Alexandria Engineering Journal (2015) 54, 473–480

Alexandria University

Alexandria Engineering Journal

www.elsevier.com/locate/aej
www.sciencedirect.com

ORIGINAL ARTICLE

NDPA: A generalized efficient parallel in-place N-Dimensional Permutation Algorithm

Muhammad Elsayed Ali [a,*], **Saleh El-shehaby** [b], **Mohamed S. Abougabal** [a]

[a] *Department of Computers and Systems Engineering, Alexandria University, Alexandria, Egypt*
[b] *Department of Biomedical Engineering (MRI), Alexandria University Alexandria, Egypt*

Received 31 December 2012; revised 15 January 2015; accepted 8 March 2015
Available online 30 April 2015

KEYWORDS

3D matrix transpose;
N-dimensional volume transpose;
Many-core systems;
GPU programming;
CUDA parallel programming

Abstract N-dimensional transpose/permutation is a very important operation in many large-scale data intensive and scientific applications. These applications include but not limited to oil industry i.e. seismic data processing, nuclear medicine, media production, digital signal processing and business intelligence. This paper proposes an efficient in-place N-dimensional permutation algorithm. The algorithm is based on a novel 3D transpose algorithm that was published recently. The proposed algorithm has been tested on 3D, 4D, 5D, 6D and 7D data sets as a proof of concept. This is the first contribution which is breaking the dimensions' limitation of the base algorithm. The suggested algorithm exploits the idea of mixing both logical and physical permutations together. In the logical permutation, the address map is transposed for each data unit access. In the physical permutation, actual data elements are swapped. Both permutation levels exploit the fast on-chip memory bandwidth by transferring large amount of data and allowing for fine-grain SIMD (Single Instruction, Multiple Data) operations. Thus, the performance is improved as evident from the experimental results section. The algorithm is implemented on NVidia GeForce GTS 250 GPU (Graphics Processing Unit) containing 128 cores. The rapid increase in GPUs performance coupled with the recent and continuous improvements in its programmability proved that GPUs are the right choice for computationally demanding tasks. The use of GPUs is the second contribution which reflects how strongly they fit for high performance tasks. The third contribution is improving the proposed algorithm performance to its peak as discussed in the results section.

1. Introduction

MT (Matrix transpose) operation is used frequently in many multimedia and high performance applications. Therefore, the use of a fast MT operation results in a shorter execution time of these applications. Higher dimensional volume transpose is used heavily in oil industry, nuclear medicine such as both 3D and 4D PET (Positron Emission Tomography) [1].

* Corresponding author.
E-mail addresses: muhammad.elsayed.ali@gmail.com (M.E. Ali), sshehaby@mcit.gov.eg (S. El-shehaby), mohamed.abougabal@alexu.edu.eg (M.S. Abougabal).
Peer review under responsibility of Faculty of Engineering, Alexandria University.

http://dx.doi.org/10.1016/j.aej.2015.03.024

It is used also in CT (computed Tomography) imaging on 3D scanners. Protein folding is one of the major fields that use higher dimensional matrix operations [2] like the transpose discussed in this paper. Media production such as filming, 4D cinema and 3D TV is a great example of such industry. It uses heavily both the 3D and 4D FFT (Fast Fourier Transform) that depend in first place on 3D and higher dimensional transpose. In Business Intelligence, any OLAP (On Line Analytical Processing applications) system has the concept of an OLAP cube (also called a multi-dimensional cube or a hypercube).

The OLAP cube uses multi-dimensional transposition or permutation in the processing and storing of its data. In general, the transpose operation can be implemented using either logical or physical approaches. In logical permutation, no physical data reordering/swapping is required, only the components of the element subscript are swapped according to the permutation string. For example, in a 3D volume, when permuting an element whose subscript is (i,j,k) with permutation string ZYX along the three axes (x,y,z), then the element at subscript (k,j,i) should be provided to the end user as the result of the permutation of the original element. This eliminates the overhead of physical reordering, but it deals with a data block of one element which increases the per-access data element cost. In physical reordering, data elements are physically moved (single element at a time) to their new locations inside the volume. Single element physical movement does not achieve more memory bandwidth. In addition, there is a need for an out-of-place swapping in memory. The authors in [3] attempted to solve these problems by proposing a novel in-place 3D transpose algorithm. There are some limitations of the base algorithm like being restricted to perform transposition in three dimensions only. In addition, it is implemented on Cell BE processor which is not straightforward in its programmability. Cell BE includes only maximum of seven symmetric multiprocessors. In this paper, these limitations are overcome through proposing a general efficient algorithm that performs the transposition/permutation in any number of dimensions. This is, as of today, the first known generalized parallel in-place n-dimensional permutation algorithm. As a proof of concept, the proposed algorithm is tested on volumes of dimensions up to 7D. In addition, the proposed algorithm is implemented on NVidia GPU that has many processing cores (128 cores) and is simpler in its programmability. The performance of the proposed algorithm is measured and boosted to its peak as will be discussed later.

The remaining of the paper is organized into five sections. A survey of related research to transpose algorithms is provided in Section 2. A proposed extension to the algorithm reported in [3], is described in Section 3. The results and performance gain in are presented in Section 4. Finally, the conclusions and possible future work are provided in Section 5.

2. Related work

Beyond the year of 2008, there were some transposition approaches in the literature focusing only on the physical transpose level while the other approaches were focusing only on logical transpose level [3].

In the literature, 3D transposition approaches focus on physical transpose. Eleftheriou et al. [4] described an algorithm for multi-dimensional transpose, where the transposition is performed along one axis at a time, with partial processing and communication among processors. Then transposition is performed on another axis.

Wapperoma et al. [5] proposed a parallelization algorithm for 3D FFT. That involves a transpose operation. The proposed algorithm divides the 3D data into planes, splitting each plane among multiple processors. Moreover, no logical transposition is performed. I-Jui Sung et al. [6] proposed a four-stages parallel in-place physical transpose algorithm for rectangular 2D matrices, where each permutation stage is done using elementary tile-wise transposition.

It is also worth noting that there are other approaches for distributed memory architecture such as Choi et al. [7] and Cohl et al. [8], but they are not relevant to the target domain of shared-memory architecture typical in the multicore paradigm.

The base algorithm developed in [3] focused on a combination of both physical and logical transposition levels. In physical transpose approaches, accessing the transposed data many times provides low overhead. It leads to duplication of used memory because of the out-of-place data reordering. Accessing of non-contiguous data blocks leads to lower performance. Logical transpose approaches have no memory management problems but there is high overhead in repetitive accesses of transposed data. Its performance on parallel memory systems is poor, because of accessing data blocks of singular elements.

The base algorithm in [3] differs from the algorithm proposed by Eleftheriou et al. [4] in dividing the work among processors, each transposes along all axes until the full transpose is completed, without requiring inter-process communications. Moreover, it performs the transpose 'in-place' using hybrid/integrated logical and physical transpositions.

It also differs from the algorithm proposed by Wapperom et al. [5] in the decomposition of a 3D volume into smaller sub-volumes, each processed by a processor.

Mixing both transposition approaches, used in [3], is the trend being followed in this paper. The proposed algorithm is intended for many-core SIMD hardware architecture (typically implemented on GPUs).The 3D transpose algorithm in [3] is intended for multi-core SIMD hardware architecture (typically implemented on Cell BE processor[2]).

2.1. The base transpose algorithm [3]

The transpose algorithm is concerned about transposing only 3D volumes of data; each is called 'cuboid'. The cuboid has dimensions L x M x N along the axes X, Y and Z, which need not to be equal. As mentioned above, the algorithm mixes both logical and physical transpose approaches. Physical transpose is applied on elements' level, and logical transpose is applied on larger blocks of data called cubes. A cube has equal dimensions of ***p x p x p***, where ***p*** divides all the cuboid dimensions L, M, N. Thus, the original cuboid can be considered as a 3D grid of cubes [3]. In the algorithm, detailed below, any element (x, y, z) is accessed by locating its cube index first, then its location inside the cube, respectively as follows:

1. Cube index $(i, j, k) = (\lfloor x/p \rfloor, \lfloor y/p \rfloor, \lfloor z/p \rfloor)$;
2. Data element is accessed via (i mod p, j mod p, k mod p).

Algorithm 1: The base 3D transpose algorithm [3].

With reference to Fig. 1 below.
1 – Divide the cuboid into bars, each of dimensions $p \times p \times L$.
2 – Distribute bars evenly among working processors.
3 – Each processor loads bars into its local memory.
4 – Each processor chops back the bars into cubes into its local memory.
5 – Each processor transposes each cube into its local memory (using three consecutive 2D primitive transpose operations).
6 – Each processor combines back the bar from local memory into the global shared memory in its original place.

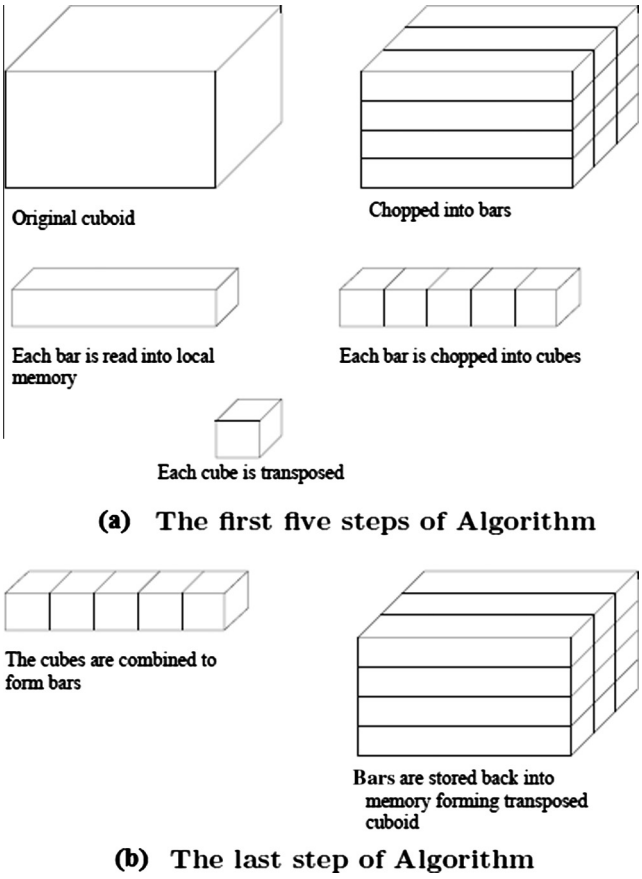

Figure 1 The base 3D transpose algorithm steps (replicated from [3]).

In the following section, a detailed discussion of the proposed n-dimensional algorithm extension will be introduced.

3. N-dimensional permutation algorithm (NDPA)

The proposed algorithm is a general case of the base algorithm in [3] presented above. It is implemented on NVidia GTS 250 graphics processing card containing 128 cores. It exploits the parallel nature of the SIMD architecture [9–11] by working on linearized form of the input data volume. In other words, non-linear n-dimensional volume can be considered as if it was a linear vector. This is the first important idea in the generalization of the original algorithm.

Linearization means that there is a need for a mapping, between elements subscripts in n-dimensional input volume, and their equivalent linear indexes. This mapping will be discussed below in detail. The second important idea of generalization is dividing the non-linear n-dimensional volume logically to some smaller equally sided subspaces; each has n dimensions. This results in n-dimensional grid of n-dimensional subspaces; each of them called a cell. The proposed algorithm extension in [2] follows the same methodology of the original algorithm in [3]. It does not assume any restrictions on data in return for the generalization. These restrictions can include but not limited to "values should not be zeros", "values should follow certain statistical distribution" or "a value should have certain length in bytes". This ensures that the same data sets used in verifying the base algorithm should work fine with the proposed extension regardless of using different hardware. The following subsection provides more details about the terminology used across the paper regarding the proposed extension.

3.1. Proposed algorithm terminology

Some terms should be defined to explain how elements are accessed and permuted within their containers. Those terms fell into two categories. The first category is concerned with data block units. The second category is concerned with both offsets and intra-offsets among block units and whole volume. Those offsets are called strides or scan vectors. The word scan reflects the way of calculating those vector elements where each element's calculation depends on a previous scanned element. The required terminology is defined below.

- *Scan*

Scan is used heavily in converting element's linear index (as if element is located into a vector) to its corresponding element's subscript (as if element is located into a matrix/volume) and vice versa. The scan vector for any dimensions' vector (Dims) of length N is calculated as follows:

$$Scan[0] = 1$$

$$\forall_{i=1}^{N}(Scan[i] = Scan[i-1] * Dims[i-1])$$

The element's linear index is calculated from its subscript as follows:

$$LinearIndex = \sum_{i=0}^{N}(Subscript[i] * scan[i]).$$

The element's subscript is calculated from its linear index as below:

$$\forall_{i=N-1}^{0}\left[Subscript[i] = \frac{Linear\ Index}{Scan[i]},\ Linear\ Index = Linear\ Index\%scan[i]\right]$$

The above formulas are the author's simple compact form of what is mentioned into "The Art of Computer Programming" book [12].

- *Cuboid*

Cuboid is n-dimensional volume having dimensions not necessarily to be equal.

- *Cell*

Cell is n-dimensional subspace of the cuboid. All its dimensions (sides) must be of equal length that divides all cuboid

dimensions. Cuboid can be considered as n-dimensional grid of n-dimensional cells.

- *Bar*

Bar is a group of consecutive cells along with the cuboid *x*-axis direction.

- *Cell-based kernel*

It is the first complete version of the proposed algorithm. The permutation is done on every thread by reading a cell, permuting the cell and then writing it back to the device global memory.

- *Bar-based kernel*

It is the second complete modified version of the proposed algorithm. A bar is the unit of data access instead of the cell in the cell-based kernel. The permutation is done on every thread by reading a bar, permuting the bar and then writing it back to the device global memory. This required more computations to detect the bar and deal with it.

- *Rote learning bar-based kernel*

It is the third and final parallel kernel of the proposed algorithm whose performance is the best among all serial and parallel kernels including the Matlab benchmark. This improvement in performance happens via exploiting the rote learning technique. The rote learning is a dynamic programming technique that boosts the performance through remembering the most frequent complex computations into lookup tables. This is achieved by caching the most frequent computations in lookup tables that are small to some extent and stored into the device texture memory. Texture memory is a portion of a very slow global device memory but it is cached and optimized well for 2D data like 2D images [2]. Rote learning saves the time wasted in performing too frequent complex computations, and thus boosted the performance to its peak.

The main steps of cell-based kernels are shown in algorithm 2 and Fig. 2.

Algorithm 2: Cell-based ND transpose algorithm.

1. Divide the cuboid into n-dimensional cells
2. Distribute cells evenly among working GPU cores keeping work load balanced.
3. Each GPU core loads one cell at a time into its local shared memory.
4. Each GPU core transposes each cell into its local shared memory in a simple out-of-place manner using 3D primitive transpose operation.
5. Each GPU core writes back the permuted cell from local shared memory into the global device memory in its original place.

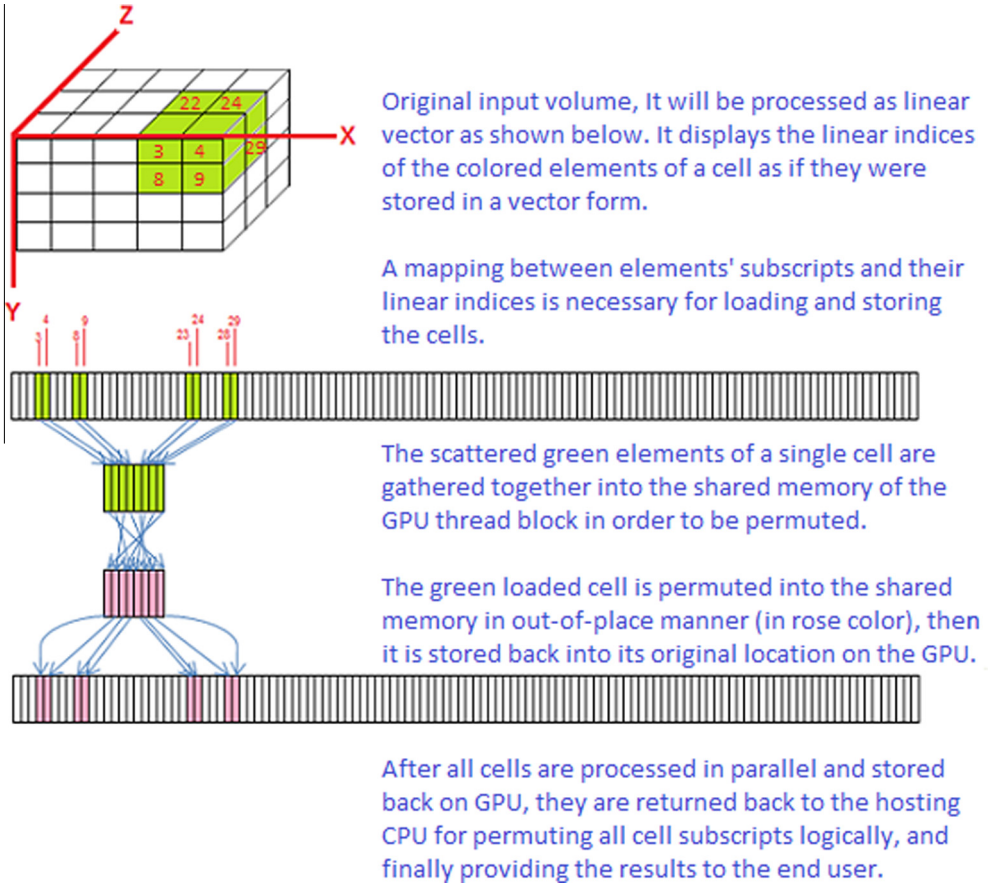

Figure 2 Proposed ND cell-based transpose kernel steps [2].

The main steps of bar-based kernels are shown in algorithm 3 and Fig. 3.

Algorithm 3: Bar-based ND transpose algorithm [2].

1 – Divide cuboid into n-dimensional bars and distributed evenly among GPU cores.
2 – Each core loads one bar at a time from global device memory and stores it into its thread block's shared memory.
3 – Each core divides the bar into cells to be loaded one at a time into the shared memory.
4 – Each core permutes the cell into shared memory in a simple out-of-place manner.
5 – A permuted cell will be stored back into its original location into the bar in shared memory.
6 – Each core combines back the bar as group of permuted cells from shared memory and writes it back to the global device memory in its original place. After permutation is finished, cells indexes must be swapped or permuted logically to access fully permuted volume.

4. Experimental results

This section introduces some permutation experiments on 3D, 4D, 5D, 6D and 7D volumes. Various volume sizes are inspected starting from 512 elements and up to 124 million of elements. In addition, this section explains the effect of data block size on performance via blocks of sizes 8, 16, 32, 64, 128, 243 and 256. The focus will be much on the total execution time of all the three implemented kernels and how fast a kernel is compared to the others [13–18]. The development environment settings are described below.

1. *Host environment*
   - Dual Core PC, 3.4 GHz.
   - GBs of DDR2 ram.
   - Windows seven Ultimate, 32 bits.
   - Microsoft Visual Studio 2008 (C++).
   - DirectX 11.

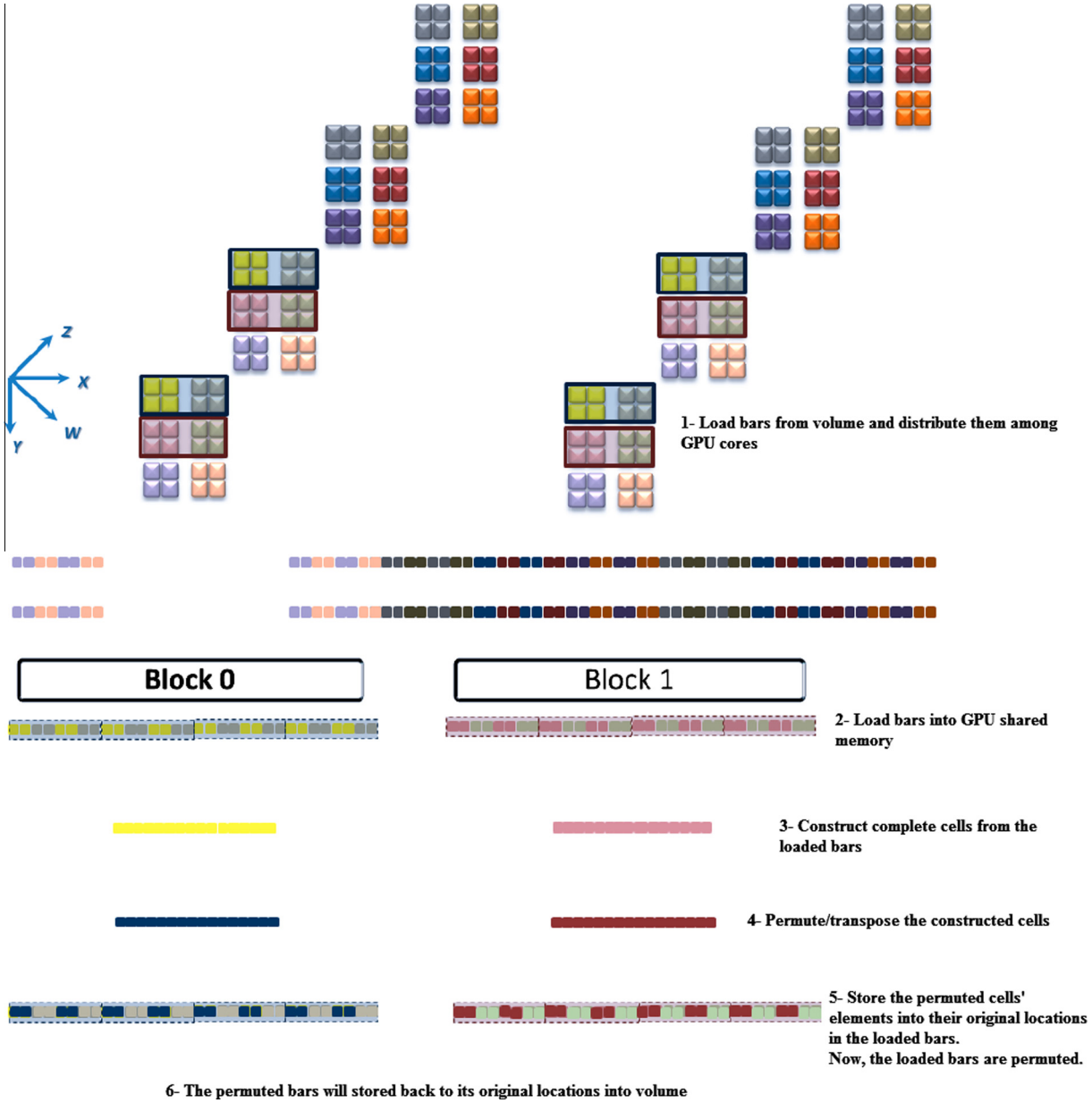

Figure 3 Proposed n-dimensional bar-based transpose kernel (4D example) [2].

2. *Device environment*
   - NVidia GeForce GTS250 GPU with 128 cores.
   - 1 GBs of DDR3 ram.
   - CUDA 3.2 environment (Toolkit and SDK).
   - NVidia display driver 260.93.

4.1. Kernels Speedup tables

A comparison among all implemented kernels was performed to express the performance in terms of how GPU kernel is faster than CPU kernel (Speedup).

Speedup is defined in [2] as, Speedup

$$= \frac{\text{Serial CPU Kernel's Total Execution Time}}{\text{Parallel GPU Kernel's Total Execution Time}} \quad (1)$$

Measuring the parallel kernels' speedup against Matlab follows Eq. (1). Total execution time is the average of number of runs. It was found that 1000 runs are sufficient for the standard deviation to stabilize at 5–7%.

Parallel kernels are abbreviated as follows:

1. *CB* denotes cell-based kernel.
2. *BB* denotes bar-based kernel.
3. *RLBB* denotes rote learning bar-based kernel.

Tables 1–5 present the speedup results defined in Eq. (1) for different block sizes and volume sizes for 3D, 4D, 5D, 6D and 7D dimensions, respectively [2].

The execution times which result in these speedups above (Tables 1–5) are plotted in logarithmic scale as depicted in Fig. 4 below.

Close study of the results depicted in Tables 1–5 and Fig. 4 indicates performance improvement, especially with enormous increase of data volume size. One reason for that is the parallel nature of the proposed kernels, which utilize the parallel architecture of the GPU.

Table 4 6D Speedup of parallel kernels vs. serial kernels.

Speed up	Speed up vs. serial kernel			Speed up vs. serial Matlab		
Block size	64			64		
Volume size	CB	BB	RLBB	CB	BB	RLBB
1,000,000	41.83	19.47	65.16	198.09	92.21	308.61

Table 5 7D Speedup of parallel kernels vs. serial kernels.

Speed up	Speed up vs. serial kernel			Speed up vs. serial Matlab		
Block size	128			128		
Volume size	CB	BB	RLBB	CB	BB	RLBB
10,000,000	54.80	12.97	88.68	305.44	72.31	494.26

Table 1 3D Speedup of parallel kernels vs. serial kernels at various volume and block sizes.

Speed up	Speed up vs. serial kernel						Speed up vs. serial Matlab					
Block size	8			64			8			64		
Volume size	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB
512	0.09	0.08	0.09	0.10	0.09	0.10	0.02	0.02	0.02	0.02	0.02	0.02
4,096	0.80	0.66	0.78	0.64	0.52	0.62	0.13	0.11	0.13	0.13	0.11	0.13
32,768	4.99	4.46	5.46	4.55	3.90	4.48	0.69	0.62	0.76	0.78	0.67	0.77
262,144	15.17	17.34	26.35	20.36	18.47	23.83	2.18	2.49	3.78	3.66	3.32	4.28
2,097,152	17.66	28.63	55.09	31.56	32.25	51.75	2.69	4.36	8.40	6.97	6.10	9.79

Table 2 4D Speedup of parallel kernels vs. serial kernels at various volume and block sizes.

Speed up	Speed up vs. serial kernel						Speed up vs. serial Matlab					
Block size	16			256			16			256		
Volume size	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB
4096	0.86	0.65	0.75	0.74	0.55	0.69	0.15	0.12	0.13	0.15	0.11	0.14
65,536	9.09	7.62	10.51	9.05	6.06	9.34	1.45	1.22	1.68	1.64	1.10	1.69
1,048,576	26.31	29.43	52.42	33.43	22.32	52.61	4.19	4.69	8.36	5.96	3.98	9.38
4,194,394	28.42	27.78	64.89	39.53	17.62	68.30	4.54	4.44	10.37	6.85	3.05	11.83

Table 3 5D Speedup of parallel kernels vs. serial kernels at various block sizes.

Speed up	Speed up vs. serial kernel						Speed up vs. serial Matlab					
Block size	32			243			32			243		
Volume size	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB	CB	BB	RLBB
112,021,056	43.42	39.87	88.33	47.08	29.97	88.27	213.28	195.84	433.92	242.69	154.47	455.02

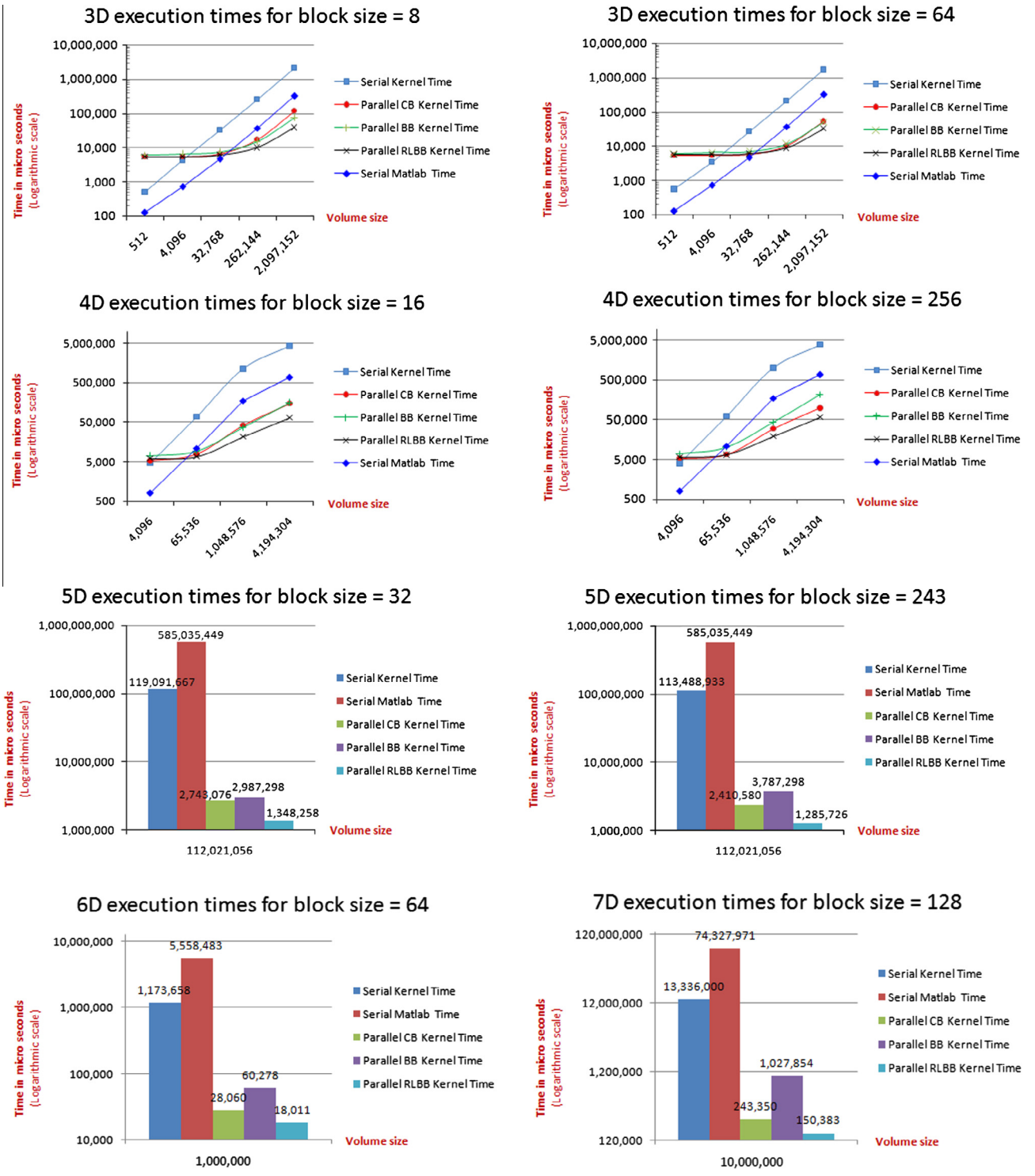

Figure 4 N-dimensional volume permutation experiments [2].

Cell-based kernel introduces low memory bandwidth because of the frequent read/write accesses to the device global memory.

Bar-based kernel overall performance is better than the cell-based kernel performance. This improvement is expressed in terms of high speed up and shorter execution time. The reason is the bar as a unit of data access is used instead of a cell; this improves the memory bandwidth and minimizes the execution time. The use of bar as a data access unit requires much complex computations to be done frequently. These complex computations prevent the overall performance from being the ultimate. This observation is captured through profiling the bar-based kernel performance via NVidia profiler.

Rote learning bar-based kernel overcomes the problem of time consumed in frequent complex computations needed to access the bar. This is achieved by the use of the rote learning technique. This dynamic programming technique helps in caching the frequent complex computations results; instead of re-computing them each time; into a look table that is stored in the GPU texture memory. The texture memory is a portion of very slow global device memory but it is cached and optimized well for 2D data like 2D images [2]. This memorization improves the kernel's performance to be the best among all developed parallel kernels discussed earlier.

5. Conclusion and future work

In this paper, an extension to an efficient in-place 3D transpose algorithm [3] is presented. The proposed algorithm performs permutation in N-dimensions. It is a generalization of the base algorithm in [3]. The proposed algorithm keeps the benefits of the base algorithm without introducing any extra restrictions or special conditions. It performs both physical and logical permutations on software-managed memory many-cores SIMD architecture called GPU. It exploits the SIMD instructions on such processors. Three parallel kernels are discussed; cell-based, bar-based and rote learning bar-based kernels. They show the positive effect of shared memory and *rote learning programming* on the overall performance. This improvement has been achieved in two steps. The first step is caching more data into shared memory to reduce the frequent expensive accesses to device global memory. The second step is reducing the execution time through storing most of the frequently used computations in lookup table. This performance improvement was evident from the experimental results. Finally, the experimental results verified the utility of GPUs and how useful they were for the data intensive and scientific parallel applications. There are too many fields of science and technology that require the processing power of GPUs such as bioinformatics, computational fluid dynamics, supercomputing centers and seismic exploration, imaging, computational finance and geographic information systems [19]. Future work will consider improving the algorithm performance by tuning the GPU through memory alignment, streaming, etc. The use of recent hardware like Kepler [20] and Tesla K80 [21], the world's recent and fastest GPU (24 GB of GDDR5 RAM, 4992 cores on two internal chips), would help a lot in improving the performance through introducing large amount of shared memory and on-chip cores. An important future extension is to add support for n-dimensional volume padding when volume dimensions are not dividable by cell side. Final suggestion is to integrate the proposed algorithm with an application that depends on n-dimensional volume permutation such as seismic applications, medical imaging, digital signal processing, multimedia industry, OLAP applications and protein folding or any suitable vital industrial application.

Acknowledgements

The authors acknowledge the very useful discussions with Prof. Dr. Ahmad El-Mahdy, one of the coauthors of [3]. This work was achieved as part of the M.Sc. degree of the first author.

References

[1] B. Hanounik, X. Hu, Linear-time matrix transpose algorithms using vector register file with diagonal registers, in: The 15th International Parallel & Distributed Processing Symposium, IPDPS '01, Washington, DC, USA, 2001, pp. 10036.
[2] M.S. Ali, A generalized efficient parallel in-place N-dimensional transpose algorithm, M.Sc. Thesis, Computers and Systems Engineering Department, Faculty of Engineering, Alexandria University, Alexandria, Egypt, 2011.
[3] A. El-Moursy, A. El-Mahdy, H. El-Shishiny, An efficient in-place 3D transpose for multicore processors with software managed memory heirarchy, in: The 1st International Forum on Next-Generation Multicore/Manycore Technologies, New Yourk, NY, USA, 2008, pp. 10:1–10:6.
[4] M. Eleftheriou, B.G. Fitch, R.S. Germain, A. Rayshubskiy, T.J.C. Ward, Multi-dimensional transpose for distributed memory network, US Patent 0010181, January 12, 2006.
[5] P. Wapperom, A.N. Beris, M.A. Straka, A new transpose split method for three-dimensional FFTs: performance on an origin2000 and alphaserver cluster, Parallel Comput. 32 (2006) 1–13.
[6] I.-J. Sung, J. Gomez-Luna, J.M. Gonzalez-Linares, N. Guil, W.-M.W. Hwu, In-place transposition of rectangular matrices on accelerators, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, 2014, pp. 207–218.
[7] J.J. Dongarra, D.W. Waler, J. Choi, Parallel matrix transpose algorithms on distributed memory concurrent computers, in: Scalable Parallel Libraries Conference, 1993.
[8] H.S. Cohl, X.-H. Sun, J.E. Tohline, Parallel implementation of a data-transpose technique for the solution of Poisson's equation in cylindrical coordinates, in: The Eighth SIAM conference on Parallel Processing for Scientific Computing, 1997.
[9] J.L. Hennessy, D.A. Patterson, Computer Architecture, A Quantitative Approach, fourth ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
[10] J.D. Dumas, Computer Architecture: Fundamentals and Principles of Computer Design, CRC Press, 2005.
[11] B. Barney, Lawrence Livermore National Laboratory, High Performance Computing, Flynn's Classical Taxonomy, <https://computing.llnl.gov/tutorials/parallel_comp/#Flynn>, 2009.
[12] D.E. Knuth, third ed., Art of Computer Programming, Fundamental Algorithms, vol. 1, Addison-Wesley Professional, 1997.
[13] J. Sanders, E. Kandro, CUDA by example, an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010.
[14] Nvidia, CUDA C Best Practice Guide, v 3.2, 2010.
[15] Nvidia, CUDA technical training, vol. I, Q2, 2008.
[16] Nvidia, CUDA compute unified device architecture programming, v 1.0, 2010.
[17] Nvidia, CUDA reference manual, v 3.2 beta, 2010.
[18] Nvidia, Cuda C programming guide, v 3.2, 2010.
[19] NVidia, Tesla case studies, <http://www.nvidia.co.uk/object/tesla-case-studies-uk.html>, 2014.
[20] NVidia, NVidia Kepler achitecture, <http://www.nvidia.com/object/nvidia-kepler.html>, 2012.
[21] NVidia, Tesla GPU test drive, <http://www.nvidia.co.uk/object/k80-gpu-test-drive-uk.html>, 2014.