Technical Report 2011-001

Extremely Scalable GEMM-based Algorithms for 3-Dimensional Discrete Transforms in Torus Networks

Stanislav G. Sedukhin

August 26, 2011

Graduate School of Computer Science and Enginering
The University of Aizu
Tsuruga, Ikki-Machi, Aizu-Wakamatsu City
Fukushima, 965-8580 Japan

Technical Report 2011-001

Title:
Extremely Scalable GEMM-based Algorithms for 3-Dimensional Discrete Transforms in Torus Networks

Authors:
Stanislav G. Sedukhin

Key Words and Phrases:
3-dimensional discrete transforms, general matrix-matrix multiplication, 3-dimensional data decomposition, orbital matrix-matrix multiplication, torus networks, extremely scalable algorithms, algorithm/architecture co-design

Abstract:
The 3-dimensional (3D) forward/inverse separable discrete transforms, such as Fourier transform, cosine/sine transform, Hartley transform, and many others, are frequently the principal limiters that prevent many practical applications from scaling to the large number of processors. Existing approaches, which are based on a 1D or 2D data decomposition, do not allow the 3D transforms to be scaled to the maximum possible number of processors. Based on the newly proposed 3D decomposition of an $N \times N \times N$ initial data into $P \times P \times P$ blocks, where $P = N/b$ and $b \in [1, 2, ..., N]$ is a blocking factor, we design unified, highly scalable, GEMM-based algorithms for parallel implementation of any forward/inverse 3D transform on a $P \times P \times P$ network of toroidally interconnected nodes in $3P$ "compute-and-roll" time-steps, where each step is equal to the time of execution of b^4 fused multiply-add (`fma`) operations and movement of $\mathcal{O}(b^3)$ scalar data between nearest-neighbor nodes. The proposed 3D orbital algorithms can be extremely scaled to the maximum number of N^3 simple nodes (`fma`-units) which is equal to the size of data.

Report Date:
8/26/2011

Written Language:
English

Any Other Identifying Information of this Report:
Manuscript is submitted to the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA) November, 2011, Seattle, WA, USA

Distribution Statement:
First Issue: 10 copies

Supplementary Notes:

Distributed Parallel Processing Laboratory
The University of Aizu
Aizu-Wakamatsu
Fukushima 965-8580
Japan

Extremely Scalable GEMM-based Algorithms for 3-Dimensional Discrete Transforms in Torus Networks

Stanislav G. Sedukhin

Abstract

The 3-dimensional (3D) forward/inverse separable discrete transforms, such as Fourier transform, cosine/sine transform, Hartley transform, and many others, are frequently the principal limiters that prevent many practical applications from scaling to the large number of processors. Existing approaches, which are based on a 1D or 2D data decomposition, do not allow the 3D transforms to be scaled to the maximum possible number of processors. Based on the newly proposed 3D decomposition of an $N \times N \times N$ initial data into $P \times P \times P$ blocks, where $P = N/b$ and $b \in [1, 2, ..., N]$ is a blocking factor, we design unified, highly scalable, GEMM-based algorithms for parallel implementation of any forward/inverse 3D transform on a $P \times P \times P$ network of toroidally interconnected nodes in $3P$ "compute-and-roll" time-steps, where each step is equal to the time of execution of b^4 fused multiply-add (`fma`) operations and movement of $\mathcal{O}(b^3)$ scalar data between nearest-neighbor nodes. The proposed 3D orbital algorithms can be extremely scaled to the maximum number of N^3 simple nodes (`fma`-units) which is equal to the size of data.

1 Introduction

Three-dimensional (3D) discrete transforms (DT) such as Fourier Transform, Cosine/Sine Transform, Hartley Transform, Walsh-Hadamard Transform, etc., are known to play a fundamental role in many application areas such as spectral analysis, digital filtering, signal and image processing, data compression,

medical diagnostics, etc. Increasing demands for high speed in many real-world applications have stimulated the development of a number of Fast Transform (FT) algorithms, such as Fast Fourier Transform (FFT), with drastic reduction of arithmetic complexity [1]. These recursion-based FT-algorithms are deeply serialized by restricting data reuse almost entirely to take advantage from the sequential single-core processing.

A recent saturation of the performance of single-core processors, due to physical and technological limitations such as memory and power walls, demonstrates that a further sufficient increase of the speed of FT-algorithms is only possible by porting these algorithms into massively-parallel systems with many-core processors. However, by reason of complex and non-local data dependency between butterfly-connected operations, the existing deeply serialized FT-algorithms are not well adapted for the massively-parallel implementation. For example, it was recently reported in [2] that, by using two quad-core Intel Nehalem CPUs, the direct convolution approach outperforms the FFT-based approach on a $512 \times 512 \times 512$ data cube by at least five times, even when the associated arithmetic operation count is approximately two times higher. This result demonstrates that, because of the higher regularity and locality in the computation and data access (movement), the convolution achieves significantly better performance than FFT even with a higher arithmetic complexity. Because cost of arithmetic operations becomes more and more negligible with respect to the cost of data access, we envision that the conventional matrix-based DT-algorithms with simple, regular and local data movement would be more suitable for scalable massively-parallel implementation.

As three-dimensional, an $N \times N \times N$ discrete transforms are computationally intensive problems with respect to N they are often calculated on large, massively parallel networks of computer nodes, each of which includes at least one processor that operates on a local memory. The computation of a transform on a network having a distributed memory requires appropriate distribution of the data and work among the multiple nodes in the network as well as orchestrating of data movement during processing.

For parallel implementation of the 3D discrete transforms (at least for the very popular Fourier transform), there are two distinct approaches which are differ in the way of data decomposition over the physical network of computer

nodes. One approach is the 1D or "slab" decomposition of a 3D $N \times N \times N$ initial data which allows scaling (or distribution of work) among P nodes, where a 2D slab of size $N \times N \times (\frac{N}{P})$, $2 \leq P \leq N$, is assigned to each node. Different implementations of the 3D FFT with a "slab" decomposition can be found, for example, in [3], and [4]. Although this approach has relatively small communication cost, the scalability of the slab-based method or the maximum number of nodes is limited by the number of data elements along a single dimension of the 3D transform, i.e. $P_{\max} = N$.

Another approach is the 2D or "pencil" decomposition of a 3D $N \times N \times N$ initial data among a 2D array of $P \times P$ nodes, where a 1D "pencil" of size $N \times (\frac{N}{P}) \times (\frac{N}{P})$, $2 \leq P \leq N$, is assigned to each node [5], [6]. This approach overcomes the scaling limitation inherent into previous method since it increases the maximum number of nodes in the system from N to N^2. However, this increasing the number of nodes leads to rising of the communication cost.

It is important to note that in the both above mentioned, so-called "transpose" approaches, the computational and communication parts are separated. Moreover, a computational part is usually implemented inside each node of a network by using the 2D or 1D *fast* DXT algorithms for "slab"- or "pencil"-based decomposition, respectively, without any inter-node communication. However, after completion of each computational part, a transposition of the 3D matrix is required to put data in an appropriate dimension(s) into each node. That is, one or two 3D matrix transpositions would be needed for the 1D or 2D data decomposition approaches, respectively. Each of the 3D matrix transposition is implemented by "all-to-all" inter-node, message-passing communication. This *global* communication imposes an overhead which is proportional to the number of nodes and can be a dominant factor in the total time of processing for even small number of nodes [7].

The proposed in this paper a new, transpose-free approach for parallel implementation of the 3D discrete transforms increases even further a scalability in the maximum number of nodes from N^2 to the theoretical limit of N^3 by using, firstly, the 3D or "cubical" decomposition of an $N \times N \times N$ initial data and, secondly, fusion of a local, intra-node computation with a *nearest-neighbor* inter-node communication at each step of processing. The 3D transforms are represented as three chained sets of (general) matrix-matrix multiplications

(GEMM) which are executed in a 3D torus network by the fastest orbital algorithms. An algorithm is *extremely* scalable if it can be scaled to the maximum number of nodes which is limited only by the size of initial data, i.e. N^3.

The paper is organized as follows. In Section 2, the scalar and block notations of the 3D forward and inverse, separable transforms are described. Section 3 shortly discusses a systematic way for selection the fastest and extremely scalable matrix-matrix multiplication algorithms and its chaining. Section 4 introduces a 3D block data partition and proposes orbital, highly-scalable GEMM-based algorithms for the 3D forward and inverse transforms on a 3D network of toroidally interconnected nodes. The paper concludes in Section 5.

2 3D Separable Transforms

Let $X = [x(n_1, n_2, n_3)]$, $0 \leq n_1, n_2, n_3 < N$, be an $N \times N \times N$ volume of input data. A separable, *forward* 3D transform of X is another volume of an $N \times N \times N$ matrix $\dddot{X} = [\dddot{x}(k_1, k_2, k_3)]$ where for all $0 \leq k_1, k_2, k_3 < N$:

$$\dddot{x}(k_1, k_2, k_3) = \sum_{n_3=0}^{N-1} \sum_{n_2=0}^{N-1} \sum_{n_1=0}^{N-1} x(n_1, n_2, n_3) \cdot c(n_1, k_1) \cdot c(n_2, k_2) \cdot c(n_3, k_3) \quad (1)$$

In turn, a separable, *inverse* 3D transform is expressed as:

$$x(n_1, n_2, n_3) = \sum_{k_3=0}^{N-1} \sum_{k_2=0}^{N-1} \sum_{k_1=0}^{N-1} \dddot{x}(k_1, k_2, k_3) \cdot c(n_1, k_1) \cdot c(n_2, k_2) \cdot c(n_3, k_3) \quad (2)$$

where $0 \leq n_1, n_2, n_3 < N$ and $X = [x(n_1, n_2, n_3)]$ is an output $N \times N \times N$ matrix.

The separable transforms differ only by the transform coefficient matrix $C = [c(n_s, k_s)] = [c(n, k)], s = 1, 2, 3$, which can be

- *symmetric*, i.e. $C = C^T$, and *unitary*, i.e. $C^{-1} = C^{*T}$, C^* is a complex conjugate of C, like in the Discrete Fourier Transform (DFT), where $c(n, k) = \exp\left[-\frac{2\pi i}{N}\ (n \cdot k)\right] = \cos(\frac{2\pi nk}{N}) - i \sin(\frac{2\pi nk}{N})$ and $i = \sqrt{-1}$, or in the Discrete Hartley Transform (DHT), where $c(n, k) = \cos(\frac{2\pi nk}{N}) - \sin(\frac{2\pi nk}{N})$;

- unitary and *real*, i.e. *orthogonal*, like in the Discrete Cosine Transform (DCT), where coefficient $c(n,k) = \cos\left[\frac{\pi}{2N}(2n+1)\cdot k\right]$ and $C \neq C^T$;

- consists only ± 1 and be symmetric and orthogonal, like in the Discrete Walsh-Hadamard Transform (DWHT).

We will abbreviate the generic form of a separable transform as DXT without taking into account the specific features of a coefficient matrix, i.e. ignore the "fast" DXT algorithms.

The first step to formulate a 3D DXT in scalable form is to represent a given transform in block matrix notation such that a scalar form (1) or (2) would be a specific case only. To implement this block notation we divide an $N \times N \times N$ input data volume $X = [x(n_1, n_2, n_3)]$ into $P \times P \times P$ data cubes, where each cube $X(N_1, N_2, N_3)$, $0 \leq N_1, N_2, N_3 < P$, has the size of $b \times b \times b$, i.e. $b = N/P$ and $1 \leq b \leq N/2$. Then a block notation of the forward 3D DXT (3D FDXT) can be expressed as:

$$\dddot{X}(K_1, K_2, K_3) = \sum_{N_3=0}^{P-1} \sum_{N_2=0}^{P-1} \sum_{N_1=0}^{P-1} X(N_1, N_2, N_3) \times C(N_1, K_1) \times C(N_2, K_2) \times C(N_3, K_3), \tag{3}$$

where $0 \leq K_1, K_2, K_3 < N$ and $C = [C(N_i, K_i)] = [C(N, K)], i = 1, 2, 3$, is an (N_i, K_i)-th block of the transform matrix C.

It is clear that a 3D block inverse transform (3D IDXT) can be written as:

$$X(N_1, N_2, N_3) = \sum_{K_3=0}^{P-1} \sum_{K_2=0}^{P-1} \sum_{K_1=0}^{P-1} \dddot{X}(K_1, K_2, K_3) \times C(N_1, K_1) \times C(N_2, K_2) \times C(N_3, K_3), \tag{4}$$

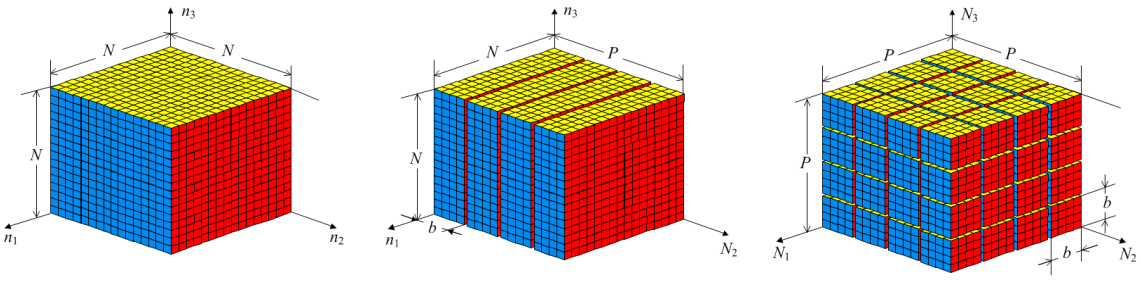

Figure 1: Partitioning of an $N \times N \times N$ initial data volume (left) into P slabs with the size of an $N \times b \times N$ each (middle) and $P \times P \times P$ cubes with the size of $b \times b \times b$ each (right) for parallel 3D DXT.

where $0 \leq N_1, N_2, N_3 < P$.

Now it is possible to formulate a 3D transform as conventional matrix-matrix multiplication. An initial $P \times P \times P$ data volume $X(N_1, N_2, N_3)$, $0 \leq N_1, N_2, N_3 < P$, is divided into P 1D slabs, for example, along N_2-axis, i.e. $X(N_1, N_2, N_3)$ can be referred as $X(N_1, N_3)_{N_2}$ for $N_2 \in [0, P)$ (see Fig. 1 for details). Then a 3D FDXT (3) can be computed in three data-dependent stages as chaining sets of matrix-matrix products:

1. **for all** *cubes* $0 \leq N_1, K_3 < P$ **at** *slabs* $N_2 \in [0, P)$ **do**

$$\dot{X}(N_1, K_3)_{N_2} = \sum_{N_3=0}^{P-1} X(N_1, N_3)_{N_2} \times C(N_3, K_3);$$

   This stage is implemented in a 4D index space
   $\mathcal{I}^{\mathrm{I}}_{4D} = \{(N_1, N_2, N_3, K_3) : 0 \leq N_1, N_2, N_3, K_3 < P\}$.

2. **for all** *cubes* $0 \leq K_1, K_3 < P$ **at** *slabs* $N_2 \in [0, P)$ **do**

$$\ddot{X}(K_1, K_3)_{N_2} = \sum_{N_1=0}^{P-1} C(N_1, K_1)^T \times \dot{X}(N_1, K_3)_{N_2};$$

   This stage is implemented in a 4D index space
   $\mathcal{I}^{\mathrm{II}}_{4D} = \{(N_1, N_2, K_3, K_1) : 0 \leq N_1, N_2, K_3, K_1 < P\}$.

3. **for all** *cubes* $0 \leq K_1, K_2 < P$ **at** *slabs* $K_3 \in [0, P)$ **do**

$$\dddot{X}(K_1, K_2)_{K_3} = \sum_{N_2=0}^{P-1} \ddot{X}(K_1, N_2)_{K_3} \times C(N_2, K_2);$$

   This stage is implemented in a 4D index space
   $\mathcal{I}^{\mathrm{III}}_{4D} = \{(K_1, N_2, K_3, K_2) : 0 \leq K_1, N_2, K_3, K_2 < P\}$.

The first two stages implement a classical 2D FDXT of a cubical matrix $\mathbf{X}$ in the form of triple matrix multiplication with transposition, $\ddot{X} = C^T \times X \times C$. Before the third stage, an 1D partition along N_2-axis is logically changed to an 1D partition along K_3-axis since $\ddot{X}(K_1, K_3)_{N_2} \equiv \ddot{X}(K_1, N_2)_{K_3}$, i.e. the same data cube simultaneously belongs to N_2- and K_3-slabs. Actually, a slab's sub-index is used only to represent a 3D DXT computing in the form of canonical (planar) matrix-matrix multiplication where the index agreement is required.

This slab's sub-index would be converted into corresponding main index when multiplication of a 3D matrix by a 2D matrix and vice-verse are utilized.

It is clear that a 3D IDXT (4) is implemented as rolling back of a 3D FDXT:

1. **for all** *cubes* $0 \le K_1, N_2 < P$ **at** *slabs* $K_3 \in [0, P)$ **do**

$$\ddot{X}(K_1, N_2)_{K_3} = \sum_{K_2=0}^{P-1} \ddot{\ddot{X}}(K_1, K_2)_{K_3} \times C(N_2, K_2)^T;$$

   This stage is implemented in a 4D index space
   $\mathcal{I}_{4D}^{\mathrm{III}} = \{(K_1, K_2, K_3, N_2) : 0 \le K_1, K_2, K_3, N_2 < P\}$.

2. **for all** *cubes* $0 \le N_1, K_3 < P$ **at** *slabs* $N_2 \in [0, P)$ **do**

$$\dot{X}(N_1, K_3)_{N_2} = \sum_{K_1=0}^{P-1} C(N_1, K_1) \times \ddot{X}(K_1, K_3)_{N_2};$$

   This stage is implemented in a 4D index space
   $\mathcal{I}_{4D}^{\mathrm{II}} = \{(K_1, N_2, K_3, N_1) : 0 \le K_1, N_2, K_3, N_1 < P\}$.

3. **for all** *cubes* $0 \le N_1, N_3 < P$ **at** *slabs* $N_2 \in [0, P)$ **do**

$$X(N_1, N_3)_{N_2} = \sum_{K_3=0}^{P-1} \dot{X}(N_1, K_3)_{N_2} \times C(N_3, K_3)^T.$$

   This stage is implemented in a 4D index space
   $\mathcal{I}_{4D}^{\mathrm{I}} = \{(N_1, N_2, K_3, N_3) : 0 \le N_1, N_2, K_3, N_3 < P\}$.

It is easy to verify that the total number of $P \times P$ block matrix-matrix multiplications in a forward or inverse 3D DXT is $3P^4$. Each $(b \times b) \times b$ block matrix-matrix multiplication requires an execution of $b^3 \cdot b = b^4$ scalar fused (indivisible) multiply-add (`fma`) operations, where $b = N/P$ is a blocking factor. The total number of such `fma`-operations for a 3D DXT is, therefore, $3P^4 \cdot b^4 = 3N^4$, i.e. blocking doesn't change an arithmetic complexity of the transformation. Obviously, it is because a 3D DXT is based completely on a matrix-matrix multiplication. At each stage of computing, the maximal degree of reusing of the N^3 data elements or an *arithmetic density* can be estimated as $N^4/N^3 = N$ multiply-adds/ (data element). Note that a 3D Fast DXT, like FFT, required an execution of the $\mathcal{O}(N^3 \log N)$ scalar arithmetic operations, at the best, i.e. an arithmetic density is only $\mathcal{O}(\log N)$. This low degree of data reusing is one of the reasons of low efficiency and low scalability of parallel implementations of the fast DXT algorithms.

3 Matrix Multiplication and 3D Transforms

From the above discussion we can conclude that in the matrix form a 3D FDXT can be expressed as

$$\dddot{X} = \overbrace{\underbrace{\Big(C^T \times \overbrace{\big(X \times C\big)}^{\text{1D FDXT: } N_3 \to K_3}\Big)}_{\text{2D FDXT: } N_1 \to K_1} \times C}^{\text{3D FDXT: } N_2 \to K_2},$$

and a 3D IDXT as

$$X = \overbrace{\underbrace{\Big(C \times \overbrace{\big(\dddot{X} \times C^T\big)}^{\text{1D IDXT: } K_2 \to N_2}\Big)}_{\text{2D IDXT: } K_1 \to N_1} \times C^T}^{\text{3D IDXT: } K_3 \to N_3}.$$

Note that this fixed order of parentetization directly corresponds to an 1D decomposition of the 3D initial data along N_2-axis which was selected above as one possible case.

From the forward and inverse 3D DXT equations it is clear that to implement a 3D transform under *unlimited* parallelism, i.e. when the maximum number of simultaneous operations is limited only by the size of data, the following should be found:

- The *fastest* matrix-matrix multiply-add (MMA) algorithm or algorithms with a maximal data reuse;
- These MMA algorithms should include the different forms of *transposition*, like in a General Matrix-matrix Multiplication (GEMM) from the Level-3 BLAS [8];
- A time-space coordinated *chain* of MMA multiplications for the quadruple matrix products;
- Mapping three, concatenated by data-dependency, 4D *computational* index spaces $\mathcal{I}^{\mathrm{I}}_{4D} \bigcup \mathcal{I}^{\mathrm{II}}_{4D} \bigcup \mathcal{I}^{\mathrm{III}}_{4D}$ into a single 3D physical *processor* space.

The solutions to all these problems can be found in our previous paper [9] where the fastest scalar MMA algorithms for the 2D DXT on a 2D torus array processor have been systematically designed and investigated. Moreover, it

was formally shown that for the following forms of the $n \times n$ matrix-matrix multiply-add:

$$C \leftarrow A \times B + C; \tag{5a}$$

$$A \leftarrow C \times B^T + A; \tag{5b}$$

$$B \leftarrow A^T \times C + B; \tag{5c}$$

there exist three classes of the fastest MMA algorithms which are implemented on an $n \times n$ mesh/torus array processor in n time-steps and which are differ in scheduling of the partially-ordered set of n^3 indivisible, scalar multiply-add operations in a 3D computational index space. These three classes are

- Broadcast-Broadcast-Compute (BBC) class which is based on the rank-1 updates [10], [11];
- Broadcast-Compute-Roll (BCR) class, which includes a few Fox-based algorithms [12];
- Compute-Roll-All (CRA) or orbital class, which includes a big variety of Cannon like algorithms [13].

Note that if transposition is required, a corresponding MMA operation (5b) or (5c) is implemented in a 3D computational index space without actual matrix transposition. Note also that MMA's computational index space is a 3D mesh for the BBC class, a 3D cylinder for the BCR class, and a 3D torus for the CRA class.

Different scheduling of the MMA computations imposes a different data reuse of the matrices A and B for updating n^2 elements of a matrix C [1]. That is, at each of n iterations or time-steps:

- an algorithm from the BBC class reuses (by replication, i.e. by broadcast) only n data elements (one column) of a matrix A and n data elements (one row) of a matrix B;
- algorithms from the BCR class reuse n data elements (diagonal) of a matrix A or B (by broadcast) and all n^2 elements of a matrix B or A (by rolling), respectively;

[1] This example is provided for the canonical form (5a), but other MMA forms can be equally considered.

- algorithms from the CRA class reuse all elements of matrices A and B (by rolling), i.e. $2n^2$ data elements.

Note that the well-known planar systolic MMA algorithms, which belong to the All-Shift-Compute (ASC) class, require $3n-2$ time-steps on a mesh array processor and, therefore, is not as fast as other discussed algorithms.

From the provided classification, the only Compute-Roll-All schedule defines at each time-step a circular movement of all $3n^2$ matrix elements without any data broadcast or replication. From the physical constrains it is clear that a global data broadcast or its alternative, data sharing, should be avoided in any extremely (unlimited) scalable system. The CRA class of the orbital MMA algorithms is selected for designing the scalable GEMM-based 3D transforms.

Due to cyclical nature of the orbital CRA-algorithms, the all rolled data elements are finally returned to the initial positions in a 3D torus index space. This unique for the orbital processing characteristic is effectively used for chaining of the different matrix products.

Below, a nontrivial extension of the scalar orbital MMA algorithms to the block MMA algorithms for the 3D DXT on a 3D array of toroidally interconnected nodes is introduced.

4 3D Data Partition and Scalable Computing

The proposed in this paper a new approach increases a scalability in the number of nodes from N^2 to the theoretical maximum of N^3 by using the 3D or "cubical" decomposition of an $N \times N \times N$ initial data and fusion of the computation with a local, *nearest-neighbor* communication at the each time-step. In our approach, an $N \times N \times N$ volume of initial data is, firstly, partitioned into the $P \times P \times P$ data cubes with the size of each is $(\frac{N}{P}) \times (\frac{N}{P}) \times (\frac{N}{P})$ and $2 \leq P \leq N$. Then each of P 1D "slab-of-cubes" is assigned not to a single node as in the "slab" decomposition above, but to the $P \times P$ array of toroidally interconnected nodes such that each data cube is assigned to the appropriate node. The P of such $P \times P$ torus arrays of nodes independently implement the set of P 2D DXTs in $2P$ block "compute-and-roll" time-steps in the same way as it was described in [9] for the scalar ($P = N$) case. For the block implementation, however, each computational step requires an execution of b^4

scalar, indivisible (fused), multiply-add operations, where $b = \frac{N}{P}$ is a blocking factor. Moreover, to provide a 3D data consistency with the remaining 1D DXT, all these P independent 2D DXTs should be appropriate skewed in space-time. The discussed below algorithms for the 3D DXTs are designed with this required skewing.

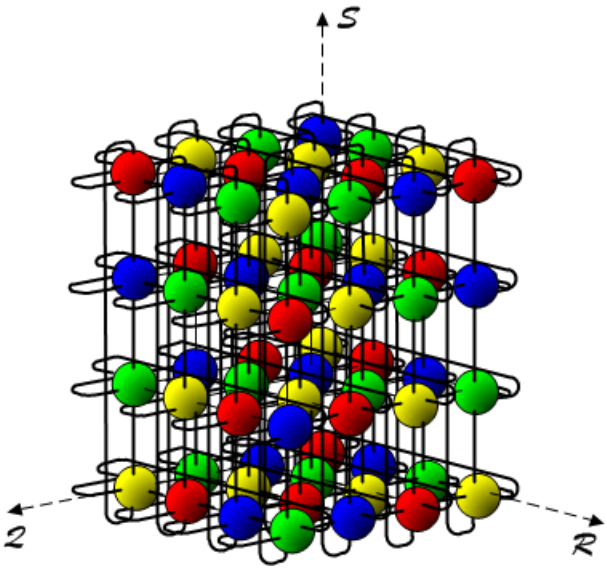

Figure 2: A $4 \times 4 \times 4$ array of computing nodes with toroidal interconnect in a 3D $(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ space.

4.1 3D Block Forward Transform

At the beginning, each computing node CN$(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ in a $P \times P \times P$ torus array (see Fig. 2 for $P = 4$) has in the local memory the four $b \times b \times b$ data cubes:

- $X = X(\mathcal{Q}, \mathcal{R}, \mathcal{S}) = X(N_1, N_2, N_3)$,
- $\dot{X} = \dot{X}(\mathcal{Q}, \mathcal{R}, \mathcal{T}) = \dot{X}(N_1, N_2, \mathcal{T}) = \emptyset$,
- $\ddot{X} = \ddot{X}(\mathcal{S}, \mathcal{R}, \mathcal{T}) = \ddot{X}(N_3, N_2, \mathcal{T}) = \emptyset$,
- $\dddot{X} = \dddot{X}(\mathcal{S}, \mathcal{Q}, \mathcal{T}) = \dddot{X}(N_3, N_1, \mathcal{T}) = \emptyset$,

as well as the three $b \times b$ matrices of transform coefficients:

- $C_{\mathrm{I}} = C(\mathcal{S}, \mathcal{T})$, $C_{\mathrm{II}} = C(\mathcal{Q}, \mathcal{S})$, and $C_{\mathrm{III}} = C(\mathcal{R}, \mathcal{Q})$,

where $\emptyset$ is a $b\times b\times b$ matrix of all zeros and $\mathcal{T}$ is one of the orbital or modular scheduling functions:

$$\mathcal{T} = (\alpha^T \cdot p) \bmod P \in (\pm\mathcal{Q} \pm \mathcal{R} \pm \mathcal{S}) \bmod P,$$

$\alpha = (\alpha_{\mathcal{R}}, \alpha_{\mathcal{Q}}, \alpha_{\mathcal{S}})^T = (\pm 1, \pm 1, \pm 1)^T$ is the scheduling vector and $p = (\mathcal{Q}, \mathcal{R}, \mathcal{S})^T$ is a node position in a 3D physical index space. Each computing node $\mathrm{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ in a torus system has six connection links labeled as $\pm\mathcal{Q}, \pm\mathcal{R}, \pm\mathcal{S}$. During processing a block of matrix data is rolled along or opposite corresponding axis (orbit) according to the scheduling vector α (for more details see [9]).

A proposed here three-stage orbital implementation of the 3D *forward* DXT on a 3-dimensional network of toroidally interconnected nodes $\{\mathrm{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S}) : 0 \le \mathcal{Q}, \mathcal{R}, \mathcal{S} < P\}$ under the scheduling function $\mathcal{T} = (\mathcal{Q} + \mathcal{R} + \mathcal{S}) \bmod P$, i.e. $\alpha = (\alpha_{\mathcal{Q}}, \alpha_{\mathcal{R}}, \alpha_{\mathcal{S}})^T = (1, 1, 1)^T$, is described below (see also Fig. 3 for $P = 2$ where red colored links show an accumulation or summation direction):

Stage I. $\dot{X}(\mathcal{Q}, \mathcal{R}, \mathcal{T}) = \sum_{0\le\mathcal{S}<P} X(\mathcal{Q}, \mathcal{R}, \mathcal{S}) \times C(\mathcal{S}, \mathcal{T})$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ **do** P times:
  1. **compute**: $\dot{X} \leftarrow X \times C_{\mathrm{I}} + \dot{X}$
  2. **data roll**: $\overset{+\mathcal{S};\alpha_{\mathcal{S}}=1}{\Longleftarrow} \dot{X} \overset{-\mathcal{S}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{Q};\alpha_{\mathcal{Q}}=1}{\Longleftarrow} C_{\mathrm{I}} \overset{-\mathcal{Q}}{\Longleftarrow}$

Stage II. $\ddot{X}(\mathcal{S}, \mathcal{R}, \mathcal{T}) = \sum_{0\le\mathcal{Q}<P} C(\mathcal{Q}, \mathcal{S})^T \times \dot{X}(\mathcal{Q}, \mathcal{R}, \mathcal{T})$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ **do** P times:
  1. **compute**: $\ddot{X} \leftarrow C_{\mathrm{II}}^T \times \dot{X} + \ddot{X}$
  2. **data roll**: $\overset{+\mathcal{Q};\alpha_{\mathcal{Q}}=1}{\Longleftarrow} \ddot{X} \overset{-\mathcal{Q}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{S};\alpha_{\mathcal{S}}=1}{\Longleftarrow} \dot{X} \overset{-\mathcal{S}}{\Longleftarrow}$

Stage III. $\dddot{X}(\mathcal{S}, \mathcal{Q}, \mathcal{T}) = \sum_{0\le\mathcal{R}<P} \ddot{X}(\mathcal{S}, \mathcal{R}, \mathcal{T}) \times C(\mathcal{R}, \mathcal{Q})$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ **do** P times:
  1. **compute**: $\dddot{X} \leftarrow \ddot{X} \times C_{\mathrm{III}} + \dddot{X}$
  2. **data roll**: $\overset{+\mathcal{R};\alpha_{\mathcal{R}}=1}{\Longleftarrow} \dddot{X} \overset{-\mathcal{R}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{Q};\alpha_{\mathcal{Q}}=1}{\Longleftarrow} \ddot{X} \overset{-\mathcal{Q}}{\Longleftarrow}$

Note that due to an orbital (cyclical) nature of processing, on completion of each stage, i.e. after every P "compute-and-roll" steps, all rotated data are

returned to the same nodes and, therefore, are ready for the next stage of processing. Note also that an initial cubical matrix X is assigned to the nodes in the canonical (in-order) layout whereas all intermediate and final matrices, $\dot{X}, \ddot{X}$ and $\dddot{X}$, will be in the skewed (out-of-order) layouts.

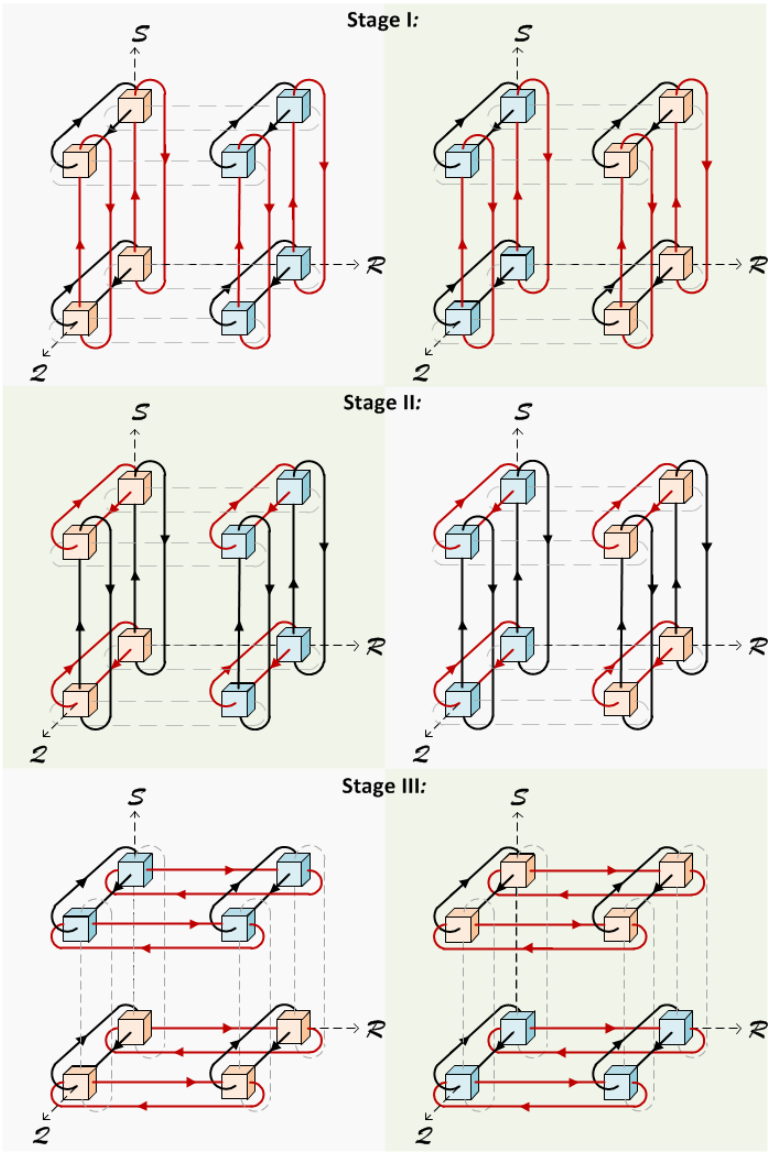

Figure 3: An orbital communication between nodes for the 3D FDXT.

4.2 3D Block Inverse Transform

It is clear that an orbital computing of the 3D *inverse* DXT can be implemented as rolling back of the above described process for a 3D forward DXT (see also Fig. 4 for $P = 2$) with the same initial data distribution among nodes:

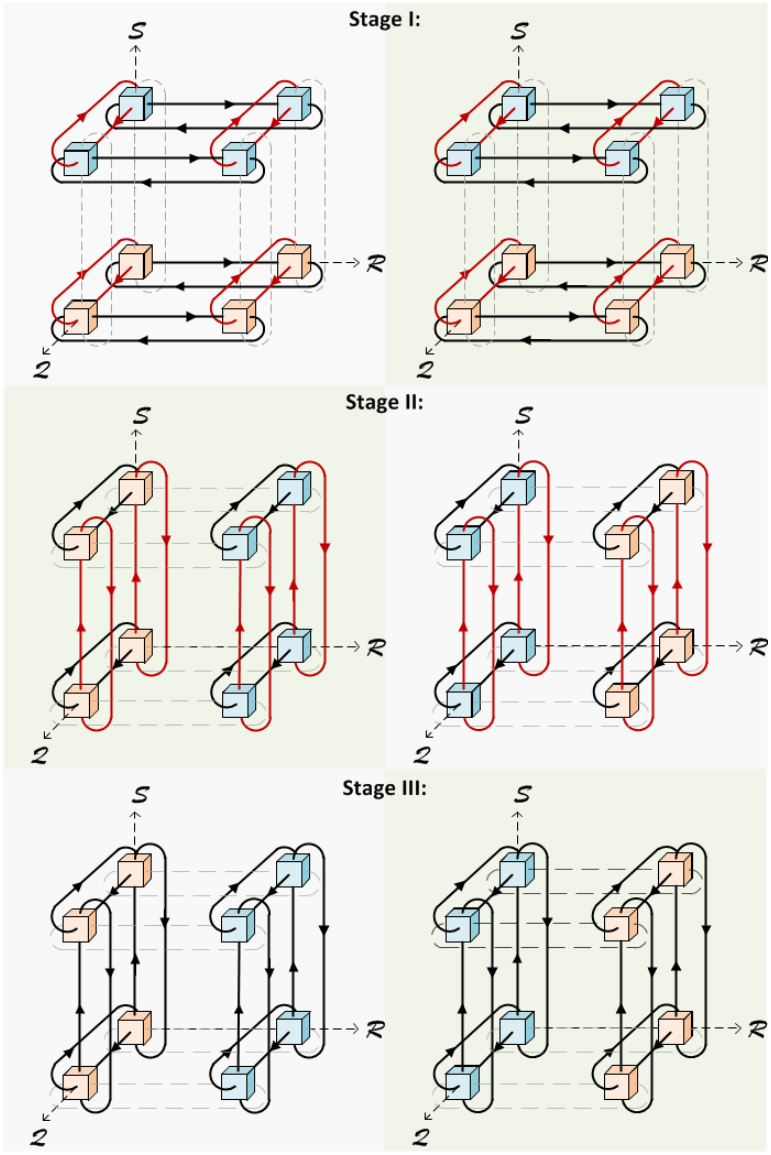

Figure 4: An orbital communication between nodes for the 3D IDXT.

Stage I. $\ddot{X}(\mathcal{S},\mathcal{R},\mathcal{T}) = \sum_{0\le\mathcal{Q}<P} \dddot{X}(\mathcal{S},\mathcal{Q},\mathcal{T}) \times C(\mathcal{R},\mathcal{Q})^T$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q},\mathcal{R},\mathcal{S})$ **do** P times:
  1. **compute**: $\ddot{X} \leftarrow \dddot{X} \times C_{\mathrm{III}}^T + \ddot{X}$
  2. **data roll**: $\overset{+\mathcal{R}}{\Longleftarrow} \dddot{X} \overset{-\mathcal{R}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{Q}}{\Longleftarrow} \ddot{X} \overset{-\mathcal{Q}}{\Longleftarrow}$

Stage II. $\dot{X}(\mathcal{Q},\mathcal{R},\mathcal{T}) = \sum_{0\le\mathcal{S}<P} C(\mathcal{Q},\mathcal{S}) \times \ddot{X}(\mathcal{S},\mathcal{R},\mathcal{T})$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q},\mathcal{R},\mathcal{S})$ **do** P times:
  1. **compute**: $\dot{X} \leftarrow C_{\mathrm{II}} \times \ddot{X} + \dot{X}$
  2. **data roll**: $\overset{+\mathcal{Q}}{\Longleftarrow} \ddot{X} \overset{-\mathcal{Q}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{S}}{\Longleftarrow} \dot{X} \overset{-\mathcal{S}}{\Longleftarrow}$

Stage III. $X(\mathcal{Q},\mathcal{R},\mathcal{S}) = \sum_{0\le\mathcal{T}<P} \dot{X}(\mathcal{Q},\mathcal{R},\mathcal{T}) \times C(\mathcal{S},\mathcal{T})^T$:

- **for all** P^3 $\mathrm{CN}(\mathcal{Q},\mathcal{R},\mathcal{S})$ **do** P times:

  1. **compute**: $X \leftarrow \dot{X} \times C_{\mathrm{I}}^T + X$
  2. **data roll**: $\overset{+\mathcal{S}}{\Longleftarrow} \dot{X} \overset{-\mathcal{S}}{\Longleftarrow} \quad \| \quad \overset{+\mathcal{Q}}{\Longleftarrow} C_{\mathrm{I}} \overset{-\mathcal{Q}}{\Longleftarrow}$

Note that at the last stage of the 3D IDXT, an accumulation or $\mathcal{T}$-summation is implemented inside each computing node. Note also that an initial 3D matrix $\ddot{X}$ is defined here in a skewed layout as unchanged result of a 3D forward transform, but on completion, a resulting 3D matrix X will be in the canonical (in-order) layout. A frequently required by many real-world applications looping: 3D FDXT $\rightleftharpoons$ 3D IDXT, is a very natural in this orbital implementation.

4.3 Complexity and Scalability Analysis

An orbital implementation of the forward or inverse block $N \times N \times N$ 3D DXT on a $P \times P \times P$ torus array of nodes requires totally $3P$ "compute-and-roll" time-steps, where $P = N/b$ and $1 \le b \le N$ is a blocking factor. Each "compute-and-roll" time-step involves the left or right multiplication of a 3D $b \times b \times b$ matrix by a $b \times b$ coefficient matrix with a possible transpose. In turn, each of this matrix-matrix multiplication requires execution by every node exactly b^4 scalar, fused multiply-add (`fma`) operations and movement (rolling) of either $b^3 + b^2$ or $2b^3$ scalar data between nearest-neighbor nodes. The required memory space in one node, which is proportionally to $4b^3 + 3b^2$, might be reduced to $2b^3 + 3b^2$ by using the well-known "compute-in-place" approach.

Note that we have provided our solution under the assumption that there is no direct memory access (DMA) between nearest-neighbor nodes, i.e. local data must be exchanged (rolled) by using the message-passing. However, if upgrading of a data (a "compute" part) precedes a movement of this data (a "roll" part) and DMA is allowed then data upgrading can be directly done in a shared between nearest-neighbor nodes memory, which will eliminate a "roll" part of this data. For the other unchangeable, but moved data, "roll" part(s) can be implemented concurrently with a "compute" part (see description of

the DXT algorithms above), making a data movement totally overlapped with a computing.

If blocking factor $b = 1$, i.e. $P = N$, our 3D transform algorithms have the highest degree of parallelization with a total processing time of $3N$ "compute-and-roll" steps on an $N \times N \times N$ torus array of simple nodes. This fastest, fine-grained implementation will require the `fma`-unit in each node to be with one-step or one-cycle latency which is possible for fixed-point arithmetic, like in Anton computer [14]. For floating-point arithmetic, the pipelined `fma`-units in today advanced microprocessors have a few cycles of latency [15, 16, 17, 18, 19, 20, 21, 22, 23, 24], which forces the size of blocking factor b to be more then one to hide this pipeline latency by concurrent execution of a few *independent* `fma`-instructions on the same `fma`-unit (usually, $b \geq 4$).

On the other hand, if $b = N$, i.e. $P = 1$, the 3D transform is represented as a serial algorithm which is executed on a sequential computer in $3N^4$ `fma`-steps. It is clear that independently on the size of a blocking factor $b \in [1, 2, ..., N]$ and, therefore, degree of parallelization, any implementation of the 3D transform requires execution of $3N^4$ `fma`-operations. The degree of the data reuse is $P^4/P^3 = P$, i.e. the maximal data reuse will be when $b = 1$ and, therefore, $P = N$.

Depending on the node's architecture, the time and space optimal parallel MMA algorithms for the *intra-node* implementation may be not from the CRA-class, i.e. orbital, but from the BBC-class, i.e. broadcast-based. Due to the minimal amount of reusing data (see discussion above), broadcast or rank-1 update MMA algorithms are the fastest and most efficient ones for computers with a hierarchical memory organization, e.g. for multi-core CPUs with a shared memory and relatively small number of `fma`-units [25],[26],[20] as well as for GPUs with many-hundred of `fma`-units, hierarchically interconnected by shared cache memories [27], [28], [29], [30]. It is clear that here, a shared between multiple `fma`-units memory replaces the required by BBC algorithm data broadcast ("one-to-all") by a shared access to this data ("all-to-one"). This replacement, however, is technically justified only when the number of these "all" is relatively small. Moreover, a deep memory hierarchy leads to the multiple data replication which increases the size of a problem to be effective solved on this computer. Currently, a parallel, cache-based MMA

implementation on the advanced GPUs is only effective (around 90% of peak performance) for the relatively big matrices with the size of $\mathcal{O}(10^3)$. It is obvious that a cache-coherent, hierarchical approach can not be extremely scaled to unlimited parallelism.

It is important to note that length of the existing wrap-around toroidal connections, which is proportional to the torus size (see Fig. 2) and, therefore, might be a physical obstacle for the fast data movement, can be equalized (localized) with a length of the internal (nearest-neighbor) connections by using double (for a 2D torus) or triple (for a 3D torus) folding [31]. This well-known technique, however, destroys an integrity (locality and regularity) of multidimensional data which are stored in an unfolded, mesh-like torus. Recall that a cubical matrix data for a 3D DXT is stored initially in a canonical (in-order) layout which will be destroyed after folding. To make torus interconnect to be regular, modular, and, therefore, scalable, like a 2D or 3D mesh, the proposed recently adaptable "mesh-of-tori" interconnect [32] and corresponding folding/unfolding data manipulation can be used for effective local implementation of the discussed multidimensional transforms (see Fig. 5 as an example of a 2D "mesh-of-tori" which can be easily extended to the 3D case).

5 Conclusions

We have introduced massively-parallel, efficient, and extremely-scalable orbital block algorithms to perform any 3D separable transform and its inverse in a 3D network of toroidally interconnected nodes. These proposed algorithms preserve all advantages of the previously popular systolic processing such as simplicity, regularity, nearest neighbor data movement, scalable parallelism, pipelining, etc. However, unlike systolic processing, an orbital processing keeps all initial, intermediate, and final data inside a 3D torus array processor. Moreover, because our orbital GEMM-based algorithms are logically represented and scheduled in a 3D torus computational index space, these algorithms are naturally (one-to-one) mapped into a 3D network of toroidally interconnected nodes.

A multidimensional torus interconnect has always been widely used in the past and present supercomputers, for example, in Cray T3E (3D torus) [33],

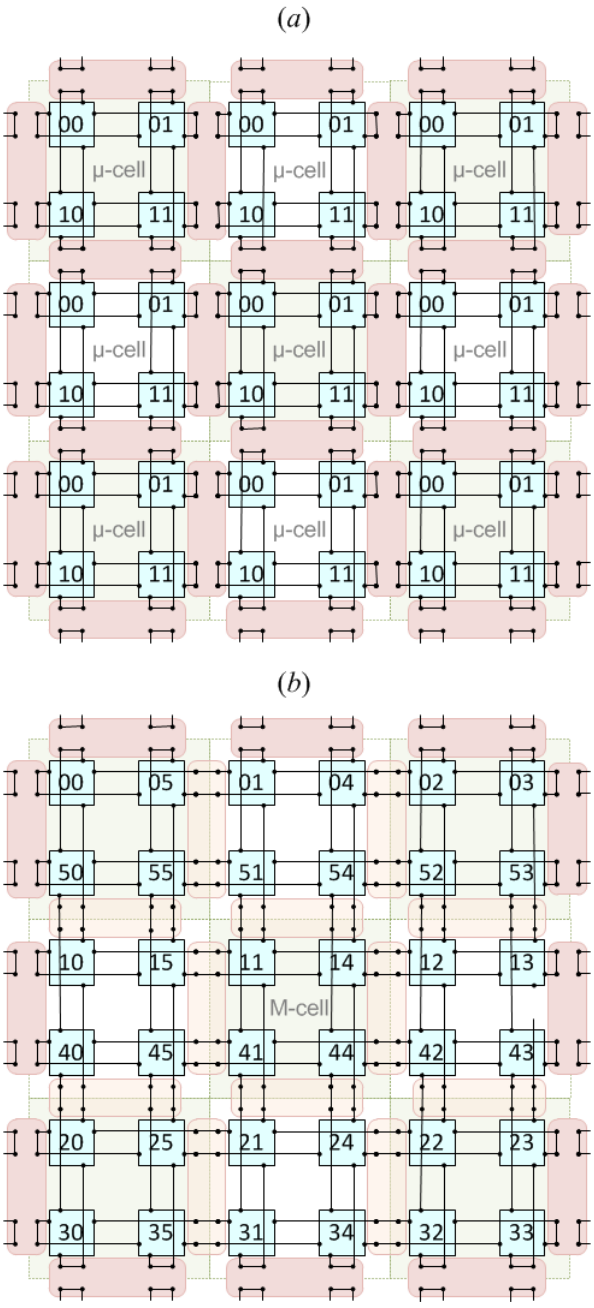

Figure 5: A 2D mesh of 3×3 μ-cells isolated by membranes, where each μ-cell is a smallest 2×2 torus (a); a single macro-cell as 6×6 double-folded torus which was formed by *fusion* of 3×3 μ-cells. Cell *division* is an opposite process.

IBM Blue Gene/L (3D torus) [34] and Blue Gene/Q (5D torus), Fujitsu K-computer (6D torus) [35], [36]. We expect that direct implementation of our coarse-grained 3D DXT block algorithms ($b > 1$) on supercomputers with a 3D torus interconnect would be beneficial with respect to other existing implementations with a 1D or 2D data decomposition and we reserve this porting and analysis as a future work.

On the other hand, our extremely scalable solution ($b = 1$) for a fine-grained 3D DXT implementation on a 3D "mesh-of-tori" of very simple processors (`fma`-units) is perfectly suited for forthcoming 3D micro/macro-electronics technologies [37], [38] and real-world applications with multidimensional data. Note that the preliminary results of FPGA implementation of a 3D torus array processor for the fine- and coarse-grained 3D Discrete Cosine Transform (3D DCT) has recently been reported [39].

One of the notable characteristics of our unified 3D DXT algorithms is in the possibility to implement concurrently a few distinct transforms on the same cubical data by using differently defined matrices of transform coefficients. Moreover, if a new discrete separable transform is proposed, it can be easily tested, verified, and compared with other existing transforms.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[2] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, pp. 41–49, March 2011. [Online]. Available: http://dx.doi.org/10.1109/MM.2011.17

[3] S. Filippone, "The IBM parallel engineering and scientific subroutine library," in *PARA*, ser. Lecture Notes in Computer Science, J. Dongarra, K. Madsen, and J. Wasniewski, Eds., vol. 1041. Springer, 1995, pp. 199–206.

[4] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, "A Volumetric FFT for BlueGene/L," in *HiPC*, ser. Lecture Notes in Computer Science, T. M. Pinkston and V. K. Prasanna, Eds., vol. 2913. Springer, 2003, pp. 194–203.

[6] D. Pekurovsky, "P3DFFT," *Website*, August, 2011. [Online]. Available: http://www.sdsc.edu/us/resources/p3dfft/

[7] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain, "Performance measurements of the 3D FFT on the Blue Gene/L supercomputer," in *Euro-Par*, ser. Lecture Notes in Computer Science, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Springer, 2005, pp. 795–803.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, pp. 1–17, March 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170

[9] S. G. Sedukhin, A. S. Zekri, and T. Myiazaki, "Orbital algorithms and unified array processor for computing 2D separable transforms," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 127–134, 2010. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.29

[10] 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.*, vol. 38, no. 6, pp. 673–681, 1994.

[11] R. van de Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," The University of Texas, Tech. Rep. TR-95-13, Apr. 1995.

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

[13] L. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[14] C. Young, J. A. Bank, R. O. Dror, J. P. Grossman, J. K. Salmon, and D. E. Shaw, "A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 23:1–23:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654083

[15] E. Quinnell, E. Swartzlander, and C. Lemonds, "Floating-point fused multiply-add architectures," in *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, nov. 2007, pp. 331 –337. [Online]. Available: http://dx.doi.org/10.1109/ACSSC.2007.4487224

[16] V. Kreinovich, "Itanium's new basic operation of fused multiply-add: theoretical explanation and theoretical challenge," *SIGACT News*, vol. 32, pp. 115–117, March 2001. [Online]. Available: http://doi.acm.org/10.1145/568438.568461

[17] F. G. Gustavson, J. E. Moreira, and R. F. Enenkel, "The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to java and high-performance computing," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. IBM Press, 1999, pp. 4–. [Online]. Available: http://portal.acm.org/citation.cfm?id=781995.781999

[18] M. Cornea, J. Harrison, and P. T. P. Tang, "Intel Itanium floating-point architecture," in *Proceedings of the 2003 workshop on Computer architecture education: Held in conjunction with the 30th International Symposium on Computer Architecture*, ser. WCAE '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/1275521.1275526

[19] S. Chatterjee, L. R. Bachega, P. Bergner, K. A. Dockser, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. Mendell,

R. Nair, C. D. Wait, T. J. C. Ward, and P. Wu, "Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 377 –391, march 2005.

[20] H.-J. Oh, S. Mueller, C. Jacobi, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong, "A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 4, pp. 759 – 771, april 2006.

[21] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong, "The vector floating-point unit in a synergistic processor element of a CELL processor," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, june 2005, pp. 59 – 67.

[22] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM J. Res. Dev.*, vol. 34, pp. 59–70, January 1990. [Online]. Available: http://dx.doi.org/10.1147/rd.341.0059

[23] S. Vangal, Y. Hoskote, N. Borkar, and A. Alvandpour, "A 6.2-GFlops floating-point multiply-accumulator with conditional normalization," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 10, pp. 2314 –2323, oct. 2006.

[24] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 913 –922, july 2011. [Online]. Available: http://dx.doi.org/10.1109/TC.2010.121

[25] D. Hackenberg, "Fast Matrix Multiplication on Cell (SMP) Systems," *Website*, August, 2011. [Online]. Available: http://www.tu-dresden.de/zih/cell/matmul

[26] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, pp. 12:1–12:25, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053

[27] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '04. New York, NY, USA: ACM, 2004, pp. 133–137. [Online]. Available: http://doi.acm.org/10.1145/1058129.1058148

[28] V. Allada, T. Benjegerdes, and B. Bode, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –9.

[29] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413370.1413402

[30] K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, and S. G. Sedukhin, "Multi-level optimization of matrix multiplication for GPU-equipped systems," *Procedia CS*, vol. 4, pp. 342–351, 2011.

[31] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[32] A. A. Ravankar and S. G. Sedukhin, "Mesh-of-tori: A novel interconnection network for frontal plane cellular processors," in *Proceedings of the 2010 First International Conference on Networking and Computing*, ser. ICNC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 281–284. [Online]. Available: http://dx.doi.org/10.1109/IC-NC.2010.30

[33] S. Scott and G. Thorson, "The Cray T3E network: Adaptive routing in a high performance 3D torus," in *Proceedings of HOT Interconnects IV*, 1996, pp. 147–156.

[34] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, "Blue Gene/L torus interconnection network," *IBM J. Res. Dev.*, vol. 49, pp. 265–276, March 2005. [Online]. Available: http://dx.doi.org/10.1147/rd.492.0265

[35] FUJITSU, "K computer," *Website*, August, 2011. [Online]. Available: http://www.fujitsu.com/global/about/tech/k/

[36] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers," *Computer*, vol. 42, pp. 36–40, 2009.

[37] M. Koyanagi, T. Fukushima, and T. Tanaka, "Three-dimensional integration technology and integrated systems," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, Jan. 2009, pp. 409–415.

[38] R. H. Reuss and at al., "Macroelectronics: perspectives on technology and applications," *Proceedings of the IEEE*, vol. 93, no. 7, pp. 1239–1256, 2005.

[39] Y. Ikegaki, T. Miyazaki, and S. G. Sedukhin, "3D-DCT Processor and its FPGA Implementation," *IEICE Transactions on Information and Systems*, vol. E94.D, no. 7, pp. 1409–1418, 2011. [Online]. Available: http://dx.doi.org/10.1587/transinf.E94.D.1409