. We scheduled the primitives of the algorithm so that only the matrix multiplication part is executed on the matrix unit and all other parts on the scalar unit. This required returning results to the main memory before switching computing between the two processing units. We included this data movement overhead in our timing model. Our analysis showed that the fraction of matrix multiplication in this algorithm dominates as the size of the matrix increases and consequently, the speed of computing is asymptotically determined by the speed of the matrix co-processor on executing the MMA kernel. The analytical results showed the speed of the LU factorization with partial pivoting for large matrix size 32 K ranges from around 50–90% of the maximum speed depending on the model parameters which we will discuss.

In Section 2, a description of the proposed matrix processor and its timing model is pre-

sented. In Section 3, three optimal data distributions to execute the b×b MMA operation on the matrix unit are presented. The formulation of the alignment operations as matrix multiply-add operations is introduced, and an algorithm of one variant of the GEMM operation is described and its predicted execution time is calculated. In Section 4, the standard LU factorization algorithm is discussed, and we designed a blocked right-looking algorithm and show how its primitives are scheduled between the scalar and matrix units. In addition, an algorithm for each primitive is described using software prefetching techniques, and the expected times are computed. Performance evaluation of the proposed algorithms is demonstrated in Section 5. Section 6 concludes the paper.

2. The Matrix Processor

2.1 Architecture Model

The proposed matrix processor is regarded as a 2D matrix register file augmented to a general-purpose scalar processor to accelerate matrix-based applications. The matrix register file is partitioned into b×b banks where each bank is directly connected to a simple core or PE. This partitioning of the matrix register file is called element-partitioned [19] since a matrix (or vector) is partitioned into b×b blocks (or b-element segments) so that each element resides in a different bank. The PEs are tightly coupled to form a 2D b×b torus matrix unit where the latency of the long wrap-around connections can be overcomed by folding the torus such that all connections between PEs become equal [14]. All PEs are working synchronously in a Single Instruction Multiple Data (SIMD) mode. A matrix instruction causes the PEs to process the elements in the same position at all register file banks simultaneously. Therefore, the matrix register file is viewed as a number of layers or matrix registers (see **Fig. 1**). Communications between PEs are local, and each PE can send and receive data to/from its four NEWS neighbors (North, East, West, and South) at the same time. For example, it can simultaneously receive data from South and East and send data and/or partial results to North and West, respectively.

The basic operation of the matrix unit is executing fine-grained b×b MMA operations in b multiply-add-roll time-steps where the scalar multiply-add $c \leftarrow c + a \times b$ is performed while the appropriate data is rolled to neighbor PEs. Our

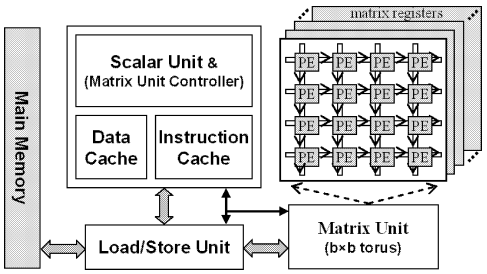

Fig. 1 The proposed matrix processor block diagram.

idea is to merge and overlap the rolling of data with computing. That is, the rolling of c is done at the end of the multiply-add-roll operation while the rolling of a and b is done simultaneously with computing c. We propose three forms of the multiply-add-roll operation corresponding to the three optimal ways (see Section 3) to execute the b×b MMA operation on the matrix processor. The difference between the three forms is the direction of data rolling. In one form, both b and a are rolled northward and westward, respectively. In the other two forms, b and c, and c and a are rolled northward and westward, respectively. (Other forms of data rolling may be useful in other data-parallel algorithms not considered in this paper. In addition, element-wise operations such as matrix addition/subtraction and register copying do not require data rolling. Combining all these multiply-add-roll forms in one generalized form is a matter of further investigations.)

The scalar unit of the matrix processor is a full-fledged core capable of executing the sequential part of an application beside controlling the matrix unit. The program instructions are loaded into the instruction cache for execution. The required data for the scalar execution is loaded into a software-controlled data cache near to the scalar registers. To reduce data miss penalty, we applied software pre-fetching techniques [20] where pre-fetch or pre-load instructions are inserted automatically by the compiler or manually by the programmer to bring data ahead of its use. The pre-load instruction causes a matrix block to be brought from the main memory to the data cache. This pre-load instruction looks like a load instruction except no register is specified [20].

To preserve the integrity of data between the scalar unit data cache and the main memory, the altered blocks in the data cache must be written back (or post-stored) into the main memory before switching the computing to the

matrix unit. The cache replacement policy is assumed to be software-controlled. While using software-controlled memories increases the complexity of programming, it has been demonstrated that this approach is effective in hiding memory latency using smaller cache sizes than conventional hardware-controlled caches [21].

The movement of data/results between the main memory and the data cache of the scalar unit and between the main memory and the matrix register file of the matrix unit is decoupled from processing by using a separate data access (or load/store) unit. Therefore, computing and data load/store can be partially or fully overlapped, which is required to hide the memory latency. The main components of the proposed matrix processor are shown in Fig. 1.

2.2 Timing Model

In order to evaluate the effectiveness of the proposed matrix processor especially the coprocessor unit we need to predict the execution times of the designed algorithms. We develop an execution time model for our proposed matrix processor. This model is simple, deterministic, and predictable due to using software-controlled data cache.

We decompose the total execution time of an algorithm into: 1) computing time, 2) load/store time, and 3) overhead time. The first term is the time spent in processing on both the matrix and scalar units. The second term is the time elapsed in the load/store operations. This term also includes the time of data pre-fetching and post-storing which are needed at the beginning and the end of the algorithm, and before switching computing between the matrix and scalar units. The third term of the execution time represents the overhead of data alignment needed for the processing on the matrix unit.

The MMA operations executed on the matrix unit will have the following times. Assuming each multiply-add-roll time-step takes τ clock cycles, the time of a b×b MMA is bτ cycles [16] (see also next section). Other matrix-formulated operations such as matrix skewing and transposition discussed in Subsection 3.2 will be quantified as the matrix multiply-add operation. The matrix unit operates on b×b blocks of matrices, and we assume the matrices are stored in memory in a b×b block data layout [17),18]. Moving a b×b block between memory and the matrix registers or the scalar data cache takes t_{bls} clock cycles. Since we assume a decoupled load/store unit, whenever a load/store operation is overlapped with computing or data alignment, the maximum execution time is taken.

In order to predict the execution time of the high-level language constructs used in our algorithms, we applied a prediction method similar to the one introduced in Ref. 22) as follows. The predicted execution time of the assignment statement *variable=expression* is the time of calculating the expression plus the time of loading its variables if they are not in registers. For example, the statement $y{=}z{+}1$ has the execution time $2t_{ls}{+}t_{add}$ clock cycles, where t_{ls} is the number of cycles to load/store a scalar value from cache to registers and t_{add} is the time of an addition in clock cycles. However, if z and y are in registers and will be used for next computations the time will be t_{add} cycles.

The execution time of the *if-then* statement has two possible values depending on its logical condition. Hence, a time interval $[T1,T2]$ will represent the lower and upper bounds of execution time. We will mean by adding an execution time $T0$ to an interval $[T1,T2]$ that $T0$ is added to both bounds. The same convention applies also to multiplying a time by an interval. For example, the execution time of the statement '*if* (c) *then* $s1$' is the interval $[t_{cd}{+}(t_{jp},t_{s1})]$, where t_{cd} is the time to check the condition c, t_{jp} is the number of cycles to jump around $s1$ to the end of the *if* statement, and t_{s1} is the number of cycles to calculate the statement $s1$.

The execution time of the *for-loop* statement equals the total number of the loop iterations multiplied by the sum of the execution times of the statements within the loop body and the loop overhead time, t_{loop}, in clock cycles.

3. Level-3 BLAS

Level-3 BLAS are targeted to the basic operations of $O(n^3)$ such as matrix-matrix multiplication, rank-k update and the solution of triangular systems of equations [23]. The most important of these operations is the general MMA (GEMM) $C{\leftarrow}\alpha C{+}\beta\mathrm{op}(A){\cdot}\mathrm{op}(B)$ where $op(X)$ is the matrix X or its transpose and α and β are scalars. In this section, we present our implementation of the GEMM operation on our proposed matrix processor.

3.1 Matrix Multiply-add Operation

Consider the MMA operation $C{\leftarrow}C{+}A{\cdot}B$ where $A{=}[a(i,k)]$, $B{=}[b(k,j)]$, and $C{=}[c(i,j)]$ are b×b dense matrices. The goal is to find optimal data distributions of the three matrices

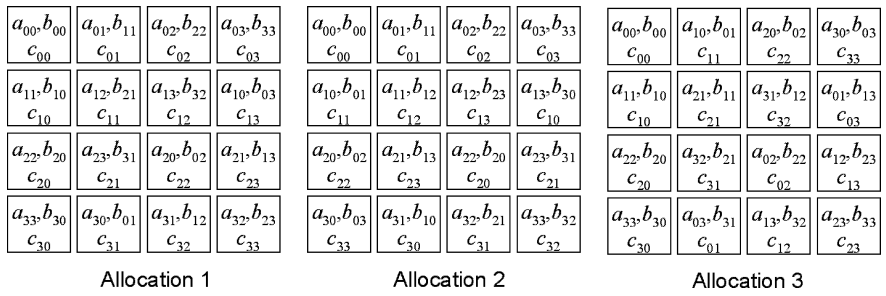

Fig. 2 Three data allocations to perform the b×b MMA operation $C \leftarrow C + A \cdot B$ on the b×b torus matrix unit, b=4.

to compute matrix C on the b×b torus matrix unit in the minimum number of steps.

We showed in prior works [15),16)] that the index space of the b×b MMA operation can be represented as a 3D b×b×b torus where modular scheduling is used to describe the distribution of the active index points at each scheduling step. All optimal modular scheduling functions are determined [15)]. At each scheduling step, b^2 points are active from the total b^3 points of the index space. To increase the efficiency of computing, we used the linear projection method to map the scheduled computations at each step to the 2D processor space where a b×b torus array processor (or matrix unit) is used to perform the matrix multiply-add operation in the minimal time, b time-steps of multiply-add-roll. For each optimal 3D scheduling function, three 2D data distributions are obtained by projection along i, j, and k axes (see Ref. 15), 16) for all possible projections). All the optimal 2D distributions are classified into three categories. In one category matrix C elements remain stationary during processing while the other two matrices are rolled. This category results from projection along the k-axis and the well-known Cannon's distribution [24)] belongs to this category. The other two categories keep either A or B stationary (i.e., projection along j or i-axis, respectively) while rotating the other two matrices.

In this paper, we will use one optimal 2D data distribution from each category. The three selected optimal data allocations are shown in **Fig. 2**. In Allocation 1, the rows of A and the columns of B are initially skewed using circular shifts. Then at each step, all PEs perform simultaneously the multiply-add $c(i,j)=c(i,j)+a(i,k)\cdot b(k,j)$ while the elements $a(i,k)$ and $b(k,j)$ are rolled westward and northward, respectively, to neighbor PEs. After b=4 multiply-add-roll time-steps, matrix C is computed and data returns to the initial state shown in the figure. In Allocation 2, after aligning B and C by skewing, matrix A remains while matrices B and C are rolled northward and westward, respectively, at each step of computing. In Allocation 3, matrices A and C are skewed as shown in the figure. During computing, matrix B remains while matrices A and C are rolled westward and northward, respectively.

3.2 Alignment

Given the three matrices A, B, and C in the canonical (conventional) layout, some alignment overhead is required before using the three optimal allocations in Fig. 2. In addition, matrices B and A need transposition in Allocations 2 and 3, respectively. Our aim is to hide this overhead. We do so by formulating both the skewing and transposition operations as b×b matrix multiply-add operations so that they can be overlapped with data loading/storing, as we will show in the algorithms.

The Skew Operation. To skew the rows/columns of an b×b matrix horizontally/vertically, the i^{th} row/column, i = 0,1,...,b-1, is rolled i times using circular shifts. We formulate the skewing operation as an MMA operation on the 2D torus matrix unit where C is initially set to zero and either A or B is set to some predefined 0-1 matrix register. For example, to skew the columns of the b×b matrix B to the North, we perform the MMA operation $C \leftarrow C + R_{\mathrm{N}} \cdot B$, where C is initially set to zero. During computing, the matrix register R_{N}, which has ones on the leftmost column and zeros elsewhere, is rolled westward while matrix C is rolled northward. After b=4 multiply-add-roll time-steps, C is replaced with the required result. To skew the rows of matrix A to the West, we perform $C \leftarrow C + A \cdot R_{\mathrm{W}}$, where C is ini-

tially set to zero and the matrix register R_{W}, which has ones on the topmost row and zeros elsewhere, is rolled northward while matrix C is rolled westward.

The two registers R_{N} and R_{W} can also be used to return, respectively, a northward skewed matrix X and a westward skewed matrix Y to their canonical forms.

The Transpose Operation. To transpose a b×b matrix X on the torus matrix unit, three MMA operations are performed in sequence. First, matrix X is skewed to the North by performing $C \leftarrow C+R_{\mathrm{N}} \cdot X$, C is initially zero, as explained above. Second, the MMA $X \leftarrow X+R_{\mathrm{I}} \cdot C$, X is initially zeroed, is performed. During computing, the matrix register R_{I}, which hold the identity matrix, is kept stationary while matrices X and C are rolled westward and northward, respectively. Now, matrix X holds the westward skewed form of matrix X^T. To get X^T in canonical form, matrix X is skewed eastward by performing an appropriate MMA operation.

For aligning matrix B in Allocation 2, we need to only apply the first two steps of the aforementioned transpose algorithm and this takes 2b multiply-add-roll time-steps. A similar procedure can be done to align matrix A in Allocation 3. For more details on the alignment operations on the torus matrix unit refer to Ref. 25).

3.3 GEMM Implementation on the Matrix Unit

Using the alignment operations discussed above, the four variants of the GEMM operation, $C \leftarrow C+A \cdot B$, $C \leftarrow C+A^T \cdot B$, $C \leftarrow C+A \cdot B^T$, and $C \leftarrow C+A^T \cdot B^T$, can be performed on the torus matrix unit. In Ref. 16) we described the four variants for different initial data layouts of A and B inside the torus matrix unit. However, the sizes of matrices were the same size as the matrix unit, b.

In this paper, we design a blocked fine-grained algorithm to implement the main variant $C \leftarrow C+A \cdot B$, where A is an $n_1 \times n_3$ matrix, B is an $n_3 \times n_2$ matrix, and n_1, n_2, $n_3 \gg$ b, on the matrix unit. The other three variants can be treated in a similar way.

Assume that the three matrices are partitioned and stored by square blocks as the matrix unit size, b, and that d blocks from A, B or C can be loaded into the matrix register file and reused many times. Then, we have two different ways to calculate the blocks of C on the matrix unit. One way is to update d blocks from C using d blocks from A (or B) and one block from B (or A). Updating the C blocks in this way is called blocked saxpy MMA [11]. The second way is to compute one block of C using a dot-product of one block row of A and one block column of B. This is called a blocked dot-product MMA [11]. Choosing which way to implement depends mainly on the sizes of involved matrices. We will describe our implementation of a fine-grained blocked saxpy algorithm when discussing the LU factorization. Here we show our implementation of a fine-grained blocked dot-product algorithm on the matrix unit assuming for simplicity that one block row of A or one block column of B can be loaded into the matrix register file and reused. This assumption will be changed when discussing the saxpy version in Subsection 4.3.

Assume the b×b blocks are stored in the column-major order. Initially, the matrix blocks are stored into the main memory. Before computing on the matrix unit, the needed blocks should be loaded into matrix registers by the load/store unit. Results after computing must be stored back to memory before it can be used on the scalar unit. We include these initial and final data movement operations in our timing model. Algorithm 1 below describes our implementation of a jik-version of the blocked dot-product of the GEMM kernel $C \leftarrow C+A \cdot B$ on the matrix unit. We applied Allocation 2, where $K_1 = \lceil n_1/\mathrm{b} \rceil$, $K_2 = \lceil n_2/\mathrm{b} \rceil$, and $K_3 = \lceil n_3/\mathrm{b} \rceil$.

Since the matrix unit and the load/store unit overlap execution, we applied a loop splitting technique called loop peeling [26] to correctly insert the load/store instructions within the algorithm. In this technique, a loop is divided into three sections: a prologue with the first iteration (and may be the second iteration too), a main body, and an epilogue with the last iteration. The statements, which are marked with the symbols '$\star$' or '$*$' at the beginning of the algorithm lines, will overlap execution as in lines 4 and 5, and lines 7 and 8 of Algorithm 1.

Algorithm 1.

01. **for** j=1 **to** K_2
02. Load $B_{1,j}$ $\{t_{bls}\}$
03. **for** k=2 **to** K_3
$\{(K_3\text{-}1) \cdot (t_{loop}+\max(t_{bls},\mathrm{b}\tau)+\mathrm{b}\tau)\}$
04.$\star$ Load $B_{k,j}$ $\{t_{bls}\}$
05.$\star$ Transpose & Skew $B_{k-1,j}$ northward $\{2\mathrm{b}\tau\}$
06. **end for**
07.$*$ Transpose & Skew $B_{K_3-1,j}$ northward $\{2\mathrm{b}\tau\}$

```
      % Prologue
08.∗  Load C1,j {tbls}
09.⋆  Load A1,1 {tbls}
10.⋆  Skew C1,j westward {bτ}
11.   for k=1 to K3 − 1
      {(K3-1)· (tloop+max(tbls,bτ)}
12.⋆    Compute C1,j+=A1,k·Bk,j {bτ}
13.⋆    Load A1,k+1 {tbls}
14.   end for
15.⋆  Compute C1,j+=A1,K3·BK3,j {bτ}
16.⋆  Load C2,j {tbls}
      % Main Body
17.   for i=2 to K1-1
      {(K1-2)(tloop+2max(tbls,bτ)+time of line 11)}
18.⋆    Store Ci−1,j {tbls}
19.⋆    Load Ai,1 {tbls}
20.⋆    Skew Ci,j westward {bτ}
21.     repeat lines 11-14 where X1,.=Xi,.
22.⋆    Compute Ci,j+=Ai,K3·BK3,j {bτ}
23.⋆    Load Ci+1,j {tbls}
24.   end for
      % Epilogue
25.⋆  Store CK1−1,j {tbls}
26.⋆  Load AK1,1 {tbls}
27.⋆  Skew CK1,j westward {bτ}
28.   repeat lines 11-14 where X1,.=XK1,.
      {time of line 11}
29.⋆  Compute CK1,j+=AK1,K1·BK1,j {bτ}
30.⋆  Store CK1,j {tbls}
31. end for
```

Note that, $w\pm=1$ means $w=w\pm1$. In lines 5 and 7, transposing the given block and skewing it northward requires performing two MMA operations as discussed before where, one of them will be overlapped with the loading at lines 4 and 8. As we see in Algorithm 1 expected execution times of each statement are calculated. Also, we can see that the alignment overhead is almost hidden with the load/store operations. In Section 5, we will evaluate the performance of the GEMM operation with different parameter values .

4. LU Factorization

In solving the linear system of equations $Ax=b$ where A is a real dense $n\times n$ matrix and both x and b are real vectors of length n, first the factorization of matrix A into a unit lower triangular matrix L and an upper triangular matrix U is done. Then, the two triangular systems $Ly=b$ and $Ux=y$ are solved to find x.

In this paper, we consider the Gaussian Elimination with Partial Pivoting (GEPP) to decompose matrix A into $PA=LU$ where P is an $n\times n$ rows permutation matrix. Using partial pivoting guarantees the stability of the solution of the system but complicates the implementation, see Ref. 27) for more details. Algorithm 2 is the standard GEPP algorithm assuming A is non-singular [11),27)]. We used MATLAB colon notation to describe ranges of array indices. In this notation an entire *for-loop* is written in compact form as in line 6, while line 7 expresses two nested *for-loops*. After the completion of Algorithm 2, the lower triangular part of A will be overwritten by matrix L with unit diagonals (which are not stored in the diagonals of A), and the upper triangular part will be overwritten by matrix U.

Algorithm 2.

```
01. for k=1 to n-1
02.   find and record γ, |a(γ,k)|=max_{k≤j≤n} |a(j,k)|
03.   if γ>k
04.     swap a(k,1:n), a(γ,1:n)
05.   end if
06.   a(k+1:n,k)=a(k+1:n,k)/a(k,k)
07.   a(k+1:n,k+1:n)-=
        a(k+1:n,k)·a(k,k+1:n)
08. end for
```

The amount of work (floating-point operations, FLOPs) in the standard GEPP algorithm is $2n^3/3+O(n^2)\approx 2n^3/3$ [11)] where one FLOP means a scalar floating-point addition, subtraction, multiplication and/or division operation. The $O(n^2)$ term includes the cost of pivoting which can be neglected compared with the $O(n^3)$ arithmetic cost of the algorithm. However, the pivoting cost may not be negligible when the data are not sequentially stored in memory.

4.1 Blocked LU Factorization

In Algorithm 2, line 6 represents a vector-scaling operation (i.e., Level-1 BLAS) while line 7 is a rank-one update of the trailing submatrix (Level-2 BLAS). In order to increase the amount of FLOPs per memory reference, i.e., the degree of data reuse, matrix-matrix multiplication (Level-3 BLAS) should be used.

There are three variants of the blocked LU factorization known as the i,j,k variants relative to the outermost loop index [10),28)]. The k variant or the so-called right-looking algorithm [10)] is generally described as follows. The $n\times n$ matrix A is segmented into four blocks where A_{11} is an nb×nb block (nb is the blocking size) and A_{22} is an $(n-\mathrm{nb})\times(n-\mathrm{nb})$ block. First, the unblocked GEPP algorithm is applied to the current block column of A. This means we compute $P_1\cdot A_{1:2,1}=L_{1:2,1}\cdot U_{11}$ where

L_{11} and U_{11} are nb×nb lower and upper triangle matrices, respectively, and P_1 is a permutation matrix representing the effects of pivoting. Second, the permutation P_1 is also applied to the right (and to the left when factorizing the next block columns) of the current block column of A to produce $\hat{A}_{1:2,2}$. Third, the triangle system with multiple right-hand-sides (RHS) $L_{11}{\cdot}U_{12}{=}\hat{A}_{12}$ is solved for U_{12}. In practice, the sub-matrices of L and U are stored on the corresponding blocks of A to save space and therefore, $\hat{A}_{12} \leftarrow U_{12}$ and $\hat{A}_{21}{\leftarrow}L_{21}$. Fourth, matrix $\hat{A}_{22}$ is updated using the matrix multiply-subtract operation $\hat{A}_{22}{\leftarrow}\hat{A}_{22}{-}\hat{A}_{21}{\times}\hat{A}_{12}$. Next, $\hat{A}_{22}$ is segmented into blocks and the above steps are repeated to factorize all matrix A.

We will implement the above k variant algorithm assuming that matrix A is stored as b×b blocks into memory. We selected this variant since it can be scheduled to run on both the matrix and scalar units provided that they don't overlap execution. Suppose that a partial factorization of A has been obtained so that the first ℓ-1 block columns of L and the first ℓ-1 block rows of U have been evaluated and stored into A (see **Fig. 3**). Algorithm 3 describes the blocked GEPP algorithm assuming for simplicity that $n{=}\mathrm{b}{\cdot}m$.

Algorithm 3.
1. **for** $\ell{=}1$ **to** m-1
2. Factor $P_\ell{\cdot}A_{\ell:m,\ell}{=}L_{\ell:m,\ell}{\cdot}U_{\ell,\ell}$
3. Pivot $A_{\ell:m,\ell+1:m}{=}P_\ell{\cdot}A_{\ell:m,\ell+1:m}$
4. Pivot $A_{\ell:m,1:\ell-1}{=}P_\ell{\cdot}A_{\ell:m,1:\ell-1}$, % $\ell > 1$
5. Solve $L_{\ell,\ell}{\cdot}A_{\ell,\ell+1:m}{=}A_{\ell,\ell+1:m}$
6. Update $A_{\ell+1:m,\ell+1:m}{-}{=}A_{\ell+1:m,\ell}{\cdot}A_{\ell,\ell+1:m}$
7. **end for**
8. Factor $P_m{\cdot}A_{m,m}{=}L_{m,m}{\cdot}U_{m,m}$
9. Pivot $A_{m,1:m-1}{=}P_m{\cdot}A_{m,1:m-1}$

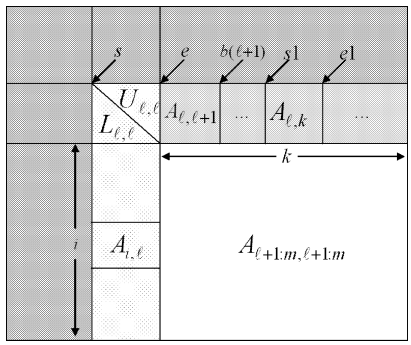

Fig. 3 Matrix A after ℓ-1 iterations of Algorithm 3; the dark grey area is already factorized. The *Factor* primitive is applied to the white dotted area, the *Solve* primitive is applied to the light grey area, the *Update* primitive is applied to the white area.

In line 2 the unblocked factorization is applied to a rectangular matrix of size (n-(ℓ-1)·b)×b. In this case, Algorithm 2 should be adjusted so that the last value of the columns index k is b-1, and the last value of the rows index will be n as it is. Then, after k=b-1, one more iteration at k=b is applied without the updating in line 7 of Algorithm 2.

In solving the m-ℓ lower triangle systems each with b RHS in line 5, we changed the column version of the forward substitution algorithm given in Ref. 11) so that it can handle b multiple RHS. Algorithm 4 describes the Solve primitive applied to block $A_{\ell,k}$ at Fig. 3. Note that $L_{\ell,\ell}$ is the lower triangular part of $A_{\ell,\ell}$ but with diagonal elements equals to one.

Algorithm 4.
1. **for**i=1 **to** b-1
2. $a(i,:){=}a(i,:)/l(i,i)$
3. $a(i+1:\mathrm{b},:){-}{=}l(i+1:\mathrm{b},i){\cdot}a(i,:)$
4. **end for**
5. $a(\mathrm{b},:){=}a(\mathrm{b},:)/l(\mathrm{b},\mathrm{b})$

In general Algorithm 4 can be applied to any lower triangle system with multiple RHS. However, when the triangular matrix has unit values (as in Algorithm 3), we can omit lines 2 and 5 in Algorithm 3 since division by the constant value 'one' gives the same result. We will describe our implementation of all primitives of Algorithm 3 in Subsection 4.3.

4.2 Level-3 BLAS Fraction

In this subsection, we will calculate the cost of Algorithm 3 in terms of the arithmetic operations. Then we will analyze the impact of the percentage of the matrix-matrix multiplication (i.e., the Update primitive) on the performance of the blocked Gaussian elimination algorithm. For simplicity, we will not consider the overhead of pivoting, however, in Subsection 4.3 we will predict its execution time.

Table 1 demonstrates the amount of arithmetic operations in the primitives of Algorithm 3. In line 2 of Algorithm 3, the Factor primitive is applied to a rectangular submatrix whereas in line 8 the last square diag-

Table 1 The arithmetic cost of Algorithm 3.

primitive	#FMA	#mult.	#div.
Factor	$\frac{\mathrm{b}-1}{4}n^2-\frac{3\mathrm{b}^2+3\mathrm{b}-2}{12}n$	$\frac{1}{2}n^2+\frac{4\mathrm{b}^2-3}{6}n$	n
Solve	$\frac{\mathrm{b}-1}{4}n^2-\frac{\mathrm{b}^2-\mathrm{b}}{4}n$	-	-
Update	$\frac{1}{3}n^3-\frac{\mathrm{b}}{2}n^2+\frac{\mathrm{b}^2}{6}n$	-	-

onal block of matrix A is factorized applying Algorithm 2 directly. As we see, the overall FLOPs count (2×#FMA+#mult.+#div.) of the Factor primitive on Algorithm 2 is $O(n^2)$ because the operations can be implemented using Level-2 and Level-1 BLAS. The primitive Solve in line 4 solves a b×b triangle system with $(m-\ell)$·b multiple RHS. The overall complexity of this primitive is $O(n^2)$ arithmetic operations because it can be implemented using Level-2 and Level-1 BLAS (see Algorithm 4). The third primitive Update is the GEMM operation $C \leftarrow C\text{-}A \cdot B$. The amount of FLOPs contributed by this primitive is $O(n^3)$. From Table 1 we can see that the Update primitive is the most intensive computational part. Therefore, it is expected that any enhancement to this primitive will enhance the total performance of the LU decomposition algorithm.

Table 1 also shows the amount of the fused multiply-add/subtract (FMA) included in each primitive. We can see that the Solve and Update primitives are entirely composed of FMAs where each FMA equals two FLOPs. More than half the FLOPs of the Factor primitive are FMA and this percentage increases as b increases. This large percentage of FMA on Algorithm 3 suggests that using an FMA instruction will have a significant advantage over using a floating-point multiply followed by a floating-point add. This is because using separate multiply and add instructions costs 1.7 times more than an FMA [29].

Now, we perform a simplified timing analysis of Algorithm 3 disregarding the pivoting process since it has less work than the updating of the trailing sub-matrix. We also assume that an FMA operation (c=c+a×b) takes μ cycles which will also be the same time for multiplication (c=1+a×b), addition/subtraction (c=a±b×1) operations. Since multiplication is faster than division, the number of divisions in line 6 of Algorithm 2 can be reduced by computing the reciprocal of the pivot element separately and then multiplying the result by the elements of matrix A at line 6.

The time of executing Algorithm 3 on the matrix processor is the total time of the three primitives. The time to execute a b×b matrix multiply-add operation on the b×b torus matrix unit is b·τ clock cycles (without the overhead of data alignment). Therefore, the expected execution time of the Update primitive on the matrix unit is approximately:

$$\left(\tfrac{1}{3}n^3 - \tfrac{\mathrm{b}}{2}n^2 + \tfrac{\mathrm{b}^2}{6}n\right) \cdot \tfrac{1}{\mathrm{b}^2}\tau. \quad (1)$$

The approximate execution time of Algorithm 3 (ignoring pivoting) on the matrix processor is:

$$\left(\tfrac{\mathrm{b}}{2}n^2 + \tfrac{\mathrm{b}^2-2}{6}n\right) \cdot \mu + n \cdot \lambda + \left(\tfrac{1}{3}n^3 - \tfrac{\mathrm{b}}{2}n^2 + \tfrac{\mathrm{b}^2}{6}n\right) \cdot \tfrac{1}{\mathrm{b}^2}\tau. \quad (2)$$

Now, the speed of computing Algorithm 3 on the matrix processor measured in the amount of FLOPs/cycle can be calculated as the ratio of the amount of FLOPs in the sequential algorithm, $2n^3/3$, and the parallel time of Algorithm 3 given in Eq. (2). The speed depends mainly on τ and if operations of the matrix unit can be scheduled so that we get one multiply-add-roll each clock cycle, then the maximum speed of computing asymptotically becomes the speed of matrix-matrix multiplication (or the Update primitive), i.e., $2\mathrm{b}^2$ FLOPs/cycle.

In the next section we will implement the four primitives of Algorithm 3 on the scalar and matrix units in order to predict the overall execution time of Algorithm 3 and hence evaluate the merits of adding the matrix unit as an accelerator for matrix kernels.

4.3 Primitives Implementation

In this subsection we design our algorithms to implement the four primitives, Factor, Pivot, Solve and Update, of the blocked LU factorization algorithm (Algorithm 3) on the proposed matrix processor. The idea behind our design is that the matrices are stored as b×b blocks rather than row- or column-major formats, and any block loaded into registers and data cache must be maximally reused before returning to memory. We scheduled each iteration of Algorithm 3's main loop so that the three primitives Factor, Pivot, and Solve are executed on the scalar core and then switching to the matrix unit to execute the Update primitive. Note that, before switching computing, partial results should return to memory.

To compute on the scalar core, the required blocks are moved by the load/store unit from memory to the data cache one b×b block at a time. Then, during processing, the scalar data is loaded into scalar registers and stored back to the data cache. We assume the data cache is controlled by software so that a preload operation causes the load/store unit to pre-fetch one block from main memory to the data cache. Pre-fetching data ahead of its use

reduces the stall time which happens when data is not in cache. The pre-fetch operations are hand-inserted into the algorithms in order to pre-fetch the data before it is actually needed. In addition, we applied a post-store operation to control the replacement of blocks in the data cache. Note that, if some of the statements which are overlapped with the pre-load and post-store instructions should be serialized, we put the symbol '$\circ$' at the beginning of their corresponding lines as in lines 2 and 3 of Algorithm 5. (The global constant c is the reciprocal of b.)

The Factor Primitive. Algorithm 5 below is the implementation of Algorithm 2 (GEPP) to factorize the ℓ^{th} block column of matrix A (dotted area in Fig. 3) on the scalar core. In Algorithm 5, the processed block column is divided into two parts. The first one is the block $A_{\ell,\ell}$ and the second is the rest of the blocks. We apply the pre-fetching technique to the second part to correctly insert the pre-load and post-store instructions. Each iteration of the main loop at line 4 takes two passes on the blocks. The first pass finds the pivot row in the block column while the second computes the multipliers (elements of matrix L) and stores them in the current column. Then the trailing sub-matrix of the ℓ^{th} block column of A is updated. Note that altered blocks are written back to memory at the end of Algorithm 5.

Algorithm 5.

01.$\star$ Pre-load $A_{\ell,\ell}$ $\{t_{bls}\}$
02.$\star\circ$ s=b$\cdot(\ell-1)$+1 $\{2\,\mu\}$
03.$\star\circ e$=b$\cdot\ell$ $\{\mu\}$
04. **for** j=s **to** e-1
% Find the pivot element in column j
05. $amax = abs(a(j,j))$ $\{t_{ls}$+$t_{abs}\}$
06. $pivot(j)$=0 $\{t_{mv}\}$
% Prologue
07.$\star$ Pre-load $A_{\ell+1,\ell}$ $\{t_{bls}\}$
08.$\star$ **for** kk=j+1 **to** e
$\{(e$-$j)\cdot(t_{ls}$+t_{abs}+t_{cd}+t_{loop}+$(t_{jp}, 2t_{mv}))\}$
09. t=abs$(a(kk,j))$ $\{t_{ls}$+$t_{abs}\}$
10. **if** $t > amax$ $\{t_{cd}$+$(t_{jp}, 2t_{mv})\}$
11. $amax$=t $\{t_{mv}\}$
12. $pivot(j)$=j $\{t_{mv}\}$
13. **end if**
14. **end for**
15.$*$ Pre-load $A_{\ell+2,\ell}$ $\{t_{bls}\}$
16.$*$ **for** kk=b$\cdot\ell+1$ **to** b$\cdot(\ell$+1)
{b$\cdot(t_{ls}$+t_{abs}+t_{cd}+t_{loop}+$(t_{jp}, 2t_{mv}))\}$
17. repeat lines 9-13
18. **end for**
% Main Body
19. **for** i=ℓ+2 **to** m-1
$\{(m$-ℓ-2)$\cdot(t_{loop}$+max$(t_{bls}$, time of line 16))$\}$
20.$\star$ Post-store $A_{i-1,\ell}$ $\{t_{bls}\}$
21.$\star$ Pre-load $A_{i+1,\ell}$ $\{t_{bls}\}$
22.$\star$ **for** kk=b$\cdot(i$-1)+1 **to** b$\cdot i$ {time of line 16}
23. repeat lines 9-13
24. **end for**
25. **end for**
% Epilogue
26.$*$ Post-store $A_{m-1,\ell}$ $\{t_{bls}\}$
27.$*$ Pre-load $A_{pivot(j)\cdot c+1,\ell}$ $\{t_{bls}\}$
28.$*$ **for** kk=b$\cdot(m$-1)+1 **to** b$\cdot m$ {time of line 16}
29. repeat lines 9-13
30. **end for**
% Swap the j^{th} row and the pivot row
31.$\star$ Post-store $A_{m,\ell}$ $\{t_{bls}\}$
32.$\star$ **if** $pivot(j)$ >0
$\{t_{cd}$+$(t_{jp}$, b$\cdot(t_{loop}$+$4t_{ls}$+$3t_{mv}))\}$
33. **for** k=s **to** e $\{$b$\cdot(t_{loop}$+$4t_{ls}$+$3t_{mv})\}$
34. t=$a(j,k)$ $\{t_{ls} + t_{mv}\}$
35. $a(j,k)$=$a(pivot(j),k)$ $\{2t_{ls}$+$t_{mv}\}$
36. $a(pivot(j),k)$=t $\{t_{mv}$+$t_{ls}\}$
37. **end for**
38. **end if**
% Compute the multipliers and update
% the trailing sub-matrix
39.$*$ Post-store $A_{pivot(j)\cdot c+1,\ell}$ $\{t_{bls}\}$
40.$*$ Pre-load $A_{\ell+1,\ell}$ $\{t_{bls}\}$
41.$*\circ$ piv=$1/a(j,j)$ $\{t_{ls}$+$\lambda\}$
42.$*\circ$ **for** ii=j+1 **to** e $\{(e$-$j)^2(t_{loop}$+$3t_{ls}$+$\mu)$+
$(e$-$j)(t_{loop}$+$2t_{ls}$+μ+$t_{mv})\}$
43. $a(ii,j)$=$piv\cdot a(ii,j)$ $\{t_{ls}$+$\mu\}$
44. t=$a(ii,j)$ $\{t_{mv}$+$t_{ls}\}$
45. **for** kk=j+1 **to** e $\{(e$-$j)(t_{loop}$+$3t_{ls}$+$\mu)\}$
46. $a(ii,kk)$-=$t\cdot a(j,kk)$ $\{3t_{ls}$+$\mu\}$
47. **end for**
48. **end for**
49.$\star$ Pre-load $A_{\ell+2,\ell}$ $\{t_{bls}\}$
50.$\star$ repeat lines 42-48 where ii=b$\cdot\ell$+1 to b$\cdot(\ell$+1)
$\{$b$(e$-$j)(t_{loop}$+$3t_{ls}$+$\mu)$+b$(t_{loop}$+$2t_{ls}$+μ+$t_{mv})\}$
51. **for** $i = \ell$+2 **to** m-1
$\{(m$-ℓ-2)$\cdot(t_{loop}$+max$(t_{bls}$, time of line 50))$\}$
52.$\star$ Post-store $A_{i-1,\ell}$ $\{t_{bls}\}$
53.$\star$ Pre-load $A_{i+1,\ell}$ $\{t_{bls}\}$
54.$\star$ repeat lines 42-48 where ii=b$\cdot(i$-1)+1 to b$\cdot i$
{time of line 50}
55. **end for**
56.$\star$ Post-store $A_{m-1,\ell}$ $\{t_{bls}\}$
57.$\star$ repeat lines 42-48 where ii=b$\cdot(m$-1)+1 to b$\cdot m$
{time of line 50}
58. Post-store $A_{m,\ell}$ $\{t_{bls}\}$
59. **end for**
60. repeat lines 4-58 where j=e,
except the for-loops starting with j+1
61. Post-store $A_{\ell,\ell}$ $\{t_{bls}\}$

The Pivot Primitive. After factoring the current block column of A, the pivoting effect should be applied to all the left and right col-

umn blocks, too. Algorithm 6 below describes the pivoting of a number of adjacent block columns of matrix A on the scalar unit. Note that, the execution time of line 10 is the same as the lines 33–37 of Algorithm 5.

Algorithm 6.

01. **for** k=k1 **to** k2
02.⋆ Pre-load $A_{\ell,k}$ $\{t_{bls}\}$
03.⋆○ $s1$=b·(k-1)+1 $\{2\mu\}$
04.⋆○ $e1$=b·k $\{\mu\}$
05.⋆○ piv=$pivot$($s1$)·c+1 $\{\mu\}$
06.∗ Pre-load $A_{piv,k}$ $\{t_{bls}\}$
07.∗ $piv1$=$pivot$($s1$+1)·c+1 $\{2\,\mu\}$
08.∗ Pre-load $A_{piv1,k}$ $\{t_{bls}\}$
09.⋆ **if** $pivot(s1) > 0$
$\{t_{cd}+(t_{jp}$, b · $(t_{loop}+4t_{ls}+3t_{mv}))\}$
10. swap rows $s1$ and $pivot(s1)$
$\{$b·$(t_{loop}+4t_{ls}+3t_{mv})\}$
11. **end if**
12. **for** i=$s1$+1 **to** $e1$-1
$\{$(b-1)·$(t_{loop}+4\,\mu+(t_{cd}+(t_{jp},t_{bls}))+$
time of line 9)$\}$
13. $piv1$=$pivot(i-1)$·c+1 $\{2\,\mu\}$
14. $piv2$=$pivot(i+1)$·c+1 $\{2\,\mu\}$
15. **if** $piv1$ <> $piv2$
16.⋆ Post-store $A_{piv1,k}$ $\{t_{bls}\}$
17.⋆ Pre-load $A_{piv2,k}$ $\{t_{bls}\}$
18. **end if**
19. repeat lines 9-11 where $s1 = i$
$\{$time of line 9$\}$
20. **end for**
19. piv=$pivot(e1-1)$·c+1 $\{2\,\mu\}$
20.⋆ Post-store $A_{piv,k}$ $\{t_{bls}\}$
21.⋆ repeat lines 9-11 where $s1$=$e1$
$\{$time of line 9$\}$
22. Post-store $A_{piv2,k}$ $\{t_{bls}\}$
23. Post-store $A_{\ell,k}$ $\{t_{bls}\}$
24. **end for**

The Solve Primitive. In Algorithm 4, we have described the solution of a b×b triangular system with b multiple RHS. However, in line 5 of Algorithm 3, we need to solve the same system but with b×$(m-\ell)$·b multiple RHS. In Algorithm 7 below, we describe how the latter triangular system is implemented on the scalar unit of the matrix processor. See Fig. 3 for the workspace of the algorithm.

Algorithm 7.

01.⋆ Pre-load $A_{\ell,\ell}$ $\{t_{bls}\}$
02.⋆○ $s1$=s+b $\{\mu\}$
03.⋆○ $e1$=e+b $\{\mu\}$
04. Pre-load $A_{\ell,\ell+1}$ $\{t_{bls}\}$
05. **for** i=s **to** e-1 $\{\sum_{i=s}^{e-1}(t_{loop}$+time of line 6)$\}$
06. **for** ii=i+1 **to** e
$\{(e\text{-}i)\cdot(t_{loop}+t_{mv}+t_{ls}+$b·$(t_{loop}+3t_{ls}+\mu))\}$
07. t=$a(ii,i)$ $\{t_{mv}+t_{ls}\}$
08. **for** j=$s1$ **to** $e1$ $\{$b·$(t_{loop}+3t_{ls}+\mu)\}$
09. $a(ii,j)$-=t·$a(i,j)$ $\{3t_{ls}+\mu\}$
10. **end for**
11. **end for**
12. **end for**
13. Pre-load $A_{\ell,\ell+2}$ $\{t_{bls}\}$
14. **for** k=ℓ+2 **to** m-1 $\{(m\text{-}\ell\text{-}2)\cdot(t_{loop}+2\cdot\mu+$
$\max(t_{bls}$, time of line 5))$\}$
15. $s1$=$s1$+b $\{\mu\}$
16. $e1$=$e1$+b $\{\mu\}$
17.⋆ Post-store $A_{\ell,k-1}$ $\{t_{bls}\}$
18.⋆ Pre-load $A_{\ell,k+1}$ $\{t_{bls}\}$
19.⋆ repeat lines 5-12 $\{$time of line 5$\}$
20. **end for**
21. $s1$=$s1$+b $\{\mu\}$
22. $e1$=$e1$+b $\{\mu\}$
23.⋆ Post-store $A_{\ell,m-1}$ $\{t_{bls}\}$
24.⋆ repeat lines 5-12 $\{$time of line 5$\}$
25. Post-store $A_{\ell,m}$ $\{t_{bls}\}$

The Update Primitive. This primitive will be executed on the matrix unit. Algorithm 8 below is our implementation of a fine-grained blocked saxpy version of Algorithm 1. We used Allocation 1 to find the blocks of C. To reduce the loading time of blocks from main memory to the matrix registers, we assume at least d blocks from A are loaded and reused many times. In this case, the updating of the trailing sub-matrix at each iteration of ℓ may be divided into two parts if the number of its block rows is not an exact multiple of d. We apply Algorithm 8 to update the main part. To update the remaining part (where the number of block rows are not multiple of d), we also apply Algorithm 8 but the loop index i will start from ℓ+(⌊(m-ℓ)/d⌋)·d+1 to m and step equals to 'one' while i+d-1 is replaced with m. It should be noted that when ℓ approaches m, m-ℓ becomes less than d. In this case, d must be reset to the value m-ℓ to correctly apply Algorithm 8.

Algorithm 8.

01. **for** i=ℓ+1 **to** ℓ+⌊(m-ℓ)/d⌋·d **step** d
02. Load $A_{i,\ell}$ $\{t_{bls}\}$
03. **for** j=i+1 **to** i+d-1
04.⋆ Skew $A_{j-1,\ell}$ westward $\{$b$\tau\}$
05.⋆ Load $A_{j,\ell}$ $\{t_{bls}\}$
06. **end for**
07.⋆ Skew $A_{i+d-1,\ell}$ westward $\{$b$\tau\}$
08.⋆ Load $A_{\ell,\ell+1}$ $\{t_{bls}\}$
09. **for** k=ℓ+1 **to** m-1
% Prologue
10.⋆ Skew $A_{\ell,k}$ northward $\{$b$\tau\}$
11.⋆ Load $A_{i,k}$ $\{t_{bls}\}$
12.∗ $A_{i,k}$-=$A_{i,\ell}$·$A_{\ell,k}$ $\{$b$\tau\}$

13.∗ Load $A_{i+1,k}$ $\{t_{bls}\}$
% Main Body
14. **for** j=i+1 **to** i+d-2
15.⋆ $A_{j,k}$-=$A_{j,\ell}\cdot A_{\ell,k}$ $\{\mathrm{b}\tau\}$
16.⋆ Load $A_{j+1,k}$ $\{t_{bls}\}$
17.⋆ Store $A_{j-1,k}$ $\{t_{bls}\}$
18. **end for**
% Epilogue
19.⋆ $A_{i+d-1,k}$-=$A_{i+d-1,\ell}\cdot A_{\ell,k}$ $\{\mathrm{b}\tau\}$
20.⋆ Store $A_{i+d-2,k}$ $\{t_{bls}\}$
21.∗ Store $A_{i+d-1,k}$ $\{t_{bls}\}$
22.∗ Load $A_{\ell,\ell+1}$ $\{t_{bls}\}$
23. **end for**
24. repeat lines 10-21 where k=m
25. **end for**

Modifications. We should mention that because of using software pre-fetching techniques in designing the previous algorithms, some minor modifications should be done to the algorithms for proper working at iterations m-1 and m of Algorithm 3. This is due to the limited number of blocks which doesn't allow applying the pre-fetching. We mention here the changes to the algorithms.

To apply Algorithm 5 at iteration ℓ=m-1, line 7 can be moved outside the main loop. Lines 15, 19–31, 39–40, 49, and 51–58 shouldn't be applied. At the end of the algorithm the block $A_{\ell+1,\ell}$ should be post-stored to memory. At iteration ℓ=m of Algorithm 5, i.e., the factoring of block $A_{m,m}$, the algorithm is applied without the lines 7, 15–31, 39–40, and 49–60.

In algorithm 6, at each iteration k, the two blocks $A_{\ell,\ell}$ and $A_{\ell,\ell+1}$ are pre-loaded. Then, the middle loop at line 12 runs from s1 to e1 for rows interchange. After that, the two blocks are post-stored. A similar change is done when ℓ=m, but the middle loop runs from s1 to e1-1.

At iteration ℓ=m-1 of Algorithm 3, only one triangular system with b RHS is solved. In this case, Algorithm 7 is applied without the lines 13–24. Also, in Algorithm 8, only the block update $A_{m,m}$-=$A_{m,m-1}\cdot A_{m-1,m}$ is needed which requires three block loads and one store. The skewing of $A_{m,m-1}$ and $A_{m-1,m}$ is overlapped with two of the three block loads.

5. Performance Evaluation

In Section 2, we have developed a predictable execution time model for the proposed matrix processor. In this model, we have decomposed the total execution time of an algorithm into the sum of: computing time, load/store time, and overhead time. In Sections 3 and 4, we have predicted the execution times of the statements of the designed algorithms so that the total execution times can be easily calculated. In this section, we analyze the performance of the GEMM operation and the blocked LU factorization with partial pivoting on the proposed matrix processor. Also, we analyse the effect of changing the memory bandwidth on the GEMM operation, which is the basic operation in the LU factorization algorithm.

We measure the performance of Level-3 BLAS and the blocked LU factorization on the proposed matrix processor by the speed of computing in FLOPs/cycle. For the parameters related to architectural implementation, we assume the time of the operations t_{ls}, t_{mv}, t_{jp}, t_{cd}, and t_{abs} is one clock cycle. Since most computations of the algorithms are FMA, we assume the floating-point functional units are fully-pipelined and the compiler is able to schedule the independent operations using loop unrolling and software pipelining techniques. Hence, the throughput of the floating-point FMA, μ, and the multiply-add-roll, τ, time-step is assumed one FMA (or two FLOPs) each cycle and one multiply-add-roll operation each cycle, respectively. However, this is not the case with the floating-point division since its amount in the algorithms is small and we assume its execution time, λ, is twenty clock cycles.

Level-3 BLAS (GEMM). We discuss the case that all matrices A, B, and C are $n \times n$. This means K_1=K_2=K_3=$\lceil n/\mathrm{b} \rceil$ in Algorithm 1. For other cases, we can select between applying Allocations 2 and 3 in Algorithm 1 or applying Allocation 1 in Algorithm 8.

An important parameter for the efficiency of the matrix processor is the time of loading/storing a b×b matrix block, t_{bls}. Assuming the load/store unit can access ω contiguous elements in one clock cycle, then t_{bls} is equal to b^2/ω clock cycles. **Figure 4** (a) shows the performance of the GEMM operation for different matrix sizes n and different block sizes b, where ω=b and the number of blocks that can be loaded and reused, d, is equal to m. It is clear from the figure that the speed of computing increases as b increases. Meantime, as n increases the maximum speed $2\mathrm{b}^2$ FLOPs/cycle is approached. This means that the alignment overhead associated with the b×b MMA operations is almost hidden. The main reason for that is the overlap between the skewing oper-

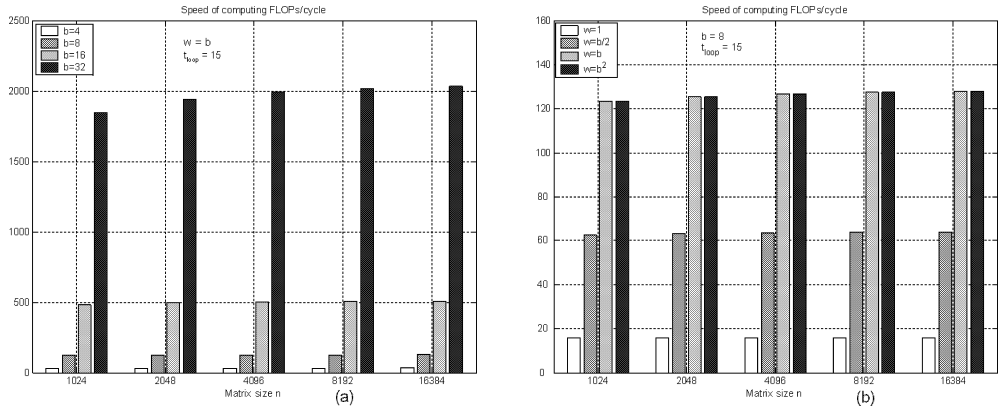

Fig. 4 Performance of the GEMM operation $C \leftarrow C + A \cdot B$ on the b×b matrix unit.

ations and the loading of data. However, decreasing d will degrade the performance relatively as we expect.

Figure 4 (b) shows the effect of changing ω on the performance of the $n \times n$ GEMM operation on the torus matrix unit, b=8. We see from the figure that the speed approaches the maximum value when $\omega \geq$b. Otherwise, the speed decreases till 2b FLOPs/cycle when the blocks are sequentially loaded/stored, i.e., ω=1.

LU Factorization. In the primitives such as Factor and Pivot, there are two bounds of the execution time depending on the values of matrix A. The lower execution time will be the case when no swapping of the matrix rows occurs. This means all lower bound execution time of all *if-then* statements are taken. This lower bound of execution time is the upper bound of the speed of computing. Also, the upper bound of the execution time (when all comparisons to find the pivot hold) is the lower bound of the speed of computing. **Figures 5** (b,d) and **6** (b,d) show the lower bounds of speed while (a,c) in both figures show the upper bounds. We see in Figs. 5 and 6 that for relatively large size matrix, n=32 K, the difference between the speed bounds is small. However, a noticeable difference exists for small size matrices, for example when n=1 K which is due to the pivoting overhead.

Both Figs. 5 and 6 show the increase of the speed of computing by increasing the matrix size, n, for different values of b. This is due to the increase in matrix multiplication portion as n increases. For the selected values of n ($\leq$32 K), the most appropriate values of b seem to be b=4 and b=8. b=4 is better in terms of the percentage of the maximum speed attained, i.e., the efficiency of the matrix processor. However, b=8 can be chosen if we favor speed with moderate efficiency. At small values of n, we see that the speed and efficiency are not high because the time of the Factor and Solve primitives became apparent. In addition, our model assumes a scalar unit with two FLOPs/cycle and no overlap between computing and registers load/store. Since the two primitives are mainly Level-1 and Level-2 BLAS, we expect that using a scalar unit with a SIMD extension will enhance the performance of the blocked LU factorization at small matrix sizes.

In Figs. 5 and 6, the effect of the loop overhead, t_{loop}, is shown. In Fig. 5, d is fixed at the best case d=m while in Fig. 6 it is fixed on the least number we assumed, d=8. Both figures show that a large percentage of speed is gained by decreasing t_{loop} from 15 cycles going down to zero cycles, the case of complete loop unrolling. In addition, the two figures show that increasing the value of d enhances the speed due to increasing the amount of data reuse.

6. Conclusions

Augmenting scalar cores with specialized coprocessors for enhancing compute-intensive kernels have gained the attention of chip manufacturers due to approaching the physical limits of increasing clock frequency. We have proposed a matrix processor model consisting of a scalar unit and a SIMD matrix unit of b×b simple cores tightly connected in a 2D torus topology to process b×b MMA operations. The matrix unit is scalable since there are no global connections between PEs. An execution model for the proposed matrix processor is developed. We presented three optimal data distributions to

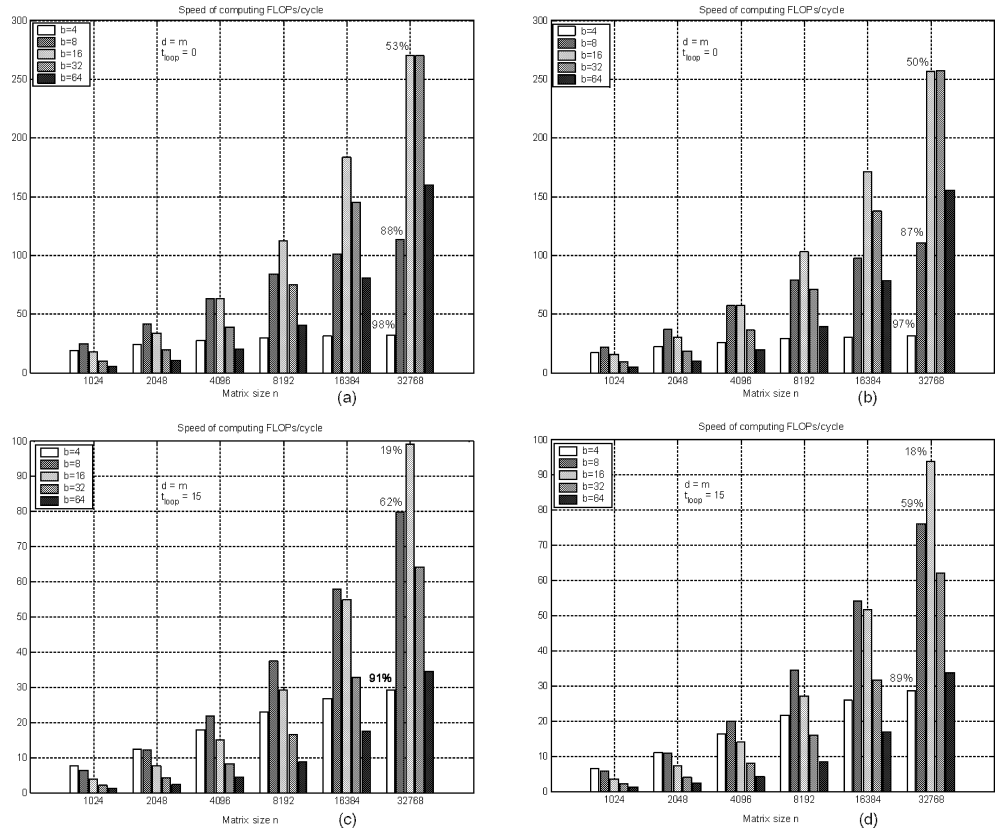

Fig. 5 Performance of the blocked LU factorization with different matrix sizes and block sizes where d=m. The numbers on top of bars at n=32 K are the percentages of the maximum speed attained.

execute the MMA operation in the minimum time. We formulated the alignment operations including matrix transpose as MMA operations and could overlap them with the load/store operations to hide their overhead.

We have designed a fine-grained blocked algorithm for the GEMM operation (Level-3 BLAS) as an application or kernel that is run completely on the matrix unit. Our analytical results showed that the maximum speed of computing is attained. Increasing the matrix unit size, b, increases the speed of computing. We have also designed a blocked right-looking algorithm for the LU factorization with partial pivoting, which represents an example of an application that needs to run on both scalar and matrix units. Only the matrix updating primitive of the algorithm is scheduled to execute on the matrix unit while the other primitives run on the scalar unit. We didn't consider execution overlap between the matrix and scalar units.

Since the percentage of the MMA increases as the matrix size increases, around 50–90% of the maximum computing speed is attained by varying the model parameters. Getting high performance requires increasing the amount of data reuse, i.e., the number of blocks, d, that can be loaded into registers and reused before replacement. Moreover, decreasing the loop overhead (using loop unrolling techniques) greatly enhances the performance.

In our algorithms, we have used software pre-fetching techniques and software-controlled data cache replacement policy in order to hide memory latency by exploiting the decoupling between the load/store unit and the processing (scalar and matrix) units. Although this may complicate programming and increases program sizes, it may be useful in hiding memory latency and using smaller cache sizes (since only three b×b blocks may exist at the same time inside the data cache as shown in the algorithms).

In our future works, we will consider implementing and evaluating other factorization al-

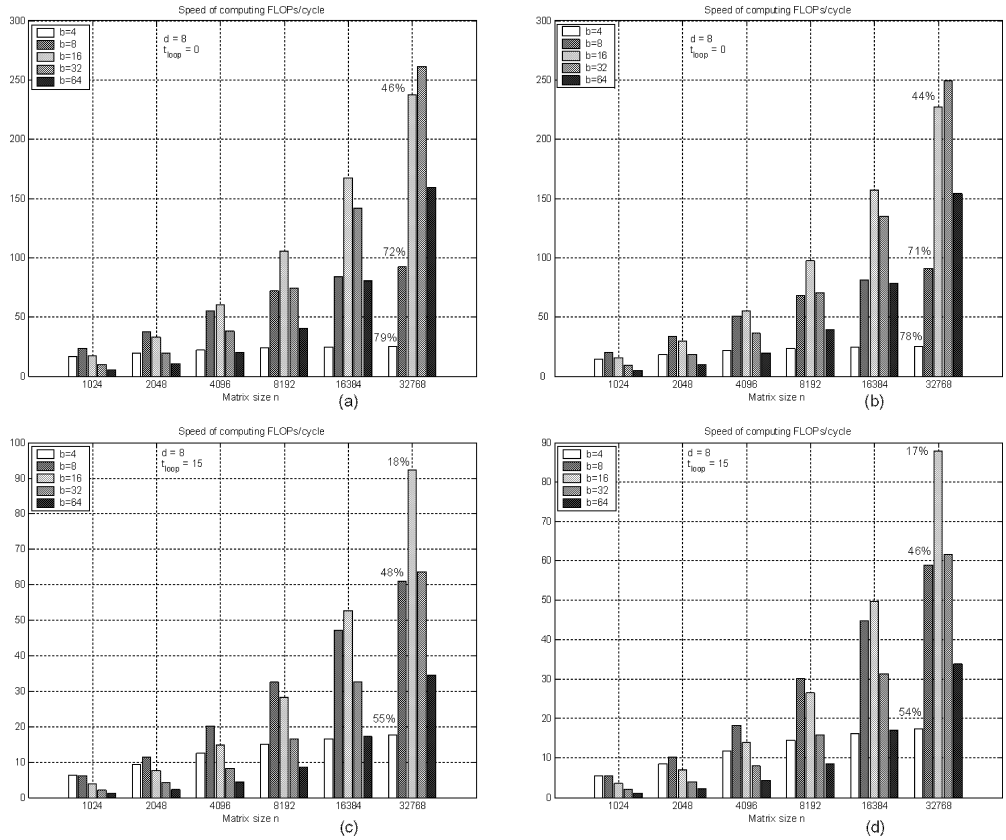

Fig. 6 Performance of the blocked LU factorization with d=8.

gorithms on the proposed matrix processor including Cholesky, QR, and singular value decompositions.

References

1) Williams, S., et al.: The potential of the cell processor for scientific computing, *CF '06: Proc. 3rd conference on Computing frontiers*, New York, NY, USA, pp.9–20, ACM Press (2006).
2) Gustafson, J.L. and Greer, B.S.: A hardware accelerator for the Intel Math Kernel, White paper, ClearSpeed Technology Inc., at http://www.clearspeed.com (2006).
3) The Berkeley Intelligent RAM (IRAM) Project: http://iram.cs.berkeley.edu.
4) Lawson, C.L., Hanson, R.J., Kincaid, R.J. and Krogh, F.T.: Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Software*, Vol.5, pp.308–323 (1979).
5) Dongarra, J.J., Croz, J.D., Hammarling, S. and Hanson, R.J.: An extended set of FORTRAN basic linear algebra subprograms, *ACM Trans. Math. Software*, Vol.14, pp.1–17 (1988).
6) Dongarra, J.J., Croz, J.D., Duff, I. and Hammarling, S.: A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Software*, Vol.16, pp.1–17 (1990).
7) Kågström, B., Ling, P. and Loan, C.V.: High Performance GEMM-based level 3 BLAS: Sample Routines for Double Precision Real Data, *High Performance Computing II*, North-Holland, pp.269–281 (1991).
8) Kågström, B., Ling, P. and Loan, C.V.: GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, *ACM Trans. Math. Software*, Vol.24, No.3, pp.268–302 (1998).
9) Gustavson, F.G., Henriksson, A., Jonsson, I., Kågström, B. and Ling, P.: Superscalar GEMM-based Level 3 BLAS — The Ongoing Evolution of a Portable and High-Performance Library, *PARA '98: Proc. 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, London, UK, pp.207–215, Springer-Verlag (1998).
10) Dongarra, J.J., Duff, I.S., Sorensen, D.C., and van der Vorst, H.A.: *Numerical Linear Algebra for High-Performance Computers*, SIAM,

Philadelphia, Pennsylvania (1998).
11) Golub, G.H. and Loan, C.F.V.: *Matrix Computations*, John Hopkins, Baltimore, Maryland (1996).
12) Gallivan, K.A., Plemmons, R.J. and Sameh, A.H.: Parallel algorithms for dense linear algebra computations, *SIAM Rev.*, Vol.32, No.1, pp.54–135 (1990).
13) Kung, S.: *VLSI Array Processors*, Prentice Hall (1988).
14) Dally, W. and B.Towles: *Principles and Practices of Interconnection Networkks*, Elsevier (2004).
15) Zekri, A.S. and Sedukhin, S.G.: Computationally Efficient Parallel Matrix-Matrix Multiplication on the Torus, *the 6^{th} Int. Symp. on High Performance Computing*, Nara City, Japan, (being published as LNCS4759) (2005).
16) Zekri, A.S. and Sedukhin, S.G.: The General Matrix Multiply-Add Operation on 2D Torus, *the 20^{th} IEEE IPDPS Symp., PDSEC-06 Workshop*, Rhodes Island, Greece, IEEE Computer Society (2006).
17) Gustavson, F.G.: High-performance linear algebra algorithms using new generalized data structures for matrices, *IBM J. Res. Dev.*, Vol.47, No.1, pp.31–55 (2003).
18) Park, N., Hong, B. and Prasanna, V.K.: Analysis of Memory Hierarchy Performance of Block Data Layout, *ICPP '02: Proc. 2002 International Conference on Parallel Processing (ICPP'02)*, p.35 (2002).
19) Asanovic, K.: Vector Microprocessors, PhD Thesis, Univ. of California at Berkeley (1998).
20) Callahan, D., Kennedy, K. and Porterfield, A.: Software prefetching, *ASPLOS-IV: Proc. 4^{th} Int. Conf. on Architectural support for programming languages and operating systems*, pp.40–52 (1991).
21) Kondo, M., Okawara, H. and Nakamura, H., et al.: SCIMA: A novel processor architecture for high performance computing, *4^{th} International conference on HPC in the Asia Pacific Region*, Vol.1, pp.355–360 (2000).
22) Shaw, A.C.: Reasoning About Time in Higher-Level Language Software, *IEEE Trans. Softw. Eng.*, Vol.15, No.7, pp.875–889 (1989).
23) Dongarra, J.J., Croz, J.D., Duff, I. and Hammarling, S.: A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Software*, Vol.16, pp.1–17 (1990).
24) Cannon, L.: A Cellular Computer to Implement the Kalman Filter Algorithm, Ph.D. Thesis, Montana State University (1969).
25) Zekri, A.S. and Sedukhin, S.G.: Matrix Transpose on 2D Torus Array Processor., *The 6^{th} IEEE International Conference on Computer and Information Technology*, Seoul, Korea, p.45 IEEE Computer Society (2006).
26) Wolfe, M.J.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995).
27) Demmel, J.W.: *Applied numerical linear algebra*, SIAM, Philadelphia, PA (1997).
28) Dongarra, J.J., Gustafson, F.G. and Karp, A.: Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Review*, Vol.26, No.3, pp.91–112 (1984).
29) Montoye, R.K., Hokenek, E. and Runyon, S.L.: Design of the IBM RISC System/6000 Floating-Point Execution Unit., *IBM J. of Res. and Develop.*, Vol.34, No.1, pp.59–70 (1990).

(Received July 23, 2007)
(Accepted October 31, 2007)

Ahmed S. Zekri received his M.Sc. degree in computer science from Alexandria University, Egypt in 1999. He is an assistant lecturer with the department of Mathematics and Computer Science, Faculty of Science, Alexandria University. Currently, he is a Ph.D. candidate at the University of Aizu. His current research interests include parallel and distributed computing, vector/matrix processing, and parallel algorithm design.

Stanislav G. Sedukhin is a professor and Director of the Computer Software Department at the University of Aizu. He received his Ph.D. in Computer Science and Dr.Sci. (Physics and Mathematics) from the Russian Academy of Sciences in 1982 and 1993, respectively. His research interests are in parallel and distributed computing, architectural synthesis of VLSI-oriented processors, and design of highly-parallel algorithms and application-specific array processors. Dr. Sedukhin is a member of ACM, IEEE Computer Society, and IEICE.