Technical Report 2012-001

Co-design of Extremely Scalable Algorithms/Architecture for 3-Dimensional Linear Transforms

Stanislav G. Sedukhin

July 2, 2012

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

Technical Report 2012-001

Title:

Co-design of the Extremely Scalable Algorithms/Architecture for 3-Dimensional Linear Transforms

Authors:

Stanislav G. Sedukhin

Key Words and Phrases:

3-dimensional linear transforms, multilinear tensor-by-matrix multiplication, computational index space, cubical data decomposition, modular scheduling, data reusing, torus interconnect, 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 1D or 2D data decomposition, do not allow the 3D transforms to be scaled to the maximum possible number of computer nodes. 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 the blocking factor, we systematically design unified, highly scalable algorithms for parallel implementation of any forward/inverse 3D transform on the one-to-one correspondent $P \times P \times P$ torus network of computer nodes. All designed algorithms require $3P$ "compute-and-roll" time-steps, where each step is equal to the time of execution in each node b^4 fused multiply-add (`fma`) operations and concurrent movement of $\mathcal{O}(b^3)$ scalar data between nearest-neighbor nodes. The proposed 3D orbital algorithms gracefully avoid a required 3D data transposition and can be extremely scaled up to the maximum number of N^3 simple nodes (`fma`-units) which is equal to the size of initial data.

Report Date:

7/2/2012

Written Language:

English

Any Other Identifying Information of this Report:

Revised and extended version of the Technical Report 2011-001, the University of Aizu, 2011, 24 p.

Distribution Statement:

First Issue: 10 copies

Supplementary Notes:

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

Co-design of Extremely Scalable Algorithms/Architecture for 3-Dimensional Linear Transforms

Stanislav G. Sedukhin

Distributed Parallel Processing Lab., The University of Aizu,
Aizuwakamatsu City, Fukushima 965-8580, Japan
e-mail: sedukhin@u-aizu.ac.jp

July 2, 2012

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 1D or 2D data decomposition, do not allow the 3D transforms to be scaled to the maximum possible number of computer nodes. 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 the blocking factor, we systematically design unified, highly scalable algorithms for parallel implementation of any forward/inverse 3D transform on the one-to-one correspondent $P \times P \times P$ torus network of computer nodes. All designed algorithms require $3P$ "compute-and-roll" time-steps, where each step is equal to the time of execution in each node b^4 fused multiply-add (`fma`) operations and concurrent movement of $\mathcal{O}(b^3)$ scalar data between nearest-neighbor nodes. The proposed 3D orbital algorithms gracefully avoid a required 3D data transposition and can be extremely scaled up to the maximum number of N^3 simple nodes (`fma`-units) which is equal to the size of initial 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 dramatic reduction of arithmetic complexity [9]. These recursion-based FT-algorithms are deeply serialized by restricting data reuse almost entirely to take advantage from the sequential 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 [34] 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 or data movement, the conventional matrix-based DT-algorithms with a simple, regular and local data movement are expected to be more suitable for scalable massively-parallel implementation. As three-dimensional, $N \times N \times N$ 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 $n_p = P = N/b$ nodes, where a 2D slab of size $N \times N \times b$, is assigned to each node (see Fig. 1 (*a*)). Here and below, $b \in \{1, 2, ..., N\}$ is the blocking factor. Different implementations of the 3D FFT with a "slab" decomposition can be found, for example, in [14, 16, 17]. Although this approach has relatively small inter-node 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. $n_p^{\max} = N$ when $b = 1$.

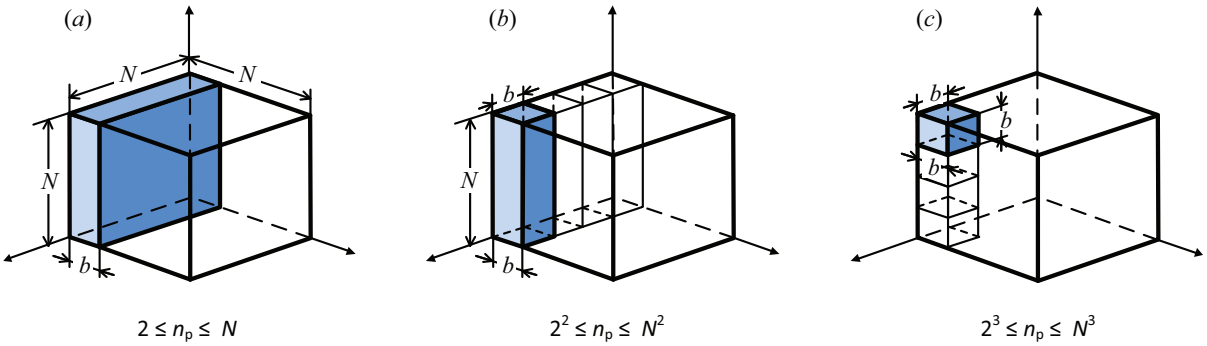

Figure 1: 3D data distribution over n_p computer nodes: (a) 1D or "slab' decomposition, (b) 2D or "pencil" decomposition, and (c) 3D or "cube" decomposition.

Another approach, shown in Fig. 1 (*b*), is the 2D or "pencil" decomposition of a 3D $N \times N \times N$ initial data among a 2D array of $n_p = P \times P$ nodes $(P = N/b)$ where a 1D "pencil" or "rod" of size $N \times b \times b$, $b \in \{1, 2, ..., N\}$, is assigned to each node. Parallel 3D FFT implementations with a 2D data decomposition are discussed in [13, 40, 32]. 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. As a consequence, each node requires less memory than in a "slab"-based approach. However, this increasing the number of nodes leads to rising of the communication cost.

It is important to note that in both of these so-called "transpose" approaches, the computational part and inter-node communication part are separated [6]. Moreover, a computational part on an assigned data is implemented inside each node of a network by using either 2D or 1D *fast* (*recursive*) algorithms for "slab"- or "pencil"-based decomposition, respectively, without any inter-node communication. However, on completion of each computational part, in order to support contiguity of memory access, a transposition of the 3D data array is required to put data in an appropriate dimension(s) into each node. At least one or

two transpositions would be needed for the 1D or 2D data decomposition approaches, respectively. Each of this 3D data transposition is implemented by "all-to-all" inter-node, message-passing communication. This *global* data exchange 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 [12]. As it was recently reported [41], by using a 2D data decomposition approach, the communication part of a 512^3 FFT on the BlueGeene/P computer with up to $512 \times 512 = 262,144$ cores (processors) may require more than 95% of the runtime.

The proposed in this paper a new, transpose-free approach for parallel implementation of the 3-dimensional discrete transforms improves even further a (strong) scalability of the 3D transforms by increasing the maximum number of computer nodes from N^2 to the extreme number of N^3. It becomes possible due to

1. the 3D or "*cubic*" decomposition of an $N \times N \times N$ initial data among $n_p = P \times P \times P$ computer nodes where a 3D data "cube" of size $b \times b \times b$ is assigned to each node (see Fig. 1 (*c*));
2. computing of the basic one-dimensional N-size transform not on a single, but on the $P = N/b$ cyclically interconnected (for data reuse) nodes of a 3D torus network by employing not recursive, but *blocked* matrix multiplication based algorithms with a well-structured data access/movement at the expense of increasing the number of arithmetic operations;
3. integration of a local, intra-node computation with a *nearest-neighbor* inter-node communication at each step of 3-dimensional processing.

A 3D transform is represented as three chained sets of the cubical tensor-by-matrix or matrix-by-tensor multiplications which are executed in a 3D torus network of computer nodes by the fastest and extremely scalable orbital algorithms. We call an algorithm as *extremely* scalable if it can be scaled to the maximum number of computer nodes which is limited only by the size of initial data, i.e. in our case, N^3. In other words, the number of simultaneously executed scalar (basic for a given algorithm) operations may be equal to the amount of initial data. Of course, exascale does not mean exaflop.

The paper is organized as follows. In Section 2, the scalar and block notations of the 3D forward and inverse, separable transforms are described. It is shown that these notations are based on the multilinear matrix multiplication. 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 of the three-way tensor and proposes orbital, highly-scalable algorithms for parallel implementation of the 3D forward and inverse transforms on a 3D network of toroidally interconnected computer nodes. The paper concludes in Section 5.

2 3D Separable Transforms

Let $\mathbb{X} = [x(n_1, n_2, n_3)]$, $0 \le n_1, n_2, n_3 < N$, be an $N \times N \times N$ cubical grid of input data or three-way data tensor [24]. A separable *forward* 3D transform of $\mathbb{X}$ is another cubical grid of an $N \times N \times N$ data or three-way tensor $\dddot{\mathbb{X}} = [\dddot{x}(k_1, k_2, k_3)]$ where for all $0 \le k_1, k_2, k_3 < N$:

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

In turn, a separable *inverse* or backward 3D transform of three-way tensor $\dddot{\mathbb{X}} = [\dddot{x}(k_1, k_2, k_3)]$ is expressed as:

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

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

There is a direct correspondence between the equations (1), (2) and the so-called *symmetric multilinear matrix multiplication*, which is used to represents three-way data tensor $\mathbb{X}$ or $\ddot{\mathbb{X}}$ in different bases and where an $N \times N$ matrix $C = [c(n_s, k_s)] = [c(n, k)], s = 1, 2, 3$ is a (non-singular) change-of-basis matrix (see [24, 33, 22] for more details). It is also interesting to mention that equations (1) and (2) can be viewed as the so called Tuker's 3D tensor decomposition which is represented in the form of three-way tensor-by-matrix multiplication [24, 30]. Moreover, the equation (1) or (2) represents the so-called three-way *tensor contraction* which is widely used in *ab initio* quantum chemistry models [5, 35, 47].

The separable transforms differ only by the transform coefficient (change-of-basis) matrix $C = [c(n, k)]$ 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 discrete transform as DXT without taking into account the specific features of a coefficient matrix, i.e. we will use direct algorithms for the 3D transforms (1) and (2) with an arithmetic complexity of $\mathcal{O}(N^4)$ instead of using the so-called "fast" DXT algorithms, like 3D FFT, with $\mathcal{O}(N^3 \log N)$ complexity.

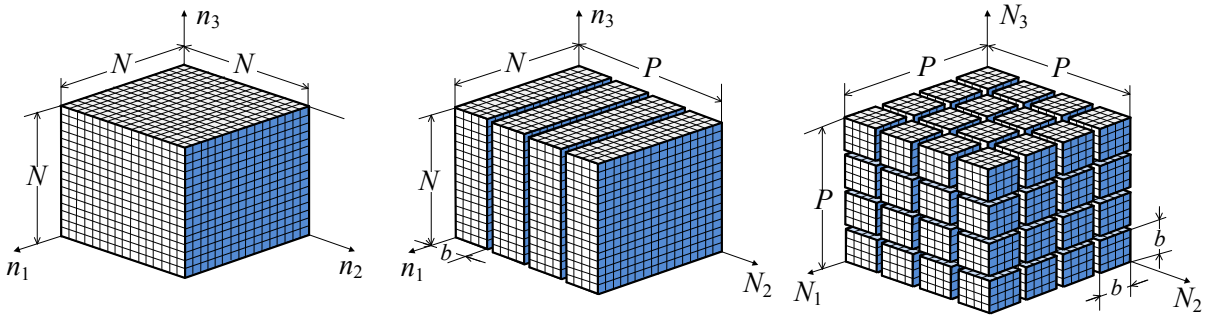

Figure 2: Partitioning of an $N \times N \times N$ initial data (a) into P slabs along N_2-axis with the size of an $N \times b \times N$ each (b) and $P \times P \times P$ cubes with the size of $b \times b \times b$ each (c).

The first step to design a scalable 3D DXT is to represent it in a block matrix notation where the conventional scalar form (1) or (2) is the only one extreme case of a block notation. To formulate our problem in this block notation we firstly divide an $N \times N \times N$ input data volume or three-way tensor $\mathbb{X} = [x(n_1, n_2, n_3)]$ into $P \times P \times P$ data cubes, where each cube $\mathbb{X}(N_1, N_2, N_3)$, $0 \leq N_1, N_2, N_3 < P$, has the size of $b \times b \times b$, i.e. $b = N/P$ and $1 \leq b \leq N/2$ is the blocking factor. Then the forward 3D DXT

(3D FDXT) can be expressed as a block version of the multilinear matrix multiplication:

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

where $0 \leq K_1, K_2, K_3 < N$ and $C(N_s, K_s) = C(N, K), s = 1, 2, 3$, is an (N_s, K_s)-th block of the transform matrix C.

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

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

where $0 \leq N_1, N_2, N_3 < P$. As it can be seen from (3) and (4), both 3D transforms are implemented in a 6D computational index space $\mathcal{I}_{6D} = \{(N_1, N_2, N_3, K_1, K_2, K_3) : 0 \leq N_1, N_2, N_3, K_1, K_2, K_3 < P\}$ with the number of index points $\|\mathcal{I}_{6D}\| = P^6$. Each index point in this 6D space is associated with three cubical tensor-by-matrix multiplications.

Due to separability of the linear transforms, the dimension of a computational index space can be reduced from six to four by splitting a 3D transform into three data dependent sets of 1D transforms as it is shown below for the 3D FDXT (3).

At the *first stage*, the $P \times P$ 1D FDXT of $\mathbb{X}(N_1, N_2, :)$ are performed for all (N_1, N_2) pairs, $0 \leq N_1, N_2 < P$, as block cubical *tensor-by-matrix multiplication*:

$$\dot{\mathbb{X}}(N_1, N_2, K_3) = \sum_{N_3=0}^{P-1} \mathbb{X}(N_1, N_2, N_3) \times C(N_3, K_3), \tag{5}$$

where $0 \leq N_1, N_2, K_3 < N$. It is clear from (5) that this stage is implemented in a 4D computational index space

$$\mathcal{I}_{4D}^{\mathrm{I}} = \{(N_1, N_2, N_3, K_3) : 0 \leq N_1, N_2, N_3, K_3 < P\}$$

with an embedded 3D input data tensor $\{\mathbb{X}(N_1, N_2, N_3), 0 \leq N_1, N_2, N_3 < P\}$ as one out of eight 3D faces of 4D index space $\mathcal{I}_{4D}^{\mathrm{I}}$. This cubical tensor is shown in red color in Fig. $3(a)$. For simplicity, a graphical example here is provided for $P = 2$, i.e. blocking factor $b = N/2$.

At the *second stage*, the $P \times P$ 1D FDXT of $\dot{\mathbb{X}}(:, N_2, K_3)$ are implemented for all (N_2, K_3) pairs, $0 \leq N_2, K_3 < P$, as second block tensor-by-matrix multiplication:

$$\ddot{\mathbb{X}}(K_1, N_2, K_3) = \sum_{N_1=0}^{P-1} \dot{\mathbb{X}}(N_1, N_2, K_3) \times C(N_1, K_1), \tag{6}$$

where $0 \leq K_1, N_2, K_3 < N$. This stage is also implemented in a 4D index space

$$\mathcal{I}_{4D}^{\mathrm{II}} = \{(N_1, N_2, K_3, K_1) : 0 \leq N_1, N_2, K_3, K_1 < P\}$$

with a common to the previous stage 3D face $\mathcal{I}_{3D}^{\mathrm{I/II}} = \mathcal{I}_{4D}^{\mathrm{I}} \bigcap \mathcal{I}_{4D}^{\mathrm{II}} = \{(N_1, N_2, K_3) : 0 \leq N_1, N_2, K_3 < P\}$ which is shown in yellow in Fig. $3(b)$.

At the *third stage*, the $P \times P$ 1D FDXT of $\ddot{\mathbb{X}}(K_1, :, K_3)$ are implemented for all (K_1, K_3) pairs, $0 \leq K_1, K_3 < P$, as third block tensor-by-matrix multiplication:

$$\dddot{\mathbb{X}}(K_1, K_2, K_3) = \sum_{N_2=0}^{P-1} \ddot{\mathbb{X}}(K_1, N_2, K_3) \times C(N_2, K_2), \tag{7}$$

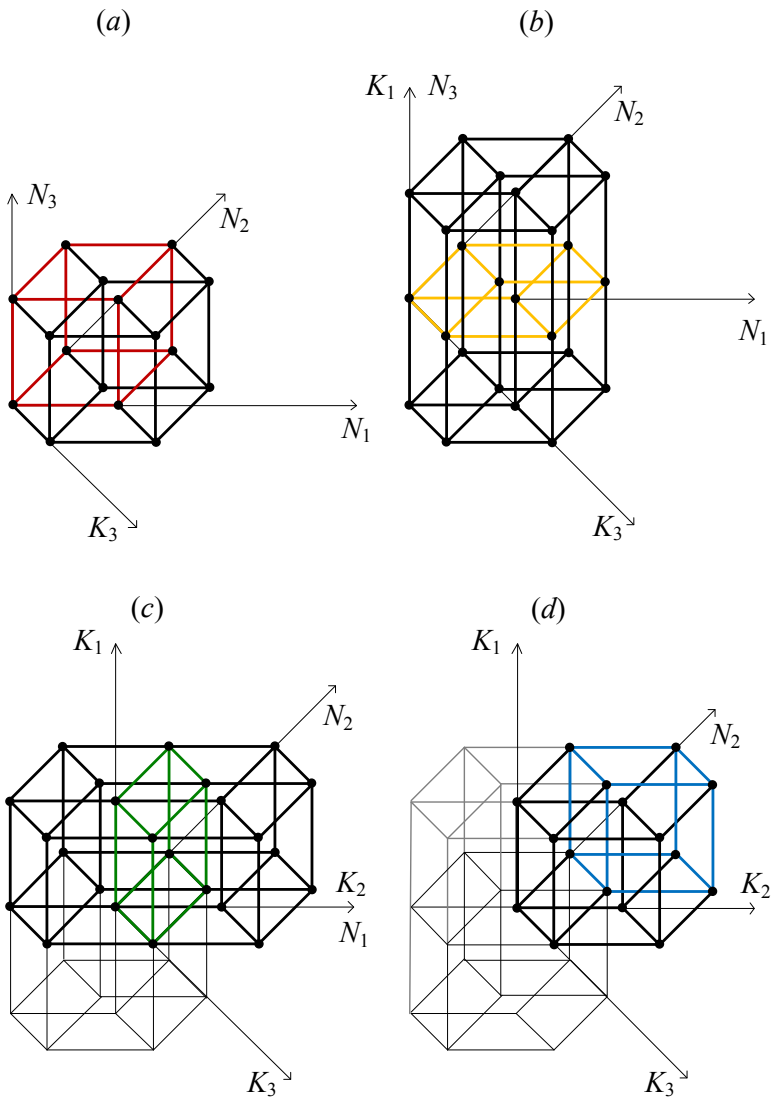

Figure 3: United 4D (mesh-based) computational index space of the 3D DXT: (a) index space for the first stage with an initial cubical data grid shown in red; concatenation of the 4D index spaces for the (b) first and second and (c) second and third stages with a common intermediate cubical data shown in yellow and green, respectively; (d) the final index space with a computed data cube shown in blue.

where $0 \leq K_1, K_2, K_3 < N$. A 4D computational index space for this final stage is

$$\mathcal{I}^{\text{III}}_{4D} = \{(K_1, N_2, K_3, K_2) : 0 \leq K_1, N_2, K_3, K_2 < P\}.$$

This index space includes a common to the previous stage 3D face $\mathcal{I}^{\text{II/III}}_{3D} = \mathcal{I}^{\text{II}}_{4D} \bigcap \mathcal{I}^{\text{III}}_{4D} = \{(K_1, N_2, K_3) : 0 \leq K_1, N_2, K_3 < P\}$ (shown in green in Fig. $3(c)$) and a 3D output data tensor $\{\dddot{\mathbb{X}}(K_1, K_2, K_3),\ 0 \leq K_1, K_2, K_3 < P\}$ which is depicted in blue color in Fig. $3(d)$.

It is clear that, in general, these three stages can be implemented in any of six possible orders of summation and the total number of block cubical tensor-by-matrix multiplications is reduced from P^6, as for the direct computing by (3) or (4), to $3P^4$ which is equal to the number of index points in a combined 4D index space

$$\mathcal{I}_{4D} = \mathcal{I}^{\text{I}}_{4D} \bigcup \mathcal{I}^{\text{II}}_{4D} \bigcup \mathcal{I}^{\text{III}}_{4D}. \tag{8}$$

By slicing cubical data, i.e. representing three-way tensors as the set of matrices, it is possible to formulate a 3D transform (3) or (4) as conventional block *matrix-by-matrix multiplication*. In this case, an initial $P{\times}P{\times}P$ data grid or three-way tensor $\{\mathbb{X}(N_1, N_2, N_3), 0 \leq N_1, N_2, N_3 < P\}$, is divided into P 1D "slices" or "slabs" or "matrices-of-cubes" along one of axises, for example, along N_2-axis, such that each $b \times b \times b$ data cube $\mathbb{X}(N_1, N_2, N_3)$ can be referred as a block element $\mathbb{X}(N_1, N_3)_{N_2}$ of the N_2-th $P \times P$ matrix, where $N_2 \in [0, P)$ (see Fig. 2). Then a 3D FDXT (3) can also be computed in three data-dependent stages as chaining sets of the block matrix-by-matrix products with a 3D computational index space for each product.

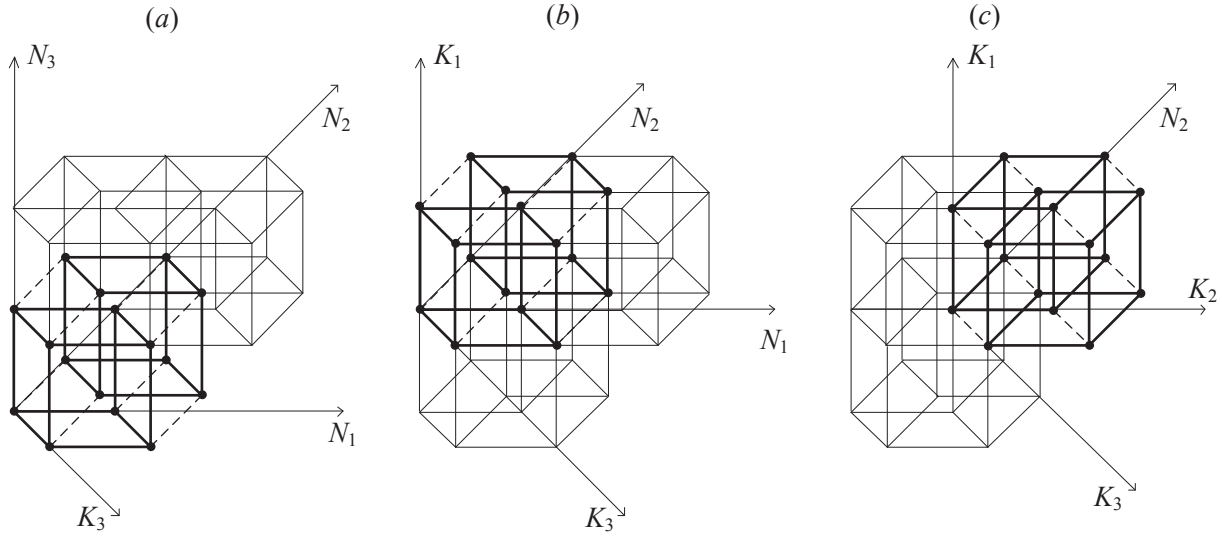

Figure 4: Partition of the mesh-based 4D computational index space into the set of 3D disjoint index spaces (opposite parallel cubical sides) for the first (a), second (b), and third (c) stages of computing.

At the **first stage**, since there is no dependency in 1D transforming data between all N_2-slabs, the cubical tensor-by-matrix multiplication (5) can be presented as P, independent on N_2, block matrix-by-matrix multiplications:

$$\dot{\mathbb{X}}(N_1, K_3)_{N_2} = \sum_{N_3=0}^{P-1} \mathbb{X}(N_1, N_3)_{N_2} \times C(N_3, K_3) \tag{9}$$

where the same change-of-basis matrix $C = [C(N_3, K_3)]$, $0 \le N_3, K_3 < P$, is used for all N_2-slices, $N_2 \in [0, P)$. As it follows from (9), each block matrix-by-matrix product in a N_2-th slab is implemented in its own 3D index space with P^3 index poins

$$\mathcal{I}^{\mathrm{I}}_{3D}(N_2) = \{(N_1, N_3, K_3) : 0 \le N_1, N_3, K_3 < P\}.$$

It is clear that all these cubical index spaces are disjoint, i.e.

$$\bigcap_{0 \le N_2 < P} \mathcal{I}^{\mathrm{I}}_{3D}(N_2) = \emptyset,$$

and, therefore, an associated with each 3D index space set of computing can be implemented independently or in parallel.

For the smallest 2×2 matrix case, i.e. when $b = N/2$ and, therefore, $P = 2$, the 3D index spaces $\mathcal{I}^{\mathrm{I}}_{3D}(0)$ and $\mathcal{I}^{\mathrm{I}}_{3D}(1)$ can be seen in Fig. $4(a)$ as opposing parallel sides (cubes in bold lines) along the N_2-axis. In these 3D grids of index points, each point $p = (N_1, N_3, K_3)^T \in \mathcal{I}^{\mathrm{I}}_{3D}(N_2)$ is associated with a conventional block matrix-by-matrix multiply-add

$$\dot{\mathbb{X}}(N_1, K_3)_{N_2} \leftarrow \mathbb{X}(N_1, N_3)_{N_2} \times C(N_3, K_3) + \dot{\mathbb{X}}(N_1, K_3)_{N_2}, \tag{10}$$

where an addition or accumulation is performed along N_3-direction and a block matrix $\dot{\mathbb{X}}(N_1, K_3)_{N_2}$ is a cubical $b \times b \times b$ data tensor which should be zeroed initially. Now, for the original first stage (5), the 4D computational index space can be defined as

$$\mathcal{I}^{\mathrm{I}}_{4D} = \bigcup_{0 \le N_2 < P} \mathcal{I}^{\mathrm{I}}_{3D}(N_2).$$

A 3D index space for the computed on this stage intermediate cubical tensor $\dot{\mathbb{X}}(N_1, N_2, K_3)$ can be seen in Fig. $3(b)$ in yellow color.

At the **second stage**, the original tensor-by-matrix product (6) is computed by the set of P, also independent on N_2, block matrix-by-matrix multiplications in the following form which keeps the required row-by-column index agreement:

$$\ddot{\mathbb{X}}(K_1, K_3)_{N_2} = \sum_{N_1=0}^{P-1} C(N_1, K_1)^T \times \dot{\mathbb{X}}(N_1, K_3)_{N_2}. \tag{11}$$

It can be seen from (11) that the same coefficient matrix $C = [C(N_1, K_1)]$, $0 \le N_1, K_1 < P$, is used for all N_2-slices, $N_2 \in [0, P)$. As in the previous stage, computing for each N_2-slab is implemented in a 3D index space

$$\mathcal{I}^{\mathrm{II}}_{3D}(N_2) = \{(N_1, K_1, K_3) : 0 \le N_1, K_1, K_3 < P\}.$$

The same as above, all these 3D index spaces are disjoint, i.e.

$$\bigcap_{0 \le N_2 < P} \mathcal{I}^{\mathrm{II}}_{3D}(N_2) = \emptyset,$$

and, therefore, the related to each 3D index space set of computing can also be implemented independently.

The 3D index spaces $\mathcal{I}^{\mathrm{II}}_{3D}(0)$ and $\mathcal{I}^{\mathrm{II}}_{3D}(1)$ are shown for $P = 2$ in Fig. $4(b)$ as two opposing parallel sides (cubes) along the same as in the prior stage N_2-axis. Each index point $p = (N_1, K_1, K_3)^T \in \mathcal{I}^{\mathrm{II}}_{3D}(N_2)$ is associated here with a block multiply-add computing

$$\ddot{\mathbb{X}}(K_1, K_3)_{N_2} \leftarrow C(N_1, K_1)^T \times \dot{\mathbb{X}}(N_1, K_3)_{N_2} + \ddot{\mathbb{X}}(K_1, K_3)_{N_2}. \tag{12}$$

In this stage, an addition is performed along N_1-direction and a block matrix $\ddot{\mathbb{X}}(K_1, K_3)_{N_2}$ is a $b \times b \times b$ data tensor which should be zeroed initially. For the original stage (6), the 4D computational index space is defined as

$$\mathcal{I}^{\text{II}}_{4D} = \bigcup_{0 \le N_2 < P} \mathcal{I}^{\text{II}}_{3D}(N_2).$$

A 3D index space for the computed on this stage intermediate cubical tensor $\ddot{\mathbb{X}}(K_1, N_2, K_3)$ can be seen in Fig. $3(c)$ in a green color.

At the **third stage**, the final tensor-by-matrix multiplication (7) should be implemented inside the 4D index space by summation data along N_2-direction. This can be done by 1D slicing or partition of the computed at the previous stage (see Fig. $3(c)$), cubical tensor $\ddot{\mathbb{X}}(K_1, N_2, K_3)$, $0 \le K_1, N_2, K_3 < P$, along K_3-axis, such that each K_3-th slab will keep the all nedeed data at the index points along N_2-direction, i.e.

$$\ddot{\mathbb{X}}(K_1, N_2, K_3) = \bigcup_{0 \le K_3 < P} \ddot{\mathbb{X}}(K_1, N_2)_{K_3}.$$

With this data partition, the tensor-by-matrix product (7) is implemented as the set of P, independent on K_3, block matrix-by-matrix multiplications:

$$\dddot{\mathbb{X}}(K_1, K_2)_{K_3} = \sum_{N_2=0}^{P-1} \ddot{\mathbb{X}}(K_1, N_2)_{K_3} \times C(N_2, K_2), \tag{13}$$

where the same $P \times P$ matrix $C = [C(N_2, K_2)]$, $0 \le N_2, K_2 < P$, is used for all K_3-slices, $K_3 \in [0, P)$. A computing in each K_3-slab is implemented in its own 3D index space

$$\mathcal{I}^{\text{III}}_{3D}(K_3) = \{(K_1, N_2, K_2) : 0 \le K_1, N_2, K_2 < P\}.$$

Again, all these 3D index spaces are disjoint, i.e.

$$\bigcap_{0 \le K_3 < P} \mathcal{I}^{\text{III}}_{3D}(K_3) = \emptyset,$$

and, therefore, the associated with each 3D index space set of computations can also be implemented independently.

For the case of $P = 2$, the 3D index spaces $\mathcal{I}^{\text{III}}_{3D}(0)$ and $\mathcal{I}^{\text{III}}_{3D}(1)$ are shown in Fig. $4(c)$ as opposing parallel sides along the K_3-axis. Here, each index point $p = (K_1, N_2, K_2)^T \in \mathcal{I}^{\text{III}}_{3D}(K_3)$ is associated with a block matrix-by-matrix multiply-add

$$\dddot{\mathbb{X}}(K_1, K_2)_{K_3} \leftarrow \ddot{\mathbb{X}}(K_1, N_2)_{K_3} \times C(N_2, K_2) + \dddot{\mathbb{X}}(K_1, K_2)_{K_3}. \tag{14}$$

In this final stage, an addition is performed along N_2-direction and a block matrix $\dddot{\mathbb{X}}(K_1, K_2)_{K_3}$ is a $b \times b \times b$ data tensor which should be zeroed initially. It is clear that for the original stage (7), the 4D index space is defined as

$$\mathcal{I}^{\text{III}}_{4D} = \bigcup_{0 \le K_3 < P} \mathcal{I}^{\text{III}}_{3D}(K_3).$$

A 3D index space for the resulting cubical tensor $\dddot{\mathbb{X}}(K_1, K_2, K_3)$ is shown in Fig. $3(d)$ in a blue color .

Note that in (9), (11) and (13) each of P products is three-way $b \times b \times b$ tensor by $b \times b$ matrix multiplication [4]. Moreover, for each N_2-slice of a cubical data $\mathbb{X}$, the first two stages perform independently the canonical 2D FDXT in the form of triple matrix multiplication with one matrix transposed, i.e., in the

matrix form, $\ddot{\mathbb{X}} = C^T \times \mathbb{X} \times C$. It is clear that in the first stage (9), N_3-axis from the original basis is converted into K_3-axis of the new basis, i.e. transforming $[N_3 \rightarrow K_3]$ is implemented. In the second stage (11), changing $[N_1 \rightarrow K_1]$ is performed (see Fig. 5 (a)). Before the third stage, where the final changing $[N_2 \rightarrow K_2]$ should be executed, an 1D slicing or working partition of the intermediate cubical tensor $\ddot{\mathbb{X}}$ along N_2-axis is algorithmically changed to the required by (13) N_2-summation by using an 1D partition along K_3-axis (see Fig. 5 (b)). This changing of partition and summation direction is possible because $\ddot{\mathbb{X}}(K_1, K_3)_{N_2} \equiv \ddot{\mathbb{X}}(K_1, N_2)_{K_3}$, i.e. the same $b \times b \times b$ data cube simultaneously belongs to N_2- and K_3-slices or slabs. Actually, a slab's sub-index in the above equations is used only to represent a 3D DXT computing in the form of canonical (planar) matrix-by-matrix multiplication where the row-by-column summation index agreement is required. It is clear that this slab's sub-index can be included in the main 3-dimensional index when direct multilinear multiplication of a three-way tensor by a 2D matrix is utilized [33, 39].

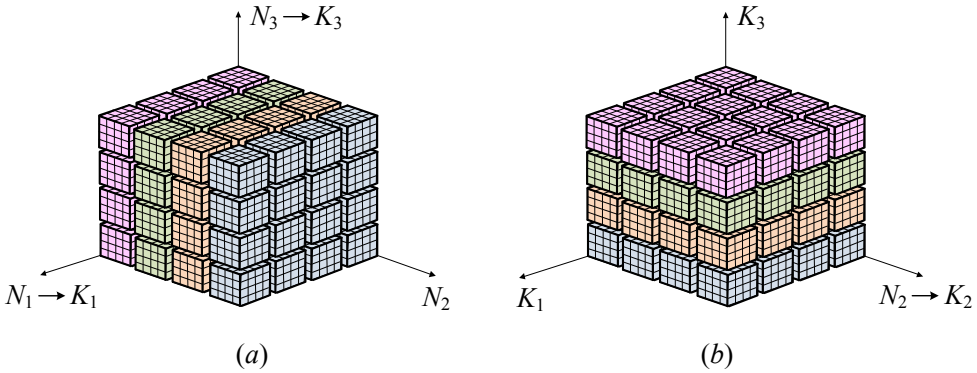

Figure 5: Partition of a 3D data tensor into slabs (slices) of cubes (a) along N_2-axis for the first and second stages and (b) along K_3-axis for the third stage.

It is important to mention that an initial slicing or 1D partition of a given cubical tensor affects the order in which the data dependent sets of 1D transforms are implemented. Excluding an existing in our cubical tensor (super)symmetry [24], the selected above initial data partition along N_2-axis or the so-called lateral slicing [24], leads to the following 3-stage transform order:

$$[N_3 \rightarrow K_3] \Rightarrow [N_1 \rightarrow K_1] \Rightarrow [N_2 \rightarrow K_2].$$

A partition of the same cubical tensor along N_1-axis or frontal data slicing, will result the order:

$$[N_3 \rightarrow K_3] \Rightarrow [N_2 \rightarrow K_2] \Rightarrow [N_1 \rightarrow K_1].$$

Finally, a cubical data partition along N_3-axis or horizontal slicing, will demand the order:

$$[N_2 \rightarrow K_2] \Rightarrow [N_1 \rightarrow K_1] \Rightarrow [N_3 \rightarrow K_3].$$

It is clear that an inverse 3D transform (IDXT) (4) is implemented in the reverse order, i.e. as rolling back of a forward 3D DXT. By keeping the slicing of an initial cubical $P \times P \times P$ tensor $\dddot{\mathbb{X}}(K_1, K_2, K_3)$ along K_3-axis, a 3D IDXT would require implementation of the following three stages:

Stage I: **for all** *pairs* (K_1, N_2) **at** *slabs* $K_3 \in [0, P)$ **do**

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

This first stage for a 3D IDXT is computed in the same as for the third stage of a 3D FDXT (7) 4D index space

$$\mathcal{I}^{\text{III}}_{4D} = \{(K_1, K_2, K_3, N_2) : 0 \le K_1, K_2, K_3, N_2 < P\}.$$

After completion, a resulting in this stage cubical tensor $\ddot{\mathbb{X}}(K_1, N_2, K_3)$ is used as sliced into 1D slabs along N_2-axis.

Stage II: **for all** *pairs* (N_1, K_3) **at** *slabs* $N_2 \in [0, P)$ **do**

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

The same as for the equation (6), this stage is implemented in a 4D index space

$$\mathcal{I}^{\text{II}}_{4D} = \{(K_1, N_2, K_3, N_1) : 0 \le K_1, N_2, K_3, N_1 < P\}$$

with a common 3D face $\mathcal{I}^{\text{III/II}}_{3D} = \mathcal{I}^{\text{III}}_{4D} \bigcap \mathcal{I}^{\text{II}}_{4D} = \{(K_1, N_2, K_3) : 0 \le K_1, N_2, K_3 < P\}$.

Stage III: **for all** *pairs* (N_1, N_3) **at** *slabs* $N_2 \in [0, P)$ **do**

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

The final third stage of the 3D IDXT is implemented in the same as for the first stage of a 3D FDXT (5) 4D index space

$$\mathcal{I}^{\text{I}}_{4D} = \{(N_1, N_2, K_3, N_3) : 0 \le N_1, N_2, K_3, N_3 < P\}$$

with a common 3D face $\mathcal{I}^{\text{II/I}}_{3D} = \mathcal{I}^{\text{II}}_{4D} \bigcap \mathcal{I}^{\text{I}}_{4D} = \{(N_1, N_2, K_3) : 0 \le N_1, N_2, K_3 < P\}$.

It is easy to verify that the total number of $P \times P$ block matrix-by-matrix multiplications in a forward or inverse 3D DXT is $3P^4$. Each block matrix-by-matrix multiplication is a $b \times b \times b$ tensor-by-matrix product which require an execution of b^4 scalar fused (indivisible) multiply-add (`fma`) operations, where $b = N/P$ is a blocking factor. The total number of such scalar `fma`-operations for a 3D DXT is, therefore, $3P^4 \cdot b^4 = 3N^4$, i.e. blocking does not change an arithmetic complexity of the transformation. Obviously, this is because our 3D DXT implementation is totally based on a matrix-by-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 $N^4/N^3 = N$ scalar multiply-add operations per data element. Note that the existing so-called "fast" algorithms for the 3D DXT, like 3D FFT, require an execution of $\mathcal{O}(N^3 \log N)$ scalar arithmetic operations, i.e. an arithmetic density is only $\mathcal{O}(\log N)$ operations per data element. This low degree of data reusing is one of the main reasons of low efficiency and poor scalability of parallel implementations of the fast DXT algorithms.

3 Exa-Scalable Matrix-by-Matrix Multiplication and 3D Transforms

3.1 Fast Matrix-by-Matrix Multiplication

It is clear from the above discussion that a 3D FDXT of a cubical data array or three-way tensor $\mathbb{X}$ can be expressed as multilinear matrix multiplication

$$\dddot{\mathbb{X}} = \overbrace{\underbrace{\Big(C^T \times \overbrace{\big(\mathbb{X} \times C\big)}^{\text{1D FDXT: } N_3 \to K_3}\Big)}_{\text{2D FDXT: } N_1 \to K_1} \times C}^{\text{3D FDXT: } N_2 \to K_2}, \tag{15}$$

and a 3D IDXT as

$$\mathbb{X} = \overbrace{\underbrace{\Big(C \times \overbrace{\big(\dddot{\mathbb{X}} \times C^T\big)}^{\text{1D IDXT: } K_2 \to N_2}\Big)}_{\text{2D IDXT: } K_1 \to N_1} \times C^T}^{\text{3D IDXT: } K_3 \to N_3}. \tag{16}$$

Recall that this fixed order of parentetization directly corresponds to 1D initial partition (slicing) of the input three-way tensor along N_2-axis which has been selected above as one of three possible 1D partitions (see Fig. 2 (b)).

From the equations (15) and (16) it is evident that to implement a 3D transform under *unlimited* or *extreme parallelism*, i.e. when the maximum number of simultaneous operations is limited only by the size of data, the answers to the following questions should be found:

- What is (are) the *fastest* and technologically justified parallel matrix-by-matrix multiply-add (MMA) algorithm(s) with the maximal data reuse and extreme scalability?
- How to design a collection of parallel MMA algorithms with different *transposed/non-transposed* scenarios, like in a famous GEneral Matrix-matrix Multiplication (GEMM) from the Level-3 BLAS [11]?
- How the different sets of MMA operations can be *chained* in time-space to effectively implement fast multilinear matrix multiplication (15) and (16) without any data transposition?
- How it can be possible to map three, concatenated by a common 3D data, 4D *computational index spaces* $\mathcal{I}^{\mathrm{I}}_{4D} \bigcup \mathcal{I}^{\mathrm{II}}_{4D} \bigcup \mathcal{I}^{\mathrm{III}}_{4D}$ of a 3D DXT (see Figures 3 and 4) into a 3-dimensional physical (spatial) space of execution or *processor space*?

Some answers to these questions can be found in our previous paper [46] where extremely scalable scalar MMA algorithms for the two-dimensional DXT on a 2D torus array processor have been systematically designed and evaluated. The main results related to the systematic design of the scalable MMA algorithms are described below. In [46] , it was formally shown that for the following three forms of the matrix-by-matrix multiply-add:

$$C \leftarrow A \times B + C, \text{ where } c(i,j) \leftarrow \sum_{k=0}^{n-1} a(i,k) \cdot b(k,j) + c(i,j); \tag{17a}$$

$$A \leftarrow C \times B^T + A, \text{ where } a(i,k) \leftarrow \sum_{j=0}^{n-1} c(i,j) \cdot b(k,j) + a(i,k); \tag{17b}$$

$$B \leftarrow A^T \times C + B, \text{ where } b(k,j) \leftarrow \sum_{i=0}^{n-1} a(i,k) \cdot c(i,j) + b(k,j); \tag{17c}$$

where $A = [a(i,k)]$, $B = [b(k,j)$, and $C = [c(i,j)]$ are matrices of the same $n \times n$ size, there exist three classes of the fastest MMA algorithms which are efficiently implemented on a planar $n \times n$ mesh/torus array processor in n time-steps. These three classes are differ in linear arrangements (time-step scheduling) of the partially-ordered set of n^3 scalar *indivisible* (fused) multiply-add operations which are algorithmically "located" in a cubical $n \times n \times n$ computational index space $\mathcal{I}_{3\mathrm{D}} = \{(i,j,k)^T : 0 \le i,j,k < n\}$. Actually, every scheduling class defines for each **step**$(p) \in \{0,1,2,...,n-1\}$ the set of n^2 "active" index points $p = (i,j,k)^T \in \mathcal{I}_{3\mathrm{D}}$ with an associated data-triplet $\{a(i,k), b(k,j), c(i,j)\}$, where partial computing of

(17) is implemented by concurrent execution of n^2 scalar multiply-add operations from the corresponding set:

$$c(i,j) \leftarrow a(i,k) \cdot b(k,j) + c(i,j); \tag{18a}$$
$$a(i,k) \leftarrow c(i,j) \cdot b(k,j) + a(i,k); \tag{18b}$$
$$b(k,j) \leftarrow a(i,k) \cdot c(i,j) + b(k,j). \tag{18c}$$

The scheduled at each time-step set of n^2 "active" index points directly defines an associated data distribution in 3D index space including an initial data placement, which is equal to the data allocation at the first time-step. Additionally, this set of index points also defines, but indirectly, a specific for each class of scheduling way of data reuse. As a result of indirect definition of data reuse, the different scenarios of its physical implementation may exist as it will be shown below.

The time-step scheduling function is used in the linear or modular form:

$$\textbf{step}(p) = (\alpha^T \cdot p) \quad \text{or} \quad \textbf{step}(p) = (\alpha^T \cdot p) \bmod n,$$

respectively, where $\alpha = (\alpha_i, \alpha_j, \alpha_k)^T$ is a scheduling vector and $(\cdot)$ is a scalar product of two vectors. Based on the different space-time data arrangements in a 3D index space $\mathcal{I}_{3D}$, the following classification of MMA algorithms can be established (see [46] for details).

- Broadcast-Broadcast-Compute (BBC) class:
  - the rank-one [48] or outer-product [19] update is used here where computational index space $\mathcal{I}_{3D}$ is represented as a 3D mesh of points;
  - there is only one linear scheduling function for this class: $\textbf{step}(p) = k$, i.e. $\alpha = (0,0,1)^T$;
  - linear projection of the 3D index space along summation axis into a 2D physical space gives a well-known broadcast- or replication-based MMA implementation on a planar array processor [2, 48];
- Broadcast-Compute-Roll (BCR) class:
  - a computational index space $\mathcal{I}_{3D}$ is a 3D cylindrically interconnected points;
  - there are two (orthogonal) modular scheduling functions: $\textbf{step}(p) = (k-i) \bmod n$ and $\textbf{step}(p) = (k-j) \bmod n$, i.e. $\alpha = (-1,0,1)^T$ or $\alpha = (0,-1,1)^T$;
  - a 3D $\rightarrow$ 2D projection of the cylindrical index space also along summation axis into a 2D physical space gives the well-known Fox's algorithm on a non-planar, semi-toroidal, array processor [15];
- Compute-Roll-All (CRA) or orbital class:
  - an index space $\mathcal{I}_{3D}$ is a 3D toroidally interconnected points;
  - excluding a symmetry, there are four distinct modular scheduling functions: $\textbf{step}(p) = (\pm i \pm j \pm k) \bmod n$, i.e. $\alpha = (\pm 1, \pm 1, \pm 1)^T$;
  - different admissible linear 3D $\rightarrow$ 2D projections of the torus index space into a 2D physical space produce a variety of Cannon-like algorithms [7, 46] on a 2D torus array processor.

Recall that all three MMA forms (17) are equally scheduled, i.e. if transposition is required, a corresponding operation (17b) or (17c) is implemented with the same initial data distribution in a 3D index space and with the same data movement pattern as for (17a), i.e. actual matrix transposition is avoided.

Different linear or modular arrangements (schedules) of n^3 data dependent scalar operations impose a different style in reusing of the matrices A and B for concurrent updating n^2 elements of a matrix C at each time-step [1]. That is, at each of n iterations or consecutive time-steps of computing:

- an algorithm from the BBC class reuses (by data replication or copying, since $\alpha_i = 0$ and $\alpha_j = 0$) n data elements (one vector-column) of a matrix A and n data elements (one vector-row) of a matrix B, i.e. totally $2n$ elements are reused for updating n^2 elements of a matrix C which are then shifted (since $\alpha_k = 1$) within a cubical index space $\mathcal{I}_{3\mathrm{D}}$ to the next iteration step;

- algorithms from the BCR class reuse n data elements (vector-diagonal) of a matrix A or B (by replication, since $\alpha_j = 0$ or $\alpha_i = 0$) and all n^2 elements of a matrix B or A, respectively, i.e. $n^2 + n$ elements are reused for updating n^2 elements of a matrix C and, after updating, the all $2n^2$ elements of matrices C and B or A, respectively, are circularly shifted or rolled (since $\alpha_k = 1$ and $\alpha_i = -1$ or $\alpha_j = -1$) inside cylindrical index space $\mathcal{I}_{3\mathrm{D}}$ to the next step along or opposite corresponding direction (orbit) according to the sign;

- algorithms from the CRA class reuse all elements of matrices A and B, i.e. totally $2n^2$ data elements are reused here for updating n^2 elements of a matrix C and, after updating, the all $3n^2$ elements of the matrices A, B and C are circularly shifted (since $\alpha_i = \pm 1, \alpha_j = \pm 1, \alpha_k = \pm 1$) inside torus index space $\mathcal{I}_{3\mathrm{D}}$ to the next step also along or opposite corresponding direction (orbit) according to the sign.

Note that the well-known planar systolic MMA algorithms [26, 28, 27], which are belong to the so-called All-Shift-Compute (ASC) class, require $3n - 2$ time-steps on a mesh array processor and, therefore, they are not as fast as other discussed here n-steps algorithms. Moreover, in the systolic array processors, the initial data streams should be appropriately skewed and, in some cases, scattered before main processing is started. An additional, non-trivial parallel memory, surrounding a systolic array processor, is usually needed to keep and form these initial/final data streams. These challenging requirements sufficiently complicate a practical implementation of systolic array processors. Note that ASC class is a localized (pipelined) version of the BBC class where the time-step function is defined as $\mathbf{step}(p) = i + j + k$, i.e. $\alpha = (1, 1, 1)^T$.

The provided classification makes it clear that MMA algorithms from BBC class are the most economical in terms of the quantity of reusing data but, at the same time, this class require a global, i.e. depending on the size of the problem, data *replication* among n^2 different operations. This global data replication can be physically implemented by either *data broadcast* ("one-to-all") or by its antipode, *concurrent access* ("all-to-one") to this reusable data in a shared memory. If MMA computing is implemented on a synchronous system where the signal propagation time between different components is no longer than the clock period then signal/data replication by broadcast or by shared access can be implemented, in principle, within a single cycle. In other words, MMA algorithms from the BBC class would be effective in any synchronous system which size is limited and adjusted for the single clock period. This synchronous system acts as a single node. Because technically, concurrent access to the shared memory is much faster than data replication by broadcast within the same limited size of system, a hierarchical shared memory approach is widely used in today high-performance CPU and GPU processors. However, it is clear from the existing physical constrains, such as finiteness of the speed of light, that a global data broadcast or its alternative, global data sharing, will prevent a computer system to be extremely scaled to any relatively big size (far beyond the

[1] An explanation here is related to the canonical form (17a), but form (17b) or (17c) can be equally considered.

synchronization area or volume) when the number of concurrent operations is equal to the size of problem, i.e. the size of matrices.

From the above classification it is also follows that the only Compute-Roll-All schedule defines at each time-step a *global* circulation by *local* movement of all $3n^2$ matrix elements without any redundant data replication. Therefore, only MMA algorithms from the CRA class, which are supported by appropriate asynchronous local data exchange, can be extremely scaled on the system without a global clock. Note that due to cyclical (modular) nature of the orbital CRA-based MMA algorithms, the all rolled at each time-step data elements are finally (after n "compute-and-roll" time-steps) returned to the originating (initial) positions in a 3D torus index space. This unique for the orbital processing property is effectively used for chaining of the different matrix products (see [46] for details). It is clear that BCR class of parallel MMA algorithms is a combination of n-data replication and n^2-data circulation, i.e. mixing of the BBC and CRA classes.

Note that to effectively realize the set of highly-repetitive in MMA algorithms scalar operations (18), the following forms of fused multiply-add (`fma`) instructions should be considered: form 123 does $c \leftarrow a \cdot b + c$, form 321 does $a \leftarrow c \cdot b + a$ and form 132 does $b \leftarrow a \cdot c + b$. The ordering number is just the order of the operands on the right side of the expression.

3.2 Exa-Scalable Implementation of the 3D Transforms in a 4D Computational Space

Due to extreme scalability, a Compute-and-Roll (CRA) approach will be used for the orbital implementation of a block multilinear matrix multiplication (15) and (16). As it was previously shown (see Fig. 4), a 4D computational index space $\mathcal{I}_{4D}$ of the 3D DXT from (8) can be represented in each of three data dependent stages as disjoint sets of the 3D index spaces. Each 3D index space represents an independent matrix-by-matrix multiplication. All these disjoint index spaces should be properly, i.e. conflict-free, planned for execution by applying one of a few possible for CRA class modular scheduling functions which will define an orbital way of processing. A scheduled for execution 4D torus computational space, which is composed of many 3D tori of index points, should be finally mapped into a 3D space of physical implementation by using a corresponding 4D $\rightarrow$ 3D linear projection for each consecutive stage of processing.

At the **first stage** of a 3D DXT computing (9), the index points $p = (N_1, N_2, K_3)^T \in \mathcal{I}_{3\mathrm{D}}$ from each N_2-th 3D computational index space

$$\mathcal{I}^{\mathrm{I}}_{3D}(N_2) = \{(N_1, N_3, K_3) : 0 \leq N_1, N_3, K_3 < P\}, N_2 \in [0, P),$$

are scheduled for execution and data transferring by using one of the admissible modular functions. This modular (circular) scheduling converts each cubical, mesh-based index space $\mathcal{I}^{\mathrm{I}}_{3D}(N_2)$ into 3D torus or orbital index space. The same scheduling function is used as for all P disjoint 3D index spaces as well as for all stages of processing. As an example, the modular scheduling function $\mathbf{step}(p) = (\alpha^T \cdot p) \bmod P$ with the scheduling vector $\alpha = (1, 1, 1)^T$ is used here for the smallest case when $P = N/b = 2$. For the first stage (see equation (9)) of orbital computing, the result of this space-time scheduling is shown schematically in Fig. 6 for each of two disjoint 3D orbital index spaces by indicating "active" index points, which are filled in black, and directions of data movement in two different time-steps (a) and (b). At every time-step, each "active" index point is associated with a matrix-by-matrix multiply-add which is defined by (10). After *computing* part, blocks of the matrix data $\mathbb{X}(N_1, N_3)_{N_2}, C(N_3, K_3)$ and $\dot{\mathbb{X}}(N_1, K_3)_{N_2}$ are *rolled* along K_3-, N_1-, and N_3-orbits, respectively. These positive directions of data rolling are defined by the given scheduling vector $\alpha = (1, 1, 1)^T$. Recall that in the first stage, an accumulation of the matrix-by-matrix products is implemented along N_3-axis or orbit.

It is clear that at each time-step only P^2 index points are active in each 3D torus index space $\mathcal{I}^{\mathrm{I}}_{3D}(N_2)$, and, therefore, totally, P^3 index points are active in all index spaces. As it can be seen later, this condition

holds for all three stages of orbital processing. Therefore, for every stage of the 3D DXT processing, no more than P^3 computer nodes would be needed at each of P time-steps to implement all P^4 operations associated with all index points in each $\mathcal{I}_{4D}^{\text{I/II/III}}$. For the first stage, the P unconnected (disjoint) orbital index spaces $\mathcal{I}_{3D}^{\text{I}}(N_2)$ is scheduled such that "compute-and-roll" processing in each N_2-th 3D torus index space begins at **step**$(p) = N_2$, i.e. starting time is skewed for each index space. As it can be seen from Fig. 6 on time-step (a), a cubical index space $\mathcal{I}_{3D}^{\text{I}}(0)$ is scheduled to start from **step**$(p) = 0$ while an index space $\mathcal{I}_{3D}^{\text{I}}(1)$ is planned to start from **step**$(p) = 1$. This initial time skewing in the orbital scheduling guaranties that all index points with the same coordinates in different 3D orbital spaces will have different time-steps during all processing. After P "compute-and-roll" time-steps, i.e. after finishing of the first stage, blocks of intermediate matrix data $\dot{\mathbb{X}}(N_1, K_3)_{N_2}$ will be located in a 4D index space as it is required by the second stage. Actually, due to the cyclical nature of data movement, these updated blocks are finally, i.e. after P time-steps, returned to the original positions (indexes).

The **second stage** of orbital computing (11) is performed by following the same as for the first stage scheduling policy for each of P disjoint by N_2-axis 3D torus index spaces

$$\mathcal{I}_{3D}^{\text{II}}(N_2) = \{(N_1, K_1, K_3) : 0 \leq N_1, K_1, K_3 < P\}, N_2 \in [0, P).$$

A time-space scheduling of "active" index points is shown in Fig. 6 for time-steps (c) and (d). It can be seen that there is no difference between two stages as in positioning of the independent 3D torus index spaces in a 4D space as well as in scheduling of the index points with the same coordinates, despite the fact that N_3-axis was replaced after the first stage by K_1-axis. Here, at each consecutive time-step, every "active" index point is associated with a given by (12) matrix-by-matrix multiplication. For this stage, according to the scheduling vector $\alpha = (1, 1, 1)^T$, a corresponding initial distribution and cyclical movements of an intermediate matrix $\ddot{\mathbb{X}}(K_1, K_3)_{N_2}$ along N_1-axis and a coefficient matrix $C(N_1, K_1)^T$ along K_3-axis are required. A previously computed matrix $\dot{\mathbb{X}}(N_1, K_3)_{N_2}$ is rolled at this stage along the "same" K_1-axis. An updating of the intermediate matrix $\ddot{\mathbb{X}}(K_1, K_3)_{N_2}$ is implemented by accumulation of the matrix-by-matrix products along N_1-axis. It is clear that P additional "compute-and-roll" time-steps are needed to finish second stage of orbital processing in a 4D index space. On completion of this stage, N_1-axis is replaced by K_2-axis.

For the **third stage** of a 3D DXT computing, we keep the same scheduling function for all index points in a 4D index space $\mathcal{I}_{4D}^{\text{III}}$ as for two previous stages. However, a required by (13) accumulation of products along N_2-axis has to be implemented in the differently oriented 3D torus index spaces

$$\mathcal{I}_{3D}^{\text{III}}(K_3) = \{(K_1, N_2, K_2) : 0 \leq K_1, N_2, K_2 < P\},$$

which are disjoint now along K_3-axis, $K_3 \in [0, P)$. The "active" index points and directions of data movement are shown schematically in Fig. 6 for the final time-steps (e) and (f) . Each "active" by the given scheduling index point from a 3D torus index space is associated here with a matrix-by-matrix multiplication (14). Besides, a selected at the beginning of processing modular scheduling function defines not only an initial data distribution, but also directions of moving of a resulting matrix $\dddot{\mathbb{X}}(K_1, K_2)_{K_3}$ along N_2-axis and a coefficient matrix $C(N_2, K_2)$ along K_1-axis. An updated at previous stage matrix $\ddot{\mathbb{X}}(K_1, N_2)_{K_3}$ is rolled along the "same" K_2-axis. It is clear that P additional "compute-and-roll" time-steps is required for this final stage and, therefore, totally, $3P$ such steps are needed for computing of a 3D DXT in a 4D $P \times P \times P \times P$ orbital index space.

4 A Scalable 3D DXT in 3D Torus Network of Nodes

The next step in the Algorithm/Architecture co-design is mapping a described above 4D logical implementation into the 3D physical space. This mapping is achieved by proper projection of the 4D scheduled orbital

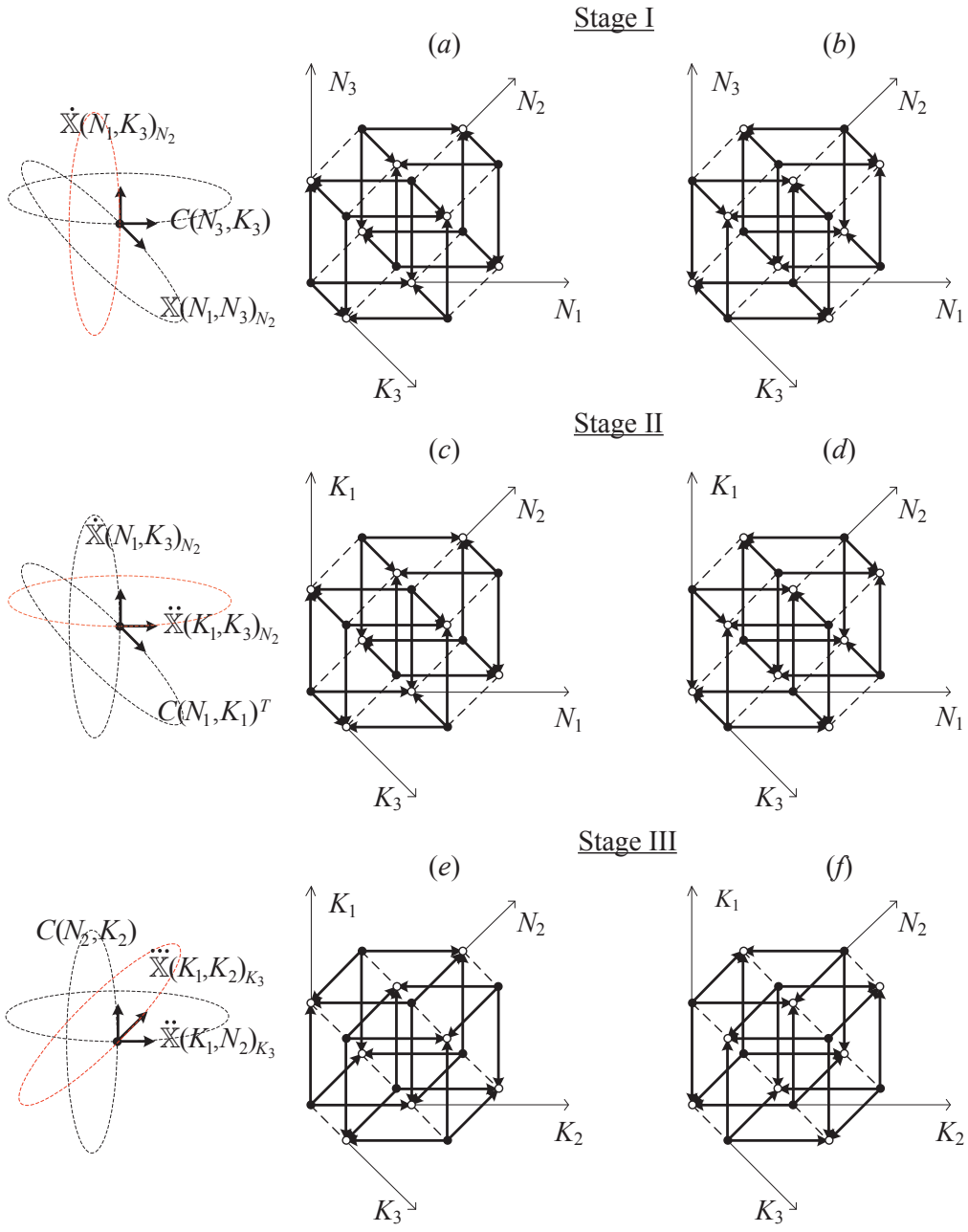

Figure 6: A 4D computational index space for the 3D DXT as disjoint sets of the 3D orbital index spaces; distribution of active index points (filled in black) as well as directions (orbits) of circular data movement at each "compute-and-roll" time-step $(a), (b), (c), (d), (e), (f)$ for the scheduling vector $\alpha = (1, 1, 1)^T$ and $P = 2$; orbits for products accumulation are shown in red; for the given smallest $2 \times 2 \times 2$ torus grid, wrap-around connections are just opposite to specified direction of data circulation.

index space to the 3D *computer space* where no more than one "active" index point is projected into a single computer node. We used the well-known $3D \rightarrow 2D$ linear projection method [28, 31, 45] which is extended here to support a $4D \rightarrow 3D$ transform. For the first and second stages of processing, each N_2-th 3D torus index space is projected onto 2D space along K_3-axis (see Fig. 6 $(a), (b), (c), (d)$) This parallel mapping results in P different 2D torus networks of $P \times P$ computer nodes each such that N_2-slices of a combined 3D torus of P^3 simple nodes independently and concurrently implements two transforms in $2P$ "compute-and-roll" time-steps. The third stage is implemented on all P parallel K_3-slices (which are orthogonal to N_2-slices) of this 3D torus of computer nodes by mapping each K_3-th 3D torus index space (see Fig. 6 $(e), (f)$) into corresponding K_3-slice along K_1-axis. The location of computer nodes in a 3D physical space are indicated by triplet $(\mathcal{Q}, \mathcal{R}, \mathcal{S})$, where $0 \leq \mathcal{Q}, \mathcal{R}, \mathcal{S} < P$. Below, an orbital implementation of the Forward and Inverse $N \times N \times N$ linear transforms on a 3D torus of $P \times P \times P$ computer nodes, where $P = N/b$ and $b \in [1, N]$ is a blocking factor, is formally described.

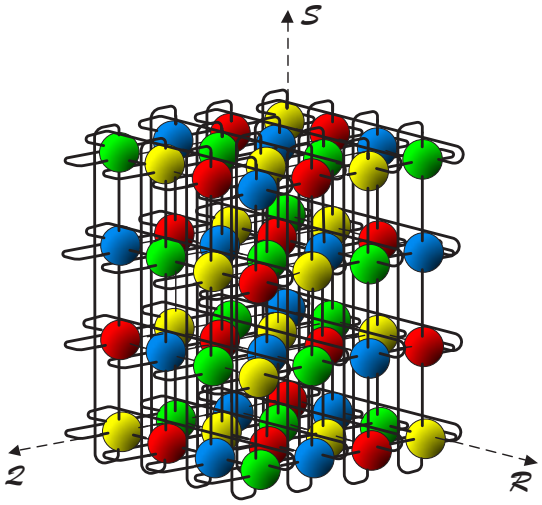

Figure 7: A $4 \times 4 \times 4$ array of computer nodes with toroidal interconnection in a 3D $(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ space; different colors are used to indicate nodes with distinct initial values of the given orbital function $\mathcal{T} = (\mathcal{Q} + \mathcal{R} + \mathcal{S}) \bmod 4 \in \{0$ (red), 1 (blue), 2 (green), 3 (yellow)$\}$.

4.1 3D Block Forward Transform

At the beginning, each computer node CN$(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ in a $P \times P \times P$ torus array (see Fig. 7 for $P = 4$) holds in a local memory the four $b \times b \times b$ data cubes:

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

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

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

where $\mathbb{O}$ is a cubical $b \times b \times b$ tensor with all its entries being zero. The initially selected and used modular function for space-time scheduling of a 4D computational index space defines one of possible functions

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

where $\alpha = (\alpha_{\mathcal{R}}, \alpha_{\mathcal{Q}}, \alpha_{\mathcal{S}})^T = (\pm 1, \pm 1, \pm 1)^T$ is the scheduling vector and $p = (\mathcal{Q}, \mathcal{R}, \mathcal{S})^T$ is a node's position in a 3D physical index space (see Fig. 7). Each computer node CN$(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ in a 3D torus network has six bi-directional links labeled as $\pm\mathcal{Q}, \pm\mathcal{R}, \pm\mathcal{S}$. During processing some blocks of tensor and matrix data are rolled, i.e. cyclically shifted, along $(+)$ or opposite $(-)$ axis (orbit) according to the scheduling vector α (see [46] for more details). As it can be seen from the assignment above, the $P \times P$ matrices $C_{\rm I}$ and $C_{\rm II}$ are replicated among P parallel along $\mathcal{R}$-axis slabs of computer nodes while $P \times P$ matrix $C_{\rm III}$ is duplicated among P parallel along $\mathcal{S}$-axis slabs. These change-of-basis matrices $C_{\rm I}$, $C_{\rm II}$, and $C_{\rm III}$ are preloaded into torus array only once for practically possible many 3D transforms.

A three-stage orbital implementation of the 3D *forward* DXT in a 3-dimensional network of toroidally interconnected nodes $\{\text{CN}(\mathcal{Q}, \mathcal{R}, \mathcal{S}) : 0 \le \mathcal{Q}, \mathcal{R}, \mathcal{S} < P\}$ 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. 8 for the smallest case: $P = 2$, where red colored links show an accumulation or summation direction).

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

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

Stage I is an orbital implementation in a physical space of the equation (5) or (9). This implementation is achieved by linear projection of the disjoint 3D torus index spaces $\mathcal{I}^{\rm I}_{3D}(N_2), 0 \le N_2 < P$, which are shown for different time-steps in Fig 6 (a) and (b), onto planar processor or execution space along K_3-axis. Each mapped 3D torus index space $\mathcal{I}^{\rm I}_{3D}(N_2)$ defines its own N_2-th 2D torus processor.

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

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

Stage II is an orbital implementation of the equation (6) or (11) by projection of the 3D torus index spaces $\mathcal{I}^{\rm II}_{3D}(N_2), 0 \le N_2 < P$, shown in Fig 6 (c), (d), onto the same as above 2D processor space along K_3-axis.

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

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

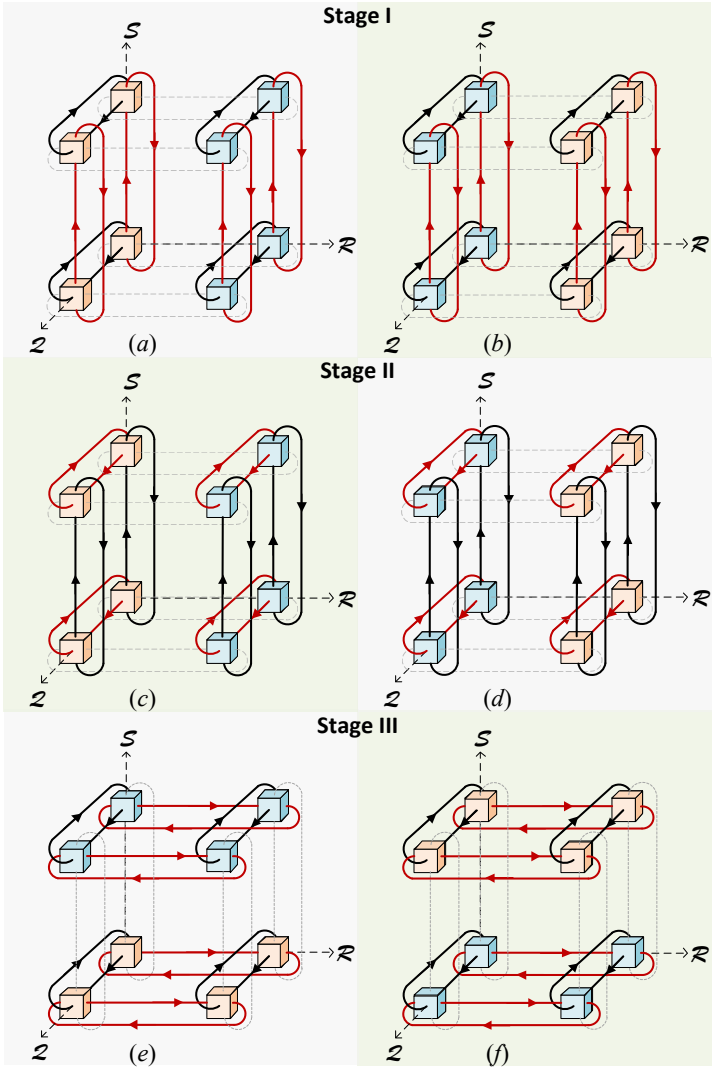

Figure 8: An orbital or circular data communication between nodes for the 3D FDXT.

Stage III represents an orbital implementation in a physical space of the equation (7) or (13) by using linear projection of the 3D torus index spaces $\mathcal{I}^{\rm III}_{3D}(K_3)$, $0 \le K_3 < P$, shown in Fig 6 $(e), (f,)$ onto a new 2D execution space along K_1-axis.

Recall that due to the orbital (cyclical or rotational) nature of processing, after completion of each stage, i.e. after P "compute-and-roll" steps, all rotated data are returned to the same originated nodes and, therefore, a computed intermediate cubical tensor can be immediately used for the next stage of a cyclical 3D processing. Note that initial tensor $\mathbb{X} = \mathbb{X}(N_1, N_2, N_3) = \mathbb{X}(\mathcal{Q}, \mathcal{R}, \mathcal{S})$ is assigned to the nodes in the canonical (in-order) layout whereas all intermediate and final tensors, $\dot{\mathbb{X}}, \ddot{\mathbb{X}}$ and $\dddot{\mathbb{X}}$, will be distributed in the skewed (out-of-order) layouts. It is also important to note that because at any stage each cubical tensor-by-matrix multiplication is separated into the set of independent matrix-by-matrix products, the "compute" and "data roll" parts can also be splitted and effectively overlapped by software pipelining.

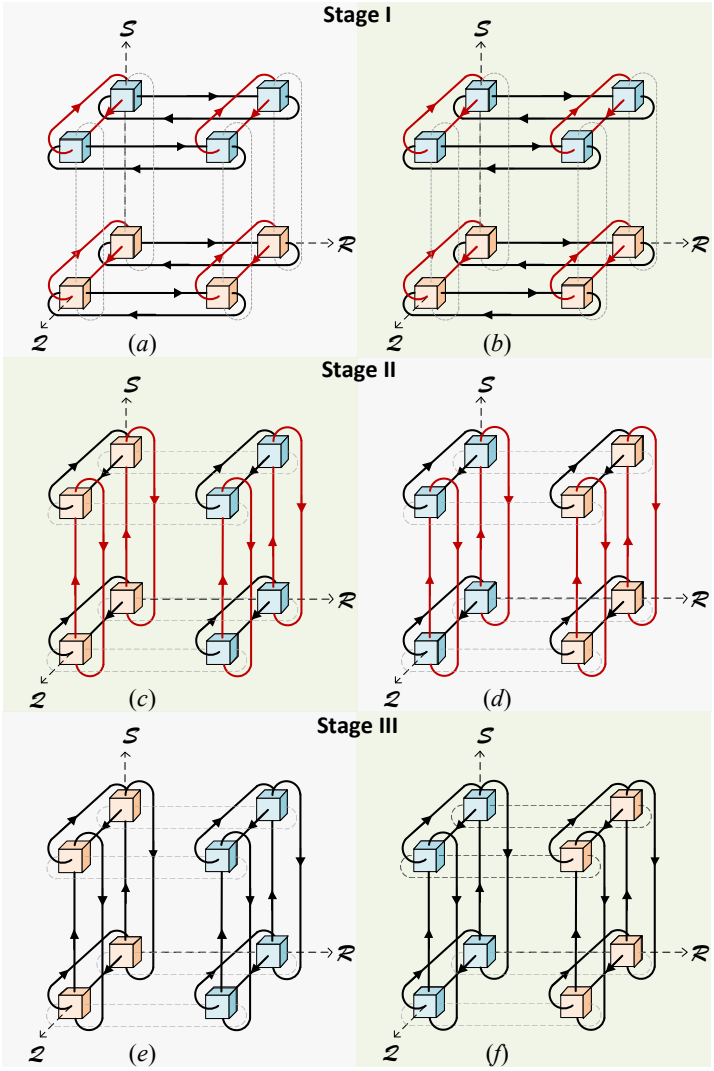

Figure 9: An orbital or circular data communication between nodes for the 3D IDXT.

The first two stages implement the set of P space-independent 2D FDXTs on P parallel along $\mathcal{R}$-axis (orbit) slabs, $0 \le \mathcal{R} < P$, with the $P \times P$ toroidally interconnected computer nodes $\{\mathrm{CN}(\mathcal{Q}, *, \mathcal{S})_{\mathcal{R}} : 0 \le \mathcal{Q}, \mathcal{S} < P\}$ each (see Fig. 8 for the stages I and II). Totally, $2P$ "compute-and-roll" time-steps are needed for each $\mathcal{R}$-th slab-of-nodes to independently implement a 2D FDXT (stages I and II) of the $\mathcal{R}$-th slab-of-cubes $\mathbb{X}(N_1, N_3)_{\mathcal{R}=N_2}$.

As it was discussed, to make data distribution of the intermediate cubical tensor $\ddot{\mathbb{X}}$ consistent with the required distribution of this tensor on Stage III, each $\mathcal{R}$-th 2D FDXT is implemented in time-skewed manner such that after $2P$ time-steps, the $b \times b \times b$ blocks of a cubical tensor $\ddot{\mathbb{X}}$ will be correctly distributed among all P^3 computer nodes in a 3D torus network. This required initial, intermediate and final data distributions are controlled by the modular (orbital) scheduling function $\mathcal{T}$. By using this scheduling, no data redistribution is needed during processing and the final P independent sets of 1D FDXTs in Stage III can be immediately started by P, now parallel along $\mathcal{S}$-axis $(0 \le \mathcal{S} < P)$, slabs of the $P \times P$ toroidally interconnected computer nodes $\{\mathrm{CN}(\mathcal{Q}, \mathcal{R}, *)_{\mathcal{S}} : 0 \le \mathcal{Q}, \mathcal{R} < P\}$ each (see Stage III in Fig. 8 for $P = 2$).

4.2 3D Block Inverse Transform

An orbital computing of the 3D *inverse* DXT is implemented as rolling back of the described above 3D forward DXT where skewed (out-of-order) distribution of the final cubical tensor $\dddot{\mathbb{X}}$ among computer nodes will correspond to the initial three-way tensor distribution for the 3D IDXT (see also Fig. 9 for $P = 2$). Obviously, the change-of-basis matrices $C_{\rm I}$, $C_{\rm II}$, and $C_{\rm III}$ are the same as for the 3D FDXT.

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

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

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

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

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

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

Note that at the last stage of the 3D IDXT, an accumulation or $\mathcal{T}$-summation is implemented inside each computer node. After $3P$ "compute-and-roll" time-steps, i.e. after completion of a 3D IDXT, the resulting cubical tensor $\mathbb{X}$ will be distributed in a 3D processor space in the canonical (in-order) layout. As a consequence, the frequently required by many real-world applications Forward/Backward looping, 3D FDXT $\rightleftharpoons$ 3D IDXT, can be implemented on a 3D torus network of computer nodes in a very natural and efficient way.

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 network of computer nodes requires totally $3P$ block "compute-and-roll" time-steps, where $P = N/b$ and $1 \le b \le N$ is the blocking factor. Each "compute-and-roll" time-step involves the left or right multiplication of a 3D $b \times b \times b$ tensor by a $b \times b$ coefficient matrix with a possible matrix transpose. In turn, each of this tensor-by-matrix multiplication requires execution by every computer node exactly b^4 scalar fused multiply-add (`fma`) operations and rolling of either $b^3 + b^2$ or $2b^3$ scalar data. Each node implements a data rolling by concurrently sending and receiving data to/from two predefined nearest-neighbor nodes. Actually, each computer node doesn't "feel" the size of a network, because at each time-step only local computing and topologically nearby (contact) data movement operations are implemented which can be done independent from the total size of a system. In this respect and by analogy with Strassen's 2×2 matrix multiplication

algorithm [2], it is enough to organize a parallel data processing for the smallest, but fundamentally basic, $2 \times 2 \times 2$ case which can be recursively scaled then to the bigger sizes with a proportional escalating of algorithm's time and space complexity.

The required memory space in one computer node, which is proportionally to $4b^3 + 3b^2$, might be sufficiently reduced by using the well-known "compute-in-place" approach. Note that for simplicity, we assumed here that there is no shared memory between nearest-neighbor nodes, i.e., after computing, a local data must be exchanged (rolled) by using a message-passing. However, if upgrading of data (a "compute" part) precedes a movement of this data (a "roll" part) and there exist a shared memory between nearest-neighbor nodes then data upgrading can be directly done in this shared memory, which will replace a time-consuming, message-passing, "roll" part of this data by quick inter-node synchronization. For the other unchangeable, but moved data, "roll" part(s) can be implemented concurrently with a "compute" part (see description of the 3D DXT algorithms above), making all nearest-neighbor data movements totally overlapped with a local computing. This way of sharable data exchange should be very promising for massively-parallel solution of relatively small problems.

If blocking factor $b = 1$, i.e. $P = N$, our 3D transform algorithms achieve the highest or extreme degree of parallelization with a total processing time equals to $3N$ scalar "compute-and-roll" steps on an $N \times N \times N$ torus array of very simple computer nodes. This fastest, fine-grained implementation will require an `fma`-unit in each computer node to be with one-step or one-cycle latency which is possible for fixed-point arithmetic, like in "Anton" supercomputer [49]. Moreover, only in this extreme case, the size of required in each computer node memory is independent from the size of problem. For floating-point arithmetic, however, the pipelined `fma`-units usually require a few cycles of latency as in today advanced microprocessors [42, 20, 10, 8, 38, 37], which force the size of blocking factor b to be more then one in order to hide this pipeline latency by concurrent execution of a few *independent* `fma`-instructions on the same `fma`-unit (usually, $b \geq 4$).

If, in contrast, a blocking factor $b = N$, i.e. $P = 1$, the proposed implementation of a 3D transform is degenerated to a serial algorithm which is executed on a single computer node in $3N^4$ `fma`-steps. It is clear that independently on the size of the blocking factor $b \in [1, 2, ..., N]$ and, therefore, degree of parallelization, any implementation of a 3D transform in the form of three-way tensor-by-matrix product requires execution of $3N^4$ `fma`-operations.

5 Conclusions

We have systematically designed massively-parallel and extremely-scalable orbital block algorithms to efficiently perform any 3D separable transform and its inverse in a 3D network of toroidally interconnected computer nodes. The designed orbital algorithms preserve all advantages of the previously popular systolic array processing [28, 26] such as simplicity, regularity, nearest-neighbor data movement, scalable parallelism, pipelining, etc. However, unlike systolic processing, our orbital computing keeps all initial, intermediate, and final data inside the same 3D torus array processor allowing a simultaneous access to all data, reusing data by its circulation, and an effective space-time chaining of different parts of processing.

Our basic 3D DXT algorithms are structurally represented in a 4-dimensional computational index space which is partitioned into the 3D disjoint sets of index points. A modular (orbital) scheduling of these index points for execution and data movement as well as appropriate 4D→3D projections allow to map a complex three-stage data-dependent processing into a single 3D physical network of toroidally interconnected

[2] In Strassen's algorithm, the time of a 2×2 matrix-by-matrix multiplication is decreased by reducing the number of multiplications in the price of increasing the number of more cheap addition/subtractions operations and irregular data access; in our case, however, we directly reduce this time by overlapping (indivisible) multiply-add operations combined with a local and regular data movement.

nodes. A multidimensional torus interconnection has always been widely used in the past and present distributed-memory supercomputers, for example, in Cray T3E (3D torus) [44], IBM Blue Gene/L (3D torus) [1] and Blue Gene/Q (5D torus), D.E. Shaw Research "Anton" (3D torus) [49], Fujitsu K-computer (6D torus) [18, 3]. It is expected that direct implementation of our coarse-grained $(b \gg 1)$ forward and inverse three-dimensional DXT algorithms on supercomputers with a multidimensional torus interconnect would be beneficial with respect to other existing implementations which are based on a 1D or 2D data decomposition. We reserve this porting and performance evaluation as a future work. Note that instead of using widely available and deeply optimized "fast", but internally serial, DXT algorithms (as in 1D or 2D decomposition approaches), our 3D decomposition technique allows to use within each node, also widely available and practically even more deeply optimized and, therefore, more fast and efficient, GEMM-based algorithms [29, 36].

Moreover, our extremely scalable solution $(b \cong 1)$ for a fine-grained 3D DXT implementation on a cubical "mesh-of-tori" network [3] [43] of very simple execution nodes (each with one multiply-add unit and a few, inter-node connected, registers) is perfectly suited for forthcoming 3-dimensional VLSI technology [25], Giga-size sensor arrays with parallel read-out [21] and many real-world applications (including embedded) with multidimensional data. Note that preliminary, but promising 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 [23].

One of the notable characteristics of our unified and extremely scalable 3D DXT algorithms is in the possibility to implement concurrently a few distinct transforms on the same cubical data by using different matrices of transform coefficients. Additionally, if it is required by application, the different transforms for different cubical data dimensions can be easily implemented at the same time.

References

[1] 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

[2] 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.

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

[4] B. W. Bader and T. G. Kolda, "Algorithm 862: Matlab tensor classes for fast algorithm prototyping," *ACM Trans. Math. Softw.*, vol. 32, pp. 635–653, December 2006. [Online]. Available: http://doi.acm.org/10.1145/1186785.1186794

[5] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276 –292, feb. 2005.

[3] i.e. a tile-based regular network without "long" wrap-around connections.

[6] E. Brachos, "Parallel FFT Libraries," Master's thesis, The University of Edinburgh, 2011. [Online]. Available: http://www.brachos.gr/files/E-Brachos_FFT.pdf

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

[8] 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.

[9] 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.

[10] 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

[11] 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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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".

[17] ——. (2012, June) FFTW for version 3.3.2. [Online]. Available: http://www.fftw.org

[18] FUJITSU, "K computer," *Website*, June, 2012. [Online]. Available: http://www.fujitsu.com/global/about/tech/k/

[19] G.H.Golub and C. Loan, *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, 1996.

[20] 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

[21] E. H. Heijne, "Gigasensors for an Attoscope: Catching Quanta in CMOS," *IEEE Solid-State Circuits Newsletter*, vol. 13, pp. 28–34, 2008.

[22] C. Hillar and L.-H. Lim, "Most tensor problems are NP hard," 2009. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0911.1393

[23] 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

[24] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[25] 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.

[26] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982. [Online]. Available: http://dx.doi.org/10.1109/MC.1982.1653825

[27] H. Kung and C. Leiserson, "Systolic array apparatuses for matrix computations," Patent US 4,493,048, Jan. 8, 1985. [Online]. Available: http://www.patentlens.net/patentlens/patent/US_4493048/

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

[29] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMMs for Fermi," University of Tennessee, Tech. Rep. UT-CS11-671, April 2011. [Online]. Available: http://icl.cs.utk.edu/news_pub/submissions/lawn245.pdf

[30] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl*, vol. 21, pp. 1253–1278, 2000.

[31] D. Lavenier, P. Quinton, and S. Rajopadhye, "Advanced systolic design," in *Digital Signal Processing for Multimedia systems*, ser. Signal Processing Series. Marcel Dekker, 1999, ch. 23, pp. 657–692.

[32] N. Li and S. Laizet, "2DECOMP&FFT - A Highly Scalable 2D Decomposition: Library and FFT Interface," in *Cray User Group 2010 conference*, 2010, pp. 1–13. [Online]. Available: http://www.2decomp.org/pdf/17B-CUG2010-paper-Ning_Li.pdf

[33] L.-H. Lim, "Numerical Multilinear Algebra: a new beginning?" in *Matrix Computations and Scientific Computing Seminar*, 2006, pp. 1–51. [Online]. Available: http://galton.uchicago.edu/~lekheng/work/mcsc1.pdf

[34] 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

[35] Q. Lu, X. Gao, S. Krishnamoorthy, G. Baumgartner, J. Ramanujam, and P. Sadayappan, "Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions," *J. Parallel Distrib. Comput.*, vol. 72, no. 3, pp. 338–352, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.09.006

[36] 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.

[37] 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

[38] 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.

[39] I. V. Oseledets, D. V. Savostyanov, and E. E. Tyrtyshnikov, "Linear algebra for tensor problems," *Computing*, vol. 85, pp. 169–188, July 2009. [Online]. Available: http://dx.doi.org/10.1007/s00607-009-0047-6

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

[41] M. Pippig and D. Potts, "Scaling Parallel Fast Fourier Transform on BlueGene/P," Jülich Supercomputing Centre, Tech. Rep. FZJ-JSC-IB-2010-03, May 2010.

[42] 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

[43] 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

[44] 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.

[45] S. G. Sedukhin and I. S. Sedukhin, "Systematic approach and software tool for systolic design," in *CONPAR*, ser. Lecture Notes in Computer Science, B. Buchberger and J. Volkert, Eds., vol. 854. Springer, 1994, pp. 172–183.

[46] 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

[47] E. Solomonik, J. Hammond, and J. Demmel, "A preliminary analysis of cyclops tensor framework," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-29, Mar 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-29.html

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

[49] 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