Optimizing Matrix Transpose in CUDA

Greg Ruetsch

gruetsch@nvidia.com

Paulius Micikevicius

pauliusm@nvidia.com

January 2009

Chapter 1. Introduction

Optimizing CUDA Memory Management in Matrix Transpose

This document discusses aspects of CUDA application performance related to efficient use of GPU memories and data management as applied to a matrix transpose. In particular, this document discusses the following issues of memory usage:

- coalescing data transfers to and from global memory
- shared memory bank conflicts
- partition camping

There are other aspects of efficient memory usage not discussed here, such as data transfers between host and device, as well as constant and texture memories.

Both coalescing and partition camping deal with data transfers between global device and on-chip memories, while shared memory bank conflicts deal with on-chip shared memory.

The reader should be familiar with basic CUDA programming concepts such as kernels, threads, and blocks, as well as a basic understanding of the different memory spaces accessible by CUDA threads. A good introduction to CUDA programming is given in the CUDA Programming Guide as well as other resources on CUDA Zone (http://www.nvidia.com/cuda).

The matrix transpose problem statement is given next, followed by a brief discussion of performance metrics, after which the remainder of the document presents a sequence of CUDA matrix transpose kernels which progressively address various performance bottlenecks.

Matrix Transpose Characteristics

In this document we optimize a transpose of a matrix of floats that operates out-of-place, i.e. the input and output matrices address separate memory locations. For simplicity and brevity in presentation, we consider only square matrices whose dimensions are integral multiples of 32 on a side, the tile size, through the document. However, modifications of code required to accommodate matrices of arbitrary size are straightforward.

Code Highlights and Performance Measurements

The host code for all the transpose cases is given in Appendix A. The host code performs typical tasks: data allocation and transfer between host and device, the launching and timing of several kernels, result validation, and the deallocation of host and device memory.

In addition to different matrix transposes, we run kernels that execute matrix copies. The performance of the matrix copies serve as benchmarks that we would like the matrix transpose to achieve.

For both the matrix copy and transpose, the relevant performance metric is the effective bandwidth, calculated in GB/s as twice the size of the matrix – once for reading the matrix and once for writing – divided by the time of execution. Since timing is performed in loops executed **NUM_REPS** times, which is defined at the top of the code, the effective bandwidth is also normalized by **NUM_REPS**.

Looping **NUM_REPS** times over code for measurement is done in two different fashions: looping over kernel launches, and looping within the kernel over the load and stores. The host code for these measurements is given below:

```
// take measurements for loop over kernel launches
    cudaEventRecord(start, 0);
    for (int i=0; i < NUM_REPS; i++) {
      kernel<<<grid, threads>>>(d_odata, d_idata,size_x,size_y,1);
    }
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float outerTime;
    cudaEventElapsedTime(&outerTime, start, stop);

    ...

 // take measurements for loop inside kernel
    cudaEventRecord(start, 0);
    kernel<<<grid,threads>>>
        (d_odata, d_idata, size_x, size_y, NUM_REPS);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float innerTime;
    cudaEventElapsedTime(&innerTime, start, stop);
```

The first timing is done by a **for** loop in the host code, the second by passing **NUM_REPS** as a parameter to the kernel. A simple copy kernel is shown below:

```
__global__ void copy(float *odata, float* idata, int width,
                                     int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

  int index  = xIndex + width*yIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    }
  }
}
```

The difference between these two timings is the overhead of the kernel launch, which should be consistent between different kernels, as well as the time spent in calculating the matrix indices at the beginning of each kernel. In addition, looping over kernel launches also acts as a synchronization mechanism. When the kernel is launched multiple times from a loop in host code, all blocks from one kernel launch must complete execution before any block of a following launch can begin. As a result the set of active blocks and hence memory access patterns resets every loop iteration. When the loop is performed within the kernels, the set of active thread blocks has more opportunity to diverge as execution progresses through the timing loop.

Both methods of timing code provide useful measurements, the first indicating what one would typically use as an overall performance metric, and the second as a means of comparing the data movement times between kernels.

In the following section we present different kernels called from the host code, each addressing different performance issues. All kernels in this study launch thread blocks of dimension 32x8, where each block transposes (or copies) a tile of dimension 32x32. As such, the parameters **`TILE_DIM`** and **`BLOCK_ROWS`** are set to 32 and 8, respectively. Using a thread block with fewer threads than elements in a tile is advantageous for the matrix transpose in that each thread transposes several matrix elements, four in our case, and much of the cost of calculating the indices is amortized over these elements.

2. Copy and Transpose Kernels

Simple copy

The first two cases we consider are a naïve transpose and simple copy, each using blocks of 32x8 threads on a 32x32 matrix tiles. The copy kernel was given in the previous section, and shows the basic layout for all of the kernels. The first two arguments **odata** and **idata** are pointers to the input and output matrices, **width** and **height** are the matrix x and y dimensions, and **nreps** determines how many times the loop over data movement between matrices is performed. In this kernel, the global 2D matrix indices **xIndex** and **yIndex** are calculated, which are in turn used to calculate **index**, the 1D index used by each thread to access matrix elements. The loop over **i** adds additional offsets to **index** so that each thread copies multiple elements of the array, and the loop over **r** is used for timing the data transfer from input to output array multiple times.

Naïve transpose

The naïve transpose:

```
__global__ void transposeNaive(float *odata, float* idata,
                         int width, int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i] = idata[index_in+i*width];
    }
  }
}
```

is nearly identical to the copy kernel above, with the exception that **index**, the array index used to access elements in both input and output arrays for the copy kernel, is replaced by the two indices **index_in** (equivalent to **index** in the copy kernel), and **index_out**. Each thread executing the kernel transposes four elements from one column of the input matrix to their transposed locations in one row of the output matrix.

The performance of these two kernels on a 2048x2048 matrix using a GTX280 is given in the following table:

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

The minor differences in code between the copy and naïve transpose kernels have a profound effect on performance - nearly two orders of magnitude. This brings us to our first optimization technique: global memory coalescing.

Coalesced Transpose

Because device memory has a much higher latency and lower bandwidth than on-chip memory, special attention must be paid to how global memory accesses are performed, in our case loading data from **`idata`** and storing data in **`odata`**. All global memory accesses by a half-warp of threads can be coalesced into one or two transactions if certain criteria are met. These criteria depend on the compute capability of the device, which can be determined, for example, by running the **`deviceQuery`** SDK example. For compute capabilities of 1.0 and 1.1, the following conditions are required for coalescing:

- ❑ threads must access either 32- 64-, or 128-bit words, resulting in either one transaction (for 32- and 64-bit words) or two transactions (for 128-bit words)
- ❑ All 16 words must lie in the same aligned segment of 64 or 128 bytes for 32- and 64-bit words, and for 128-bit words the data must lie in two contiguous 128 byte aligned segments
- ❑ The threads need to access words in sequence. If the **`k`**-th thread is to access a word, it must access the **`k`**-th word, although not all threads need to participate.

For devices with compute capabilities of 1.2, requirements for coalescing are relaxed. Coalescing into a single transaction can occur when data lies in 32-, 64-, and 128-byte aligned segments, regardless of the access pattern by threads within the segment. In general, if a half-warp of threads access N segments of memory, N memory transactions are issued.

In a nutshell, if a memory access coalesces on a device of compute capability 1.0 or 1.1, then it will coalesce on a device of compute capability 1.2 and higher. If it doesn't coalesce on a device of compute capability 1.0 or 1.1, then it may either completely coalesce or perhaps result in a reduced number of memory transactions, on a device of compute capability 1.2 or higher.

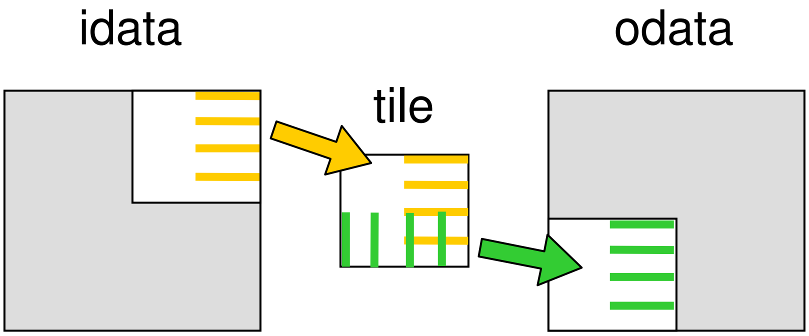
For both the simple copy and naïve transpose, all loads from **`idata`** coalesce on devices with any of the compute capabilities discussed above. For each iteration within the **`i`**-loop, each half warp reads 16 contiguous 32-bit words, or one half of a row of a tile. Allocating device memory through **`cudaMalloc()`** and choosing **`TILE_DIM`** to be a multiple of 16 ensures alignment with a segment of memory, therefore all loads are coalesced.

Coalescing behavior differs between the simple copy and naïve transpose kernels when writing to **`odata`**. For the simple copy, during each iteration of the **`i`**-loop, a half warp writes one half of a row of a tile in a coalesced manner. In the case of the naïve transpose, for each iteration of the **`i`**-loop a half warp writes one half of a column of floats to different segments of memory, resulting in 16 separate memory transactions, regardless of the compute capability.

The way to avoid uncoalesced global memory access is to read the data into shared memory, and have each half warp access noncontiguous locations in shared memory in order to write contiguous data to **`odata`**. There is no performance penalty for noncontiguous access patters in shared memory as there is in global memory, however the above procedure requires that each element in a tile be accessed by different threads, so a **`__synchthreads()`** call is required to ensure that all reads from **`idata`** to shared memory have completed before writes from shared memory to **`odata`** commence. A coalesced transpose is listed below:

```
__global__ void transposeCoalesced(float *odata,
            float *idata, int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```

A depiction of the data flow of a half warp in the coalesced transpose kernel is given below. The half warp writes four half rows of the **`idata`** matrix tile to the shared memory 32x32 array “**`tile`**” indicated by the yellow line segments. After a **`__syncthreads()`** call to ensure all writes to **`tile`** are completed, the half warp writes four half columns of **`tile`** to four half rows of an **`odata`** matrix tile, indicated by the green line segments.



With the improved access pattern to memory in **`odata`**, the writes are coalesced and we see an improved performance:

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

While there is a dramatic increase in effective bandwidth of the coalesced transpose over the naïve transpose, there still remains a large performance gap between the coalesced transpose and the copy. The additional indexing required by the transpose doesn’t appear to be the cause for the performance gap, since the results in the “Loop in kernel” column, where the index calculation is amortized over 100 iterations of the data movement, also shows a large performance difference. One possible cause of this performance gap is the synchronization barrier required in the coalesced transpose. This can be easily assessed using the following copy kernel which utilizes shared memory and contains a **`__syncthreads()`** call:

```
__global__ void copySharedMem(float *odata, float *idata,
                              int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

  int index  = xIndex + width*yIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] =
        tile[threadIdx.y+i][threadIdx.x];
    }
  }
}
```

The `__syncthreads()` call is not needed for successful execution of this kernel, as threads do not share data, and is included only to assess the cost of the synchronization barrier in the coalesced transpose. The results are shown in the following modified table:

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

The shared memory copy results seem to suggest that the use of shared memory with a synchronization barrier has little effect on the performance, certainly as far as the "Loop in kernel" column indicates when comparing the simple copy and shared memory copy. When comparing the coalesced transpose and shared memory copy kernels, however, there is one performance bottleneck regarding how shared memory is accessed that needs to be addressed: shared memory bank conflicts.

Shared memory bank conflicts

Shared memory is divided into 16 equally-sized memory modules, called banks, which are organized such that successive 32-bit words are assigned to successive banks. These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory the threads in a half warp should access shared memory associated with different banks. The exception to this rule is when all threads in a half warp read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads of the half warp in one transaction.

One can use the **`warp_serialize`** flag when profiling CUDA applications to determine whether shared memory bank conflicts occur in any kernel. In general, this flag also reflects use of atomics and constant memory, however neither of these are present in our example.

The coalesced transpose uses a 32x32 shared memory array of floats. For this sized array, all data in columns **`k`** and **`k+16`** are mapped to the same bank. As a result, when writing partial columns from **`tile`** in shared memory to rows in **`odata`** the half warp experiences a 16-way bank conflict and serializes the request. A simple to avoid this conflict is to pad the shared memory array by one column:

```
  __shared__ float tile[TILE_DIM][TILE_DIM+1];
```

The padding does not affect shared memory bank access pattern when writing a half warp to shared memory, which remains conflict free, but by adding a single column now the access of a half warp of data in a column is also conflict free. The performance of the kernel, now coalesced and memory bank conflict free, is added to our table below:

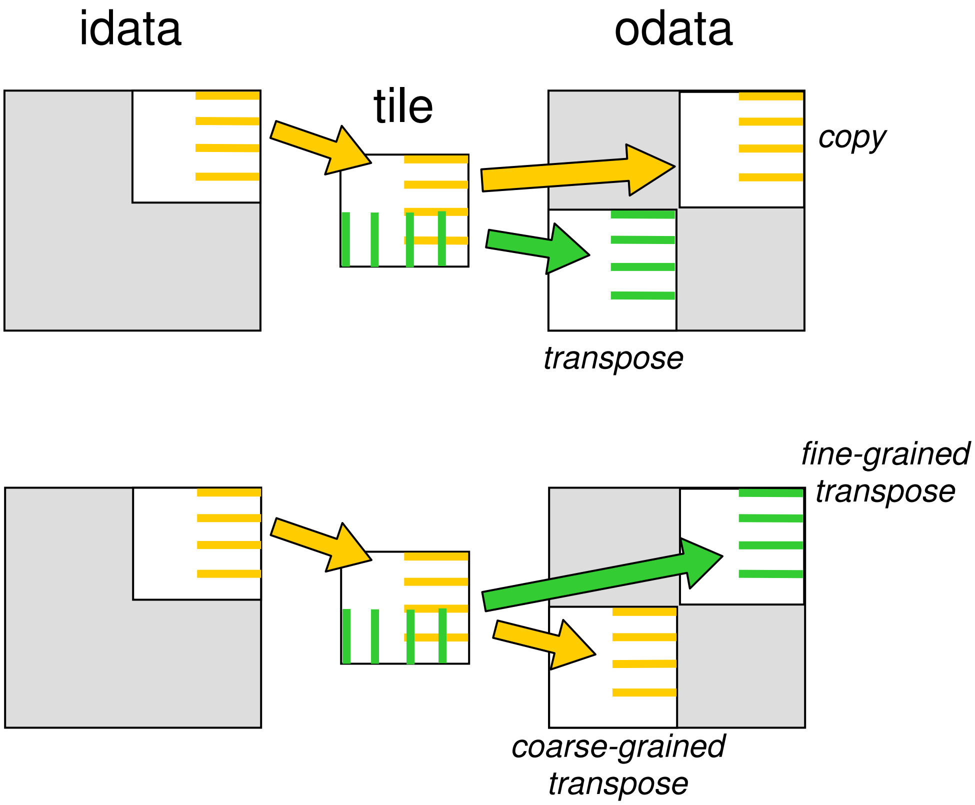
	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2

While padding the shared memory array did eliminate shared memory bank conflicts, as was confirmed by checking the **`warp_serialize`** flag with the CUDA profiler, it has little effect (when implemented at this stage) on performance. As a result, there is still a large performance gap between the coalesced and shared memory bank conflict free transpose and the shared memory copy. In the next section we break the transpose into components to determine the cause for the performance degradation.

Decomposing Transpose

There is over a factor of four performance difference between the best optimized transpose and the shared memory copy in the table above. This is the case not only for measurements which loop over the kernel launches, but also for measurements obtained from looping within the kernel where the costs associated with the additional index calculations are amortized over the 100 iterations.

To investigate further, we revisit the data flow for the transpose and compare it to that of the copy, both of which are indicated in the top portion of the diagram below. There are essentially two differences between the copy code and the transpose: transposing the data within a tile, and writing data to transposed tile. We can isolate the performance between each of these two components by implementing two kernels that individually perform just one of these components. As indicated in the bottom half of the diagram below, the fine-grained transpose kernel transposes the data within a tile, but writes the tile to the location that a copy would write the tile. The coarse-grained transpose kernel writes the tile to the transposed location in the **`odata`** matrix, but does not transpose the data within the tile.



The source code for these two kernels is given below:

```
__global__ void transposeFineGrained(float *odata,
           float *idata, int width, int height,  int nreps)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index = xIndex + (yIndex)*width;

  for (int r=0; r<nreps; r++) {
    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
      block[threadIdx.y+i][threadIdx.x] =
        idata[index+i*width];
    }

    __syncthreads();

    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
      odata[index+i*height] =
        block[threadIdx.x][threadIdx.y+i];
    }
  }
}


__global__ void transposeCoarseGrained(float *odata,
      float *idata, int width, int height, int nreps)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r<nreps; r++) {
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      block[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      odata[index_out+i*height] =
        block[threadIdx.y+i][threadIdx.x];
    }
  }
}
```

Note that the fine- and coarse-grained kernels are not actual transposes since in either case **odata** is not a transpose of **idata**, but as you will see they are useful in analyzing performance bottlenecks. The performance results for these two cases are added to our table below:

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*

The fine-grained transpose has performance similar to the shared memory copy, whereas the coarse-grained transpose has roughly the performance of the coalesced and bank conflict free transposes. Thus the performance bottleneck lies in writing data to the transposed location in global memory. Just as shared memory performance can be degraded via bank conflicts, an analogous performance degradation can occur with global memory access through partition camping, which we investigate next.

Partition Camping

Just as shared memory is divided into 16 banks of 32-bit width, global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width. We previously discussed that to use shared memory effectively, threads within a half warp should access different banks so that these accesses can occur simultaneously. If threads within a half warp access shared memory though only a few banks, then bank conflicts occur.

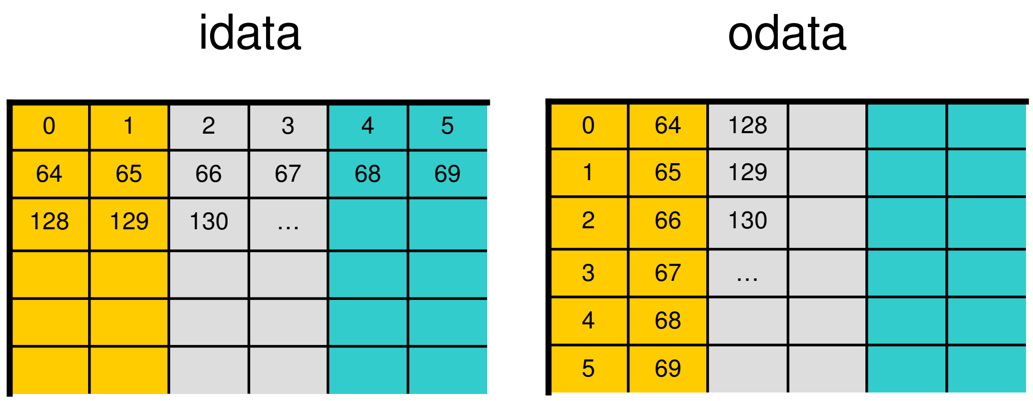
To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions. The term partition camping is used to describe the case when global memory accesses are directed through a subset of partitions, causing requests to queue up at some partitions while other partitions go unused.

While coalescing concerns global memory accesses within a half warp, partition camping concerns global memory accesses amongst active half warps. Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important. When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as:

```
bid = blockIdx.x + gridDim.x*blockIdx.y;
```

which is a row-major ordering of the blocks in the grid. Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed. How quickly and the order in which blocks complete cannot be determined, so active blocks are initially contiguous but become less contiguous as execution of the kernel progresses.

If we return to our matrix transpose and look at how tiles in our 2048x2048 matrices map to partitions on a GTX 280, as depicted in the figure below, we immediately see that partition camping is a problem.



idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	…		

odata

0	64	128			
1	65	129			
2	66	130			
3	67	…			
4	68				
5	69				

With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition. Any float matrix with an integral multiple of 512 columns, such as our 2048x2048 matrix, will contain columns whose elements map to a single partition. With tiles of 32x32 floats (or 128x128 bytes), whose one-dimensional block IDs are shown in the figure, all the data within the first two columns of tiles map to the same partition, and likewise for other pairs of tile columns (assuming the matrices are aligned to a partition segment).

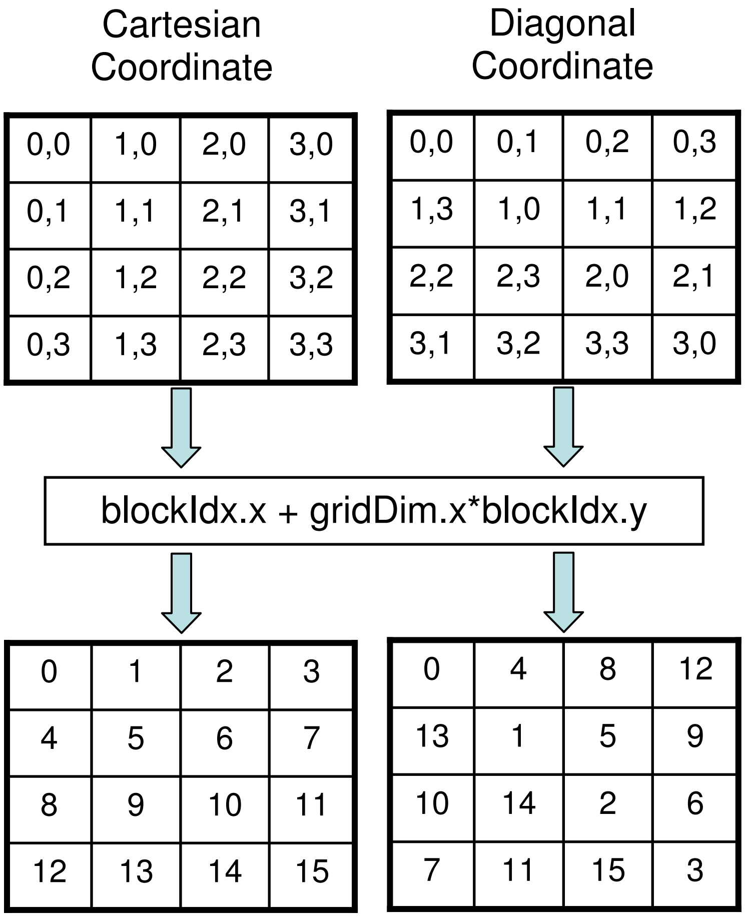
Combining how the matrix elements map to partitions, and how blocks are scheduled, we can see that concurrent blocks will be accessing tiles row-wise in **`idata`** which will be roughly equally distributed amongst partitions, however these blocks will access tiles column-wise in **`odata`** which will typically access global memory through just a few partitions.

Having diagnosed the problem as partition camping, the question now turns to what can be done about it. Just as with shared memory, padding is an option. Adding an additional 64 columns (one partition width) to **`odata`** will cause rows of a tile to map sequentially to different partitions. However, such padding can become prohibitive to certain applications. There is a simpler solution that essentially involves rescheduling how blocks are executed.

Diagonal block reordering

While the programmer does not have direct control of the order in which blocks are scheduled, which is determined by the value of the automatic kernel variable **`blockIdx`**, the programmer does have the flexibility in how to interpret the components of **`blockIdx`**. Given how the components **`blockIdx`** are named, i.e. **`x`** and **`y`**, one generally assumes these components refer to a cartesian coordinate system. This does not need to be the case, however, and one can choose otherwise. Within the cartesian interpretation one could swap the roles of these two components, which would eliminate the partition camping problem in writing to **`odata`**, however this would merely move the problem to reading data from **`idata`**.

One way to avoid partition camping in both reading from **`idata`** and writing to **`odata`** is to use a diagonal interpretation of the components of **`blockIdx`**: the **`y`** component represents different diagonal slices of tiles through the matrix and the **`x`** component indicates the distance along each diagonal. Both cartesian and diagonal interpretations of **`blockIdx`** components are shown in the top portion of the diagram below for a 4x4-block matrix, along with the resulting one-dimensional block ID on the bottom.



0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

0,0	0,1	0,2	0,3
1,3	1,0	1,1	1,2
2,2	2,3	2,0	2,1
3,1	3,2	3,3	3,0

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

Before we discuss the merits of using the diagonal interpretation of **blockIdx** components in the matrix transpose, we briefly mention how it can be efficiently implemented using a mapping of coordinates. This technique is useful when writing new kernels, but even more so when modifying existing kernels to use diagonal (or other) interpretations of **blockIdx** fields. If **blockIdx.x** and **blockIdx.y** represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by the following mapping:

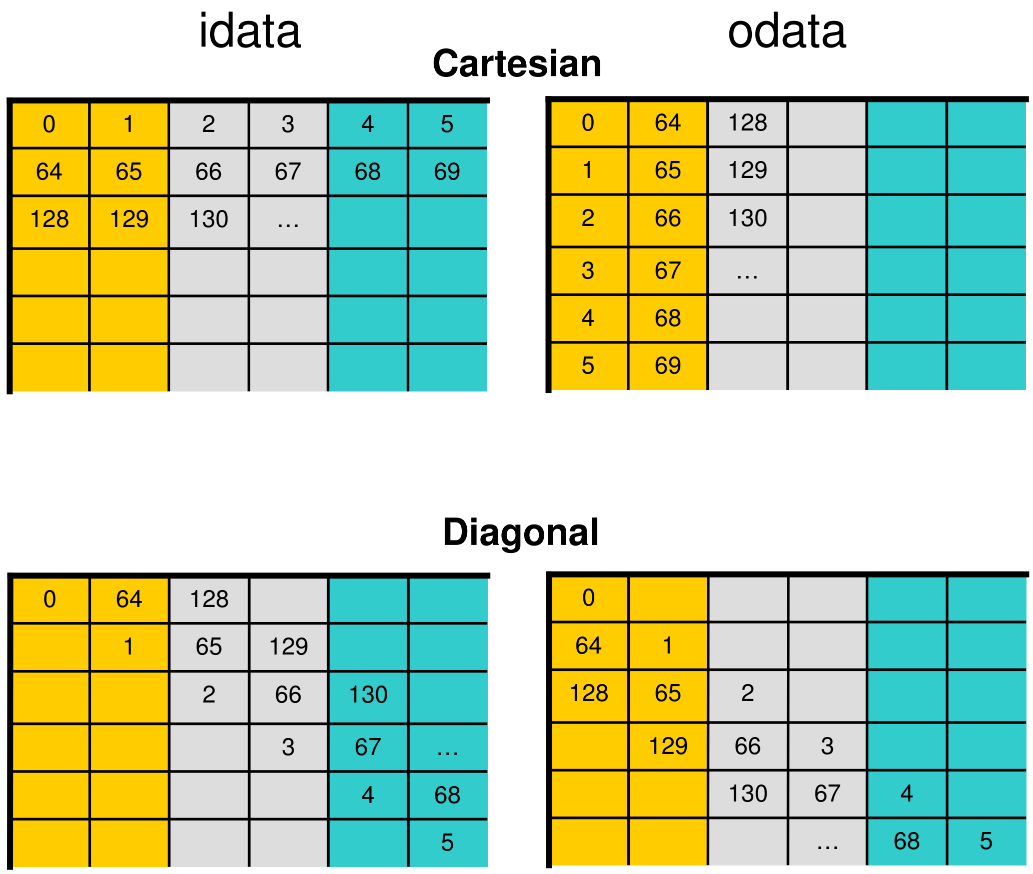
```
blockIdx_y = blockIdx.x;
blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
```

One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of **blockIdx** fields, except using **blockIdx_x** and **blockIdx_y** in place of **blockIdx.x** and **blockIdx.y**, respectively, throughout the kernel. This is precisely what is done in the transposeDiagonal kernel below:

```
__global__ void transposeDiagonal(float *odata,
            float *idata, int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int blockIdx_x, blockIdx_y;

  // diagonal reordering
  if (width == height) {
    blockIdx_y = blockIdx.x;
    blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
  } else {
    int bid = blockIdx.x + gridDim.x*blockIdx.y;
    blockIdx_y = bid%gridDim.y;
    blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
  }

  int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
  yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```

Here we allow for both square and nonsquare matrices. The mapping for nonsquare matrices can be used in the general case, however the simpler expressions for square matrices evaluate quicker and are preferable when appropriate.

If we revisit our 2048x2048 matrix in the figure below, we can see how the diagonal reordering solves the partition camping problem. When reading from **idata** and writing to **odata** in the diagonal case, pairs of tiles cycle through partitions just as in the cartesian case when reading data from **idata**.



Cartesian – idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	…		

Cartesian – odata

0	64	128			
1	65	129			
2	66	130			
3	67	…			
4	68				
5	69				

Diagonal – idata

0	64	128			
	1	65	129		
		2	66	130	
			3	67	…
				4	68
					5

Diagonal – odata

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	
			…	68	5

The performance of the diagonal kernel in the table below reflects this. The bandwidth measured when looping within the kernel over the read and writes to global memory is within a few percent of the shared memory copy. When looping over the kernel, the performance degrades slightly, likely due to additional computation involved in calculating **blockIdx_x** and **blockIdx_y**. However, even with this performance degradation the diagonal transpose has over four times the bandwidth of the other complete transposes.

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*
Diagonal	69.5	78.3

Summary

In this paper we have discussed several aspects of GPU memory management through a sequence of progressively optimized transpose kernels. The sequence is typical of performance tuning using CUDA. The first step in improving effective bandwidth is to ensure that global memory accesses are coalesced, which can improve performance by an order of magnitude.

The second step was to look at shared memory bank conflicts. In this study eliminating shared memory bank conflicts appeared to have little effect on performance, however that is largely due to when it was applied in relation to other optimizations: the effect of bank conflicts were masked by partition camping. By removing the padding of the shared memory array in the diagonally reordered transpose, one can see that bank conflicts have a sizeable effect on performance.

While coalescing and bank conflicts will remain relatively consistent as the problem size varies, partition camping is dependent on problem size, and varies across different generations of hardware. The particular sized matrix in this example will experience far less performance degradation due to partition camping on a G80-based card due to the different number of partitions: 6 partitions on the 8-series rather than 8 on the 200-series.

The final version of the transpose kernel by no means represents the highest level of optimization that can be achieved. Tile size, number of elements per thread, and instruction optimizations can improve performance, both of the transpose and the copy kernels. But in the study we merely focused on the issues that have the largest impact.

Appendix A - Host Code

```
#include <stdio.h>

// kernels transpose/copy a tile of TILE_DIM x TILE_DIM elements
// using a TILE_DIM x BLOCK_ROWS thread block, so that each thread
// transposes TILE_DIM/BLOCK_ROWS elements.  TILE_DIM must be an
// integral multiple of BLOCK_ROWS

#define TILE_DIM 32
#define BLOCK_ROWS 8

// Number of repetitions used for timing.

#define NUM_REPS  100

int
main( int argc, char** argv)
{
  // set matrix size
  const int size_x = 2048, size_y = 2048;

  // kernel pointer and descriptor
  void (*kernel)(float *, float *, int, int, int);
  char *kernelName;

  // execution configuration parameters
  dim3 grid(size_x/TILE_DIM, size_y/TILE_DIM),
       threads(TILE_DIM,BLOCK_ROWS);

  // CUDA events
  cudaEvent_t start, stop;

  // size of memory required to store the matrix
  const int mem_size = sizeof(float) * size_x*size_y;

  // allocate host memory
  float *h_idata = (float*) malloc(mem_size);
  float *h_odata = (float*) malloc(mem_size);
  float *transposeGold = (float *) malloc(mem_size);
  float *gold;

  // allocate device memory
  float *d_idata, *d_odata;
  cudaMalloc( (void**) &d_idata, mem_size);
  cudaMalloc( (void**) &d_odata, mem_size);

  // initalize host data
  for(int i = 0; i < (size_x*size_y); ++i)
    h_idata[i] = (float) i;

  // copy host data to device
  cudaMemcpy(d_idata, h_idata, mem_size,
             cudaMemcpyHostToDevice );
```

```
// Compute reference transpose solution
computeTransposeGold(transposeGold, h_idata, size_x, size_y);

// print out common data for all kernels
printf("\nMatrix size: %dx%d, tile: %dx%d, block: %dx%d\n\n",
      size_x, size_y, TILE_DIM, TILE_DIM, TILE_DIM, BLOCK_ROWS);

printf("Kernel\t\t\tLoop over kernel\tLoop within kernel\n");
printf("------\t\t\t----------------\t------------------\n");

//
// loop over different kernels
//

for (int k = 0; k<8; k++) {
  // set kernel pointer
  switch (k) {
  case 0:
    kernel = &copy;
    kernelName = "simple copy           "; break;
  case 1:
    kernel = &copySharedMem;
    kernelName = "shared memory copy    "; break;
  case 2:
    kernel = &transposeNaive;
    kernelName = "naive transpose       "; break;
  case 3:
    kernel = &transposeCoalesced;
    kernelName = "coalesced transpose   "; break;
  case 4:
    kernel = &transposeNoBankConflicts;
    kernelName = "no bank conflict trans"; break;
  case 5:
    kernel = &transposeCoarseGrained;
    kernelName = "coarse-grained        "; break;
  case 6:
    kernel = &transposeFineGrained;
    kernelName = "fine-grained          "; break;
  case 7:
    kernel = &transposeDiagonal;
    kernelName = "diagonal transpose    "; break;
  }

  // set reference solution
  // NB: fine- and coarse-grained kernels are not full
  //     transposes, so bypass check
  if (kernel == &copy || kernel == &copySharedMem) {
    gold = h_idata;
  } else if (kernel == &transposeCoarseGrained ||
             kernel == &transposeFineGrained) {
    gold = h_odata;
  } else {
    gold = transposeGold;
  }


  // initialize events, EC parameters
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  // warmup to avoid timing startup
```

```
    kernel<<<grid, threads>>>(d_odata, d_idata, size_x,size_y, 1);

    // take measurements for loop over kernel launches
    cudaEventRecord(start, 0);
    for (int i=0; i < NUM_REPS; i++) {
      kernel<<<grid, threads>>>(d_odata, d_idata,size_x,size_y,1);
    }
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float outerTime;
    cudaEventElapsedTime(&outerTime, start, stop);

    cudaMemcpy(h_odata,d_odata, mem_size, cudaMemcpyDeviceToHost);
    int res = comparef(gold, h_odata, size_x*size_y);
    if (res != 1)
      printf("*** %s kernel FAILED ***\n", kernelName);

    // take measurements for loop inside kernel
    cudaEventRecord(start, 0);
    kernel<<<grid,threads>>>
        (d_odata, d_idata, size_x, size_y, NUM_REPS);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float innerTime;
    cudaEventElapsedTime(&innerTime, start, stop);

    cudaMemcpy(h_odata,d_odata, mem_size, cudaMemcpyDeviceToHost);
    res = comparef(gold, h_odata, size_x*size_y);
    if (res != 1)
      printf("*** %s kernel FAILED ***\n", kernelName);

    // report effective bandwidths
    float outerBandwidth =
       2.*1000*mem_size/(1024*1024*1024)/(outerTime/NUM_REPS);
    float innerBandwidth =
       2.*1000*mem_size/(1024*1024*1024)/(innerTime/NUM_REPS);
    printf("%s\t%5.2f GB/s\t\t%5.2f GB/s\n",
       kernelName, outerBandwidth, innerBandwidth);
  }

  // cleanup

  free(h_idata); free(h_odata); free(transposeGold);
  cudaFree(d_idata); cudaFree(d_odata);
  cudaEventDestroy(start); cudaEventDestroy(stop);

  return 0;
}
```

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

Macrovision Compliance Statement

NVIDIA Products that are Macrovision enabled can only be sold or distributed to buyers with a valid and existing authorization from Macrovision to purchase and incorporate the device into buyer's products.

Macrovision copy protection technology is protected by U.S. patent numbers 5,583,936; 6,516,132; 6,836,549; and 7,050,698 and other intellectual property rights. The use of Macrovision's copy protection technology in the device must be authorized by Macrovision and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Macrovision. Reverse engineering or disassembly is prohibited.

Copyright