Matrix Transpose and Multiplication in Cuda

Xiaoxu Na

Abstract – Using Navidia GPU to parallelize matrix transpose and multiplication.
Index Term – Navidia GPU, Cuda, matrix transpose, matrix multiplication

1 Introduction

In this project, it is important to explore performance programming of matrix transpose and multiplication running on NVIDIA's Kepl. In this project, we consider only square matrices whose dimensions are integral multiples of 32 on a side. Finally, we'll test the kernels with the side dimension of 521, 1024, 2048 and 4096. In addition to performing several different matrix transposes, we run simple matrix copy kernels because copy performance indicates the performance that we would like the matrix transpose to achieve.

2 Algorithms and Result Analysis

2.1 Matrix Transpose

2.1.1 Naïve Kernel

The naïve algorithm is straightforward that each thread takes care of 4 elements, and output[j][i] = input[i][j].

2.2 The Shared Memory Kernel

Because it is on-chip, shared memory is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency, provided that there are no bank conflicts between the threads. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory [1].

Based on this situation, we may speed up the naïve transpose by cached them into the shared memory. In order to make the kernel running faster, we should utilize the coalescing memory, since when accessing global memory, peak bandwidth utilization occurs when all threads in a warp access one cache line. The warps of threads cached sub blocks of the input matrix into rows of the data matrix in the shared memory. After recalculating the array indices, a column of the shared memory data is written to contiguous addresses in the output matrix.

For a shared memory data matrix of 64 × 64 elements, all elements in a column of data map to the same shared memory bank, resulting in a worst-case scenario for memory bank conflicts: reading a column of data results in a 16-way bank conflict since each thread takes care of 4 elements. Luckily, the solution for this is simply to pad the width in the declaration of the shared memory data, making the data column 65 elements wide rather than 64. In this way, when accessing the data matrix in the shared memory, the elements are read from different banks.

2.3 The Optimized Kernel

While each thread takes care of 1 element per iteration, after 4 iterations, the threads scan the entire matrix. It is easy just to unroll the 4 loops to make the kernel run faster. The

results are shown in Result 1.1 as below. For the optimal kernel is limited by the bandwidth, the bigger the matix is, the worse speedup it gains.

Size 512 naive CPU: 0.779520 ms
Size 512 GPU memcpy: 0.033760 ms
Size 512 naive GPU: 0.058688 ms
Size 512 shmem GPU: 0.021856 ms
Size 512 optimal GPU: 0.019200 ms

Size 1024 naive CPU: 2.592096 ms
Size 1024 GPU memcpy: 0.060256 ms
Size 1024 naive GPU: 0.134496 ms
Size 1024 shmem GPU: 0.054848 ms
Size 1024 optimal GPU: 0.056352 ms

Size 2048 naive CPU: 11.028992 ms
Size 2048 GPU memcpy: 0.211840 ms
Size 2048 naive GPU: 0.514208 ms
Size 2048 shmem GPU: 0.191616 ms
Size 2048 optimal GPU: 0.205632 ms

Size 4096 naive CPU: 177.857178 ms
Size 4096 GPU memcpy: 0.741440 ms
Size 4096 naive GPU: 2.053984 ms
Size 4096 shmem GPU: 0.732224 ms
Size 4096 optimal GPU: 0.796768 ms

Result 1.1

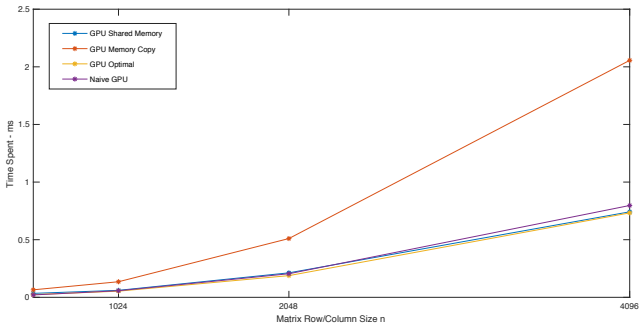


Fig. 1.1 GPU Matrix Transpose Kernels

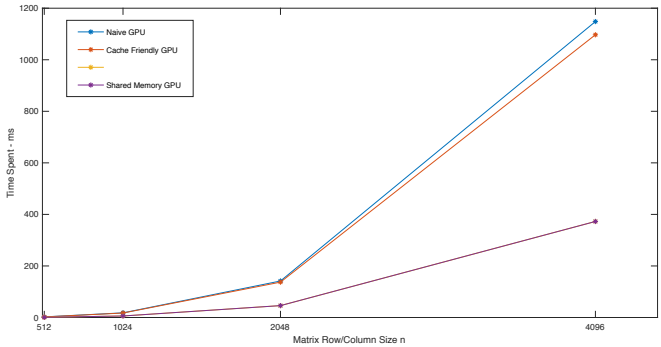
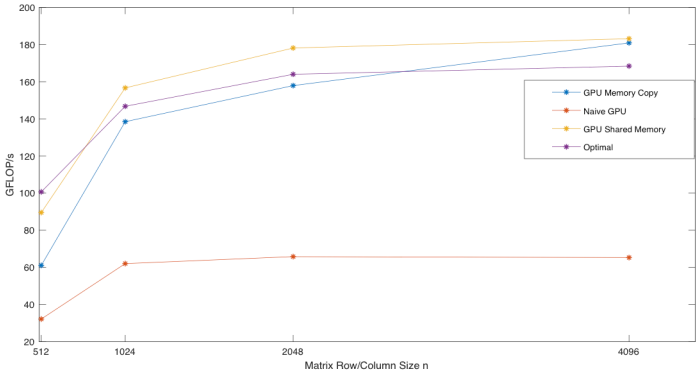


Fig. 1.2 GPU Matrix Multiplication Kernels



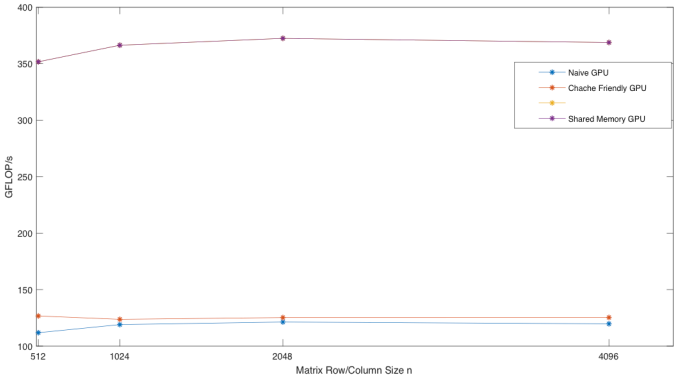


Fig. 1.3 GPU Matrix Transpose Kernels GFLOP/s Fig. 1.4 GPU Matrix Multiplication Kernels-GFLOP/s

2.2 Matrix Multiplication

2.2.1 Optimized Naïve Kernel

The optimized naïve algorithm is quite the same as the mathematic definition that each item in matrix c is the addition of multiplication of all elements in the corresponding row of a and

the column of b accordingly. The only optimization here is the mapping compared with the original given naïve kernel.

2.2.2 The Cache Friendly Kernel

The cache friendly algorithm focuses on less cache miss rate. Recalling what we had learned in lecture four, there was a ijk version of the matrix multiplication, which for any elements in a, it multiplied with the entire row of b to get the entire row of c. After n iterations, we got the final elements of c. By this way, the matrix b and c are accessed in the row order. First, this method was given a try, but the result is not satisfying. From the results shown below, we may obviously see that this method took even more time than the naïve kernel. Back taking a look of the kernel code, we found out that in every iteration, the entire b was accessed and so was the matrix c. As a result shown in Result 1.2, although the matrix b and c are all accessed by the row order, the global access time is still very long as it needs to be accessed n times.

Size 512 naive CPU: 96.126556 ms
Size 512 naive GPU: 2.393152 ms
Size 512 cache GPU: 8.708224 ms

Size 1024 naive CPU: 742.379272 ms
Size 1024 naive GPU: 17.950592 ms
Size 1024 cache GPU: 69.436668 ms

Size 2048 naive CPU: 5916.551758 ms
Size 2048 naive GPU: 141.012863 ms
Size 2048 cache GPU: 541.046814 ms

Size 4096 naive CPU: 53887.980469 ms
Size 4096 naive GPU: 1140.422852 ms
Size 4096 cache GPU: 4297.110352 ms

Result 1.2

Size 512 naive CPU: 95.523392 ms
Size 512 naive GPU: 2.400896 ms
Size 512 cache GPU: 2.119584 ms
Size 512 shared GPU: 0.763104 ms
Size 1024 naive CPU: 741.913513 ms
Size 1024 naive GPU: 18.045759 ms
Size 1024 cache GPU: 17.365856 ms
Size 1024 shared GPU: 5.860544 ms
Size 2048 naive CPU: 5976.682617 ms
Size 2048 naive GPU: 141.647995 ms
Size 2048 cache GPU: 137.167236 ms
Size 2048 shared GPU: 46.116673 ms
Size 4096 naive CPU: 53851.527344 ms
Size 4096 naive GPU: 1148.214478 ms
Size 4096 cache GPU: 1096.950806 ms
Size 4096 shared GPU: 372.610901 ms

Result 1.3

However, considering the matrix multiplication definition, only a or b may gain a better hit rate. The best way should be the same thing we did in shared transpose by sub block access, just without using the shared memory. We use the block size of 32 x 32, so that each block of c is gained by the addition of the multiplication of one row of blocks of a and a corresponding column of blocks of b. The running time is improved as shown in Result 1.3.

2.2.3 The Shared Memory Kernel

This kernel is a similar version as an improvement of cache friendly kernel, and the only difference is that we cached both sub blocks of a and b into the shared memory by defining sharedMemA and sharedMemB, the results in Result 1.3 are based on a 32*32 BLOCK_WIDTH, however, this can be changed in order to get the best results. It is defined at the top of the kernel file as #define BLOCK_WIDTH 32.

3 Conclusions

In the experiment, we defined the BLOCK_WIDTH as 64 in the matrix transpose section, and 32 in the matrix multiplication section, as displayed in Fig 1.1 and Fig. 1.2. However, the code is friendly to change this setting at the top so that more tries can be given to the parameters to find the best solution.

Acknowledges

The author would like to express the gratitude to Dr. Panruo Wu.

[1] Mark Harris, Using Shared Memory in CUDA C/C++, *https://devblogs.nvidia.com/using-shared-memory-cuda-cc/.*