In-Place Transposition of Rectangular Matrices on Accelerators

I-Jui Sung
MulticoreWare, Inc
ray@multicorewareinc.com

Juan Gómez-Luna
University of Córdoba
el1goluj@uco.es

José María González-Linares
Nicolás Guil
University of Málaga
{jgl, nguil}@uma.es

Wen-Mei W. Hwu
University of Illinois at Urbana-Champaign
w-hwu@illinois.edu

Abstract

Matrix transposition is an important algorithmic building block for many numeric algorithms such as FFT. It has also been used to convert the storage layout of arrays. With more and more algebra libraries offloaded 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 GPUs to achieve good performance. In this paper we present the first known in-place matrix transposition approach for the GPUs. Our implementation is based on a novel 3-stage transposition algorithm where each stage is performed using an elementary tiled-wise transposition. Additionally, when transposition is done as part of the memory transfer between GPU and host, our staged approach allows hiding transposition overhead by overlap with PCIe transfer. We show that the 3-stage algorithm allows larger tiles and achieves 3X speedup over a traditional 4-stage algorithm, with both algorithms based on our high-performance elementary transpositions on the GPU. We also show our proposed low-level optimizations improve the sustained throughput to more than 20 GB/s. Finally, we propose an asynchronous execution scheme that allows CPU threads to delegate in-place matrix transposition to GPU, achieving a throughput of more than 3.4 GB/s (including data transfers costs), and improving current multi-threaded implementations of in-place transposition on CPU.

Categories and Subject Descriptors G.4 [*Mathematics of Computing*]: MATHEMATICAL SOFTWARE—Parallel and vector implementations

Keywords GPU, Transposition, In-Place

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2656-8/14/02. . . $15.00.
http://dx.doi.org/10.1145/2555243.2555266

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, or computing linear algebra functions as defined by BLAS libraries.

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 also benefits from 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, such as GotoBLAS [3] and Intel MKL [4], use matrix transposition extensively as well. For instance, MKL includes out-of-place and in-place transposition routines since release 10.1. Moreover, many level 2 and level 3 functions, such as matrix multiplication (`sgemm` and `dgemm`), include a parameter to specify that input matrices are transposed before executing the operation. When offloading these BLAS libraries to the GPU, an efficient transposition becomes important.

Since 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 GPU an attractive platform to execute

the transposition because of its sheer memory bandwidth (to its global memory) compared to CPUs. A GPU implementation of matrix transposition can be used to accelerate CPU transposition as well by transferring the matrix to GPU memory, transposing on GPU, and copying back the transposed matrix to CPU memory. Lastly, it can be used as a building block in more complex GPU applications.

Implementing out-of-place matrix transposition on GPU, that achieves high fraction of peak memory bandwidth, is well understood [5]. However, the memory capacity on GPU is usually a much more constrained resource than its CPU counterpart. If an out-of-place transposition is employed, only up to 50% of the total available GPU memory could be used to hold one or several matrices, that need to be transposed, since the out-of-place transposition has at least 100% spatial overhead. This leads to the need of a general in-place transposition library for the accelerator programming models.

To avoid the high spatial overhead of out-of-place transposition, in-place transposition can be employed, which means the resulting A^T occupies the same physical storage locations as A and there should not be much temporary storage required during transposition. The spatial overhead is either none (i.e. methods that do not require bit flags but with extra computations) [6] or at most a small fraction of the input size (one bit per element) [7].

Mathematically, in-place transposition is a permutation, that can be factored into a product of disjoint cycles [8]. 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. 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. Obtaining a fast parallelization of matrix transposition requires the use of tiling, in order to take advantage of the spatial locality in scratchpads or cache memories. In this paper, we propose a new technique that tackles the general case of arbitrary rectangular matrices. Our proposal is "in-place" in the sense that A^T is placed in the same global memory space as A. Moreover, it does not need much temporary storage in global memory (except 0.1% or less overhead coordination bits), since tiles are temporarily stored in on-chip memories, which are of a very small and bounded size.

The contributions of this paper are as follows:

- This is, as of today, the first known in-place full transposition algorithm for general rectangular matrices on the GPU [9]. We present a full transposition on GPU that is a 4-stage technique based on Gustavson/Karlsson implementation [10, 11]. In each stage, elementary tiled-wise transpositions are carried out. These were developed by Sung et al. [12], but they did not explain how to use them to implement full transposition.
- We propose further a brand new 3-stage tiled transposition scheme that is optimized for GPUs [9], compared to 4-stage Gustavson/Karlsson transposition, plus insights to minimize the search space of tile sizes, which is crucial to obtain high throughput.
- We improve Sung's elementary transpositions by reducing memory contention due to atomic instructions, and by modifying the way the work is distributed among workgroups.
- We develop an asynchronous execution scheme that accelerates CPU matrix transposition by delegating the work to our GPU implementation. Further improvement is obtained by overlapping 2 stages of the GPU transposition with GPU-CPU transfer.
- We compare our in-place matrix transposition on GPU with state-of-the-art out-of-place and in-place implementations for multi-core and many-core CPUs, and discuss under which conditions executing transpositions on an accelerator is profitable.

Our 3-stage in-place transposition achieves on modern GPUs more than 20 GB/s of sustained throughput (calculated as twice the number of bytes of the matrix -once for reading the matrix and once for writing- divided by the total execution time). Moreover, a 3X speedup with respect to the baseline transposition (i.e, 4-stage Gustavson/ Karlsson-style implementation on GPU) is obtained. OpenCL codes of the baseline and the new 3-stage transpositions are publicly available[1].

From the CPU's perspective, our asynchronous execution scheme offers an effective throughput of more than 3.4 GB/s, that is, more than 20% faster than Gustavson/Karlsson implementation on a 6-core CPU. This scheme can be very profitable when in-place transposition is required on CPU due to the use of very large matrices and memory limitations. In these cases, our 3-stage GPU in-place transposition can be applied to transpose these matrices, whose size would be only limited by the GPU memory size.

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 in-place transposition of rectangular matrices can be carried out as a sequence of elementary transpositions. Section 5 explains the low-level optimizations on the elementary transpositions. Section 6 describes how our in-place transposition on GPU can be used to accelerate in-place transposition on CPU. Section 7

[1] https://bitbucket.org/ijsung/libmarshal/wiki/Home

presents the experimental results. Finally, the conclusions are stated.

2. Related Work

2.1 In-Place Transposition and Parallelization for CPUs

As indicated above, most of sequential in-place transposition algorithms can be classified as cycle-following. Berman [7] proposed a bit-table for tagging cycles that have been shifted, and it requires $O(MN)$ bits of workspace for transposing an $M \times N$ matrix. Windley [6] 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 4-stage approach [10, 11, 14], in order to improve cache locality. Moreover, Gustavson et al. [11] proposed parallelization for multicores up to 8-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 [5] studied optimizations for out-of-place transposition. Sung et al. [12] 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 [15] 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 such as ours to increase the maximum size of dataset allowed for GPU offloading.

3. Definition of Matrix Transposition

Assume that A is an $M \times N$ matrix, where $A(i, j)$ is the element in row i and column j. The transpose of A is an $N \times M$ 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 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:

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

where $\mathcal{M} = M \times N - 1$ [11].

```
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];
}
```

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

The expression in Equation (1) allows us to calculate the destination for a matrix element. Since we are moving elements in-place, the original element in the destination 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 Figure 1 shows. In this kernel, called Barrier-sync (BS), one work-group transposes a 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.

4. In-place Transposition of Rectangular Matrices

In a 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. $M = 5, N = 3, \mathcal{M} = M \times N - 1 = 14$. We start with element 1, or the location of $A(0, 1)$. The content of element 1 should be moved to the location of element 5, or the location of $A^T(1, 0)$. The original content at the location of element 5, or the location of $A(1, 2)$, is saved before being overwritten and moved to location of element 11, or the location of $A^T(2, 1)$; 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 [11] targeting multicores parallelize by assigning each cycle to a thread. As cycles by definition never

overlap, they are an obvious source of parallelism that could be exploited by parallel architectures. In [12] 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 several times the lengths of other cycles. This creates significant load imbalance problem.

Sung et al. [12] have proposed an atomic-operation-based approach to coordinate the shifting to reduce load imbalance. The gist of their method, called PTTWAC, is to have multiple threads participating in the shifting of elements in one cycle, and use atomic operations to coordinate the shifting among threads. However, the problem is that modern GPUs lack bit-addressable atomic operations as the smallest addressable unit is a 4-byte word. As we shall show in Section 5, simulating atomic bit operations naïvely can lead to significant performance loss due to the conflicts among concurrent threads. In this regard, we have also been inspired by recent works on histogram calculation, which is a class of atomic-intensive application that has attracted many research efforts in the GPU computing community. They minimize the impact of atomic conflicts by replicating the histogram in local memory in combination with the use of padding [16] or a careful layout [17].

4.1 Full Transposition As a Sequence of Elementary Tiled Transpositions

Good locality is crucial for modern memory hierarchies. Therefore staged transpositions that trade locality with extra data movements can be favorable. A full transposition of a matrix can be achieved by a series of blocked transpositions in four stages [10, 11, 14]. As shown further, on a modern NVIDIA K20 GPU, a 4-stage Gustavson/Karlsson-style in-place transposition reaches around 7 GB/s with optimized blocked transposition whereas a single-stage in-place transposition only runs at 1.5 GB/s, due to poor locality.

A Gustavson/Karlsson style transposition first considers an $M \times N$ matrix as an $M' \times m$ by $N' \times n$ matrix where $M = M' \times m$ and $N = N' \times n$. Then the elementary transpositions are designed in such a way that they only swap adjacent dimensions among the four dimensions. In this case, the problem becomes finding a sequence of elementary transpositions to reach $N' \times n \times M' \times m$. We employ the factorial numbering system [18] to refer each stage: Table 1 lists possible permutations that refer to swapping of adjacent dimensions. Intuitively, each digit of the factorial number for a particular permutation can be thought as an item from an imaginary queue of items, with offset starting from zero for the leftmost element. If we insert items from the right end of the queue and take the items from the left end of the queue, we maintain the original order. However, when an item reaches the left end, if we take its right neighbor instead for the next turn, we reverse the order between the two items. If we have 4 items, (A, B, C, D) in the queue, we can generate a sequence of 4 numbers by generating 0 whenever we remove the leftmost item (offset 0) and 1 for the item right to the leftmost item (offset 1). So if we reverse the order between B and C, we would generate $0100_!$, which is the factorial number for a permutation from (A, B, C, D) to (A, C, B, D).

Table 1. Permutations in Factorial Numbering System.

#Dimensions	From	To	Factorial Num.	Sung's terminology [12]
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 elementary transpositions were used by Sung et al. [12] 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 (where N is the number of elements in each structure), and each of these $m \times N$ tiles is assigned one work-group (in OpenCL terminology), that is in charge of transposing the corresponding tile. Thus, their AoS-to-ASTA marshaling is essentially an elementary transposition that converts $M' \times m \times N$ (AoS) to $M' \times N \times m$ (ASTA). Similarly, their SoA to ASTA (i.e. transposition $100_!$) transformation 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.

Figure 2 illustrates our 4-stage implementation of a full in-place transposition based on [10], which employs the mentioned elementary tiled transpositions. Initially, the matrix is considered as a 4D array. Then, 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. Although $m \times N'$ can be large, this stage always moves super-elements of size n that can be tuned to fit cache line and/or DRAM burst size, thus maintaining locality. The second stage employs $0010_!$ to transpose $M' \times N'$ instances of $n \times m$ matrices. This stage can be realized by holding a temporary array of $n \times m$ in fast on-chip memory of GPUs for each instance. The third stage, which applies the factorial $1000_!$, can be considered as one instance of transposition of an $N' \times M'$ array of super-element sized $n \times m$. The fourth stage is similar to the first one but with a different dimensionality.

4.2 3-Stage Full In-place Transposition on GPU

The transposition $1000_!$ in the 4-stage approach moves super-elements of $m \times n$ elements, so that its best perfor-

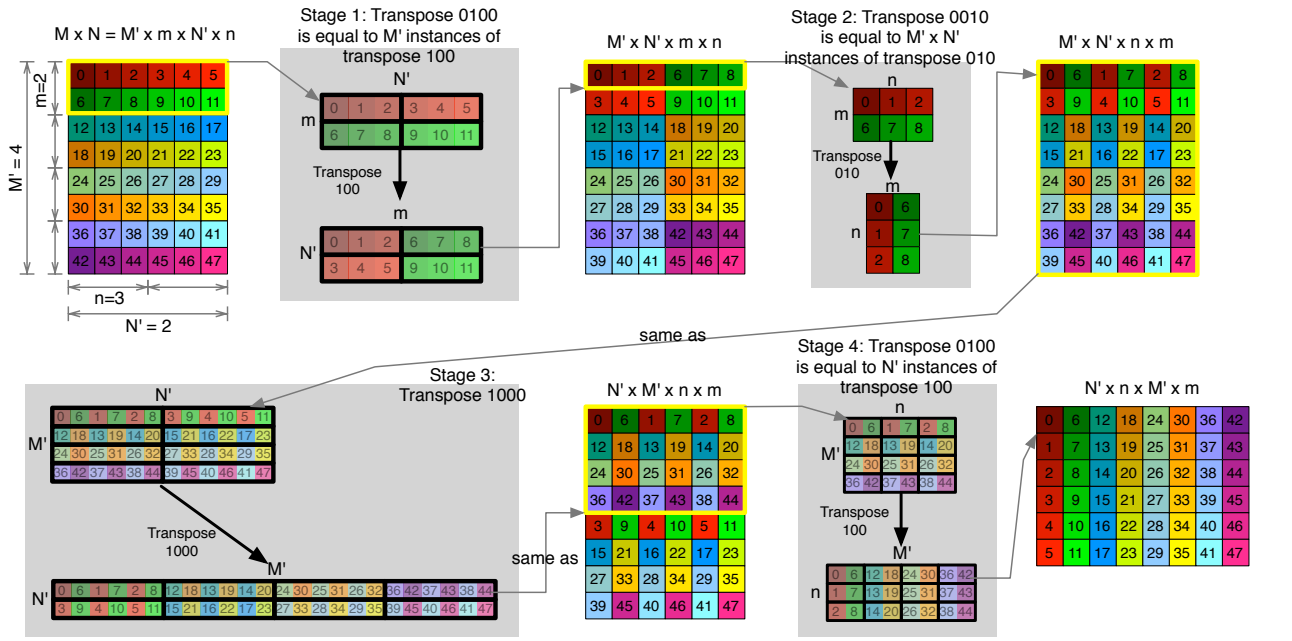


Figure 2. 4-stage full in-place transposition. In every figure, memory addresses increase from left to right and from top to bottom. Yellow halos indicate the part of the matrix that is brought into focus in the subsequent step. Black halos represent super-elements, which are shifted as a whole.

mance 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 stage 1 and 4, where the size of the super-elements is only n and m, respectively.

To address this problem, we propose to eliminate the intermediate transposition $1000_!$ without sacrificing locality. One such improved 3-stage approach is:

1. Treat matrix $M \times N$ as a 3-dimensional array of $M \times N' \times n$. Perform transposition of n-sized super-elements, i.e. $M \times N' \times n$ to $N' \times M \times n$. This is transposition $100_!$.
2. Treat matrix $N' \times M \times n$ as a 4-dimensional array of $N' \times M' \times m \times n$. Perform $N' \times M'$ instances of transposition of $m \times n$ matrices, i.e. $N' \times M' \times m \times n$ to $N' \times M' \times n \times m$. This is transposition $0010_!$.
3. Perform N' instances of transposition of m-sized super-elements, i.e. $N' \times M' \times n \times m$ to $N' \times n \times M' \times m$. This is transposition $0100_!$.

In this improved algorithm, there are only three steps, and a much larger values of m and n can be used in the first and the third stage respectively for transposition $0100_!$ without overflowing the on-chip memory.

5. Performance Improvements for Elementary Transpositions

Sung et al. [12] suggests parallelization strategies that are useful for the elementary transpositions shown in the previous section. However, their transposition algorithms suffer from the following bottlenecks, if implemented literally. On the one hand, AoS-ASTA transformation is burdened by serious atomic memory contention. On the other hand, SoA-ASTA transformation has several limitations related to the work-group size, that are detailed below. In the following sections we describe improvements to those transposition building blocks.

5.1 Transposition $010_!$ (aka AoS-ASTA)

Sung et al. present two versions of transposition $010_!$: the first one is the fast Barrier-sync (BS) kernel, that we have already shown in Figure 1. As we also pointed out, for BS the tile size (the product of sizes of the lowest two dimensions) is limited, since it cannot exceed the size of on-chip memory accessible to a work-group (i.e., OpenCL local memory or register file). The second one is devised for large tiles and is based on the PTTWAC algorithm.

In their PTTWAC-based algorithm for large tiles, each work-item of a work-group shifts scalar values inside a tile directly from global memory. In order to ensure load balancing and coalesced global memory reads, adjacent work-items start to read adjacent elements and then follow the corresponding cycle. Recall that given a current element, the next one in the cycle is calculated by Equation (1).

One 1-bit flag per element per tile is stored in local memory, so that work-items can mark the elements they shift. When one work-item finds a previously set flag, it terminates. Sung et al. pack the flag bits in local memory 32-bit words using an intuitive layout. The local memory word

$Flag\_word$, where the flag bit for element $Element\_position$ is stored, is given by Equation (2). $Element\_position$ stands for the one-dimensional index of an element within a tile.

$$Flag\_word = \left\lfloor \frac{Element\_position}{32} \right\rfloor \quad (2)$$

Reading or setting a flag is a 1-bit atomic operation. Since the smallest hardware atomic operation size available is 32 bits wide, an atomic logic OR function is used to simulate bit-addressable atomics. This will cause conflicts among work-items updating flags in the same 32-bit word. Particularly burdening are intra-warp atomic conflicts[2], as explained by Gómez-Luna et al. [19]. In that work, the authors showed the latency is roughly increased by a factor equal to the number of colliding threads, which is called position conflict degree.

5.1.1 Spreading the Flag Bits

The position conflict degree can be diminished by spreading the flag bits over more local memory words. In Equation (3), the spreading factor stands for the reduction in the number of flag bits per local memory word. Thus, the maximum spreading factor is 32, unless the local memory available becomes a constraint [3].

$$Flag\_word = \left\lfloor \frac{Element\_position \times Spreading\_factor}{32} \right\rfloor \quad (3)$$

5.1.2 Padding to Reduce Bank and Lock Conflicts

When using transposition $010!$ in the second stage of the previously proposed full in-place transposition, the tile dimensions $m \times n$ are determined by the factors of the matrix dimensions $M \times N$. Thus, typical values m and n might be power-of-2. And recall that Equation (1) multiplies the offset by m. So a power-of-two value of m will cause new conflicts that are even more frequent when spreading the flags, as explained in Figure 3 (a) and (b). These new conflicts can be categorized as bank conflicts and lock conflicts [19]. Bank conflicts are due to concurrent reads or writes to different addresses in the same local memory bank. Lock conflicts are caused by the limited number of locks associated to atomic operations that are available in the hardware. This produces a similar effect to position conflicts.

Padding can be used to remove both types of conflicts. This optimization technique consists of keeping some memory locations unused, in order to shift the bank or lock accessed by concurrent threads. For instance, as the NVIDIA Fermi architecture contains 32 local memory banks and 1024 locks, inserting one unused location each 32 words will remove most bank and lock conflicts. This is shown in Figure 3 (c).

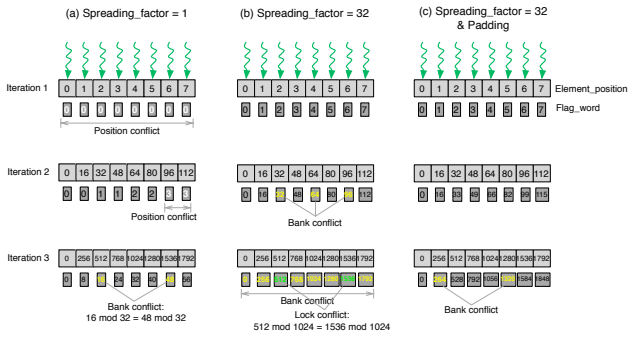


Figure 3. Consecutive work-items access consecutive elements in iteration 1. In the following iterations, the next elements in the cycle are computed with Equation (1). In this example, $m = 16$ and $n = 215$. Representative conflicts are highlighted: position conflicts (white), bank conflicts (yellow), lock conflicts (green). In case (a), the flag word is obtained through Equation (2). Many position conflicts appear. In case (b), the flag words are obtained with Equation (3). Position conflicts are removed, but bank and lock conflicts appear. 32 banks and 1024 locks are considered, as shown by Gómez-Luna et al. for NVIDIA Fermi architecture. In case (c), the use of padding avoids the lock conflicts and most bank conflicts.

5.2 Transposition $100!$ (aka SoA-ASTA)

Sung's implementation of SoA-ASTA transformation essentially converts from $N \times M' \times m$ to $M' \times N \times m$. Adjacent m elements are treated as a super-element that is then shifted in order to obtain the ASTA layout. Such a task is carried out by $N \times M'$ work-groups of m work-items. Thus, coordination between work-groups must be done with atomic operations on global memory. Hence, the shared-memory-oriented optimization techniques above are not applicable here.

Optimization efforts on this kernel can be oriented to overcome some limitations in Sung's implementation that are related to the fact that m is derived from the factors of the matrix dimensions:

1. The runtime imposes a maximum limit on the number of active work-groups in a GPU (e.g., 8 per $streaming\ multiprocessor$ in NVIDIA Fermi). This entails low occupancy (i.e., the ratio of active work-items to the maximum possible number of active work-items) because of typical values of m (between 8 and 64). For instance, $m = 32$ means 16% occupancy for Fermi while the minimum recommended is 50% [20].
2. Every m that is not a multiple of the SIMD unit size[4] entails idle work-items, that is, further occupancy reduction.

[2] Warps are SIMD units in NVIDIA devices. AMD counterparts are called wavefronts.

[3] Practically, we could use any spreading factor up to 16, because 32 would entail 100% local memory overhead, that would allow us to use the faster BS kernel.

[4] NVIDIA's warp = 32 work-items; AMD's wavefront = 64 work-items.

3. If m is larger than the SIMD unit size, barriers are needed to synchronize the SIMD units belonging to the same work-group. This degrades performance, as SIMD units need to wait for each other.
4. The maximum possible m is limited to the maximum number of work-items per work-group (only 256 in AMD devices).

5.2.1 Improving Flexibility and Performance

The aforementioned limitations can be alleviated by using one SIMD unit to move m elements, instead of one work-group as in [12]. This increases occupancy, saves costly barriers, and expands the value range of m.

In Sung's implementation, each element of a super-element was temporally stored in one register per work-item. Since m might be longer than the SIMD unit size in our approach, local memory tiling is required in the pursuit of flexibility. First, each SIMD unit will need several iterations to store its m elements in local memory. Then, the SIMD unit will move its m elements to the new location in global memory.

Further performance improvement can be achieved for particular cases where m is a divisor or a multiple of the SIMD unit size using register tiling, because register accesses are faster than local memory accesses [21].

6. Using GPU Full In-place Transposition from CPU Host

The high memory bandwidth of GPUs makes them attractive to accelerate matrix transposition. Out-of-place matrix transposition on GPU [5] has demonstrated a very high throughput, but it is not suitable for large matrices, as it needs 100% memory overhead. In these cases, our 3-stage in-place approach can be employed. It is not strictly in-place by the definition from CPU's perspective as we are using 1X memory in the accelerator, but still the in-place algorithm works for datasets up to 100% of GPU accelerator's on-board memory theoretically. Thus, in-place matrix transposition is virtually executed on CPU, but physically executed on GPU. Figure 4 shows a high-level plan on how to implement this. The entire matrix must be transferred from CPU to GPU through the PCIe bus (1). Then, the in-place transposition is executed on GPU (2). Finally, the matrix is copied from GPU memory to the same location in CPU memory (3). The total execution time (including data transfers) will determine the effective throughput from CPU's perspective.

The use of several concurrent command queues can help us to accelerate this in-place matrix transposition. Using more than one command queue in OpenCL codes allows programmers to overlap data transfers and computation on a heterogenous system. This is a way to alleviate the bottleneck caused by data transfers.

OpenCL command queues are similar to CUDA streams, which were thoroughly studied in [22]. They are defined as sequences of operations that are executed in order, while

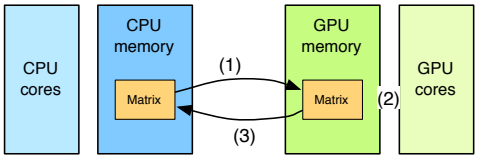


Figure 4. Scheme of an in-place matrix transposition on CPU memory using a GPU. In (1) the matrix is transferred to GPU memory. In-place matrix transpose is performed on GPU (2). In (3) the matrix is moved back to the same location in CPU memory.

different streams are executed asynchronously[5]. Data transfers and computation can be divided into a number of command queues. Thus, data transfers belonging to one command queue can be overlapped with computation belonging to a different command queue. Similar to the asynchronous scheme we propose below, in [23] data transfers are overlapped with memory layout reorganization kernels using CUDA streams.

As explained in Section 4.1, stages 2 and 3 in our 3-stage transposition execute independent instances of transpose $010_!$ and transpose $100_!$, respectively. These independent instances work with separate memory areas. Thus, they can be executed asynchronously: work-groups can be split into Q command queues (see Figure 5 (b) with $Q = 4$). This will allow us to overlap stages 2 and 3 and GPU-CPU transfer.

The number Q can range from 1 to a maximum number that still keeps the GPU multiprocessors busy (i.e., with the same occupancy as the synchronous execution) and leverages the PCIe bandwidth. Moreover, it should also be taken into account that the creation of multiple command queues entails an overhead.

Unfortunately, stage 1 (transpose $100_!$) cannot be overlapped with data transfers. This elementary transposition is made up of a number of cycles that shift super-elements across the entire memory space where the matrix is located. For this reason, transpose $100_!$ cannot be divided into independent command queues.

7. Experimental Results

Experiments in this section have been performed on two current NVIDIA devices and one AMD device, using single precision versions of the algorithms. NVIDIA GeForce GTX 580 with Fermi architecture has a peak bandwidth 192.4 GB/s. The recently released NVIDIA Tesla K20 with Kepler architecture achieves up to 208 GB/s. The AMD Radeon HD7750 Cape Verde has a peak memory bandwidth 72 GB/s.

7.1 Transposition $010_!$

The effect of spreading and padding on throughput has been measured with the same inputs used by Sung et al. [12] Fig-

[5] OpenCL also supports out-of-order execution within a command queue

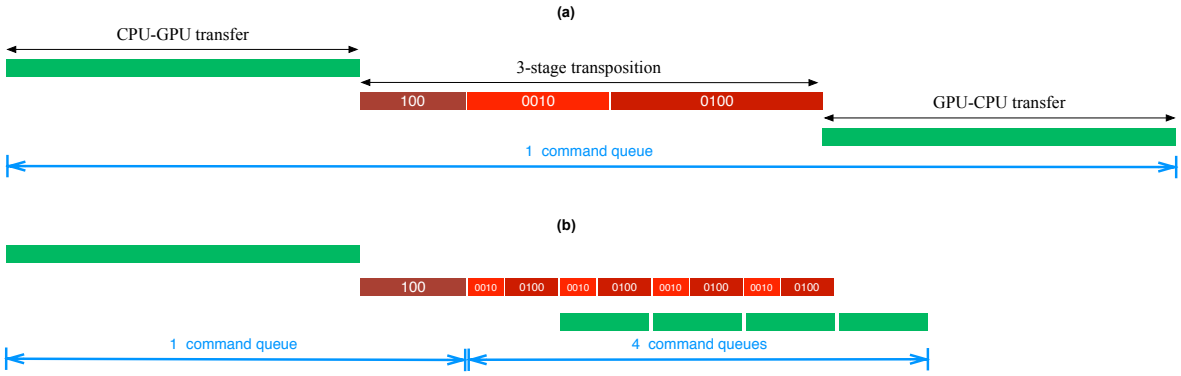


Figure 5. Timeline comparison. (a) Synchronous execution of the 3-stage in-place GPU transposition. (b) Asynchronous execution of the 3-stage in-place GPU transposition. Stages 2 and 3 and GPU-CPU transfer are divided into 4 command queues.

ure 6 shows the results on an NVIDIA Tesla K20 GPU. The experiment resizes matrices from $M' \times m \times n$ to $M' \times n \times m$. The reduction in the amount of position conflicts produces in average 1.77× increased throughput. Moreover, the use of padding minimizes the bank and lock conflicts, so that 12% additional improvement is achieved. Some significant performance drops are noticeable when increasing the spreading factor (e.g., test problem bccstk31 with m equal to 32 and 64). These are caused by an occupancy value under 50%, due to the increase of local memory needs.

We have compared our optimized PTTWAC (with spreading and padding) to the original algorithm for n between 16 and 256 (in steps of 1) and m between 16 and 64 (in steps of 1). The average (minimum / maximum) speedup is 1.85 (1.36 / 3.49) on NVIDIA GeForce GTX 580, 1.79 (1.30 / 5.29) on NVIDIA Tesla K20, and 1.90 (1.15 / 3.34) on AMD Cape Verde.

The optimized PTTWAC has been compared to the P-IPT version [12], introduced in Section 1. P-IPT outperformed the original PTTWAC algorithm in some well load-balanced cases, but it is defeated by the optimized PTTWAC.

7.2 Transposition $100_!$

The new version of the transpose $100_!$ results in impressive speedups compared to Sung's original implementation on NVIDIA devices. Experiments resize from $N \times M' \times m$ to $M' \times N \times m$. We have tested with matrices of m between 16 and 64 (in steps of 1), and M' between 16 and 256 (in steps of 1). In order to obtain the highest throughput, the work-group size has been chosen to maximize the occupancy. Analyzing the need for registers and local memory, we noticed that the occupancy is limited on Fermi by the number of registers (22 registers per thread). Thus, the highest occupancy is obtained for 192 threads/block. On Kepler, such a limitation does not appear. The highest occupancy can be obtained with a number of threads per block that is a multiple of 128.

The average (minimum / maximum) speedup is equal to 2.95 (1.97 / 4.09) on GTX 580 and 2.58 (1.54 / 3.50) on K20, when using local memory tiling. Register tiling can be applied for m that is multiple or divisor of the warp size. In these cases, performance further increases by 16% on GTX 580 and 23% on K20. The P-IPT version [12] is always outperformed by these new versions. Unfortunately, on AMD we did not observe speedups (albeit being more flexible, as explained in Section 5.2).

Table 2. Throughput of our 3-stage approach and Karlsson/-Gustavson 4-stage approach on a Kelper K20. Best performing tile sizes have been used. Both implementations include the low-level optimizations presented in Section 5.

	3-stage	4-stage (+fusion)
7200×1800	20.59 GB/s	7.11 (7.67) GB/s
5100×2500	18.49 GB/s	6.87 (7.38) GB/s
4000×3200	20.73 GB/s	7.23 (7.79) GB/s
3300×3900	18.80 GB/s	7.23 (7.79) GB/s
2500×5100	17.29 GB/s	6.86 (7.37) GB/s
1800×7200	18.70 GB/s	7.07 (7.60) GB/s

Figure 7.2 shows the throughput of transposition $100_!$ on K20 and Cape Verde for values of m and M' under 256. The best performance on K20 is obtained with m between 64 and 160. This range ensures enough work per work-item in the warp, and does not reduce the occupancy due to local memory needs for tiling. Similar results have been observed on GTX 580. On Cape Verde the best performance results occur with m over 128 (wavefront size doubles warp size). Local memory needs are not an issue, because the amount of local memory per wavefront is larger. To maximize occupancy, we typically use 40 wavefronts for 64 KB of local memory on Cape Verde [24], while 64 warps for 48 KB of local memory on K20 [20].

7.3 3-Stage and 4-Stage Transposition

Table 2 summarizes the throughput difference on a Tesla K20 of our 3-stage approach compared to the 4-stage version, which we have developed following the original ap-

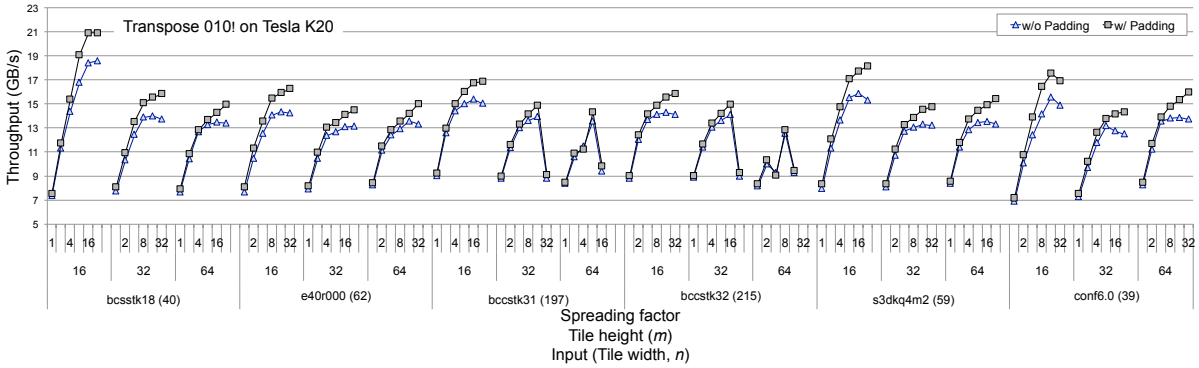
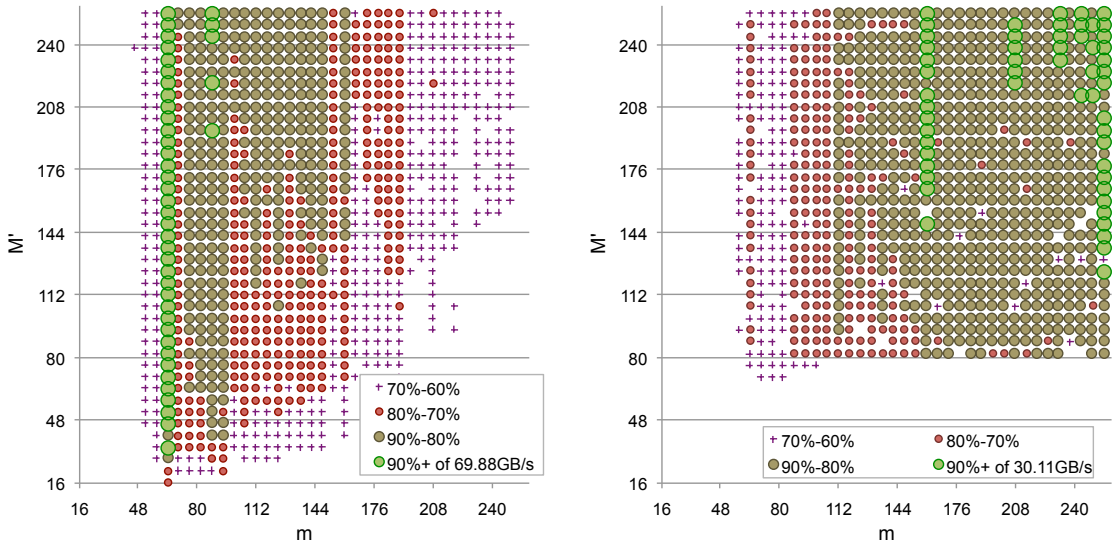


Figure 6. Effect of spreading and padding on Tesla K20. Transposition $010_!$ converts from $M' \times m \times n$ to $M' \times n \times m$. Six inputs are used [12]. The value within parentheses is the tile width n. Three values of the tile height m are tested (16, 32, 64). The spreading factor changes between 1 and 32 for every case.



(a) Tesla K20.

(b) AMD Radeon HD7750.

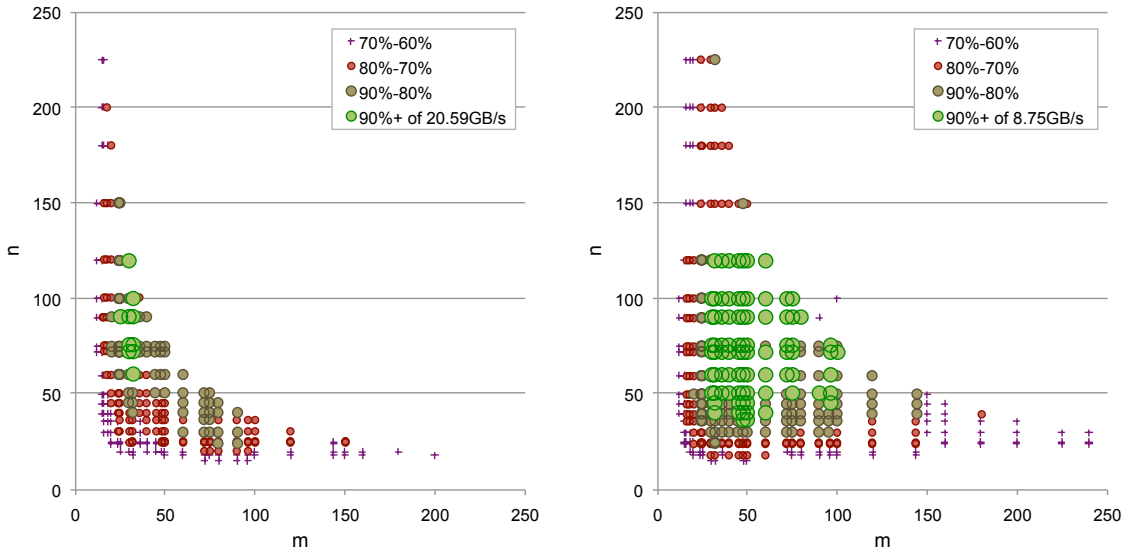
Figure 7. Throughput (GB/s) of transpose $100_!$ for m and M' under 256. The experiment converts from $N \times M' \times m$ to $M' \times N \times m$.

proach by Gustavson [11] and Karlsson [10], using the dataset configuration from their paper. Both approaches are implemented using the same set of elementary transposition routines.

Note as also pointed out by Karlsson and Gustavson, the stage 2–3 in the 4-stage approach in Figure 2 could be fused. We present the throughput of their approach with fusion in the second column (values inside parentheses) of Table 2. The reason why our 3-stage method is significantly faster than the 4-stage method is not only eliminating one stage (which can be achieved by fusion in 4-stage approach anyway), but the 3-stage algorithm allows much bigger tile sizes which is crucial for transposition $100_!$ including derived $0100_!$, and $1000_!$. For Tesla K20, the throughput of transpositions $100_!$ et al., is dominated by tile size used: 12.5 GB/s for tile size 8, 24.5 GB/s for tile size 16, 47.6 GB/s for tile size 32, 69 GB/s for tile size 64 on average. In fact, the best performing tile sizes (m, n) for transposing a 7200×1800 matrix is $(20, 16)$ for 4-stage transposition, but $(32, 72)$ for the 3-stage algorithm on a Tesla K20.

7.4 Choosing Tile Sizes for Full Transposition

Tile sizes are crucial to the throughput of full transposition. Naïvely, we could exhaustively search on all possible m and n combination and use the best one, but that is too time-consuming especially for M and N values having many possible dividends. We can prune the search space by taking into consideration these three factors: Transposition $100_!$ and $0100_!$ obtain a better throughput if the tile size is larger. This limits the stage 1 (tile size = n) and stage 3 (tile size



(a) Tesla K20. The best-performing ones all fall in a very small range of tile sizes

(b) AMD Radeon HD7750.

Figure 8. Tile Sizes versus Performance

= m). Transposition $0010_!$ obtains a better throughput if the tile (in this case $m \times n$) fits into shared memory, because the barrier-synchronization kernel can be used.

Figure 8(a) plots some of the best combinations of tile sizes (m and n) in a 7200x1800 in-place transposition on one Tesla K20 (Kepler). The best ones achieved a throughput of 20.59 GB/s in an exhaustive search. It is clear that the tile sizes that lead to best throughput (80%+ of the best performing combinations) are actually within a very small subset roughly along the curve of $m \times n < 3600$ (which is roughly the shared memory capacity) and with mostly m and n around 60. Figure 8(b) shows a similar trend but on an AMD Radeon HD7750 (Cape Verde). We can see that for AMD GPUs the best performing combinations are also confined in a small region, but the shape is different from an NVIDIA GPU. For all three GPUs, a good guess for m and n will be from 50 to 100 with $m \times n$ less than the maximal shared memory capacity: this simple heuristic can yield at least 80% of the best throughput.

That exposes the only limitation of our algorithm. When the algorithm cannot choose a good tile size (e.g., prime-number dimensions), the throughput would be degraded [6].

7.5 Comparison to Matrix Transposition on CPU

We compare our 3-stage in-place transposition on GPU with Intel MKL [4] and Gustavson/Karlsson [11] out-of-place and in-place transpositions on CPU. From Intel MKL we use `mkl_somatcopy()` and `mkl_simatcopy()`, single precision routines for out-of-place and in-place transposition, respectively. Gustavson/Karlson original implementations are double precision. We have moved them to single precision for a fair comparison. Figure 9 shows the throughput results for the best configurations. The same matrix sizes as in Section 7.3 have been used. CPU transposition implementations have run on a 6-core 3.47 GHz Intel Xeon W3690 and our 3-stage GPU in-place transposition on a Tesla K20.

It is remarkable that MKL in-place transposition is a sequential implementation. Thus, it achieves less than 0.1 GB/s, so that the corresponding column is not represented in the figure. MKL out-of-place transposition is a parallel implementation. Its throughput is not limited by the number of cores, but the memory bandwidth, since it does not scale for more than 4 threads. As it entails 100% memory overhead, its main drawback is that it is not suitable for applications in which the memory resources are constrained.

Gustavson/Karlsson implementations are also parallel and launch as many threads as CPU cores. The average throughput of their in-place transposition is 2.85 GB/s. Our 3-stage implementation on GPU is up to 6.7X faster than Gustavson/Karlsson's.

If our 3-stage implementation on GPU is used by a CPU thread, both data transfers (CPU-GPU and GPU-CPU) times must be added when using a synchronous execution scheme. For those matrix sizes, both transfers take around 15 ms to complete. Thus, the resulting average throughput from CPU's perspective is 2.87 GB/s, that is, slightly faster than Gustavson/Karlsson's method on CPU. Using an out-of-place transposition on GPU [5], instead of our 3-stage in-place approach, would only result in a minimally higher

[6] Contemporary to the preparation of this paper is another work [25] that addresses this limitation for a particular architecture (NVIDIA Kepler), obtaining a comparable performance.

Table 3. Assessment of in-place and out-of-place matrix transposition methods on CPU and GPU. Throughput figures are average results for the matrix sizes in Table 2. CPU codes have been tested on a 6-core Xeon, while GPU implementations have run on a Tesla K20.

Implementation	Executed on...	Throughput (GB/s)	CPU memory overhead	GPU memory overhead
Intel MKL out-of-place [4]	6 CPU cores	12.07	100%	-
Intel MKL in-place [4]	1 CPU core	< 0.1	0%	-
Gustavson/Karlsson out-of-place [11]	6 CPU cores	2.36	100%	-
Gustavson/Karlsson in-place [11]	6 CPU cores	2.85	0%	-
GPU out-of-place [5] + data transfers	GPU cores	3.57	0%	100%
3-stage GPU in-place + data transfers	GPU cores	3.43	0%	$\simeq$0%

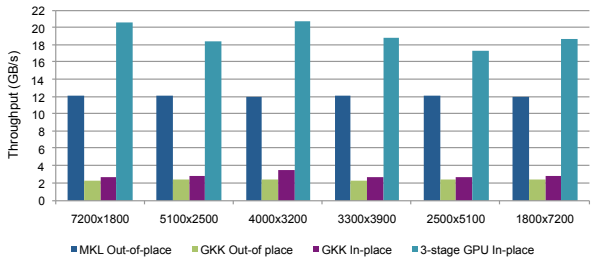


Figure 9. Throughput results for Intel MKL out-of-place transposition on CPU, Gustavson/Karlsson (GKK) out-of-place and in-place transposition on CPU, and 3-stage in-place transposition on GPU. The throughput of MKL in-place transposition is < 0.1 GB/s in all cases. CPU results have been obtained on a 6-core Xeon. GPU results correspond to a Tesla K20.

overall performance: data transfers are still the main performance bottleneck, even though the out-of-place transposition achieves more than 120 GB/s on a K20. Furthermore, our in-place approach has the additional advantage that it can transpose larger matrices, as it needs a negligible memory overhead.

7.6 Overlapping Stages 2 and 3 with GPU-CPU Transfer

As explained in Section 6, stages 2 and 3 of our 3-stage in-place GPU transposition can be overlapped with GPU-CPU data transfer. We have evaluated this asynchronous execution scheme with the same matrix sizes and all possible combinations of m and n (6444 tests).

In these tests, the asynchronous execution scheme (Q command queues) outperforms the synchronous execution scheme (1 command queue) by an average 9% and a maximum 24%. In the asynchronous execution, the best number Q is typically under 8. Larger values lower the resulting throughput because of the overhead derived from the creation of the command queues [22]. Such a throughput degradation is not attributable to an underutilization of either GPU resources (the occupancy value is maintained) or PCIe bandwidth (data transfers are still larger than 1 MB, which ensures a linear timing behavior of PCIe transfers [26]).

If only the best configurations are considered, the asynchronous execution scheme increases the effective average throughput from CPU's perspective from 2.87 to 3.43 GB/s, that is, 19% improvement. Thus, this virtual in-place transposition achieves more than 20% speedup with respect to Gustavson/Karlsson in-place implementation on CPU.

As a summary, Table 3 compares all in-place and out-of-place implementations from CPU's perspective. Throughput results and memory overheads are presented.

It can be noticed that our 3-stage GPU in-place transposition has GPU memory overhead approximately equal to 0%. This is thanks to using on-chip memory for temporary storage. An insignificant global memory overhead is due to coordination bits in elementary transposition $100_!$, as explained in Section 5.2. Such an overhead is negligible because only one bit per super-element is needed. Thus, it depends on the number of super-elements, not the size of the matrix. Assuming m and n between 50 and 100 (as indicated in Section 7.4), the overhead is less than 0.1%.

7.7 Testing our implementations on a non-GPU accelerator

As our 4-stage and 3-stage in-place matrix transpositions are implemented in OpenCL, we have tested them on a 60-core Intel Xeon Phi [27]. The average results of the best configurations for the matrix sizes in Table 2 are 2.81 and 5.02 GB/s for our 4-stage and our 3-stage implementations, respectively. Thus, the 3-stage approach improves throughput by 1.8X.

The lack of on-chip scratchpad memory on Xeon Phi makes these implementations not strictly in-place, since OpenCL local memory is emulated on the regular GDDR memory. We leave for future work an implementation with a reduced memory overhead.

8. Conclusion

We have presented the design and implementation of the first known general in-place transposition of rectangular matrices for modern GPUs. We have enhanced both the performance of building blocks proposed by earlier works as well as the overall staged approach. Combined with insights that lead to greatly improved performance of elementary tiled transformations, a new 3-stage approach that is efficient for the GPUs is presented and we have shown that this is much faster than traditional 4-step approaches. We have also observed that the tile size greatly affects performance of

in-place transposition, especially for the GPUs since it can affect the algorithm choice due to hardware limitations of on-chip resources. Though the search space for tile sizes can be big, we have also identified pruning criteria that helps user to choose good tile sizes for current GPUs. Finally, we have proposed an asynchronous execution scheme that allows CPU threads to transpose in-place, obtaining 20% speedup to the fastest state-of-the-art in-place transposition for multi-core CPUs. As a future work, besides the above mentioned implementation for Xeon Phi, we plan to extend our work to multi-GPU environments. We believe that our efficient 3-stage approach can be used as a building block for a multi-GPU version.

Acknowledgments

This project was partly supported by the STARnet Center for Future Architecture Research (C-FAR), the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), and the UIUC CUDA Center of Excellence. We also thank NVIDIA for a hardware donation to the University of Málaga under CUDA Research Center 2012-2013 Awards, and the Ministry of Education of Spain for financial support (TIN2010-16144). Access to Xeon Phi was kindly provided by J.R. Bilbao-Castro, and financed by a Ramón y Cajal fellowship by the Ministry of Economy and Competitiveness of Spain.

References

[1] Frigo, M., Johnson, S.: The design and implementation of fftw3. Proceedings of the IEEE **93**(2) (2005) 216–231

[2] Kohlhoff, K., Pande, V., Altman, R.: K-means for parallel architectures using all-prefix-sum sorting and updating steps. IEEE Transactions on Parallel and Distributed Systems **24**(8) (2013) 1602–1612

[3] Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3) (May 2008) 12:1–12:25

[4] Intel MKL: Intel Math Kernel Library (January 2013)

[5] Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA. (January 2009)

[6] Windley, P.F.: Transposing matrices in a digital computer. The Computer Journal **2**(1) (1959) 47–48

[7] Berman, M.F.: A method for transposing a matrix. J. ACM **5**(4) (October 1958) 383–384

[8] Hungerford, T.: Abstract algebra: an introduction. Saunders College Publishing (1997)

[9] Sung, I.J.: Data layout transformation through in-place transposition. PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering (May 2013) `http://hdl.handle.net/2142/44300`.

[10] Karlsson, L.: Blocked in-place transposition with application to storage format conversion. Technical report (2009)

[11] Gustavson, F., Karlsson, L., Kågström, B.: Parallel and cache-efficient in-place matrix storage format conversion. ACM Transactions on Mathematical Software **38**(3) (April 2012) 17:1–17:32

[12] Sung, I.J., Liu, G., Hwu, W.M.: DL: A data layout transformation system for heterogeneous computing. In: Innovative Parallel Computing, InPar. (May 2012) 1 –11

[13] Cate, E.G., Twigg, D.W.: Algorithm 513: Analysis of in-situ transposition [f1]. ACM Trans. Math. Softw. **3**(1) (March 1977) 104–110

[14] Kaushik, S.D., Huang, C.H., Johnson, J.R., Johnson, R.W., Sadayappan, P.: Efficient transposition algorithms for large matrices. In: Supercomputing. (November 1993)

[15] Dotsenko, Y., Baghsorkhi, S.S., Lloyd, B., Govindaraju, N.K.: Auto-tuning of fast fourier transform on graphics processors. In: Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming. PPoPP '11, New York, NY, USA, ACM (2011) 257–266

[16] Gómez-Luna, J., González-Linares, J.M., Benavides, J.I., Guil, N.: An optimized approach to histogram computation on gpu. Machine Vision and Applications **24**(5) (2013) 899–908

[17] Van den Braak, G.J., Nugteren, C., Mesman, B., Corporaal, H.: GPU-vote: A framework for accelerating voting algorithms on GPU. In Kaklamanis, C., Papatheodorou, T., Spirakis, P., eds.: Euro-Par Parallel Processing. Volume 7484 of Lecture Notes in Computer Science. (2012) 945–956

[18] Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley (1981)

[19] Gómez-Luna, J., González-Linares, J.M., Benavides, J.I., Guil, N.: Performance modeling of atomic additions on GPU scratchpad memory. IEEE Transactions on Parallel and Distributed Systems **24**(11) (2013) 2273–2282

[20] NVIDIA: CUDA C Programming Guide 5.0 (July 2012)

[21] Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Supercomputing, Piscataway, NJ, USA, IEEE Press (2008) 31:1–31:11

[22] Gómez-Luna, J., González-Linares, J.M., Benavides, J.I., Guil, N.: Performance models for asynchronous data transfers on consumer graphics processing units. Journal of Parallel and Distributed Computing **72**(9) (2012) 1117 – 1126 Accelerators for High-Performance Computing.

[23] Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: optimizing memory access patterns for heterogeneous systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, New York, NY, USA, ACM (2011) 13:1–13:11

[24] AMD: ATI Stream SDK OpenCL Programming Guide (2010)

[25] Catanzaro, B., Keller, A., Garland, M.: A decomposition for in-place matrix transposition. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '14 (February 2014)

[26] Boyer, M., Meng, J., Kumaran, K.: Improving GPU performance prediction with data transfer modeling. In: 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW). (2013) 1097–1106

[27] Intel: OpenCL design and programming guide for the Intel Xeon Phi coprocessor. (2013)