The General Matrix Multiply-Add Operation on 2D Torus

Ahmed S. Zekri and Stanislav G. Sedukhin
The Graduate School of Computer Science and Engineering
The University of Aizu, Aizu-Wakamatsu City, Fukushima 965-8580, Japan
{d8062103, sedukhin}@u-aizu.ac.jp

Abstract

In this paper, the index space of the ($n\times n$)-matrix multiply-add problem $C = C + A{\cdot}B$ is represented as a 3D $n\times n\times n$ torus. All possible time-scheduling functions to activate the computation and data rolling inside the 3D torus index space are determined. To maximize efficiency when solving a single problem, we mapped the computations into the 2D $n\times n$ toroidal array processor. All optimal 2D data allocations that solve the problem in n multiply-add-roll steps are obtained. The well known Cannon's algorithm is one of the resulting allocations. We used the optimal data allocations to describe all variants of the GEMM operation on the 2D toroidal array processor. By controling the data movement, the transposition operation is avoided in 75% of the GEMM variants. However, only one matrix transpose is needed for the remaining 25%. Ultimately, we described four versions of the GEMM operation covering the possible layouts of the initially loaded data into the array processor.

1. Introduction

Most of scientific, engineering, and image processing applications are expressed in terms of matrix operations. The basic linear algebra subprograms (BLAS) were introduced to make the performance of the dense linear algebra algorithms portable on high performance architectures [15, 7, 6]. Levels 1 and 2 of the BLAS, which describe the vector and matrix-vector operations, involve $O(n)$ and $O(n^2)$ computations accompanied with $O(n)$ and $O(n^2)$ load/store operations, respectively, where n is the size of the problem. The level 3 BLAS describe the matrix-matrix operations which involve $O(n^3)$ computations and $O(n^2)$ data movements. Because of the high ratio of operations to data movement, the level 3 BLAS are targeted to the high performance computers with hierarchical memory and parallel processing capabilities.

Due to the different organizations of the current and fore coming high performance parallel computers, it is a tedious work to optimize the different level 3 BLAS kernels on each computer system. However, it is possible to express all the level 3 BLAS kernels in terms of the highly optimized general matrix multiply-add operation (GEMM) and a small percentage of levels 1 and 2 of the BLAS [11, 12].

Different 2D algorithms have been proposed to solve the matrix multiply-add problem. These algorithms fall into three main categories based on the communication scheme. Local communication algorithms include systolic matrix multiplication and Cannon's algorithm [13, 9, 2]. The broadcast-based category includes the outer product matrix multiplication algorithm by Agarwal et al [1] and SUMMA (Scalable Universal Matrix Multiplication Algorithm) [18]. The third category combines both broadcasting and local communication such as the algorithm by Fox et. al. [8], BiMMeR [10], and PUMMA (Parallel Universal Matrix Multiplication Algorithms) [3].

In this paper, we describe four versions of the GEMM operation on the 2D $n\times n$ toroidal array processor (AP). Specifically, we exploit local communications together with the optimal data allocations that compute $C = C + A{\cdot}B$ on the toroidal AP to reduce data redistributions. We used the skewing operation to avoid the transposition operation in 75% of the GEMM variants. For the remaining 25%, one transposition operation is needed.

We assume the following for the toroidal AP. Each processing element (PE) performs a fused multiply-add-roll operation at each step. That is, each PE computes $c = c + a \times b$, where a, b, and c are scalars, then the appropriate data is rolled (or circularly shifted) to neighbour PEs. To overcome latency due to the long wrap around connections, the toroidal AP can be folded such that all connections between PEs are equal [17, 4]. For simplicity, the data allocations of the AP will be shown here in unfolded form.

In Section 2, the index space of the matrix multiply-add problem is represented as a 3D torus index space. All optimal time-scheduling functions are presented. For each function, the optimal data allocations resulting from mapping the 3D index space to the 2D AP space are discussed.

1-4244-0054-6/06/$20.00 ©2006 IEEE

Section 3 discusses the initial data layouts of the optimal AP data allocations. The pre- and post-processing needed for data alignment are also discussed. Four versions of the GEMM operation on the 2D $n\times n$ toroidal AP are described in Section 4. Section 5 concludes the paper.

2. Matrix Multiply-Add on the Torus

Consider the ($n\times n$)-matrix multiply-add problem

$$C = C + A\cdot B,$$

where $A = [a(i,j)]$, $B = [b(i,j)]$, and $C = [c(i,j)]$ are $n\times n$ dense matrices. The index space is the $n\times n\times n$ cube defined by the set of points $\Im = \{(i,j,k)^T : i,j,k \in \{0,1,\ldots,n-1\}\}$, where at each index point $p = (i,j,k)^T$ the computation $c(i,j) = c(i,j) + a(i,k)\times b(k,j)$ is performed. Scheduling the computations inside the index space $\Im$ is needed so that the elements $a(i,k)$, $b(k,j)$, and $c(i,j)$ should meet with each other in correct place and time while preserving data dependencies. To do so, we showed in [19] that all elements of matrices A and B can be reused to update matrix C on each computing step. Specifically, matrix A is rolled along the j-axis while matrices B and C are rolled along i and k axes, respectively. The three orthogonal rolling directions for all elements of matrices A, B, and C form a 3D $n\times n\times n$ torus computation space .

2.1. Time-Space Mapping

A *Time-scheduling* mapping partions all the computations of an index space into non-overlapping groups of independent computations and assigns a time-step to each group. A *Space mapping* assigns the ordered groups of computations to PEs so that each group is executed at the same time.

Different time-scheduling and space-allocation functions are proposed in the literature [13, 5, 14]. In [16], modular time-space mappings are used to derive data distribution independent programs for matrix-matrix multiplication. Here, we use modular time-scheduling to schedule the computations inside the index space $\Im$ because of the circular movement of data and results. The modular time-scheduling function is defined by:

$$\text{Step}(p) = \pi^T\cdot p \mod n,$$

where Step$(p) : \mathbb{Z}^3 \rightarrow \mathbb{Z}$, $\mathbb{Z}$ is the set of integers, $p \in \Im$, and $\pi = (\alpha,\beta,\gamma)^T$ is a time-scheduling vector with integer components. For space-allocation we use the projection method to allocate the scheduled computations to the AP. The space-allocation function is defined by the linear map:

$$\text{Allocation}(p) = S_\eta \cdot p,$$

where Allocation$(p) : \mathbb{Z}^3 \rightarrow \mathbb{Z}^2$, S_η is a 2×3 space transformation matrix to map the index points to the coordinates of the PEs, and η is a projection vector orthogonal to the AP basis, i.e., $S_\eta\cdot\eta = 0$.

2.2. Optimal Mapping

In this subsection, we determine the optimal values of the time-scheduling vector π and the minimum number of PEs in the AP to perform the computations in the minimum number of steps.

Let $r_0, r_1, \ldots, r_{m-1}$ be ordered objects connected in a ring. If we start from r_l, $0 \le l \le m-1$, then we have two directions to traverse the ring which result in the two paths $r_l, r_{l+1}, \ldots, r_{m-1}, r_0, \ldots, r_{l-1}$ and $r_l, r_{l-1}, \ldots, r_0, r_{m-1}, \ldots, r_{l+1}$. The subscript of the next object to traverse in the two paths is obtained by the modular operations $(l+1) \bmod m$, and $(l-1) \bmod m$, respectively, where l is the subscript of the current object. We will say that the rolling order of index l in the first path is *increasing* while in the second is *decreasing*. The 3D torus space comprises from three rings along i, j, and k axes. Each ring has two degrees of freedom to roll. Therefore, eight different combinations exist to roll the elements of A, B, and C simultaneously along j, i, and k axes, respectively. If the ring along a given axis is rolled in increasing/decreasing order, the corresponding component of the time schedule vector is assigned a positive/negative sign. The magnitudes of the schedule vector components must be chosen so that data dependencies are preserved while the communications are local. Meanwhile, at any step of computing the data is moving in the three rings which mean that none of the components equals to zero. In [19], we found for the vector π that each of the three components α, β, and γ equals to either $+1$ or -1, i.e., $\pi \in \{(1,1,1)^T, (-1,1,1)^T, (1,-1,1)^T, (1,1,-1)^T, (-1,-1,1)^T, (-1,1,-1)^T, (1,-1,-1)^T, (-1,-1,-1)^T\}$. Hence, we have eight different time-scheduling functions for scheduling the computations inside the 3D torus.

Solving a single matrix multiply-add problem using an $n\times n\times n$ AP is not efficient since only n^2 computations are active at each step. Therefore, we mapped the computations to the 2D AP space by an admissible projection vector η so that exactly one computation point is assigned to exactly one PE at a time, i.e., $\pi^T\cdot\eta \neq 0$. Although many admissible projection vectors can be applied, the unit vectors $e_1 = (1,0,0)^T$, $e_2 = (0,1,0)^T$, and $e_3 = (0,0,1)^T$, and their negatives are the optimal and give the minimum number of PEs, n^2. The PEs are organized in an $n\times n$ toroidal AP. In this case, each optimal 2D data allocation requires n multiply-add-roll steps to solve the matrix multiply-add problem on the resulting toroidal AP.

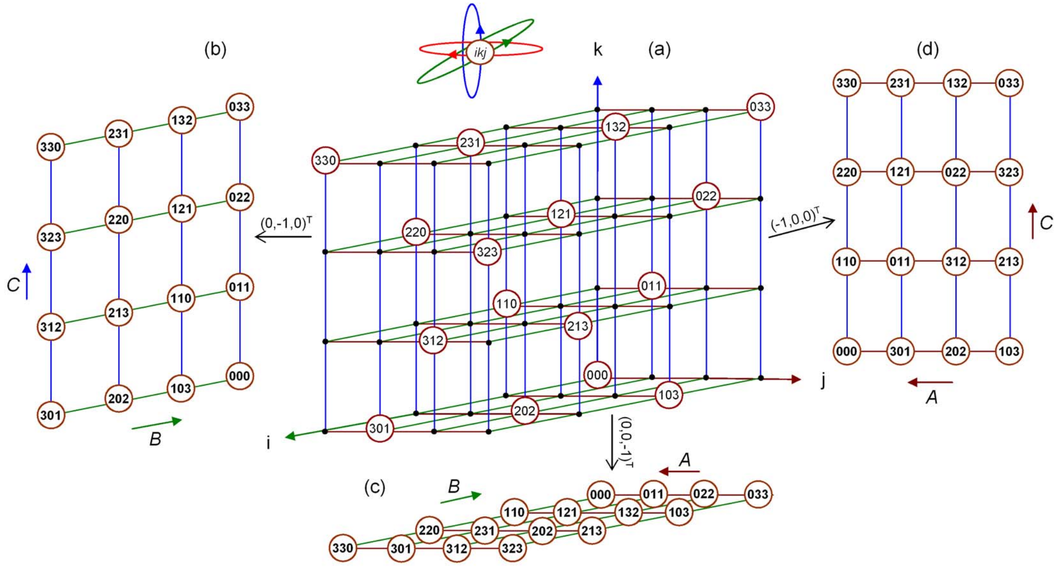

Figure 1. The distributions of A, B, and C elements for π = (-1,-1,1)^T and three different data allocations to the AP at Step(p) = 0.

2.3. Optimal Time-Space Allocations

In the previous subsection, eight optimal time-scheduling vectors are obtained. It can be easily seen that four of the eight vectors are the negatives of the other four. That is for each scheduling vector π_i and its negative -π_i, the elements of A, B, and C have the same distributions inside $\Im$ at the computing steps $\text{Step}_{\pi_i}(p) = q \mod n$, and $\text{Step}_{-\pi_i}(p) = [n-q] \mod n$ where $q = 0, 1, \ldots, n-1$. In other words, there are four distinct time-scheduling functions each activating totally n^3 computations in a sequence of n steps so that n^2 computations are active at each step. However, due to the cyclical nature of processing we can start computing from any of the different n steps within each scheduling function and evolve through the remaining $n-1$ steps in one of two possible ways. For each time-scheduling function, three optimal projections give three possible 2D data allocations. Totally, 4×3n different initial data allocations are available to solve the matrix multiply-add problem. This large number of initial distributions serves in minimizing the overhead of redistributing the data to prepare for layout of for the next computation. For each time-scheduling function we select the scheduling step that activates the computation at the point $p_0 = (0,0,0)^T$ to be the initial scheduling step, i.e., $\text{Step}(p) = 0$.

i) $\alpha = -1$**,** $\beta = -1$**, and** $\gamma = 1$**.** In this case, the time-scheduling function is given by $\text{Step}(p) = [k-i-j] \mod n$. The rings along i and j axes are rolled in decreasing order, and the ring along k-axis is rolled in increasing order. The distributions of A, B, and C elements inside the 3D torus index space at $\text{Step}(p) = 0$ are shown in Figure 1 (note that the computation at $p_0 = (0,0,0)^T$ is active at the initial step). We use the circles (*ikj*) to show the active computation points at each scheduling step. At each circle, the appropriate elements $a(i,k)$, $b(k,j)$, and $c(i,j)$ meet so that the computation $c(i,j) = c(i,j) + a(i,k) \times b(k,j)$ is performed. Then, the three elements are rolled in the corresponding directions and the subsequent computations are activated at $\text{Step}(p) = 1, 2, \ldots, n-1$.

Mapping the 3D torus index space into the 2D AP space along j, k, and i axes we get three optimal data allocations for computing matrix C as illustrated in Figure 1(b, c, and d, respectively). We can see in each allocation that one matrix remains resident inside the AP and the other two matrices are rolled. In order to use for example allocation (c), which is the classical Cannon's algorithm [2], both A and B are initially loaded in the conventional form. Then, matrix A is skewed westward and matrix B northward (see next section). Next, all PEs synchronously perform a fused multiply-add-roll operation where the elements of A and B are rolled westward and northward, respectively, at the end of the operation. The elements of matrix C remain resident during computing. After n steps of multiply-add-roll, C is obtained.

ii) $\alpha = -1$**,** $\beta = 1$**, and** $\gamma = -1$**.** The scheduling function is given by $\text{Step}(p) = [j-i-k] \mod n$. The values of the scheduling vector components indicate that the two rings along i and k axes are rolled in decreasing order while the ring along the j axis is rolled in increasing order. Projecting the 3D computation space parallel to the j, k, and i

axes gives the three allocations (a), (b), and (c) in Figure 2, respectively.

iii) $\alpha = 1$**,** $\beta = -1$**, and** $\gamma = -1$**.** The values of α, β, and γ indicate that the i-index is increasing while the values of j and k indexes are decreasing during rolling. The time-scheduling function is given by Step(p) $= [i - j - k] \mod n$. Three data allocations can be obtained from projection along $\pm e_1$, $\pm e_2$, and $\pm e_3$.

iv) $\alpha = -1$**,** $\beta = -1$**, and** $\gamma = -1$**.** The time-scheduling function is given by Step(p) $= [-i - j - k] \mod n$, where the three rings are rolled in decreasing order. Projections parallel to i, j, and k axes give three different 2D allocations.

v) Special case. When we solve the (2×2)-matrix multiply-add problem, the four initial distributions resulting from the four time-scheduling functions give the same 3D distribution of the computations. This distribution can be described by any one of the eight optimal scheduling functions. Therefore, there are only three optimal data allocations to solve the (2×2)-matrix multiply-add problem on the 2×2 toroidal AP. The reason behind this result is that the possible two rolling directions for each ring in the 3D torus have the same traverse path, i.e., r_0, r_1. This means that when $n = 2$ the direction of rolling has no effect on the distribution of the data inside the computation space.

a_{00} $b_{00}c_{00}$	a_{01} $b_{11}c_{01}$	a_{02} $b_{22}c_{02}$	a_{03} $b_{33}c_{03}$
a_{10} $b_{01}c_{11}$	a_{11} $b_{12}c_{12}$	a_{12} $b_{23}c_{13}$	a_{13} $b_{30}c_{10}$
a_{20} $b_{02}c_{22}$	a_{21} $b_{13}c_{23}$	a_{22} $b_{20}c_{20}$	a_{23} $b_{31}c_{21}$
a_{30} $b_{03}c_{33}$	a_{31} $b_{10}c_{30}$	a_{32} $b_{21}c_{31}$	a_{33} $b_{32}c_{32}$

(a)

a_{00} $b_{00}c_{00}$	a_{01} $b_{11}c_{01}$	a_{02} $b_{22}c_{02}$	a_{03} $b_{33}c_{03}$
a_{13} $b_{30}c_{10}$	a_{10} $b_{01}c_{11}$	a_{11} $b_{12}c_{12}$	a_{12} $b_{23}c_{13}$
a_{22} $b_{20}c_{20}$	a_{23} $b_{31}c_{21}$	a_{20} $b_{02}c_{22}$	a_{21} $b_{13}c_{23}$
a_{31} $b_{10}c_{30}$	a_{32} $b_{21}c_{31}$	a_{33} $b_{32}c_{32}$	a_{30} $b_{03}c_{33}$

(b)

a_{00} $b_{00}c_{00}$	a_{10} $b_{01}c_{11}$	a_{20} $b_{02}c_{22}$	a_{30} $b_{03}c_{33}$
a_{31} $b_{10}c_{30}$	a_{01} $b_{11}c_{01}$	a_{11} $b_{12}c_{12}$	a_{21} $b_{13}c_{23}$
a_{22} $b_{20}c_{20}$	a_{32} $b_{21}c_{31}$	a_{02} $b_{22}c_{02}$	a_{12} $b_{23}c_{13}$
a_{13} $b_{30}c_{10}$	a_{23} $b_{31}c_{21}$	a_{33} $b_{32}c_{32}$	a_{03} $b_{33}c_{03}$

(c)

Figure 2. The optimal AP data allocations at Step(p) = 0 for π = (-1,1,-1)^T. (a) A remains, B rolls northward, and C rolls westward. (b) C remains, B rolls northward, and A rolls eastward. (c) B remains, C rolls northward, and A rolls eastward.

3. Initial Data Layouts

In this section, we discuss the possible data layouts that may be initially loaded into the AP. We also describe the pre- and post-processing needed to align the three matrices A, B, and C such that data redistribution overhead for the subsequent computations is reduced.

We mean by an initial data layout the distribution of the loaded data relative to the coordinates of the PEs in the AP. We organize the 2D toroidal AP so that PE(0,0) is located at the top left corner and PE($n - 1$,$n - 1$) is located at the bottom right corner of the AP. We call this organization of the PEs the canonical layout. Obviously, the canonical layout matches the conventional distribution of a 2D matrix. In Subsection 2.3, we have discussed four optimal time-scheduling functions inside the 3D torus computation space for computing the matrix multiply-add problem. Three different AP data allocations resulted from each scheduling function. Although, we have totally $12n$ possible data allocations to start computing, we only consider in this paper those allocations that include the elements a(0,0), b(0,0), and c(0,0) inside PE(0,0) at the initial computing step, Step(p) $= 0$. That is only twelve optimal data allocations are considered. These allocations require the least alignment overhead among all resulting allocations because the distributions of the elements of A, B, and C are more closer to the conventional layout of a 2D matrix than in the other allocations.

The twelve optimal allocations can be classified according to the direction of projection they resulted from into three groups each containing four different allocations. In the first group resulted from the projection vector $\pm e_1$, matrix B remains inside the AP during computing while matrices A and C are rolled. In the second group resulted from $\pm e_2$, matrix A remains resident and both matrices B and C are rolled. In the third group resulted from $\pm e_3$, matrix C remains resident while both matrices A and B are rolled. The Cannon's algorithm [2] belongs to the last group. No-

tice that the elements of A, B, and C in the optimal data allocations are aligned before starting computing in order to give correct results. But, what is the layout of the three matrices that is initially loaded into the AP before any alignment is done? It is easy to see that in the first group of allocations the two matrices A^T and B are initially loaded (in the canonical form) into the AP as in allocation (c) in Figure 2. In the second group, the two matrices A and B^T are initially loaded into the AP, while A and B are initially loaded into the AP in the third group (see allocations (a) and (b), respectively, in Figure 2).

To complete our discussion about all variants of the initially loaded data layouts, we introduce the fourth expected data layout where the two matrices A^T and B^T are initially loaded into the AP. Although this layout is not included in the 2D optimal allocations that solve the problem $C = C + A{\cdot}B$, we will show later how to use it and the three initial data layouts: $A\&B$, $A\&B^T$, and $A^T\&B$ to describe four computationally equivalent versions of the GEMM operation. In describing the four initial data layouts, we did not include matrix C because the distribution of C is implicitly determined by the distributions of A and B. At the same time, the optimal scheduling of the matrix multiply-add problem inside the 3D torus index space, we build on, is determined for computing matrix C not its transpose.

From the point-of-view of the main memory storage format, the four mentioned initial data layouts cover all possible variants of storing matrices A and B into memory. According to the standard format of storing 2D matrices into the main memory of current systems , either the two matrices A and B are both stored in row-major or column-major orders, or one of the two matrices is stored row-major and the other is stored column major. Therefore, providing four versions of the GEMM operation gives more freedom to choose the one that best matches the storing format of the data inside the main memory.

3.1. The Skewing Operation

Let a matrix X has n rows and n columns. To skew, for example, the rows of X to the West, the ith row, i = 0,1,...,$n-1$, is rolled i times to the West using circular rolling of data. In general, to skew the rows/columns of a matrix, each row/column is rolled so that the diagonal elements of the matrix move to the 0th column/row. Figure 3 shows the result of skewing the rows and columns of a 4×4 matrix. The following pseudo code takes $n-1$ rolls (circular shifts) to skew the rows of the $n{\times}n$ matrix X to the West.

load matrix X into AP in canonical form
for k = 1, $n-1$ **do**
  for all $i = k, n-1$ **and** j = 0, $n-1$ **do**
    $x(i, j) = x(i, [j+1] \bmod n)$

(a)

x_{00}	x_{01}	x_{02}	x_{03}
x_{11}	x_{12}	x_{13}	x_{10}
x_{22}	x_{23}	x_{20}	x_{21}
x_{33}	x_{30}	x_{31}	x_{32}

(b)

x_{00}	x_{11}	x_{22}	x_{33}
x_{10}	x_{21}	x_{32}	x_{03}
x_{20}	x_{31}	x_{02}	x_{13}
x_{30}	x_{01}	x_{12}	x_{23}

Figure 3. (a) The rows of matrix X are skewed westward; (b) The columns of matrix X are skewed northward.

(a)

a_{00} $b_{00}c_{00}$	a_{01} $b_{11}c_{01}$	a_{02} $b_{22}c_{02}$	a_{03} $b_{33}c_{03}$
a_{11} $b_{10}c_{10}$	a_{12} $b_{21}c_{11}$	a_{13} $b_{32}c_{12}$	a_{10} $b_{03}c_{13}$
a_{22} $b_{20}c_{20}$	a_{23} $b_{31}c_{21}$	a_{20} $b_{02}c_{22}$	a_{21} $b_{13}c_{23}$
a_{33} $b_{30}c_{30}$	a_{30} $b_{01}c_{31}$	a_{31} $b_{12}c_{32}$	a_{32} $b_{23}c_{33}$

(b)

a_{00} $b_{00}c_{00}$	a_{10} $b_{01}c_{11}$	a_{20} $b_{02}c_{22}$	a_{30} $b_{03}c_{33}$
a_{11} $b_{10}c_{10}$	a_{21} $b_{11}c_{21}$	a_{31} $b_{12}c_{32}$	a_{01} $b_{13}c_{03}$
a_{22} $b_{20}c_{20}$	a_{32} $b_{21}c_{31}$	a_{02} $b_{22}c_{02}$	a_{12} $b_{23}c_{13}$
a_{33} $b_{30}c_{30}$	a_{03} $b_{31}c_{01}$	a_{13} $b_{32}c_{12}$	a_{23} $b_{33}c_{23}$

Figure 4. Optimal AP allocations at Step(p)=0. (a) π = (-1,-1,1)^T, A rolls westward, and B rolls northward. (b) π= (1,-1,-1)^T, A rolls wesward, and C rolls northward.

3.2. Pre- and Post-Processing

By analyzing the different data allocations in Subsection 2.3, we can see that some pre-processing is needed to align the loaded data into the proper layout for computing. The same is true after getting the final result C, some post-processing may be needed to return the result matrix to the canonical form. There are two types of pre- and post-processing for aligning the optimal data allocations: skewing and permuting the rows/columns of a matrix. We mean by permuting the rows/columns reversing their order except row/column #0. To illustrate, consider for example allocations (a) and (b) of Figure 2. In allocation (a), the columns of matrix B are skewed northward before computing while the rows of matrix C are skewed westward after computing. In allocation (b), before computing, the rows of ma-

trix A are skewed eastward. The columns of matrix B are skewed northward then, the order of rows is reversed excluding row #0, i.e., only rows #1 and #3 are exchanged. Now the twelve optimal allocations can be divided according to the type of pre- and post-processing into two groups. One of them contains the allocations that require the skewing operation only. This groups contains the three allocations: allocation (a) in Figure 2 and the two allocations (a) and (b) in Figure 4. The other group contains the remaining allocations that need both the skewing and the permutation operations. Nine allocations belong to this group (see for example allocations (b) and (c) in Figure 2). In the next section, we will use the skewing operation and the three optimal data allocations (Figure 2(a) and Figure 4) together with the matrix transpose algorithm [20] to describe four versions of the GEMM operation on the $n\times n$ toroidal AP where $n \geq 2$.

4. GEMM on the 2D Torus

The GEMM operation is the very important kernel of the level 3 BLAS kernels. The GEMM-based approach was introduced to express the level 3 BLAS routines in terms of one basic operation, the GEMM, and a small percentage of levels 1 and 2 of the BLAS [11, 12]. In this section we describe four versions of the GEMM operation on the 2D $n\times n$ toroidal AP.

The four different variants of the GEMM operation are:

$$\begin{aligned} C &= \lambda C + \mu A{\cdot}B, \\ C &= \lambda C + \mu A{\cdot}B^T, \\ C &= \lambda C + \mu A^T{\cdot}B, \\ C &= \lambda C + \mu A^T{\cdot}B^T, \end{aligned}$$

where λ and μ are scalars. For simplicity, we set $\lambda = \mu = 1.0$. We use the skewing operation to perform three of the above four variants of the GEMM operation without explicitly making a transposition operation. What is needed is two skewing operations at most for aligning the data. The initial data layout of matrix A and matrix B inside the AP determine which GEMM variant will be performed without applying the transposition operation. To perform the remaining GEMM variant we need to transpose only one matrix beside using the skewing operation as will be shown at the end of this section.

The importance of the skewing operation appears in solving the GEMM operation when the two matrices A and B commute, i.e., solving $C = \lambda C + \mu \text{op}(B)\ \text{op}(A)$ where op(X) denotes X or X^T. In this case, no need to reload A and B into the AP. We only need to apply the skewing operation properly to align the data for the new computation. For example, to find the product $A{\cdot}B$, assuming the layout $A\&B$ is initially loaded into the AP, matrix A is skewed westward and matrix B is skewed northward to align the data. After computing is finished, the data distribution inside the AP returns to the same alignment before starting the computing because of the rolling of data at the end of each multiply-add-roll step. Now, to compute $C = C + B{\cdot}A$ with the same initial layout $A\&B$, both A and B must be returned to their canonical distribution inside the AP which costs two skewing operations. The directions of rolling and skewing of A and B are exchanged to give the required new result. That is, matrix A and matrix B are skewed northward and westward, respectively. During computing, A is rolled northward and B is rolled westward while matrix C remains resident. We can apply the same rules in the above example to find the products $B{\cdot}A^T$ and $B^T{\cdot}A$. But, for finding the product $B^T{\cdot}A^T$ the transposition operation is needed.

4.1. Four GEMM Kernels

Four initial data layouts ($A\&B$, $A\&B^T$, $A^T\&B$, and $A^T\&B^T$) can be loaded into the AP. Therefore, we provide four versions of the GEMM operation on the toroidal AP so that the one that matches the current layout of the two matrices A and B inside the AP is selected. The differences among the four GEMM versions are the initially loaded layout of A and B (assuming C is canonically distributed), and consequently, the skewing and the rolling directions applied to the three matrices. It suffices here to discuss one of the four GEMM versions in detail and summarize the results of the others.

Consider the initial data layout $A\&B$ in which the two matrices A and B are initially loaded into the $n\times n$ 2D toroidal AP in the canonical form. By letting one of the three matrices A, B, and C resides in the AP during computing and rolling the other two matrices in predefined directions, we can compute the products $A{\cdot}B$, $A{\cdot}B^T$, and $A^T{\cdot}B$ without any explicit transposition operation. However, to compute the product $A^T{\cdot}B^T$ one transposition operation is needed as will be shown below.

i) Matrix C resides. This case is Cannon's algorithm [2] where matrix A is skewed westward and matrix B is skewed northward to align the elements for computing. At each step of computing, matrix A is rolled westward and matrix B northward. After n multiply-add-roll steps, matrix C is the solution of the problem $C = C + A{\cdot}B$.

ii) Matrix A resides. Only matrix B is skewed northward before computing. During computation, matrix B is rolled northward while matrix C westward. After computing, matrix C is the solution of problem $C = C + A{\cdot}B^T$. The first two steps of the computing are shown in Figure 5. To return matrix C to the canonical form after getting the final result, the skewing operation is applied to skew matrix C

Initial Layout	Problem	Before Computing			During Computing			After Computing
		Transpose	Skew		Roll			Skew
			A	B	A	B	C	C
$A\&B$	$C = C + A \cdot B$		W	N	W	N		
	$C = C + A \cdot B^T$			N		N	W	E
	$C = C + A^T \cdot B$		W		W		N	S
	$C = C + A^T \cdot B^T$	A		N		N	W	E
		B	W		W		N	S
$A\&B^T$	$C = C + A \cdot B$			N		N	W	E
	$C = C + A \cdot B^T$		W	N	W	N		
	$C = C + A^T \cdot B$	A		N		N	W	E
		B	W		W		N	S
	$C = C + A^T \cdot B^T$		W		W		N	S
$A^T\&B$	$C = C + A \cdot B$		W		W		N	S
	$C = C + A \cdot B^T$	A		N		N	W	E
		B	W		W		N	S
	$C = C + A^T \cdot B$		W	N	W	N		
	$C = C + A^T \cdot B^T$			N		N	W	E
$A^T\&B^T$	$C = C + A \cdot B$	A		N		N	W	E
		B	W		W		N	S
	$C = C + A \cdot B^T$		W		W		N	S
	$C = C + A^T \cdot B$			N		N	W	E
	$C = C + A^T \cdot B^T$		W	N	W	N		

Table 1. Summary of the four GEMM versions on the $n \times n$ toroidal AP.

eastward. We can see in this case that without an explicit transpose to the matrix B and with only proper application of the skewing operation, the product $A \cdot B^T$ is calculated.

iii) Matrix B resides. Here, matrix A is skewed westward before starting computing. The two matrices A and C are rolled at the end of each computing step westward and northward, respectively. After the final step, matrix C is the solution of the problem $C = C + A^T \cdot B$ where matrix A is not explicitely transposed. To return matrix C to the canonical form, the skewing operation is applied to skew its columns southward.

iv) Transpose one matrix. To solve the problem $C = C + A^T \cdot B^T$ one matrix transpose is needed. There are two approches to do this. The first is performing an explicit transpose to either matrix A or matrix B but not both. This means one matrix transposition can be avoided. If matrix A is transposed, then we proceed as if we have the layout $A^T\&B$ inside the AP which is equivalent to applying case ii) above where matrix B is skewed northward and both B and C are rolled northward and westward, respectively. But, if the transposition is applied to matrix B then, matrix B^T remains inside the AP and we apply case iii) above. The second approach is to find the product $B \cdot A$, then transpose the result to give the matrix C. Both approaches use the transposition algorithm given in [20] which transposes an $n \times n$ matrix in $3n$ multiply-add-roll steps.

In Table 1, we summarize the directions of rolling and skewing (North, West, South, and East) of the elements of the three matrices A, B, and C for the four initial data layouts. Each initial data layout corresponds to a different version of the GEMM operation. We assume that the data is aligned relative to the canonical form of the toroidal AP. For each initial data layout, we show the directions of skewing for the three matrices before and after computing (if needed). The directions of rolling during computing are also shown. For illustration, consider the initial data layout $A\&B$ where matrix A and matrix B are initially loaded in the canonical form into the AP. To solve $C = C + A \cdot B^T$ only matrix B needs northward skewing before computing. During computing, matrix B is rolled northward and matrix C is rolled westward. After n steps of multiply-add-roll, C needs eastward skewing to return to its canonical form.

The transposition information in Table 1 describes the application of the first transposition approach only. To illustrate, consider solving the problem $C = C + A^T \cdot B^T$ where the initial data layout is $A\&B$. Two choices are available for the transposition; either matrix A or matrix B is explic-

a_{00} $b_{00}c_{00}$	a_{01} $b_{11}c_{01}$	a_{02} $b_{22}c_{02}$	a_{03} $b_{33}c_{03}$
a_{10} $b_{10}c_{11}$	a_{11} $b_{21}c_{12}$	a_{12} $b_{32}c_{13}$	a_{13} $b_{03}c_{10}$
a_{20} $b_{20}c_{22}$	a_{21} $b_{31}c_{23}$	a_{22} $b_{02}c_{20}$	a_{23} $b_{13}c_{21}$
a_{30} $b_{30}c_{33}$	a_{31} $b_{01}c_{30}$	a_{32} $b_{12}c_{31}$	a_{33} $b_{23}c_{32}$

Step(p) = 0

a_{00} $b_{10}c_{01}$	a_{01} $b_{21}c_{02}$	a_{02} $b_{32}c_{03}$	a_{03} $b_{03}c_{00}$
a_{13} $b_{20}c_{12}$	a_{10} $b_{31}c_{13}$	a_{11} $b_{02}c_{10}$	a_{12} $b_{13}c_{11}$
a_{22} $b_{30}c_{23}$	a_{23} $b_{01}c_{20}$	a_{20} $b_{12}c_{21}$	a_{21} $b_{23}c_{22}$
a_{31} $b_{00}c_{30}$	a_{32} $b_{11}c_{31}$	a_{33} $b_{22}c_{32}$	a_{30} $b_{33}c_{33}$

Step(p) = 1

Figure 5. The first two steps in computing $C = C + A{\cdot}B^T$, where matrix B rolls northward and matrix C rolls westward.

itly transposed inside the AP. The selection depends on the subsequent operation and whether the transpose of A or B is needed.

5. Conclusions

We have presented the GEMM operation on the 2D $n{\times}n$ toroidal AP. Depending on the optimal 2D data allocations that solve the matrix multiply-add problem $C = C + A{\cdot}B$ in n multiply-add-roll steps, we have discussed four data layouts that can be initially loaded into the AP. For each initial data layout we provided one version of the GEMM operation. Therefore, we have four versions of the same operation so that the one that best matches the initial data layout of the current computation is selected. We used the skewing operation to describe the alignment of data before and after the computing phase. This alignment overhead requires two skewing operations which cost $2(n-1)$ data rolls using circular shift of data. We have shown that 75% of the GEMM variants for each version can be performed without any matrix transposition. However, only one matrix transpose is needed to describe the remaining 25% of the GEMM variants which costs additional $3n$ multiply-add-roll steps.

References

[1] 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.

[2] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[3] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: parallel universal matrix multiplication algorithms. *Concurrency: Practice and Experience*, 6(7):543–570, Oct. 1994.

[4] W. Dally and B.Towles. *Principles and Practices of Interconnection Networkks*. Elsevier, 2004.

[5] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8):814–822, Aug. 1994.

[6] 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.

[7] 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.

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

[9] G. H. Golub and C. F. V. Loan. *Matrix Computations*. John Hopkins, Baltimore, Maryland, 1989.

[10] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhan. Matrix multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, 6(7):571–594, Oct. 1994.

[11] B. Kågström, P. Ling, and C. V. Loan. High performance GEMM-based level 3 BLAS: Sample routines for double precision real data. In *High Performance Computing II*, pages 269–281. North-Holland, 1991.

[12] B. Kågström, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

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

[14] D. Lavenier, P. Quinton, and S. Rajopadhye. Advanced systolic design. In *Digital Signal Processing for Multimedia systems*, Signal Processing Series, chapter 23, pages 657–692. Marcel Dekker, 1999.

[15] 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.

[16] H. J. Lee and J. A. Fortes. Modular mappings and data distribution independent computations. *Parallel Processing Letters*, 7(2):169–180, 1997.

[17] C. H. Sequin. Doubly twisted torus networks for VLSI processor arrays. In *the 8th Annual Symposium on Computer Architecture*, Minneapolis, Minnesota, USA, 1981.

[18] R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, The University of Texas, Apr. 1995.

[19] A. S. Zekri and S. G. Sedukhin. Computationally efficient parallel matrix-matrix multiplication on the torus. In *the 6th Int. Symp. on High Performance Computing*, Nara City, Japan, Sept. 2005.

[20] A. S. Zekri and S. G. Sedukhin. Matrix transpose on 2D torus. Technical Report 2005-1-001, The University of Aizu, Aizu Wakamatsu, Japan, Dec. 2005.