International Journal of Networked and Distributed Computing, Vol. 2, No. 3 (August 2014), 124-134

Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm

Ayaz ul Hassan Khan , Mayez Al-Mouhamed , Allam Fatayer , Anas Almousa , Abdulrahman Baqais and Mohammed Assayony

Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia

E-mail: {ahkhan, mayez, g201003720, anasm, g201004220, g201102150}@kfupm.edu.sa

Abstract

The advances of Graphic Processing Units (GPU) technology and the introduction of CUDA programming model facilitates developing new solutions for sparse and dense linear algebra solvers. Matrix Transpose is an important linear algebra procedure that has deep impact in various computational science and engineering applications. Several factors hinder the expected performance of large matrix transpose on GPU devices. The degradation in performance involves the memory access pattern such as coalesced access in the global memory and bank conflict in the shared memory of streaming multiprocessors within the GPU. In this paper, two matrix transpose algorithms are proposed to alleviate the aforementioned issues of ensuring coalesced access and conflict free bank access. The proposed algorithms have comparable execution times with the NVIDIA SDK bank conflict - free matrix transpose implementation. The main advantage of proposed algorithms is that they eliminate bank conflicts while allocating shared memory exactly equal to the tile size (T x T) of the problem space. However, to the best of our knowledge an extra space of Tx(T+1) needs to be allocated in the published research. We have also applied the proposed transpose algorithm to recursive gaussian implementation of NVIDIA SDK and achieved about 6% improvement in performance.

Keywords: Bank conflict free, coalesced memory access, CUDA, GPU, matrix transpose, linear Algebra solvers, solving system of linear equations.

1. Introduction

GPU (Graphics Processor Unit) has been applied extensively in various scientific and engineering domains. These are the primary components in the latest supercomputers in terms of performance gain and energy efficiency [1]. Due to its high parallelism and efficiency, GPU was utilized to obtain high performance in many of linear algebra problems involving matrices. The matrix is a fundamental concept of linear algebra since they denote a linear relationship [2].

The performance of implementing naïve matrix transpose in GPU was disappointing. Though the GPU performance is very high, the utilization of memory system is essential to ensure a good application performance. To write efficient application kernels, the programmers have to consider the details of GPU memory architecture and access patterns of each memory type. Another point of consideration is the issue of shared memory bank conflicts where many requests are submitted to the same bank and hence the requests will be serialized affect-

Published by Atlantis Press

ing the parallelism performance expected from the GPU [3]. One of the common techniques proposed by NVIDIA is known as padding. It refers to the strategy of adding dump columns or rows to solve the issues of matrix divisibility or bank conflicts. This strategy was used in matrix transpose and its impact on performance was noticeable.

Matrix transpose problem involves replacing all the elements in the rows of one matrix to be placed vertically (column-wise). Some authors argue that the use of matrix transpose in itself is limited and of little use, and the power of transpose matrix is revealed in utilizing other algebraic problems like matrix multiplication in which matrix transpose is used as an intermediate step [2]. Though this view is valid to some extent, the literature has plenty of usage of matrix transpose in different applications and domains. The impact of matrix transpose to improve the performance of matrix multiplication is noticeable in [2,4]. However, matrix transpose has also been applied for utilizing the bottleneck in transferring data between host and device memory in clusters [5,6] and in other architectures such as distributed memory architecture [7]. Moreover, it has been applied in enhancing the performance of deep packet inspection (a famous problem in the domain of network security) by avoiding non-coalesced memory access [8]. Volkov and Demmel [9] studied the impact of the matrix transpose algorithms in matrix factorization, which is one important algorithm for solving system of linear equations.

When applying matrix transpose in a single processor, the elements in memory are not necessarily changed, but only the indices. However , in a distributed memory architecture, this is not the case [7]. Due to the different speed in the hierarchy of memory types where registers, cache and local memory are faster than off-processor memory and also due to the allocation size of the data segments to be given to each processor, several issues must be addressed such as load balancing, scattered decomposition and the communication scheme. In a GPU device, blocks of data are assigned to the processors. Furthermore, shared and global memory has different access speed and latency, so it simulates a distributed memory environment in a single machine. However, CUDA programming model simplifies the solutions to the aforementioned issues and the performance of matrix transpose is noticeably high comparing to the parallel matrix transpose algorithms for distributed memory concurrent computers [7]. The efficient usage of global memory and shared memory is a key factor in kernel optimizations, the efficiency is achieved through coalescing of global memory access and the bank conflict free access of shared memory [10].

In this paper, we have designed two matrix transpose algorithms for GPU. Both algorithms ensure that all global memory accesses are coalescent and all shared memory accesses are bank conflict free. The first algorithm reads a tile of the input matrix in the global memory to the shared memory transposes it locally in the shared memory and then writes it back to the output matrix in the global memory. The main step in this algorithm is the swapping of elements of the rows and columns in the tile. However, accessing successive elements of a column by successive threads in a half-warp when the tile size is multiple of 16 causes bank conflict. Our algorithm avoids the bank conflict by mapping the block threads to the tile elements diagonally. Even though the first algorithm reported good execution time, we found that it could be further improved by reducing shared memory accesses and distribute the work evenly between the threads within a warp. In the second algorithm, the transpose is performed as simple as reading a tile into the shared memory from the input matrix in the global memory and writing it back to the output matrix in the global memory. Since our goal is to ensure that all accesses are coalescent and bank conflict free, this goal is achieved by carrying out different mappings of threads within a block and the elements in a tile in read and write operations.

The remainder of the paper is organized as follows: Section 2 provides a brief background on GPU and CUDA. Section 3 presents some of the previous implementations of matrix transpose algorithms. In Section 4, an in-depth illustration and explanation of proposed algorithms has been laid out. The results are shown and discussed in Section 5 while we conclude our paper in Section 6.

2. GPU And CUDA

General Purpose Graphic Processor Unit (GPGPU) gains higher attention and recognition these days as a suitable platform for engineering and science applications. Initially, AMD & Intel microprocessors with GFLOPS were used using hypercube multiprocessors that offered a cost effective and feasible approach to supercomputing through parallelism at the processor level by direct connecting a large number of low-cost processors with local memory which communicate by message passing instead of shared variables [11], especially if bunch of these processors could be clustered together. However, power consumption was a major issue [12]. GPU on the other hand is capable of executing more GFLOPS than normal CPU. It provides a highly parallel computing environment suitable for numerous data parallel arithmetic computations such as dense linear algebraic operations [13]. However, the only restriction in earlier GPU version lies in its lack of support for IEEE FP Standards [12]. GPGPU costs are rapidly decreasing with increasing speed and wider memory. GPU has the advantage over CPU in utilizing transistors more efficiently. Though the challenges involved in programming GPU, they still provide a conductive platform for variety of applications from linear algebra to 3D gaming. It has been applied to various application domains including linear algebra [2,5,4,6,9], ray tracing and image processing [14,15,16].

CUDA ^{TM} is "a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)" [17]. It demands and requires a deep level of understanding of the organization of GPU memory and the execution context. Utilizing GPU underlying architecture and design by writing a program in CUDA is not straightforward or intuitive even for experienced programmers. CUDA requires a specific compiler to extract parallel code from sequential code and execute each of them to the corresponding device. That is, sequential code will be executed on the CPU while the parallelized segment of the code will be interpreted on the GPU device. These segments need to be implemented as isolated functions also called kernels. These kernel functions are compiled by the NVIDIA CUDA compiler and the kernel GPU object code generator [18].

3. Previous work

Ruetsch et al [19] discussed different aspects on how to improve the performance of out-of-place matrix transpose, where a matrix is transposed and stored into another matrix in the GPU's global memory. The most efficient implementation on modern GPUs they reported is the one that coalesces all global memory accesses performed by a half-warp of threads and guarantees that all threads in a half warp access shared memory locations associated with different banks.

The first condition can be achieved by accessing the input and output matrices in the global memory row-wise. The nave algorithm for matrix transpose is to read input matrix row-wise and writes the elements to the output matrix column-wise. However, the writes to the output matrix cannot be coalesced, since the half-warp threads access non-contiguous locations in the global memory.

To avoid non-coalesced accesses in writing to the global memory, the shared memory is used to temporarily store the elements read from the input matrix. To use the shared memory efficiently, a TxT matrix called tile, is allocated, where T is a multiple of 16. Each thread block reads a tile of TxT elements from the input matrix and stores them into the tile in the shared memory. The threads of a block read the elements of input matrix row-wise and write them to the shared memory matrix row-wise too. Then these threads read from the shared memory column-wise and write to the output matrix row-wise. Since shared memory locations are not necessarily be accessed continuously, the coalescent access is guaranteed. However, synchronization is necessary between the threads in write and read operations to the shared memory because the element that was written by one thread in the shared memory might be read by another thread.

Although this algorithm performs coalesced global memory accesses, it suffers from bank conflict problem in accessing the shared memory. When the tile size is multiple of 16, the locations of the

same columns are assigned to the same shared memory banks, and therefore, all read operations from the shared memory executed by a half-warp thread involve bank conflict.

To avoid the bank conflicts, the tile of size Tx(T+1) is allocated. The extra column is added only to assign a location read by a half-warp thread to different banks in the shared memory and therefore eliminates the bank conflicts in reading from the tile as shown in Figure 1.

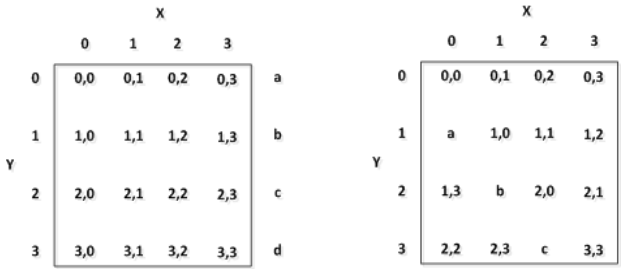

Figure 1: Shared memory tile column padding to carry out the permutation on the data

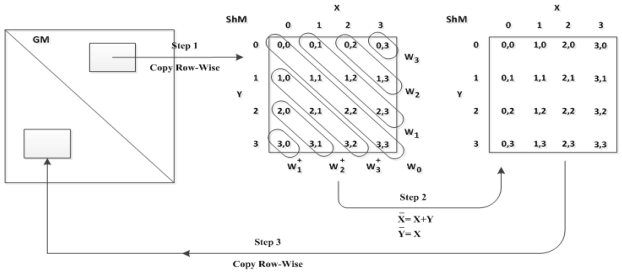

Figure 2: Threads and Elements mapping at each steps of the First Algorithm (Selective Copying)

We note that the bank conflict could be avoided without wasting shared memory by mapping the half-warp threads to different columns in the tile in expense of simple calculations. Since the matrix transpose is a memory-bound application, the computation overheads of these calculations could be hidden and does not significantly affect the execution time of the kernel.

4. Proposed Algorithms

In this section we explain our algorithms for matrix transpose.

4.1. The First Algorithm: Selective Copying

Algorithm 1 shows the pseudo code of our first implementation for matrix transpose. There are three

Algorithm 1 Selective Copying

ourTranspose1(odata, idata, width, height)

Parameters:

odata = global memory array to store the results
idata = global memory array to store the input
width = width of the matrices
height = height of the matrices

Constants and Keywords:

TILE_DIM = Dimension of the tile to load into shared memory. The allocated tile will be TILE_DIM x TILE_DIM in size
blockIdx.x = x index of current block
blockIdx.y = y index of current block
threadIdx.x = x index of current thread within a block
threadIdx.y = y index of current thread within a block

Algorithm:

```
1: xIndex = blockIdx.x * TILE_DIM + threadIdx.x
2: yIndex = blockIdx.y * TILE_DIM + threadIdx.y
3: index_in = xIndex + yIndex * width
4: xIndex = blockIdx.y * TILE_DIM + threadIdx.x
5: yIndex = blockIdx.x * TILE_DIM + threadIdx.x
6: index_out = xIndex + yIndex * height
7: tile[threadIdx.y][threadIdx.x] = idata[index_in]
8: synchronize all threads
9: x_bar = (threadIdx.x + threadIdx.y) % TILE_DIM
10: y_bar = threadIdx.x
11: if y_bar > x_bar then
12:     temp1 = tile[y_bar][x_bar]
13:     temp2 = tile[x_bar][y_bar]
14:     tile[x_bar][y_bar] = temp1
15:     tile[y_bar][x_bar] = temp2
16: end if
17: synchronize all threads
18: odata[index_out] = tile[threadIdx.y][threadIdx.x]
```

main steps: reading from the input matrix in the global memory to the tile in the shared memory (Lines 1-3, 7), transposing the tile (Lines 9-16) and writing the tile to the proper location in the output matrix in the global memory (Lines 4-6, 18).

The first step is the same step performed by the NVidia algorithm: the threads of the block read the elements from the input matrix in the global memory row-wise and write them to the tile in the shared memory row-wise (Line 7). As discussed earlier, these accesses are coalescent and bank conflict free.

In the second step, the elements of the tile are transposed inside the tile. To ensure that all accesses to the tile in order to transpose the elements are bank conflict free, the block threads are mapped to the tile elements diagonally (Lines 9, 10). The mapping between the thread (x, y) and the element (x', y') that should be accessed by this thread within the tile of dimension T is as the following:

$$x' = (x+y) mod T \quad (1)$$

$$y' = x \quad (2)$$

The transpose of the tile elements is based on swapping the elements in the upper half of the tile with the elements in the lower half. The elements lies in the diagonal are not changed (Line 11).

The swapping between upper half and lower half elements are involved by the threads that are mapped to the elements of the upper half (active threads). Other threads don't perform any work in this step (idle threads). This configuration ensures that the active threads of each half-warp are accessing elements in different banks and therefore all accesses are bank conflict free.

When all elements are transposed, the last step is to write the tile back to the output matrix in the global memory (Line 18). This step is achieved by mapping threads (Lines 4-6) to the tile elements row-wise and allowing half-warp threads to read the elements row-wise from the tile and write them row-wise to the output matrix. Since both matrices are accessed row- wise, all accesses are coalescent and bank conflict free. Figure 2 shows the threads and elements mapping at each step of the first algorithm.

4.2. The Second Algorithm: Parallel Write

The main advantage of the first algorithm we discussed in the previous sub-section is that it eliminates bank conflict while allocating shared memory space exactly equal to the tile size of TxT, while NVIDIA algorithm allocates Tx(T+1) space for a tile. However, the implementation has two drawbacks:

1. It performs many accesses to the shared memory in order to transpose the tile in the shared memory. Though all accesses are bank conflict free, accessing shared memory is counted as a computation instruction and takes execution time similar to that of computation instruction [20]. The evaluation shows that when the thread block size is relatively small, this computation might not be hidden by the data transfer and they affect the overall execution time of the kernel.

2. All accesses mentioned above are performed by less than a half of the block threads, while most of the threads stay idle. This configuration has two side effects. The first is the active threads take more time to complete the work. The second is the effect of branching, since the branch instruction divides the warp threads into two groups that should be scheduled independently [20].

To overcome these drawbacks, we designed the second algorithm as shown in Listing 2. This algorithm achieves the advantage of the first algorithm while all drawbacks are avoided. This has been achieved by carrying out different mappings of threads within a block and elements in a tile in read and write operations.

The algorithm performs two main steps: reading from the input matrix in the global memory to the tile in the shared memory (Lines 1-3, 7-9) and writing the tile elements to the proper location in the output matrix in the global memory (Lines 4-6, 11-13).

In the first step, the threads of the block read the elements from the input matrix in the global memory row-wise and write them to the tile in the shared

memory diagonally (Lines 7-9). The mapping between the thread (x, y) and the element (x', y') that should be written by this thread within the tile of dimension T is the same mapping used in the first algorithm (see Equations 1 and 2).

Algorithm 2 Parallel Write

ourTranspose2(odata, idata, width, height)

Parameters:

odata = global memory array to store the results
idata = global memory array to store the input
width = width of the matrices
height = height of the matrices

Constants and Keywords:

TILE_DIM = Dimension of the tile to load into shared memory. The allocated tile will be TILE_DIM x TILE_DIM in size
blockIdx.x = x index of current block
blockIdx.y = y index of current block
threadIdx.x = x index of current thread within a block
threadIdx.y = y index of current thread within a block

Algorithm:

```
1:  xIndex = blockIdx.x * TILE_DIM + threadIdx.x
2:  yIndex = blockIdx.y * TILE_DIM + threadIdx.y
3:  index_in = xIndex + yIndex * width
4:  xIndex = blockIdx.y * TILE_DIM + threadIdx.x
5:  yIndex = blockIdx.x * TILE_DIM + threadIdx.x
6:  index_out = xIndex + yIndex * height
7:  x_bar = (threadIdx.x + threadIdx.y) % TILE_DIM
8:  y_bar = threadIdx.x
9:  tile[y_bar][x_bar] = idata[index_in]
10: synchronize all threads
11: x_coor = (threadIdx.x + threadIdx.y) % TILE_DIM
12: y_coor = threadIdx.y
13: odata[index_out] = tile[y_coor][x_coor]
```

Since all read operations from the global memory are row- wise, the memory accesses are coalesced. And by using diagonal mapping in writing to the shared memory, we ensure that all elements accessed by half-warp threads are in different bank. Therefore, all shared memory accesses are bank conflict free.

Before writing to the transposed tile to the output matrix, we should note that by writing the tile elements diagonally, the elements stored in each row of the tile are exactly the elements of the column that are read from the input matrix, starting from the diagonal element. Therefore, to complete the transpose we need to map the threads to the tile elements properly (Lines 11, 12), read the elements from the tile row wise and write them to the output matrix row-wise. The mapping between the thread (x, y) and the element (x', y') that should be read by this thread is as the following:

$$x' = (x+y) mod T \tag{3}$$

$$y' = y \tag{4}$$

Since reads and writes operations are row-wise, they are coalesced and bank conflict free. Figure 3 shows the threads and elements mapping at each step of the second algorithm.

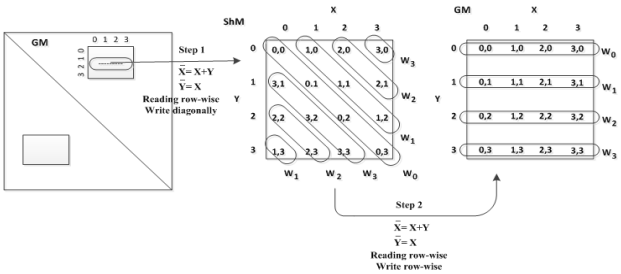

Figure 3: Threads and Elements mapping at each steps of the Second Algorithm (Parallel Write)

The proposed approach to access elements in shared memory diagonally in matrix transpose algorithms 1 and 2 can also be applied to any other application having bank conflict issue. We have defined a C macro as follows to be used in other applications to avoid shared memory bank conflicts.

```
#define tile(y,x) tile[x % TILE_DIM][(x+y) % TILE_DIM]
```

The programmer just needs to use the above macro to access shared memory variable instead of direct accessing as array.

4.3. Generalized Approach

Furthermore, using modulo (%) operator to each access in shared memory may reduce performance. So, we need to define some alternative to modulo operations. By architecture, the shared memory system (ShM) which is available within each SM consists of $S = 2^r = 2^4$ memory banks denoted by $ShM = \{M_K\}$ for k in [0,15]. These memory banks M_K are accessed in parallel by the currently executing half-warp (hw) which consists of a group of S threads. The standard row major storage of an NxN two dimensional array A(j,i) leads to map each array element A(j,i) into memory bank M_K such that k = (jxN+i) mod S. A half-warp hw = $\{th_K,.., th_K+15\}$ lying in a kernel block row. Thus hw accesses distinct memory banks because its array addresses differ in the column numbers(i). Therefore, the least significant r bits of K are all distinct which causes all the data accessed by hw to fall into distinct memory banks. However, when hw accesses a group of data elements lying in a column, the addresses generated by all the threads of hw have fixed i and differ only in j. As that k = (jxN+i) mod S there is no guaranty that these addresses fall into distinct memory banks. The question is to find a mapping function k=f(j,i) from array address (j,i) onto the memory bank k such array element a(i,j) fall into distinct memory banks for each hw that accesses a set of data patterns like a row of S elements or a column of S elements that distant by a multiple of S. If such a function f(j,i) exists then any hw will perform a conflict free access to all the S banks of ShM because all of threads generate distinct addresses. In other words the least significant r bit in the mapping k = (jxN+i) mod S must be linked to row index i as well as to column index j to cause parallel access to rows and columns. This can be achieved by changing the array mapping from the standard (j,i) to (j, i xor (j and 0F)), where xor is the exculsive-or operator and (j and 0F) gives the least significant r bits of j which are xor-ed with the lsb 4 bit of i. When hw is accessing a row of data elements the row numbering (j and 0F) will be constant for all the generated hw addresses and conflict free access is achieved due to distinct 4 lsb in i. Similarly when hw is accessing a column of data elements the 4 lsb of i will be constant for all the generated hw addresses and conflict free access will be due to distinct (j and 0F), i.e. 4 lsb in j. In both cases hw generates array addresses that are all distinct. Therefore, the mapping scheme (j,i) $\rightarrow$ (j, i xor (j and 0F)) provides a conflict-free access to all the S memory banks of ShM when accessing rows or columns.

5. Experimental Results

To evaluate the performance of our algorithms, we have implemented them, in addition to the NVIDIA algorithm, as CUDA kernels. We have used the execution time elapsed by the kernels as the performance parameter. Table 1 shows the main features of the GPU on which the implementations were run.

Table 1: GPU Specification

Property	Value
GPU Model	Quadro FX 7000
CUDA Capability Major/Minor Version	2.0
Total amount of global memory	4 GBytes
(16) Multiprocessors x (32) CUDA Cores/MP	512 CUDA Cores
Total amount of shared memory per block	48 KBytes
Maximum number of threads per block	1024

The three implementations that we used for evaluation comparisons are:

1. **No bank conflict Transpose**[17]
2. **Selective Copying**
3. **Parallel Write**

Table 2: Experimental Setup

Tile Size	Thread Block Size
16 x 16	16 x 1, 16 x 2, 16 x 4, 16 x 16
32 x 32	32 x 1, 32 x 2, 32 x 4, 32 x 8, 32 x 16, 32 x 32
48 x 48	48 x 1, 48 x 2, 48 x 4, 48 x 6, 48 x 8, 48 x 12, 48 x 16

We run these implementations several times with the following configurations:

1. Matrices of 3072 x 3072 floats that is the maximum multiple of 32 and 48 that could be allocated.

2. We have used three tile sizes: 16x16, 32x32 and 48x48. The last is the maximum tile size that can be allocated.

3. The implementations have been run with all possible thread block sizes shown in Table 2.

4. Number of iterations = 100.

Figures 4,5,6 show the execution time of running the three different implementations with different configurations.

Table 3: Shared Memory Usage per Kernel Block

Tile Size	Shared Memory Usage		% Reduced
	NVIDIA	Proposed	
16 x 16	1088	1024	5.88
32 x 32	4224	4096	3.03
48 x 48	9408	9216	2.04

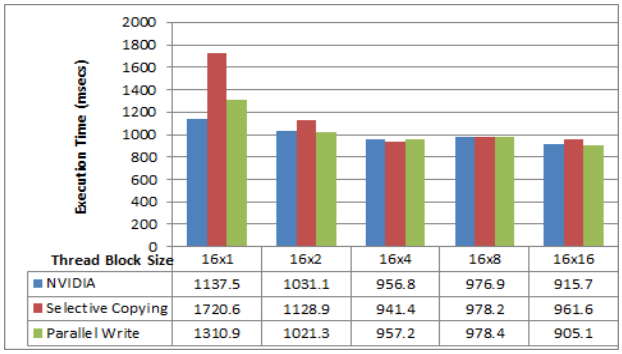

Figure 4: Exec. time for the three implementations with Tile Size = 16 x 16

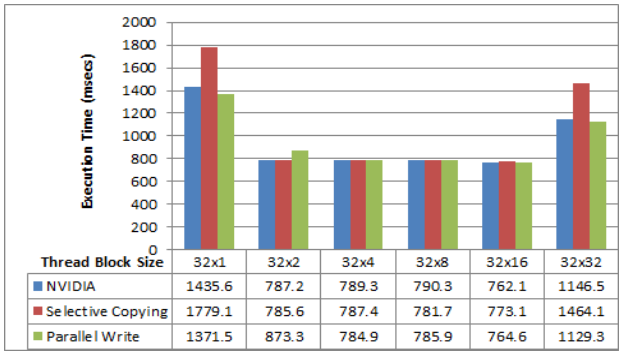

Figure 5: Exec. time for the three implementations with Tile Size = 32 x 32

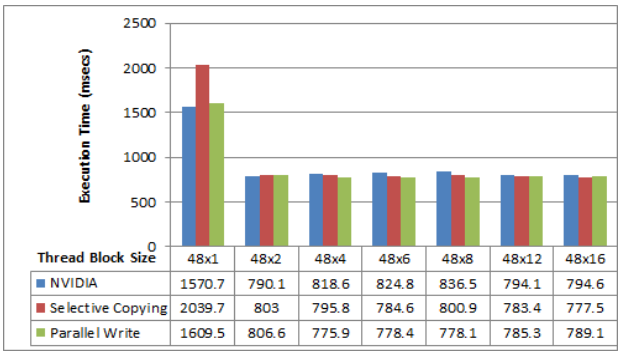

Figure 6: Exec. time for the three implementations with Tile Size = 48 x 48

The results show that the three implementations are almost have the same time to transpose a matrix of size 3072x3072, except for some configurations. But our implementations (Selective Copying and Parallel Write) uses less shared memory per block than the NVIDIA implementation as shown in Table 3 and gives more reduction in terms of percentage in case of small tile size.

When the thread block size is one-dimensional block (16x1, 32x1, 48x1), the three implementations give the maximum value for the execution time. This is because in one- dimensional block, each thread transposes more memory elements than the threads in other configurations. As an example, in the tile of 48x48 and the block of 48x1, each thread transposes 48 elements in the input matrix, while in the block of 48x16 each thread transposes 3 elements only.

The results also show that when the thread block size is one- dimensional block (16x1, 32x1, 48x1), Selective Copying gives the largest value among the three implementations. This is because in this implementation the active threads in the block execute many memory accesses in order to transpose the tile in the shared memory. When the thread block size is small, these accesses cannot be hidden by global memory accesses, and therefore their overheads appear in the execution time.

We note also in Figure 5 that for the block size of 32x32, the execution time is high and Selective Copying is the highest. This is due to the fact that when the block size of 32x32 has 1024 threads (32x32=1024) and this number with the other configurations in the implementations is beyond the oc-

cupancy of the GPU as reported by NVIDIA CUDA GPU Occupancy Calculator [21]. When the kernel has low occupancy, the performance is always reduced [22].

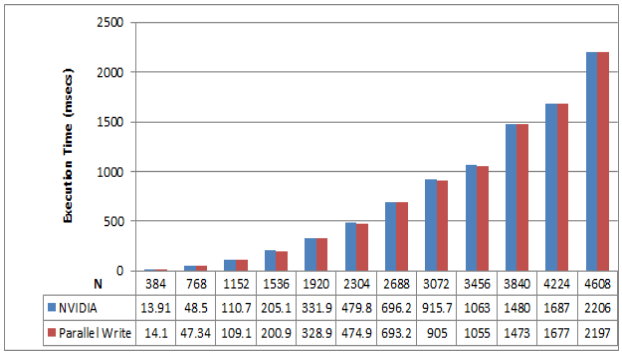

Figure 7: Exec. time with Tile Size = 16 x 16 and Thread Block Size = 16 x 16

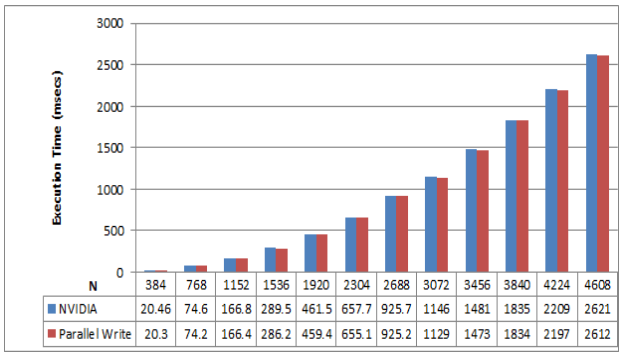

Figure 8: Exec. time with Tile Size = 32 x 32 and Thread Block Size = 32 x 32

Figure 7 and 8 show the trend of execution time for NVIDIA and Parallel Write algorithms with different space sizes. The results show approximately same execution time except even with different space sizes. However, there are some improvements in performance due to reduction in execution time for small tile size (16 x 16). Padding column in tile as in NVIDIA implementation to avoid bank conflict applies only on the multiple of 32 tile sizes. Our proposed approach to access shared memory tile diagonally avoids bank conflict even with non-multiple of 32 tile sizes. We have analyzed both the kernels using NVIDIA visual profiler event for shared bank conflicts. Table 4 shows the profiler results for bank conflicts with different tile sizes.

Table 4: Bank Conflicts with different tile sizes

Tile Size	NVIDIA	Proposed
16 x 16	1162330	0
32 x 32	0	0

We have also applied Parallel Write algorithm in the implementation of recursive gaussian filter provided in NVIDIA SDK. The code sample implements a Gaussian blur using Deriche's recursive method [23]. It processes columns of the image parallel. While, to make the coalesced read for the row pass, it transpose the image and then transpose it back again afterwards. So, it requires to perform transpose twice in each step. The results in figures 9 and 10 shows upto 6% improvement in performance with Parallel Write algorithm over the NVIDIA SDK Transpose.

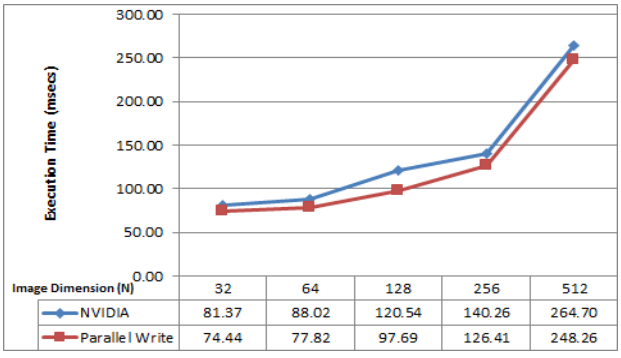

Figure 9: Exec. Time of recursive gaussian using NVIDIA Transpose and Proposed Parallel Write algorithms

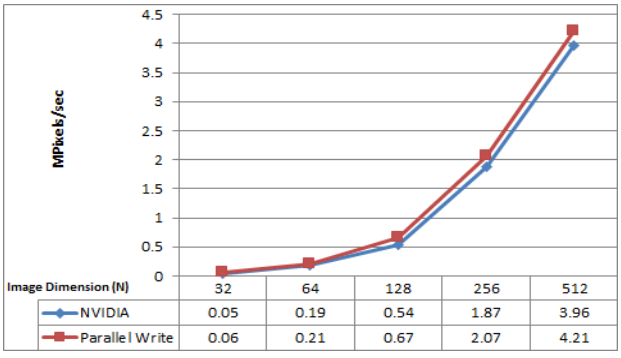

Figure 10: Performance of recursive gaussian using NVIDIA Transpose and Proposed Parallel Write algorithms

6. Conclusions

Graphic Processor Unit (GPU) gains higher attention and wider acceptance and recognition these days as a suitable platform for engineering and science application. To achieve better performance, NVIDIA introduced CUDA, a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the GPU.

In this work we have studied different implementations of matrix transpose that illustrate how to achieve efficient use of GPU memories and data management. We proposed two implementations for matrix transpose that perform efficient memory accesses to global and shared memory by ensuring all accesses to the global memory are coalesced and all accesses to the shared memory are bank conflict free. The main advantage of our algorithms is that they eliminate bank conflicts while allocating exactly the tile size memory space. However, in the literature [19] they allocate Tx(T+1) space for TxT tile. Also, padding column in shared memory tile to avoid bank conflicts applies only on the multiple of 32 tile sizes only. While our approach can also be applied on non-multiple of 32 tile sizes.

We have also applied the proposed transpose algorithm to recursive gaussian implementation of NVIDIA SDK and achieved about 6% improvement in performance.

References

References are to be listed in the order cited in the text. Use the style shown in the following examples. For journal names, use the standard abbreviations. Typeset references in 9 pt Times Roman.

1. D. Qian and D. Zhu, "Challenges and possible approaches: towards the petaflops computers." Frontiers of Computer Science in China, vol. 3, no. 3, pp. 273289, 2009.
2. A. Moravanszky and N. Ag, "Dense matrix algebra on the gpu," in In Direct3D ShaderX2, Engel W. F., (Ed.). Wordware Publishing. NovodeX AG, 2003, p. 2.
3. Y. Kim and A. Shrivastava, "Cumapz: a tool to analyze memory access patterns in cuda," in Proceedings of the 48th Design Automation Conference, ser. DAC '11. ACM, 2011, pp. 128133.
4. R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '11. ACM, 2011, pp. 6:16:10.
5. Y. Chen, X. Cui, and H. Mei, "Large-scale fft on gpu clusters," in Proceedings of the 24th ACM International Conference on Supercomputing, ser. ICS '10. ACM, 2010, pp. 315324.
6. Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in IEEE International Symposium on Parallel and Distributed Processing, 2008, Apr. 2008, pp. 110.
7. J. Choi, J. Dongarra, and D. W. Walker, "Parallel matrix transpose algorithms on distributed memory concurrent computers." Parallel Computing, vol. 21, no. 9, pp. 1387 1405, 1995.
8. L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), June 2011, pp. 366370.
9. V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, ser.SC'08. IEEEPress,2008, pp. 31:131:11.
10. K. Nakano, "Optimal parallel algorithms for computing the sum, the prex-sums, and the summed area table on the memory machine models," IEICE Transactions on Information and Systems, vol. 2013, no. 12, pp. 26262634, 2013.
11. J. Aguilar, "Heuristic algorithm based on a genetic algorithm for mapping parallel programs on hypercube multiprocessors." Comput. Syst. Sci. Eng., vol. 18, no. 4, pp. 217221, 2003.
12. S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced mri reconstructions on gpus," in Proceedings of the 5th conference on Computing frontiers, ser. CF '08. ACM, 2008, pp. 261272.
13. J. WANG, X. MA, Y. ZHU, and J. SUN, "Auto-tuning of thread assignment for matrix-vector multiplication on gpus," IEICE TRANSACTIONS on Information and Systems, vol. E96-D, no. 11, pp. 23192326, November 2013.
14. D. K. Bogolepov, D. P. Sopin, and V. E. Turlapov, "Simplified photon mapping for real-time caustics rendering," Program. Comput. Softw., vol. 37, no. 5, pp. 229235, Sep. 2011.
15. V. A. Frolov, A. A. Kharlamov, and A. V. Ignatenko, "Biased solution of integral illumination equation via irradiance caching and path tracing on gpus." Programming and Computer Software, vol. 37, no. 5, pp.

252259, 2011.
16. I. N. Skopin and D. Y. Tribis, "A method for solving mass point-in-covering problems for arbitrary coverings using gpu," Programming and Computer Software, vol. 39, no. 3, pp. 158162, 2013.
17. NVIDIA, "Nvidia developer zone," http://developer.nvidia.com, NVIDIA Corporation.
18. Z. Xu, Y. He, W. Lin, and L. Zha, "Four styles of parallel and net programming," Frontiers of Computer Science in China, vol. 3, no. 3, pp. 290301, 2009.
19. G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," NVIDIA Corporation, Tech. Rep., June 2010.
20. S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness, " SIGARCH Comput. Archit. News, vol.37, no.3, pp. 152163, Jun. 2009.
21. NVIDIA, "Cuda occupancy calculator," http://news.developer.nvidia.com/2007/03/cuda occupancy .html, NVIDIA Corporation.
22. NVIDIA, "Cuda c best practices guide," NVIDIA Corporation, Tech. Rep., 2010.
23. R. Deriche, "Recursively Implementing the Gaussian and Its Derivatives," in Proc. Second International Conference On Image Processing, Singapore, Sep. 7-11 1992, pp. 263267.