Lab 3: Matrix Transpose and Matrix Multiplication Optimization with CUDA
Team ONE

Daniel Putt (ID:1654588)
Vignesh Jeganathan (ID:1552354)

October 31, 2018

Matrix Transpose:

Naive GPU

The naive GPU version that was given breaks up the matrix into 64x64 blocks, and further divides each submatrix into 32 one by four blocks, which each of the 32 threads handle. The naive version uses the warps well, but does not coalesce memory, and does not use shared memory.

shmem GPU

This is the first version that we wrote. We create a statically sized shared array of size [32][33]. Using a similar algorithm to the naive GPU version, the shared arrays are loaded with the respective elements from global memory.
The threads are then synced, to make sure threads don't write to the output/transposed matrix before the shared arrays are filled completely.
Lastly, the shared arrays are written back to global memory, in pretty much the same way as they were read in from 'input'.

Parameter choices:

The best choice for block size is 32. It fits well with the WARPs. The threads still work over 1x4 blocks.
It is also important to note that loading the shared memory array of size [32][32] with block size 32 (after it is filled with data from global memory) results in a 32 way memory bank conflict. When we pad the second dimension by one, (i.e.[32][33]), this bank conflict is completely eliminated.

Optimized GPU

This version is almost identical to shmem. Pragma unroll directives were added to the for loops, but the compiler already does a great job at optimizing the code, including unrolling. The effect on efficiency is minimal, though noticeable.

Results and Discussion:

Computation Time(in ms)					
	Naive CPU	GPU memcpy	Naive GPU	shmem GPU	Optimized GPU
512	0.759	0.046	0.061	0.024	0.022
1024	2.417	0.063	0.138	0.053	0.053
2048	11.028	0.227	0.513	0.194	0.192
4096	176.053	0.752	2.091	0.765	0.762

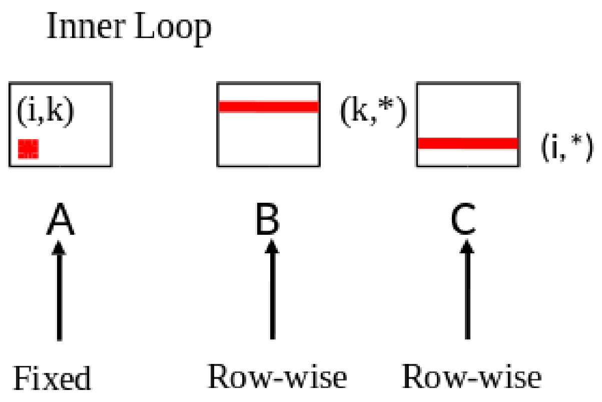
The table above shows example computation time for the different versions of matrix transpose. The computation times don't change much from run to run.
Our shmem and optimized versions are about the same or better than GPU memcpy, and about 3 times as fast as the naive GPU version.

Matrix Multiplication: Cache Management

Description:

The naive version uses ijk loop where each node is given to c matrix elements and the inner loop goes through rows and columns of A and B respectively. This leads to cache incoherence as there are so many memory jumps within the inner loop.

In order to avoid this, the following method of matrix multiplication is used. The threads are assigned to matrix A and the iteration is done along the rows of B and C respectively. Since, C program is row major, this scheme has the better cache coherence.



Methodology:

This method is challenging to implement because at any point of time, multiple threads will be trying to write to an element in C. In order to prevent this, atomic operation should be used.

Drawbacks:

This method of matrix multiplication relies on using atomic operations. This leads to bottleneck because of serialization.

Shared Memory Implementation of Cache Optimized Version:

Efforts are made to optimize the cache optimized version by using shared memory. However it was easy to implement for matrix B but since many threads will be trying to write to matrix C, making a shared memory out of C becomes difficult. Again, an atomic operation should be done in making shared memory for matrix C, which would hinder the performance. Hence, memory sharing is done only for matrix B.

Results:

There is an improvement in the performance in both a simple cache coherent version and shared memory cache coherent version. Following are an example set of results. Any given run will differ only slightly in computation time.

Computation Time(in ms)					
	Naive CPU	Naive GPU	Cache GPU	Shared-Cache-Optimized GPU	Shared GPU
512	94.72	26.58	15.90	15.40	0.78
1024	739.18	204.50	122.70	118.57	5.97
2048	5991.76	1614.44	970.12	946.01	47.13
4096	53803.05	12892.59	7993.72	7708.63	375.44

Some Notes on Results:

- The cache friendly version is 85.14% faster than the naive version, for 4096*4096 matrix.
- The shared memory implementation of the cache friendly version is 85.67% faster than the naive version for 4096*4096 matrix.

Shared Matrix Multiply:

This version is similar to the shmem transpose we've written, in that 32x32 blocks are loaded into statically sized shared memory arrays for matrices A and B.
After a sync, a temporary float variable is used to sum up the calculations for each element in the resultant matrix C.
After every loop for summing over our temporary float, the threads are synced.
Finally, each temporary float is written back to the global memory array for matrix C.

Other notes about Shared Matrix Multiply:

- Unlike the shared memory transpose code, each thread handles one element, not a 1x4 chunk of the input matrices.
- Using a [32][32] size for the shared arrays for both A and B is most efficient.
- There is only one section of the code that has memory bank conflicts. This occurs after we have loaded matrix A into shared memory. Specifically, there is a 32 way bank conflict when we're multiplying elements in shared A and shared B to get the corresponding element C.

- Luckily, since the bank conflicts are on shared memory, the impact isn't nearly as great as it would be for global reads and writes.
- By padding the shared matrix for A to a size [32][33], the shared memory bank conflicts are completely eliminated. However, the efficiency actually significantly decreases when I make this change. We have left the shared matrix for A at [32][32] in the code.
- Our results are significant. Using shared memory to store submatrices of A and B to calculate over allowed our efficiency to be on the order of 10 times faster than both the naive GPU version and our attempts at a cache GPU version of matrix multiply. Example computation times for our shared matrix multiply code are listed in the table above.