In-Place Matrix Transposition on GPUs

Juan Gómez-Luna, I-Jui Sung, Li-Wen Chang, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu, *Fellow, IEEE*

Abstract—Matrix transposition is an important algorithmic building block for many numeric algorithms such as FFT. With more and more algebra libraries offloading to GPUs, a high performance in-place transposition becomes necessary. Intuitively, in-place transposition should be a good fit for GPU architectures due to limited available on-board memory capacity and high throughput. However, direct application of CPU in-place transposition algorithms lacks the amount of parallelism and locality required by GPU to achieve good performance. In this paper we present our in-place matrix transposition approach for GPUs that is performed using elementary tile-wise transpositions. We propose low-level optimizations for the elementary transpositions, and find the best performing configurations for them. Then, we compare all sequences of transpositions that achieve full transposition, and detect which is the most favorable for each matrix. We present an heuristic to guide the selection of tile sizes, and compare them to brute-force search. We diagnose the drawback of our approach, and propose a solution using minimal padding. With fast padding and unpadding kernels, the overall throughput is significantly increased. Finally, we compare our method to another recent implementation.

Index Terms—GPU, transposition, in-place

1 Introduction

Matrix transposition converts an M-rows-by-N-columns array ($M \times N$ for brevity) to an N-rows-by-M-columns array. It is an important algorithmic building block with a wide range of applications from converting the storage layout of arrays to numeric algorithms, such as FFT and K-Means clustering.

FFT implementations typically carry out matrix transpositions before transforming each dimension [1]. This allows the transforms to access contiguous data, avoiding time-consuming strided memory accesses. K-Means clustering is also benefited by transposition when partitioning thousands or even millions of descriptors in image classification applications [2], in which typical descriptors are multidimensional vectors with up to 256 components.

BLAS libraries use matrix transposition as well. For instance, Intel MKL [3] includes out-of-place and in-place transposition routines since release 10.1.

As matrix transposition merely reorders the elements of a matrix, performance of matrix transposition is essentially determined by the sustained memory bandwidth of the system. This makes GPUs attractive platforms to execute the transposition because of their high memory bandwidth (to their global memory) compared to CPUs.

- *J. Gómez-Luna is with the University of Córdoba, Spain. E-mail: el1goluj@uco.es.*
- *I-J. Sung is with the Multicoreware, Inc. E-mail: ray@multicorewareinc.com.*
- *J. M. González-Linares and N. Guil are with the University of Málaga, Spain. E-mail: {gonzalez, nico}@ac.uma.es.*
- *L.-W. Chang and W.-M. W. Hwu are with the University of Illinois at Urbana Champaign. E-mail: dddscy@gmail.com, w-hwu@illinois.edu.*

Manuscript received 29 July 2014; revised 8 Mar. 2015; accepted 9 Mar. 2015. Date of publication 11 Mar. 2015; date of current version 12 Feb. 2016. Recommended for acceptance by D. R. Kaeli. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2015.2412549

Implementing out-of-place matrix transposition on GPUs, which achieves high fraction of peak memory bandwidth, is well understood as demonstrated in [4]. However, the memory capacity on GPUs is usually a much more constrained resource than their CPU counterparts. If an out-of-place transposition is employed, only up to 50 percent of the total available GPU memory could be used to hold one or several matrices, which need to be transposed, since the out-of-place transposition has at least 100 percent spatial overhead. This leads to the need for a general in-place transposition library for the accelerator programming models.

In-place transposition means the resulting A^T occupies the same physical storage locations as A. The spatial overhead is either none (i.e., methods that do not require bit flags but with extra computations) [5] or at most a small fraction of the input size (one bit per element) [6].

Mathematically, in-place transposition is a permutation, which can be factored into a product of disjoint cycles [7]. These cycles are "chains" of shifting, where each data element is moved to a destination that is the original location of another data element. In the special case of square matrices, the shifting consists of simply swapping symmetric elements along the diagonal, while the diagonal elements remain in the same location. There are as many cycles as elements over (or under) the diagonal, and their length is two. Thus, the GPU implementation is straightforward, as we show in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2412549, available online. The throughput of that simple implementation is in the order of magnitude of highly-optimized libraries [8] [9]. However, in the general case of rectangular matrices the number of cycles can be much lower, and their length is not uniform. These facts make parallelization a challenge.

A fast parallelization of matrix transposition can be achieved by the use of tiling, in order to take advantage of

the spatial locality in on-chip memories. We presented in [10] a four-stage approach, based on a method for multicore CPUs [11], and a three-stage approach that improved spatial locality. Both approaches used elementary tile-wise transpositions for each stage. In this paper, we explore all possible sequences of elementary tile-wise transpositions that implement a full transposition. Moreover, we develop heuristics for the selection of tile sizes. We found the drawback of our approach, and solved it by adding a negligible number of extra rows and/or columns.

The contributions of this paper are as follows:

- We derive all possible sequences of elementary tile-wise transpositions that implement full in-place matrix transposition. We compare them, and detect which of them can be advantageously used for particular matrix characteristics.
- We optimize the elementary transpositions, which were employed by our original three-stage tile-wise transposition [10], and adapt them to work with 32-bit and 64-bit elements. We also perform exhaustive tests to identify the best execution configuration for each tile size. Thus, we can obtain an optimal tuning of the elementary transpositions.
- We develop an heuristic that allows the code to choose near-optimal tile sizes in runtime, and compare it to a brute-force search.
- We detect a drawback of our approach, and solve it by adding very few extra rows and/or columns. This negligible padding ensures proper tile sizes for GPU architectures, so that the throughput is significantly boosted.
- We compare our scheme to another recent implementation [12] and show our algorithm performs better in most situations.

Our in-place transposition achieves a median throughput of 22.54 GB/s for double precision matrices on NVIDIA K20, 29.31 GB/s on NVIDIA GTX 980, and 37.40 GB/s on AMD Hawaii. In the special case of skinny matrices, or AoS-SoA and SoA-AoS conversion, the median throughput increases to more than 50 GB/s on GTX 980, more than 35 GB/s on K20, and more than 68 GB/s on Hawaii. OpenCL codes of our approach are publicly available.[1]

The rest of the paper is organized as follows. Section 2 presents the related works in the field of matrix transposition on CPU and GPU. In Section 3, the matrix transposition is defined, and a basic GPU implementation is presented. Section 4 describes how the full transposition of rectangular matrices can be carried out as a sequence of elementary transpositions. In Section 5 we tune the elementary transpositions to achieve the highest throughput, propose an heuristic to choose the most profitable tile sizes, and compare sequences of elementary transpositions. In Section 6 we estimate the throughput of our approach, and detect its main drawback. Then, we propose a solution for overcoming that drawback. Section 7 presents the experimental results. Finally, the conclusions are stated.

1. https://bitbucket.org/ijsung/libmarshal

2 Related Work

2.1 In-Place Transposition

As indicated above, most of sequential in-place transposition algorithms can be classified as cycle-following. Berman [6] proposed a bit-table for tagging cycles that have been shifted, and it requires $O(MN)$ bits of workspace. Windley [5] presented the notation of cycle-leaders as the lowest numbered element. Cate and Twigg [13] proved a theorem to compute the number of cycles in a transposition.

Achieving fast implementations of in-place transposition has attracted several research efforts. Recent works took a four-stage approach [11], [14], [15], in order to improve cache locality. Moreover, Gustavson et al. [14] proposed parallelization for multicores up to eight-cores. They noticed load imbalance issues, even for the relatively small number of threads available on multicores compared to modern GPUs. To address this problem, they proposed greedy assignment of cycles to threads and, for long cycles, splitting the shifting a priori.

2.2 In-Place and Out-of-Place Transposition for GPUs

For many-core processors, previous work [4] studied optimizations for out-of-place transposition. There are also highly-optimized libraries [8], [9] that include routines for in-place and out-of-place transposition of square matrices. For this special case, they use a straightforward algorithm, which is not generalizable for arbitrary matrices. Sung et al. [16] proposed the use of atomically-updated bit flags to solve the load-imbalance problem for GPUs and introduced elementary transposition routines that can be used to compose a multi-stage transposition. However, they do not specify how one would compose these elementary transpositions to obtain a full transposition.

Previous works on fast Fourier transform for the GPU such as [17] includes transposition to improve locality for global memory accesses; the authors did not specify whether the transposition is in-place or not, but we believe it is an out-of-place one. We also believe that their work can be enhanced by employing an in-place transposition algorithm like ours to increase the maximum size of data set allowed for GPU offloading.

Recently, a decomposition for in-place transposition has been presented by Catanzaro et al. [12]. It allows the rows and columns to be operated on independently, reducing work complexity and auxiliary space. Catanzaro et al. compare their implementation to our original three-stage approach [10], which is improved in the present work.

3 Definition of Matrix Transposition

Assume that A is a $M \times N$ matrix, where $A(i, j)$ is the element in row i and column j. The transpose of A is a $N \times M$ matrix A^T, so that the columns of A are the rows of A^T, or formally $A(i, j) = A^T(j, i)$.

In a linearized row-major layout, $A(i, j)$ is in offset location $k = i \times N + j$. When transposing, $A(i, j)$ at offset k is

```
for(int k=wi_id; k<M*N-1; k+=wg_size){
   // Transpose in a temporary array
   int k1 = (k * M) % (M * N - 1);
   temp[k1] = matrix[k];
}
// Synchronization
barrier();
// Copy to global memory
for(int i=wi_id; i<M*N-1; i+=wg_size){
   matrix[i] = temp[i];
}
```

Fig. 1. Code segment of in-place matrix transposition with barrier synchronization (BS). Input matrix `matrix` is located in global memory. The temporal array in local memory is `temp`. Each work-item `wi_id` belongs to a work-group size `wg_size`.

moved to $A^T(j,i)$ at $k' = j \times M + i$ in the transposed array A^T. The formula for mapping from k to k' is:

$$k' = \begin{cases} k \times M \bmod \mathcal{L}, & \text{if } 0 \leq k < \mathcal{L} \\ \mathcal{L}, & \text{if } k = \mathcal{L}, \end{cases} \quad (1)$$

where $\mathcal{L} = M \times N - 1$ [14].

The expression in Equation (1) allows us to calculate the destination a matrix element is moved to. Since we are moving elements in-place, each element has to be saved and further shifted (according to the involved permutation) to the next location. This generates cycles or chains of shifting.

The former transformation can be implemented on GPU by assigning matrix elements to work-items (i.e., a thread in OpenCL terminology), as the code in Fig. 1 shows. In this kernel, called barrier synchronization (BS), each work-group transposes a sub-matrix that fits the on-chip memory (registers or local memory). Although this implementation does not directly apply to arrays larger than tens of kilobytes in size, it can be used as a building block when transposing larger matrices. Section 4 explains a general transposition scheme for rectangular matrices. Moreover, the particular case of square matrices is reviewed in the supplemental material, available online.

4 In-Place Transposition of Rectangular Matrices

In the general implementation of in-place transposition of rectangular matrices, the cycles are generated using Equation (1). For instance, we can use a row-majored 5×3 matrix transposition example, i.e., $M = 5, N = 3, \mathcal{L} = M \times N - 1 = 14$. We start with element 1, or the location of $\mathtt{A}(0,1)$. The content of element 1 should be moved to the location of element 5, or the location of $\mathtt{A}^{\mathtt{T}}(1,0)$. The original content at the location of element 5, or the location of $\mathtt{A}(1,2)$, is saved before being overwritten and moved to the location of element 11, or the location of $\mathtt{A}^{\mathtt{T}}(2,1)$; The original content at the location of element 11 to the location of element 13, and so on. Eventually, we will return to the original offset 1. This gives a cycle of `(1 5 11 13 9 3 1)`. For brevity, we will omit the second occurrence of 1 and show the cycle as `(1 5 11 13 9 3)`. The reader should verify that there are five such cycles in transposing a 5×3 row-majored matrix: `(0) (1 5 11 13 9 3) (7) (2 10 8 12 4 6) (14)`.

Prior works [14] targeting multicores parallelize by assigning each cycle to a thread. As cycles by definition never overlap, it is an obvious source of parallelism that could be exploited by parallel architectures. In [16] this implementation is called P-IPT. However, for massively parallel systems that require thousands of concurrently active threads to attain maximum parallelism, this form of parallelism alone is neither sufficient nor regular. In fact, for the vast majority of other cases the amount of parallelism from the sheer number of cycles is both much lower and varying except when $M = N$ or square arrays. Even for larger M and N, the parallelism coming from cycles can be low. Also, as proven by Cate and Twigg [13], the length of the longest cycle is always multiple times the lengths of other cycles. This creates a load imbalance problem. Sung et al. [16] have proposed an atomic-operation-based approach to coordinate the shifting to reduce load imbalance. This approach is reviewed and optimized in this paper (see Section 5.1 and supplemental material, available online).

TABLE 1
Storage Formats of an $M \times N$ Matrix with Dimensions Factorized as $M = M' \times m$ and $N = N' \times n$ [14]

Format	Block order
RM	$M' \times m \times N' \times n$
RRRB	$M' \times N' \times m \times n$
RCRB	$M' \times N' \times n \times m$
CRRB	$N' \times M' \times m \times n$
CCRB	$N' \times M' \times n \times m$
CM	$N' \times n \times M' \times m$

4.1 Full Transposition As a Sequence of Elementary Tiled Transpositions

Good locality is crucial for modern memory hierarchies. For instance, on an NVIDIA Tesla K20 GPU a single-stage in-place transposition only runs at 1.5 GB/s due to poor locality. Therefore staged transpositions that trade locality with extra data movements can be favorable. Such an approach was explored in [11], [14], [15] for multi-core processors, where a full four-stage in-place transposition of a matrix, that can be achieved by a series of elementary tiled transpositions, significantly increases the throughput, thanks to a proper exploitation of the spatial locality.

In [14] the dimensions of an $M \times N$ matrix are factorized as $M' \times m \times N' \times n$ or a 4D array, where $M = M' \times m$ and $N = N' \times n$. This factorization defines a blocked format on the matrix composed by two independent formats: intra-block format and inter-block format. Intra-block format expresses the order of the elements within a block (row- or column-major) and, similarly, inter-block format defines the order of the blocks (also row- or column-major). Thus, there are only six possible storage formats combining intra-block and inter-block orders, as shown in Table 1.

Then, storage format conversions can be performed as sequences of elementary transpositions. These are designed in such a way that they only swap adjacent dimensions among the four dimensions. Thus, the problem of full in-place matrix transposition becomes finding a sequence of elementary transpositions to reach CM from RM. Table 2 employs the factorial numbering system [18] to name each elementary transposition. This table lists possible permutations that refer to the swapping of adjacent dimensions.

TABLE 2
Permutations in Factorial Numbering System

#Dimensions	From	To	Factorial Num.	Sung's terminology [16]
3D	(A, B, C)	(A, C, B)	$010_!$	AoS-ASTA transpose
	(A, B, C)	(B, A, C)	$100_!$	SoA-ASTA transpose
4D	(A, B, C, D)	(B, A, C, D)	$1000_!$	–
	(A, B, C, D)	(A, C, B, D)	$0100_!$	A instances of SoA-ASTA
	(A, B, C, D)	(A, B, D, C)	$0010_!$	A×B instances of AoS-ASTA

The equivalence between the factorial numbering system and Sung's terminology is described.

Each digit of the factorial number for a particular permutation represents each of three or four dimensions. The digit equal to 1 indicates that the corresponding dimension and the one on its right-hand side are swapped. For instance, if there are four dimensions (A, B, C, D), the factorial number $0100_!$ stands for a permutation from (A, B, C, D) to (A, C, B, D).

The elementary transpositions were used by Sung et al. [16] to transform data layouts from array-of-structures (AoS) or structure-of-arrays (SoA) to an intermediate layout called array-of-structures-of-tiled-arrays (ASTA). Thus, Sung's implementation of transposition $010_!$ considers AoS as a $M' \times m \times N$ 3-D array, and each of these $m \times N$ tiles is assigned one work-group, that is in charge of transposing the corresponding tile. Consequently, their AoS-to-ASTA marshaling is essentially a local transposition that converts $M' \times m \times N$ (AoS) to $M' \times N \times m$ (ASTA). Similarly, their SoA-to-ASTA transformation (i.e., transposition $100_!$) essentially is from $N \times M' \times m$ (SoA) to $M' \times N \times m$ (ASTA), in which every m-element tile is treated as a super-element that is then shifted in order to obtain ASTA.

Gustavson et al. [14] found two possible four-stage sequences of elementary transpositions to carry out the full in-place matrix transposition. For instance, in one of them, the first stage applies transposition $0100_!$, that is, M' instances of transpositions of $m \times N'$ matrices that are formed by super-elements of size n: from $M' \times m \times N' \times n$ to $M' \times N' \times m \times n$. The second stage employs $0010_!$ to transpose $M' \times N'$ instances of $n \times m$ matrices: from $M' \times N' \times m \times n$ to $M' \times N' \times n \times m$. The third stage, that applies the factorial $1000_!$, can be considered as one instance of transposition of an $N' \times M'$ array of super-element sized $n \times m$: from $M' \times N' \times n \times m$ to $N' \times M' \times n \times m$. The fourth stage is similar to the first one but with a different dimensionality.

In [10] we showed that the four-stage approach presents some issues that limit its throughput on GPUs. For instance, the transposition $1000_!$ in the four-stage approach moves super-elements of $m \times n$ elements, so that its best performance is obtained when these super-elements fit on-chip memory. Thus, values for m and n resulting in a high throughput for that transposition in stage 3, can perform poorly for transposition $0100_!$ in stages 1 and 4, where the size of the super-elements is only n and m, respectively.

In the present work, we explore all possible sequences of elementary transpositions between RM and CM formats. We detect nine sequences of 1, 2, 3, or 4 stages, which are shown in Fig. 2. Gustavson's original sequences are 4.1 and 4.2. One tenth sequence (4.3) results from the fusion of stages 2 and 3 in 4.1 or 4.2.

To illustrate these sequences, we explain sequence 3.1 in Fig. 3. Compared to Gustavson's sequences 4.1 and 4.2, in this sequence a much larger value of m and n can be used in the first and the third stage respectively for transposition $0100_!$ without overflowing the on-chip memory. On the other hand, two-stage sequences can be faster when the tile sizes $m \times N$ or $M \times n$ fit in local memory (assuming $N \ll M$ or $M \ll N$, respectively), since they entail one read access and one write access less than the three-stage scheme.

5 Towards a Suitable Approach for GPU Architectures

The aim of this section is to settle on the most appropriate configurations of elementary transpositions and sequences of them, in order to shape an approach to in-place matrix transposition suitable for GPUs. Thus, we initially evaluate briefly the performance of the elementary transpositions after some low-level optimizations that are explained in detail in the supplemental material, available online. Then, we analyze their throughput in a wide range

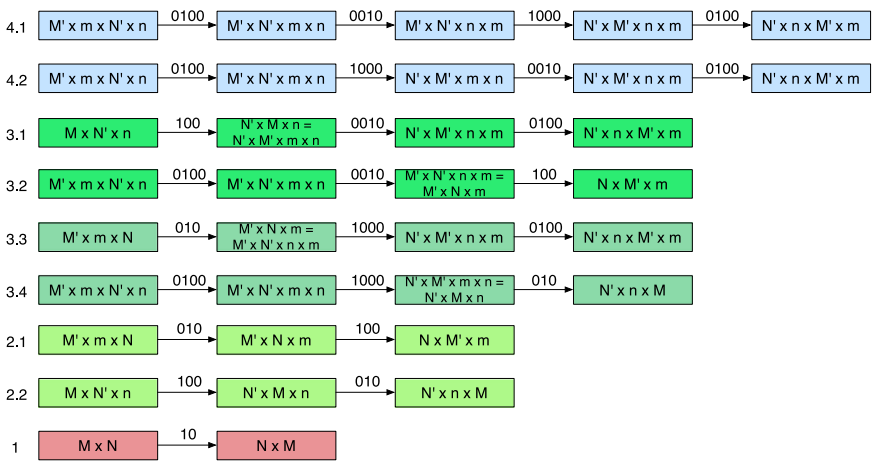


Fig. 2. Nine possible sequences of transformations between $M \times N$ and $N \times M$. An additional sequence of transformations (4.3) can be obtained if stages 2 and 3 are fused in 4.1 or 4.2, as explained in [14].

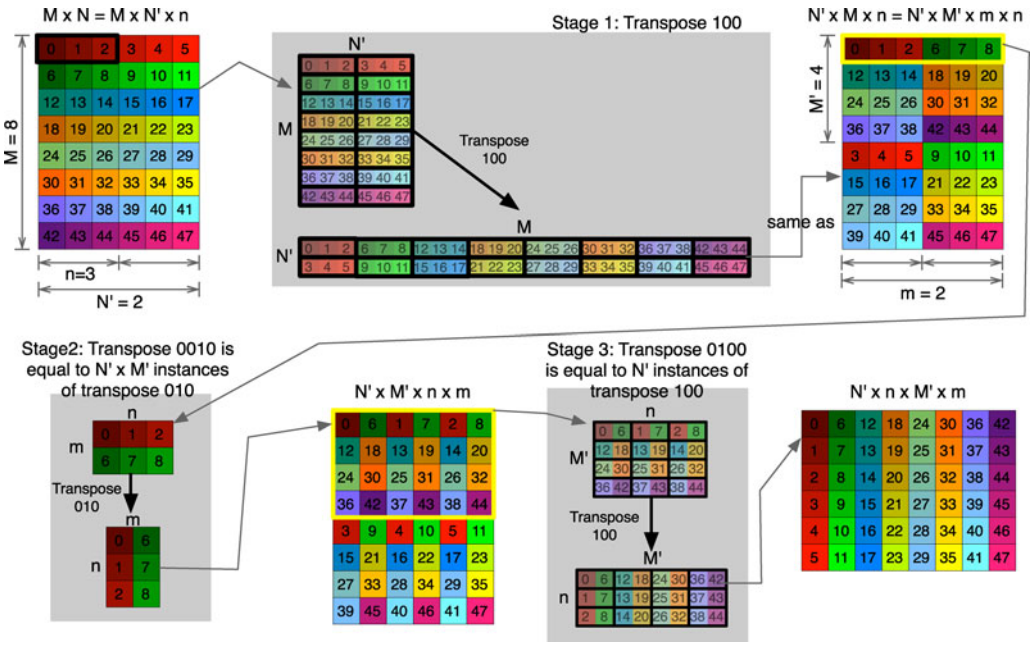


Fig. 3. Sequence 3.1 that implements full in-place transposition. In every figure, memory addresses increase from left to right and from top to bottom. Yellow halos indicate the part of the matrix that is brought into focus in the subsequent stage. Black halos represent super-elements, which are shifted as a whole. Stage 1 treats matrix $M \times N$ as a three-dimensional array of $M \times N' \times n$. It applies transposition $100_!$ moving super-elements of size n. Stage 2 treats the matrix as a four-dimensional array of $N' \times M' \times m \times n$. It is a transposition $0010_!$, where tiles (sub-matrices) of size $m \times n$ are transposed. Stage 3 applies transposition $0100_!$ to $N' \times M' \times n \times m$ with m-sized super-elements.

of super-element and tile sizes, in order to tune properly the execution configuration (i.e., the size of the set of work-items that assigns a tile or a super-element). Afterwards, we present heuristics to choose sizes for m and n. Finally, we evaluate the sequences of elementary transpositions that were presented in Section 4.1.

5.1 Performance of the Elementary Transpositions

Sung et al. [16] suggests parallelization strategies that are useful for the elementary transpositions introduced in the previous section. Transposition $010_!$ (AoS-ASTA) can be efficiently implemented with the BS kernel presented in Fig. 1, when the tile size $m \times n$ fits in GPU on-chip memory. For larger tiles they propose a method, called PTTWAC, in which multiple threads participate in the shifting of elements in one cycle, and use atomic operations to coordinate the shifting. This method is also applied to transposition $100_!$ (SoA-ASTA). Several optimizations can be applied to these elementary transpositions.

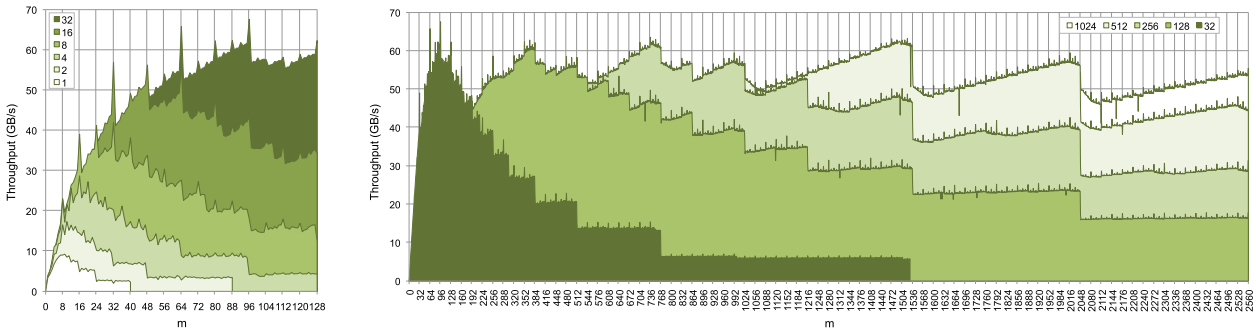
Regarding the PTTWAC-based transposition $010_!$, atomic contention in local memory burdens its performance. In [10] we presented two techniques (spreading and padding) to optimize this elementary transposition. They are detailed in the supplemental material, available online. The use of these techniques allows us to obtain an average speedup of 1.79 on a K20 over a baseline implementation without these two techniques. An alternative technique (remapping of the flag bits) can also be useful for tile sizes where the spreading and padding cannot be applied (see suplemental material for more details). This alternative technique yields up to 60 percent speedup on a K20 over the same baseline. Despite the considerable improvement obtained by the mentioned optimization techniques, the throughput of the PTTWAC-based transposition $010_!$ remains under 25 GB/s. Thus, the BS-based transposition $010_!$ will be preferably used in the full transposition approach.

BS-based transposition $010_!$ and PTTWAC-based transposition $100_!$ have in common the fact that both load chunks of matrix elements ($m \times n$-sized tiles or m-sized super-elements, respectively) into on-chip memory. In order to avoid a reduction of the available thread level parallelism (TLP), it is necessary to adapt the number of work-items that assigns one chunk to the size of the chunk. An explanation of this is included in the supplemental material, available online. With such an adaptation, the BS-based transposition $010_!$ results always in a very high throughput (typically, more than 100 GB/s on a K20 GPU), thanks to the use of the fast on-chip local memory. The throughput of transposition $100_!$ is shown in Fig. 4, which is analyzed in the next section. It can yield more than 60 GB/s on a K20, if local memory tiling is used, and more than 80 GB/s, if register tiling is used.

5.2 Tuning the Execution Configuration

One key aspect to attaining a high throughput is a proper tuning of the execution configuration, that is, the number of work-items that are assigned to one tile or one super-element. If the size of a tile (or a super-element) is too small, the number of active work-items and, consequently, the available TLP will be limited, and the throughput will be degraded. We propose to match the granularity of tiles or super-elements to the size of the sets of work-items (work-group, SIMD unit, or virtual SIMD unit) that assigns them.

Fig. 4 shows the throughput for transposition $100_!$ on an NVIDIA Tesla K20 for a wide range of super-element sizes from 1 to 2,560 (see supplemental material, available online, for the same tests on an AMD Hawaii). Local memory tiling is used in these experiments. In each sub-figure the legend indicates the number of work-items in a set that assigns a



(a) $m \leq 128$ (b) $m \leq 2560$

Fig. 4. Throughput (GB/s) of transpose $100_!$ for m under 2,560 on a K20 (32-bit elements). The experiment converts from $N \times M' \times m$ to $M' \times N \times m$. On the left (a), the throughput for different virtual SIMD unit sizes is shown. On the right (b), the throughput for different work-group sizes is presented.

super-element. Fig. 4a compares the effect of different virtual SIMD unit sizes. Virtual size 8 achieves the highest throughput under $m = 24$. From this point, the occupancy falls because of the local memory requirements. Similarly, the throughput for virtual sizes 16 and 32 is abruptly reduced in m equal to 48 and 96, respectively. Choosing the proper virtual SIMD unit size ensures a throughput average (minimum/maximum) speedup of 18 percent (3%/61%) with respect to using the actual SIMD unit size (for example, 32 work-items in NVIDIA GPUs).

Fig. 4b explores the throughput for different work-group sizes. The darker shape corresponds to the implementation that assigns super-elements to SIMD units (32 work-items), which is presented in [10]. The other four series in the graph use one entire work-group of 128, 256, 512, or 1,024 work-items per super-element. As it can be seen, each series presents some abrupt changes that are due to occupancy reductions, when the local memory requirements exceed a certain limit. This determines the ranges of m where each work-group size is more profitable.

It is remarkable in Fig. 4 that the throughput can be maintained over 40 GB/s for a wide range of super-element sizes from 24 to 2,560, if the size of the set of work-items is properly chosen. However, the throughput unavoidably degrades for smaller super-elements, despite the use of virtual SIMD units ensures some improvement, as seen above. The fact that Equation (1) determines the location where a super-element is moved to makes adjacent virtual SIMD units (which belong to the same actual SIMD unit) access distant global memory areas. Thus, the number of global memory transactions increases. This fact will entail the main performance bottleneck in our approaches to matrix transposition.

Additionally, once determined the proper size of the set of work-items for each range of super-element sizes, register tiling can be applied instead of local memory tiling, obtaining further improvement.

A similar analysis can be done for BS-based transposition $010_!$. The highest throughput can be achieved in each range of the tile size, if the size of the set of work-items is properly selected. Similarly, the bounds for each set size are given by the local memory requirements and the occupancy. Unlike transposition $100_!$, the throughput is not dramatically reduced for very small tiles. The tiles are assigned to virtual SIMD units that will write the tiles in the same locations they were read, as the transposition takes place within a tile. Thus, the number of global memory transactions is limited, because the tiles accessed by virtual SIMD units belonging to the same actual SIMD unit are in adjacent locations.

5.3 Heuristics for a Near-Optimal Throughput

In [12] Catanzaro et al. compared their implementation to our work presented in [10]. As they pointed out, our implementation [10] needed a heuristic to choose tile sizes automatically. In order to carry out the comparison, they used the heuristic in Algorithm 1. Their aim was to choose m and n that are not too small nor to large for the hardware. This way, they set a maximum of 72, so that the maximum tile size 72×72 fits in the local memory of a K20.

Algorithm 1. Heuristic used by Catanzaro et al. [12]. F_M and F_N are arrays that contain the factors of M and N, respectively, after sorting them in ascending order. $\#F_M$ and $\#F_N$ are the number of factors of each dimension.

Calculate factors of M **and** N
Sort factors of M **and** N
for $i = 0$ **to** $\#F_M$ **do**
  if $F_M[i] \leq 72$ **then**
    $k = i$
  end if
end for
$m = F_M[k]$
$M' = M/m$
for $i = 0$ **to** $\#F_N$ **do**
  if $F_N[i] \leq 72$ **then**
    $k = i$
  end if
end for
$n = F_N[k]$
$N' = N/n$

Although that heuristic allowed them to obtain an impressive maximum throughput for some cases, it does not select the best tile sizes for many matrices. As we have shown in Section 5.2, the main bottleneck of our tiled approach is in transposition $100_!$ when the super-element size (m or n) is very small. Thus, we propose a more elaborate heuristic in Algorithm 2. It tries to select m and n in a range between Min and Max. Then, it

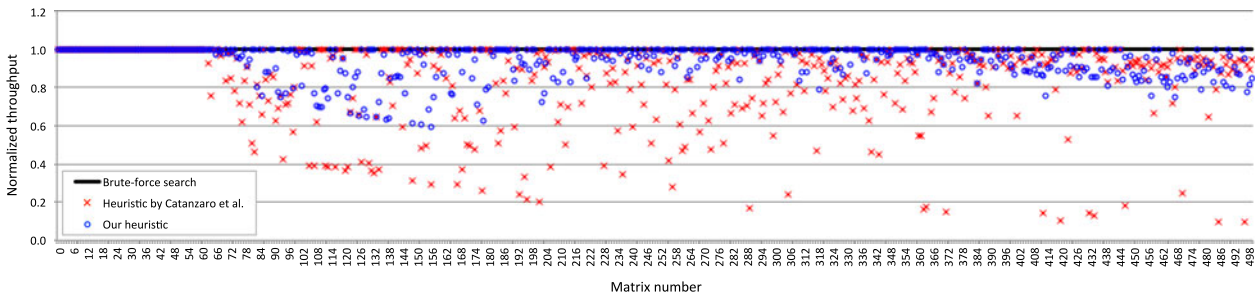


Fig. 5. Comparison of a brute-force search to the proposed heuristics for 500 random matrices on an NVIDIA Tesla K20 GPU.

evaluates all possible combinations of m and n in that range that fit local memory. Finally, it selects the tile size $m \times n$ that obtains the largest exploitation of the local memory size. This criterion is based on the intuition that the BS-based transposition $010_!$ will obtain the highest throughputs when the local memory is more intensively used. The PTTWAC-based transposition $010_!$ will only be used when no possible combinations of m and n fit in local memory (not detailed in Algorithm 2).

Algorithm 2. Our heuristic. Min and Max stand for the minimum and maximum super-element size for transposition $100_!$, respectively. $Eligible\_F_M$ and $Eligible\_F_N$ are auxiliary arrays where potentially eligible factors are stored. LM_size is the number of local memory words. $\#WG$ is the number of work-groups that are executed simultaneously according to the local memory requirements per work-group.

Calculate factors of M **and** N
Sort factors of M **and** N
$k = 0$
for $i = 0$ **to** $\#F_M$ **do**
  if $F_M[i] > Min$ **&** $F_M[i] < Max$ **then**
    $Eligible\_F_M[k] = F_M[i]$
    $k = k + 1$
  end if
end for
$l = 0$
for $i = 0$ **to** $\#F_N$ **do**
  if $F_N[i] > Min$ **&** $F_N[i] < Max$ **then**
    $Eligible\_F_N[l] = F_N[i]$
    $l = l + 1$
  end if
end for
for $i = 0$ **to** k **do**
  for $j = 0$ **to** l **do**
    if $Eligible\_F_M[i] \times Eligible\_F_N[j] < LM\_size$ **then**
      Calculate number of work-groups $\#WG$
      if $\#WG \times Eligible\_F_M[i] \times Eligible\_F_N[j] > m \times n \times \#WGMax$ **then**
        $m = Eligible\_F_M[i]$
        $n = Eligible\_F_N[j]$
        $\#WGMax = \#WG$
      end if
    end if
  end for
end for

5.3.1 Brute-Force Search versus Heuristics

We evaluate the accuracy of these heuristics with respect to a brute-force search, which is not practical in real-world applications. Tests have been conducted on a K20 for 500 random matrices with dimensions in the interval [1,000, 20,000).

As it can be seen in Fig. 5, our heuristic is very close to the maximum possible throughput. The median relative error of the throughput achieved by the heuristic with respect to that obtained by brute-force is only 2 percent. Moreover, the cost of the complete execution of the heuristic is very small, representing less than 1 percent of the total execution time.

5.4 Comparison of Sequences of Elementary Transpositions

We compare the sequences of transpositions with 2, 3, and 4 stages enumerated in Section 4.1, in order to figure out which sequences are more appropriate for GPU architectures. Tests have been carried out on a K20 for 1,000 random matrices with dimensions M and N in the interval [1,000, 20,000). For each of these matrices, it has been necessary to select m and n among the factors of M and N. This selection has been carried out with the same criteria explained in Section 5.3.

As shown in Fig. 6, sequences 3.1 and 3.2 achieve the highest throughput in a majority of cases. Sequence 3.1 is the fastest for 418 matrices, and sequence 3.2 for 374. One key observation is that sequence 3.1 is faster than 3.2 when $n > m$, while 3.2 outperforms 3.1 when $m > n$. The reason for this lies on the fact that n is the super-element size of transposition $100_!$ in sequence 3.1, and m in sequence 3.2. The larger this super-element, the higher the throughput, since super-elements are shifted over larger distances in transposition $100_!$ than in transposition $0100_!$.

The transposition of the rest of matrices is faster with sequence 2.1 (98 matrices) or 2.2 (91 matrices). The common characteristic of these matrices is that they are very skinny. If $N \ll M$, sequence 2.1 is the fastest, because $m \times N$ fits in on-chip memory. Sequence 2.2 performs likewise when $M \ll N$.

In summary, once m and n have been chosen using the heuristic of Algorithm 2, our approach to in-place matrix transposition will carry out one sequence of elementary transpositions (3.1, 3.2, 2.1, or 2.2) depending on the characteristics of the matrix (See Algorithm 3).

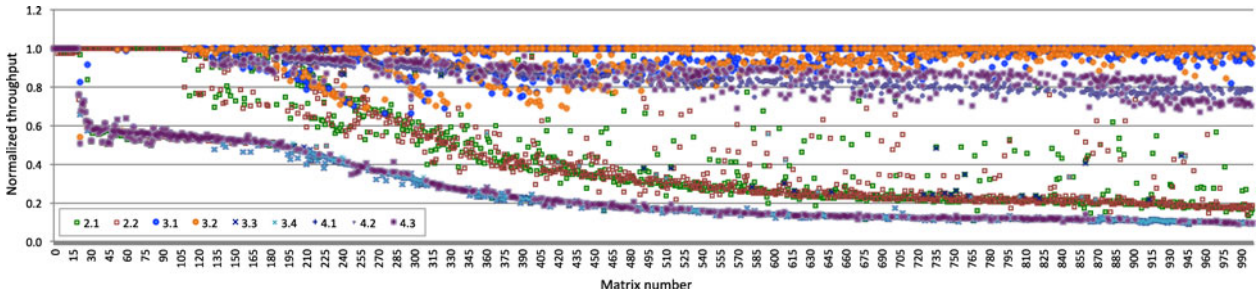


Fig. 6. Comparison of nine sequences of transformations ($M \times N$ to $N \times M$) for 1,000 random matrices on an NVIDIA Tesla K20 GPU.

Algorithm 3. Sequence selection according to matrix characteristics.

```
Execute Algorithm 2
if m × N < LM_size then
  Sequence 2.1
else if M × n < LM_size then
  Sequence 2.2
else if m < n then
  Sequence 3.1
else
  Sequence 3.2
end if
```

6 An Outstanding Throughput Boost at a Negligible Overhead

In this section, we analyze the impact on the effective throughput of the major bottleneck (i.e., very small super-element size m or n) that we have detected in the previous section. Then, we propose a simple correction based on minimal padding to boost the throughput of the affected matrices.

6.1 Throughput Estimation and Discussion

The effective throughput $T_{effective}$ of a three-stage sequence of elementary transpositions can be estimated with Equation (2), where $T_{100_!}$ and $T_{010_!}$ are respectively the throughput of transpositions $100_!$ and $010_!$. The outcome is the inverse of the total execution time per byte. According to this equation, the maximum theoretical throughput on a K20 GPU with a global memory bandwidth of 208 GB/s would be 69.33 GB/s. However, a more realistic theoretical maximum has to take into account the highest throughputs that can be obtained for transpositions $100_!$ and $010_!$. Using the figures presented in Section 5.1, the realistic maximum throughput would reach the top around 30 GB/s for 32-bit elements,

$$T_{effective} = \frac{1}{\frac{1}{T_{100_!}} + \frac{1}{T_{010_!}} + \frac{1}{T_{100_!}}} \ (B/s). \quad (2)$$

An analogous analysis can be done for the lower bound. A very small super-element size m or n makes transposition $100_!$ barely achieve 10 GB/s or less, according to Fig. 4a. This would place the peak of the effective throughput around 8 GB/s. Even worse would be the case of both m and n very small: less than 5 GB/s. For illustrative purposes, in order to figure out how many matrices would be transposed at such low rates, we have generated 5,000 random matrices with dimensions M and N in the interval [1,000, 20,000). Then, we have counted the number of them that will be factorized with very small m or n using Algorithm 2. If we arbitrarily choose m or n less or equal to 6, the resulting number of matrices is 2,235, that is, more than 44 percent of those matrices.

6.2 Padding to Obtain Better Factorizations

For those matrices where m and/or n will be very small, we can easily obtain a better factorization with a minimal overhead. As a running example, let us consider a matrix of size $6{,}203 \times 6{,}607$. Both dimensions are prime numbers. Thus, Algorithm 2 will choose $m = 1$ and $n = 1$. With such super-element sizes, the throughput that our approach to in-place matrix transposition can achieve for this matrix is 2.38 GB/s on an NVIDIA Tesla K20.

By simply padding one row and one column, the number of factors of the new dimensions 6,204 and 6,608 is 23 and 19, respectively. Among these, we can choose $m = 94$ and $n = 59$, which will ensure a much higher throughput. The effective throughput for a $6{,}204 \times 6{,}608$ matrix on K20 is 24.18 GB/s.

6.2.1 *Fast Padding and Unpadding Kernels*

Padding extra rows is trivial: allocating more space at the end of the array. However, padding columns is trickier, as it can be seen in Fig. 7. Rows should be moved sequentially, as they are shifted a few positions forward, where rows with higher index are placed. For instance, row 4 is allowed to be shifted when row 5 has been shifted, or more precisely when row 5 has been saved in temporary on-chip memory. Algorithm 4 depicts our fast padding kernel. Each work-group has an associated flag that will enable work-groups assigned to row elements with lower index to move them. A work-group i loads consecutive array elements into on-chip memory (scratch-pad memory and registers) and waits for the flag $i - 1$ to be set true. As soon as this is true, work-group i sets its

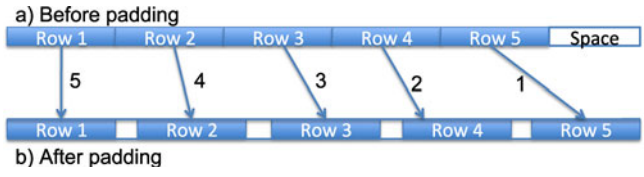


Fig. 7. Padding in-place a row-major matrix composed by five rows.

```
// Synchronization
barrier(CLK_LOCAL_MEM_FENCE);
if (wi_id == 0){
  // Wait
  while (atom_or(&flags[wg_id], 0) == 0){}
  // Set flag
  atom_or(&flags[wg_id + 1], 1);
}
// Synchronization
barrier(CLK_GLOBAL_MEM_FENCE);
```

Fig. 8. Code segment of adjacent work-group synchronization. Work-item with `wi_id = 0` executes the loop until the flag of the previous work-group is set. Then, it sets its associated flag. The first `barrier()` ensures all work-items in the work-group reach that point at the same time. The second one guarantees correct ordering of global memory operations.

flag, and stores the array elements into global memory. The code of this synchronization mechanism is in Fig. 8. Flags should be read and written atomically. The first `barrier()` ensures all work-items of the work-group have loaded array elements into on-chip memory. The second `barrier()` entails a global memory fence, which ensures correct ordering of global memory operations. Similar synchronization procedures between adjacent work-groups are explained in [19], [20].

Algorithm 4. Padding kernel. A work-item wi_id loads into registers matrix elements from locations $pos(wg\_id, wi\_id)$, that are a function of the work-group ID wg_id and wi_id. Then, they are stored in locations $pos\_pad(wg\_id, wi\_id)$.

Dynamic work-group ID allocation
for $i = 0$ **to** $\#REGS$ **do**
  $Register_i = Matrix[pos(wg\_id, wi\_id)]$
  $pos(wg\_id, wi\_id) += wg\_size$
end for
Adjacent work-group synchronization
for $i = 0$ **to** $\#REGS$ **do**
  $Matrix[pos\_pad(wg\_id, wi\_id)] = Register_i$
  $pos\_pad(wg\_id, wi\_id) += wg\_size$
end for

In order to avoid potential deadlocks due to the non-deterministic scheduling of work-groups, we deploy a dynamic work-group ID allocation [19]. The code is explained in Fig. 9.

A similar design can be used for the unpadding kernel, which is needed after transposition, if extra rows were added.

As it is shown in Fig. 10, these padding and unpadding kernels achieve a large fraction of peak bandwidth on current GPUs. Thus, the effective throughput for the $6{,}203 \times 6{,}607$ matrix is 15.36 GB/s, including padding and unpadding. The throughput is boosted 6.5x with a negligible memory overhead of 0.03 percent.

```
__local int wg_id_;
if (wi_id == 0) wg_id_ = atom_add(&S, 1);
// Synchronization
barrier(CLK_LOCAL_MEM_FENCE);
```

Fig. 9. Code segment of dynamic work-group ID allocation. As soon as a work-group is scheduled, the first work-item `wi_id = 0` gets the dynamic work-group ID by incrementing a location `S` in global memory, which was initialized to 0. The dynamic work-group ID is stored in local memory, so that it is visible to every work-item in the work-group after the synchronization.

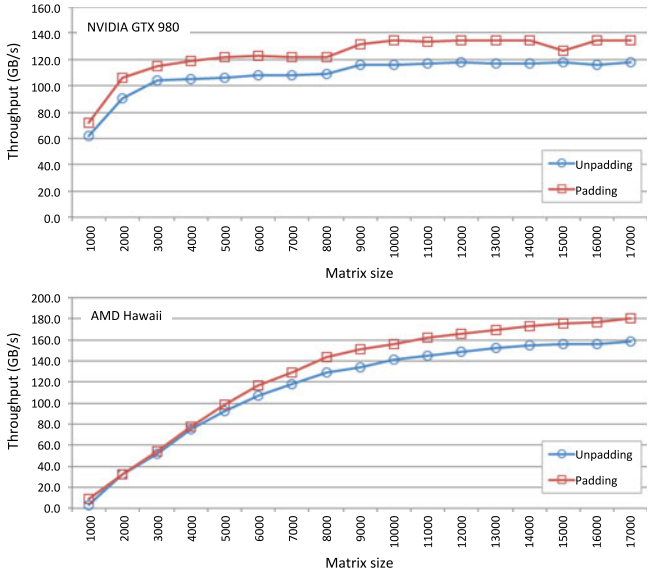


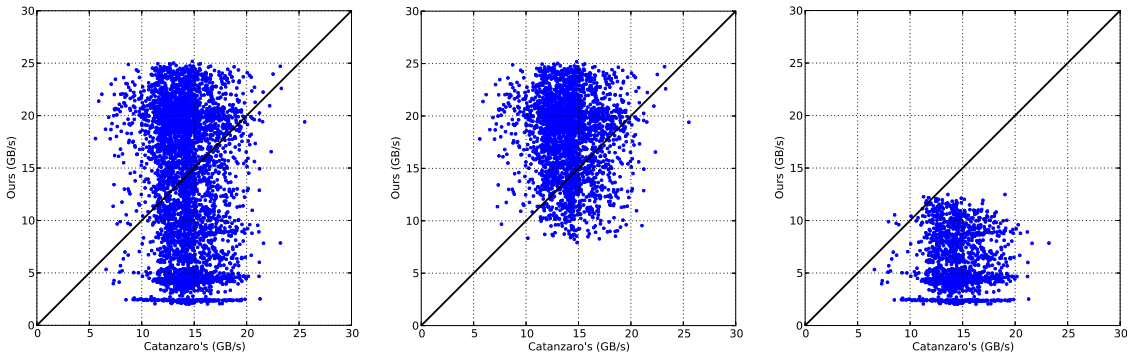
Fig. 10. Throughput of padding or unpadding one column on an NVIDIA GeForce GTX 980 GPU and an AMD Hawaii GPU. Abscissas represent the number of rows of the matrix. The number of columns is the same after padding/before unpadding.

6.2.2 A New Simpler Heuristic

Algorithm 5 defines a simple heuristic to select the super-element sizes m and n. Essentially, if it is not possible to find a super-element size in the desired range, one row or column is padded. The lower bound Min (typically, 24) ensures a considerable throughput for transpositions $100_!$ and $0100_!$. The upper bound is the square root of the local memory size, so that BS-based transposition $010_!$ can be employed. In the worst case, M will be a multiple of Min before Min iterations. As this algorithm decides padding needs, it is executed prior to memory allocation, so that the extra space is allocated contiguous to the matrix.

Algorithm 5. Simple heuristic to choose m. n is chosen independently with the same heuristic. Min stands for the minimum super-element size for transposition $100_!$ (or $0100_!$). LM_size is the number of local memory words.

$done_m = false$
while $done_m = false$ **do**
  Calculate factors of M
  Sort factors of M
  for $i = 0$ **to** $\#F_M$ **do**
    if $F_M[i] \geq Min$ **&** $F_M[i] < \sqrt{LM\_size}$ **then**
      $m = F_M[i]$
      $done_m = true$
      Break loop
    end if
  end for
  if $done_m = false$ **then**
    $M = M + 1$
  end if
end while



(a) 5000 matrices (b) 2926 matrices ($m > 4$ and $n > 6$) (c) 2074 matrices ($m \leq 4$ and $n \leq 6$)

Fig. 11. Comparison to Catanzaro et al. Each point corresponds to one matrix, and gives the throughput of Catanzaro's implementation on the horizontal axis, and the throughput of our implementation on the vertical axis. On the left, 5,000 random matrices are used. On the middle, 2,926 matrices for which our heuristic chooses m greater than 4 and n greater than 6 (for sequence 3.1, and vice versa for sequence 3.2), and $m \times n$ fits in local memory. On the right, the remaining 2,074 matrices.

7 Experimental Results

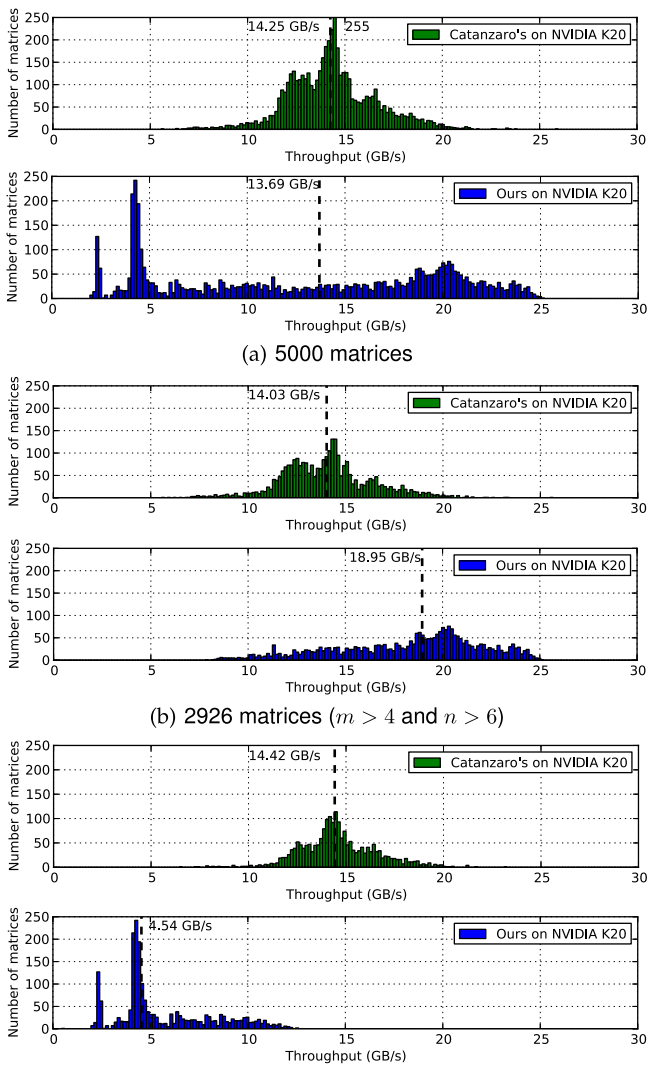
In this section we evaluate our approach and compare it to another recent in-place transposition. First, we use the heuristic in Algorithm 2, and confirm where the drawback of our approach is. Then, we utilize padding to obtain a significant improvement. Experiments in this section have been performed on NVIDIA Tesla K20 and GeForce GTX 980 GPUs, and AMD Hawaii. K20 has Kepler architecture with a peak memory bandwidth of 208 GB/s. GTX 980 is Maxwell and its memory bandwidth is 224 GB/s. Hawaii's peak memory bandwidth is 320 GB/s. Tests on NVIDIA devices have been carried out with CUDA SDK 6.5, and on AMD device with AMD SDK 2.9.1.

7.1 Evaluation of the three-Stage/two-Stage Approach

Catanzaro et al. [12] compared their implementation to our previous work [10] for 2,500 random matrices with M and N in the interval [1,000, 20,000). They reported a median throughput of 14.23 GB/s for their implementation and 5.33 GB/s for our previous one (using the heuristic in Algorithm 1), when transposing matrices of 32-bit elements. In the present work, we have described some new optimizations, such as the use of virtual SIMD units, and have tuned our elementary transpositions. Moreover, we have devised an heuristic that is more accurate. In this section, we compare our three-stage/two-stage approach using Algorithm 2[2] to Catanzaro's using 5,000 random matrices of 32-bit elements with M and N in the interval [1,000, 20,000).

As it can be seen in Fig. 12a, we measure a median throughput of 14.25 GB/s for Catanzaro's implementation, and 13.69 GB/s for our implementation on K20. It is noticeable that our implementation presents cases with throughput below 5 GB/s. As we detected in Section 6, these are due to very small super-elements m and n. Thus, Fig. 12b shows the throughput for matrices where Algorithm 2 is able to select $m > 4$ and $n > 6$.

2. In the experiments of this section, our approach does not use the padding technique presented in Section 6.



(a) 5000 matrices

(b) 2926 matrices ($m > 4$ and $n > 6$)

(c) 2074 matrices ($m \leq 4$ and $n \leq 6$)

Fig. 12. Throughput histograms for general transposition of random matrices (32-bit elements) on NVIDIA Tesla K20 GPU. On the top (a), Catanzaro's and our results for 5,000 random matrices. On the middle (b), results for 2,926 matrices that are transposed by our approach with $m > 4$ and $n > 6$ (or vice versa for sequence 3.2), and BS-based transposition $010_!$. On the bottom (c), results for the remaining 2,074 matrices.

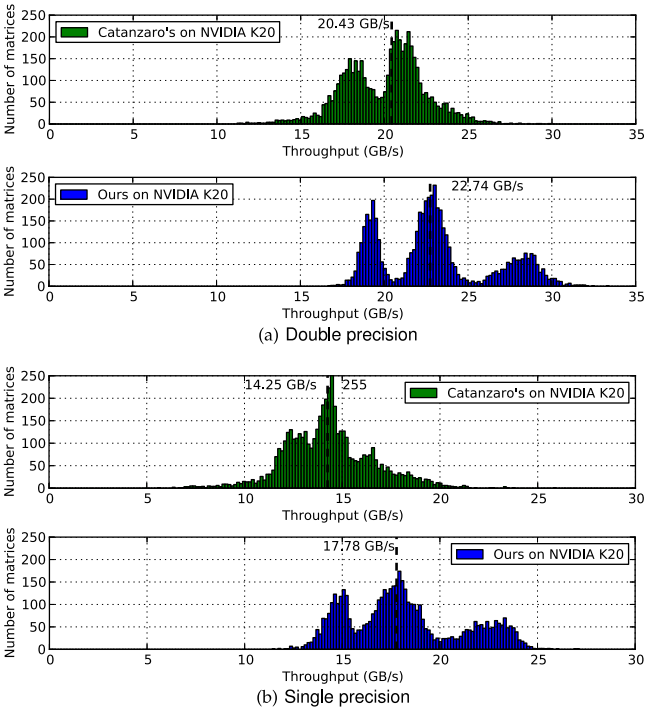


Fig. 13. Throughput histograms for general transposition on an NVIDIA Tesla K20 GPU. On the top, results for 5,000 matrices of 64-bit elements. On the bottom, results for 5,000 matrices of 32-bit elements.

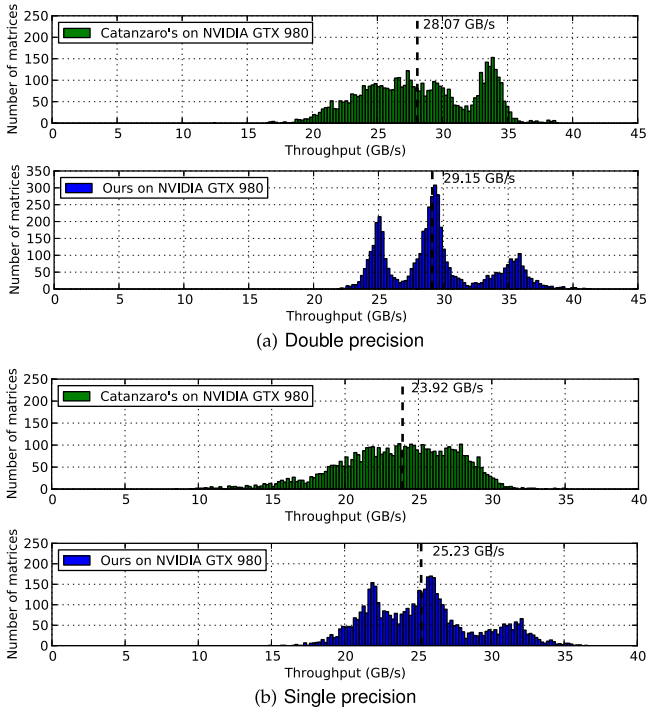


Fig. 14. Throughput histograms for general transposition on an NVIDIA GeForce GTX 980 GPU. On the top, results for 5,000 matrices of 64-bit elements. On the bottom, results for 5,000 matrices of 32-bit elements.

Catanzaro's results for the same matrices are also shown. The median throughput of our implementation for these cases (2,926 matrices) is significantly higher than Catanzaro's.

Figs. 11a, 11b, and 11c present the results using a dispersion map. Each point represents the throughput for one of the 5,000 random matrices. The projection on the horizontal axis gives the throughput of Catanzaro's implementation, and the projection on the vertical axis indicates the throughput of our implementation.

Similar trends happen for matrices of 64-bit elements, and for the particular case of skinny matrices (i.e., AoS to SoA and SoA to AoS conversions). Our three-stage/two-stage approach with Algorithm 2 results in a significantly higher median throughput than Catanzaro's implementation for almost 60 percent of the matrices, but bad factorization cases burden the overall throughput.

7.2 Throughput Boost with Padding

As explained in Section 6, a negligible memory overhead (due to padding very few columns or rows) can increase the throughput of our approach for every matrix. In this section, all our throughput figures include padding and/or unpadding times, if padding columns and/or rows is applied.[3] Fig. 13 compares Catanzaro's results to our approach using padding and Algorithm 5 on K20. Fig. 14 is the same for GTX 980. The two graphs on the bottom correspond to 5,000 matrices of 32-bit elements with dimensions in the interval [1,000, 20,000). The two graphs on the top are for 5,000 matrices of 64-bit elements with dimensions in [1,000, 15,000).[4]

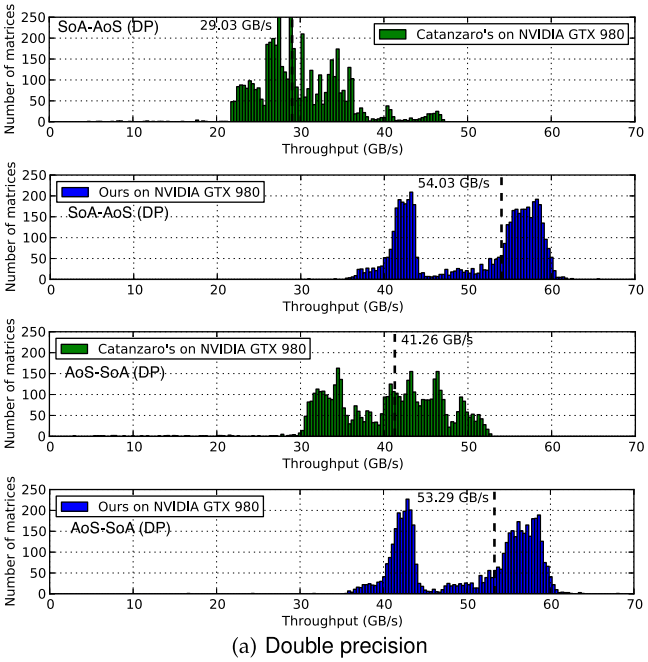
The graphs show a dashed black line that stands for the median throughput. This is considerably higher for our approach. The three bumps in our histograms roughly correspond to matrices where two dimensions, one dimension or none were padded, respectively. It is remarkable that the maximum number of extra rows and/or columns in all these cases is 8. For single precision matrices the maximum memory overhead is 0.43 percent, and the median memory overhead is 0.01 percent. For double precision matrices, the maximum is 0.47 percent and the median is 0.03 percent.

We have also tested data layout transformations, as a particular application of matrix transposition. 5,000 skinny matrices with the long dimension in the interval $[10^4, 10^7)$, and the short dimension in [2, 32) have been used in each experiment. Fig. 15 compares the throughput of our approach to Catanzaro's for SoA-AoS and AoS-SoA conversion on GTX 980. The graphs on the top (a) correspond to matrices of 64-bit elements, and on the bottom to matrices of 32-bit elements. In these cases, our approach only pads the long dimension, and two-stage sequences are used. The lower bump in our histograms corresponds to the cases where padding is needed. The maximum memory overhead in all these cases is 0.02 percent.

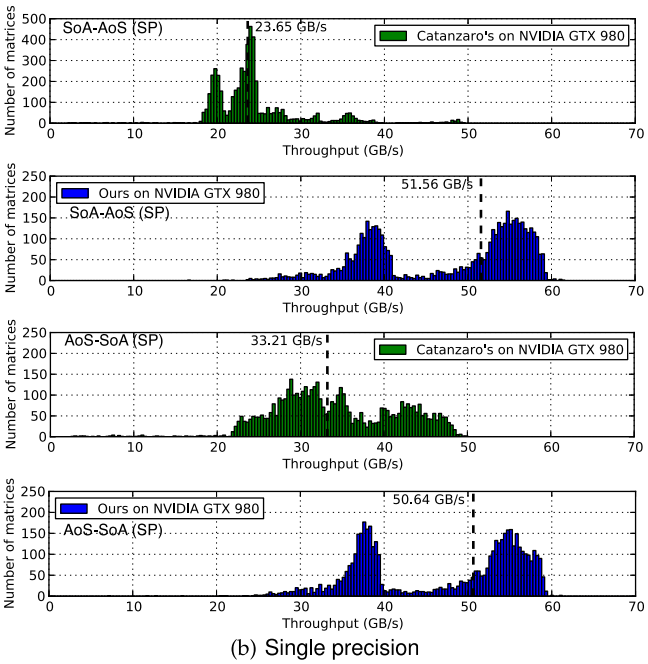
Finally, we have also tested our approach on an AMD Hawaii GPU. We cannot compare to Catanzaro et al.,

3. In some applications padding can be done at storage time, so runtime padding cost will not be added.

4. Due to OpenCL constraints on maximum allocatable memory space, it is not possible to test larger matrices.



(a) Double precision



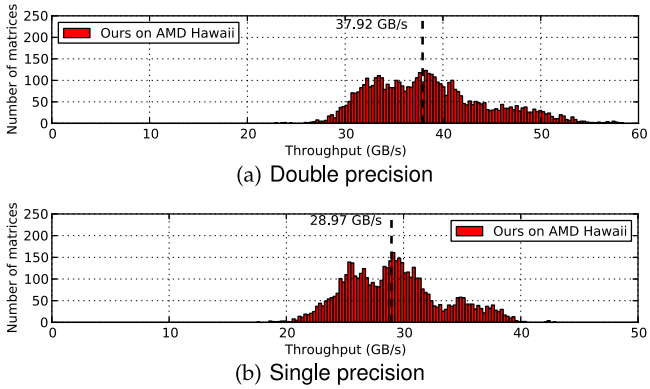
(b) Single precision

Fig. 15. Throughput histograms for SoA-AoS conversion on an NVIDIA GeForce GTX 980 GPU.

because that is a CUDA implementation. To the best of our knowledge, ours is the only in-place matrix transposition approach for AMD devices. Fig. 16 presents the throughput histograms for 5,000 matrices of 32-bit elements (bottom) and 64-bit elements (top).

8 Conclusion

This paper deals with in-place matrix transposition on GPUs. We have designed a general approach for rectangular matrices using elementary transformations. We have enhanced the performance of these building blocks proposed by earlier works. Moreover, we have explored all possible sequences of elementary transformations that



(a) Double precision

(b) Single precision

Fig. 16. Throughput histograms for general transposition of 5,000 random matrices on an AMD Hawaii GPU. On the top, the results for double precision matrices. On the bottom, the results for single precision matrices.

implement full transposition. Thus, we are able to choose the most favorable sequence according to matrix characteristics. We have observed that the tile size greatly affects performance of in-place transposition. Since the search space for tile sizes can be big, we have proposed an heuristic to choose good tile sizes. We have detected the drawback of our approach (i.e., very small super-elements in transpositions $100_!$ and $0100_!$), and we have proposed padding very few rows and/or columns to solve it. With fast padding and unpadding kernels, the throughput of our approach can be greatly improved. We have finally compared our approach to another recent implementation.

Acknowledgments

The authors thank Bryan Catanzaro for giving us early access to his implementation. We also thank the Starnet Center for Future Architecture Research (C-FAR), the UIUC CUDA Center of Excellence, NVIDIA (for hardware donation to the University of Málaga under CUDA Research Center Awards), and the Ministry of Education of Spain (TIN2010-16144) and the Junta de Andalucía of Spain (TIC-1692) for financial support. J. Gómez-Luna is the corresponding author.

References

[1] M. Frigo and S. Johnson, "The design and implementation of fftw3," in *Proc. IEEE*, 2005, vol. 93, no. 2, pp. 216–231.

[2] K. Kohlhoff, V. Pande, and R. Altman, "K-means for parallel architectures using all-prefix-sum sorting and updating steps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1602–1612, Aug 2013.

[3] Intel MKL. (2013, Jan.). Intel Math Kernel Library. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl-111-release-notes

[4] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," NVIDIA Technical Report, pp. 1–24, Jan. 2009.

[5] P. F. Windley. (1959). Transposing matrices in a digital computer. *Comput. J.* [Online]. *2(1)*, pp. 47–48. Available: http://comjnl.oxfordjournals.org/content/2/1/47.abstract

[6] M. F. Berman, "A method for transposing a matrix," *J. ACM*, vol. 5, no. 4, pp. 383–384, Oct. 1958.

[7] T. Hungerford. (1997). *Abstract Algebra: An Introduction* [Online]. Saunders College Publishing. Available: http://books.google.com/books?id=H7XuAAAAMAAJ

[8] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "Modeling and Simulation for Defense Systems and Applications V," pp. 770 502–770 502–7, Apr. 2010, Doi: 10.1117/12.850538.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. (2009). Numerical linear algebra on emerging architectures: The plasma and magma projects. *J. Phys.: Conf. Series* [Online]. *182(1)*, p. 012037. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012037
[10] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place transposition of rectangular matrices on accelerators," in *Proc. 19th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2014, pp. 207–218.
[11] L. Karlsson, "Blocked in-place transposition with application to storage format conversion," Umea University, Tech. Rep. UMINF 09.01, 2009, http://www8.cs.umu.se/research/uminf/index.cgi?year=2009&number=1
[12] B. Catanzaro, A. Keller, and M. Garland, "A decomposition for in-place matrix transposition," in *Proc. 19th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, Feb. 2014, pp. 193–206.
[13] E. G. Cate and D. W. Twigg, "Algorithm 513: Analysis of in-situ transposition [f1]," *ACM Trans. Math. Softw.*, vol. 3, no. 1, pp. 104–110, Mar. 1977.
[14] F. Gustavson, L. Karlsson, and B. Kågström, "Parallel and cache-efficient in-place matrix storage format conversion," *ACM Trans. Math. Softw.*, vol. 38, no. 3, pp. 17:1–17:32, Apr. 2012.
[15] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, "Efficient transposition algorithms for large matrices," in *Proc. Conf. Supercomput.*, Nov. 1993, pp. 656–665.
[16] I.-J. Sung, G. Liu, and W.-M. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Proc. Innovative Parallel Comput.*, May 2012, pp. 1–11.
[17] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, "Auto-tuning of fast fourier transform on graphics processors," in *Proc. 16th ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 257–266.
[18] D. E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms,* 2nd ed. Reading, MA, USA: Addison-Wesley, 1981.
[19] S. Yan, G. Long, and Y. Zhang, "Streamscan: Fast scan algorithms for gpus without global barrier synchronization," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 229–238.
[20] L.-W. Chang. (2014, Sep.). Scalable parallel tridiagonal algorithms with diagonal pivoting and their optimization for many-core architectures. MS thesis, Univ. of Illinois at Urbana-Champaign. Champaign, IL, USA. [Online]. Available: http://hdl.handle.net/2142/50588

Juan Gómez-Luna received the BS degree in telecommunication engineering from the University of Sevilla, Spain, in 2001 and the PhD degree in computer science from the University of Córdoba, Spain, in 2012. Since 2005, he is an assistant professor at the University of Córdoba. His research interests focus on GPU and heterogenous computing.

I-Jui Sung received the BS and MS degrees from the National Chiao-Tung University in Taiwan, and the PhD degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 2013. His research interests include data layout and memory systems for parallel architectures. He is a software architect at Multicoreware, Inc.

Li-Wen Chang received the BS degree in electrical engineering from National Taiwan University, Taipei, in 2007. He is currently working towards the PhD degree in electrical and computer engineering, the University of Illinois at Urbana-Champaign. His current research interests include parallel computing, heterogeneous computing, optimization for compiler, and high-performance computing.

José María González-Linares received the BS degree in telecommunication engineering from the University of Málaga, Spain, in 1995, and the PhD degree from the University of Málaga, Spain, in 2000. Since 2002, he is an associate professor at the University of Málaga. He has published more than 20 papers in international journals and conferences. His research interests are parallel computing, and video and image processing.

Nicolás Guil received the BS degree in physics from the University of Sevilla, Spain, in 1986 and the PhD degree in computer science from the University of Málaga in 1995. Currently, he is full professor with the Department of Computer Architecture in the University of Málaga. He has published more than 50 papers in international journals and conferences. His research interests are parallel computing, and video and image processing.

Wen-Mei W. Hwu received the PhD degree in computer science from the University of California, Berkeley, 1987. He is the Walter J. (Jerry) Sanders III-Advanced Micro Devices Endowed chair in electrical and computer engineering in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. His research interests are in the areas of architecture, implementation, software for high-performance computer systems, and parallel processing. He is a principal investigator(PI) for the petascale Blue Waters system, is a co-director of the Intel and Microsoft funded Universal Parallel Computing Research Center (UPCRC), and PI for the world's first NVIDIA CUDA Center of Excellence. At the Illinois Coordinated Science Lab, he is the director of the OpenIMPACT project, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. He also serves as the Soft Systems Theme leader of the MARCO/DARPA Gigascale Silicon Research Center (GSRC) and on the Executive Committees of both the GSRC and the MARCO/DARPA Center for Circuit and System Solutions (C2S2). For his contributions to the areas of compiler optimization and computer architecture, he received the 1993 Eta Kappa Nu Outstanding Young Electrical Engineer Award, the 1994 Xerox Award for Faculty Research, the 1994 University Scholar Award of the University of Illinois, the 1997 Eta Kappa Nu Holmes MacDonald Outstanding Teaching Award, the 1998 ACM SigArch Maurice Wilkes Award, the 1999 ACM Grace Murray Hopper Award, the 2001 Tau Beta Pi Daniel C. Drucker Eminent Faculty Award, the 2002 ComputerWorld Honors Archive Medal, and the 2014 B. Ramakrishna Rau Award. From 1997 to 1999, he served as a chairman of the Computer Engineering Program at the University of Illinois. In 2007, he introduced a new engineering course in massively parallel processing, which he co-taught with David Kirk, Chief Scientist of NVIDIA. In 2008, he was named co-director of one of two Universal Parallel Computing Research Centers sponsored by Microsoft and Intel. He is a fellow of the IEEE and of the ACM.