Judsen Hembree

Programming II

Due: submit the report and source codes to canvas

100 points total

All answers in red

Results were collected with the following configuration

```
#PBS -N results_runner
#PBS -l select=1:ncpus=4:mem=60gb:ngpus=1:gpu_model=v100
#PBS -l walltime=24:00:00
#PBS -j oe
```

Overview

You will practice CUDA programming in this assignment. Particularly, you will follow the basic CUDA programming and don't use unified virtual memory. Through this assignment, you will develop an understanding of CUDA execution model, memory model, and techniques for performance optimization.

References:

1. CUDA C programmer's guide https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
2. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA by Ryoo et al. https://dl.acm.org/citation.cfm?id=1345220

Environment platform

You will run the experiments on Palmetto GPU nodes. You should request the same node (or the nodes with the same GPU card, e.g., P100, V100, K40, or K80) for your experiments. For Palmetto resource request, visit CCIT website https://citi.sites.clemson.edu/infrastructure/.

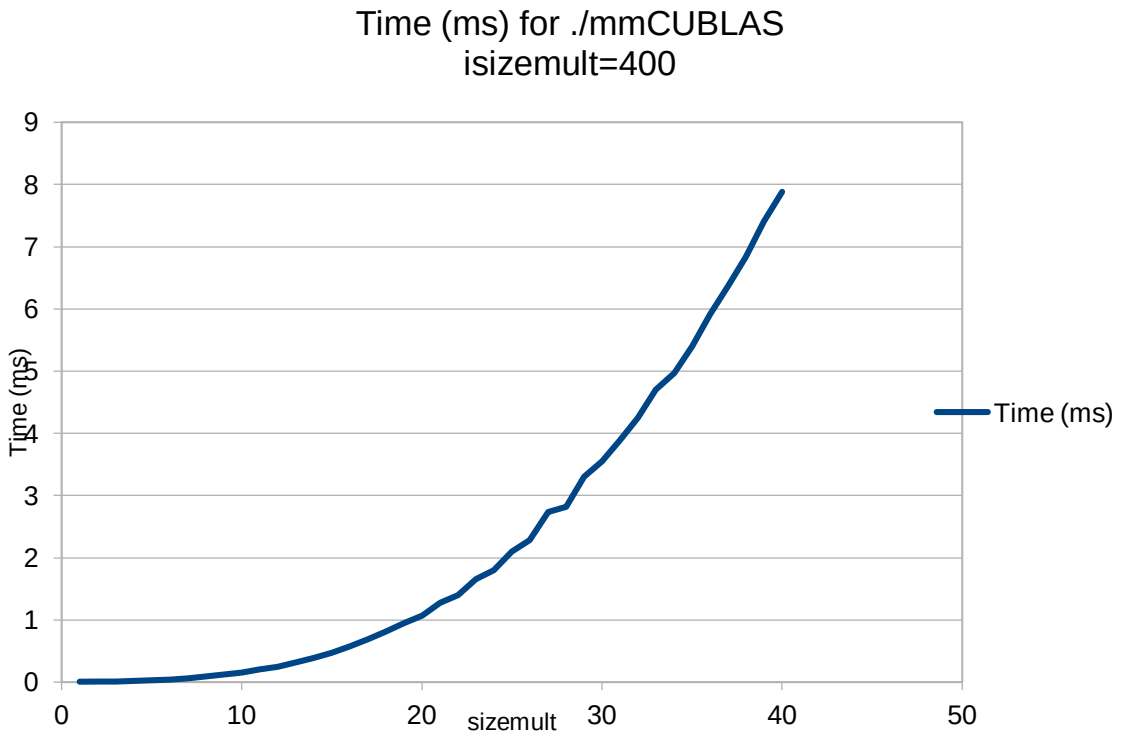
Judsen Hembree

Part 1: Compute matrix multiplication with CUDA library (10pts)

mmCUBLAS (available on canvas) uses GPU device to compute matrix multiplication matrix_result = A * B for single precision matrices. Instead of implementing from scratch, this program invokes the cublasSgemm() function.

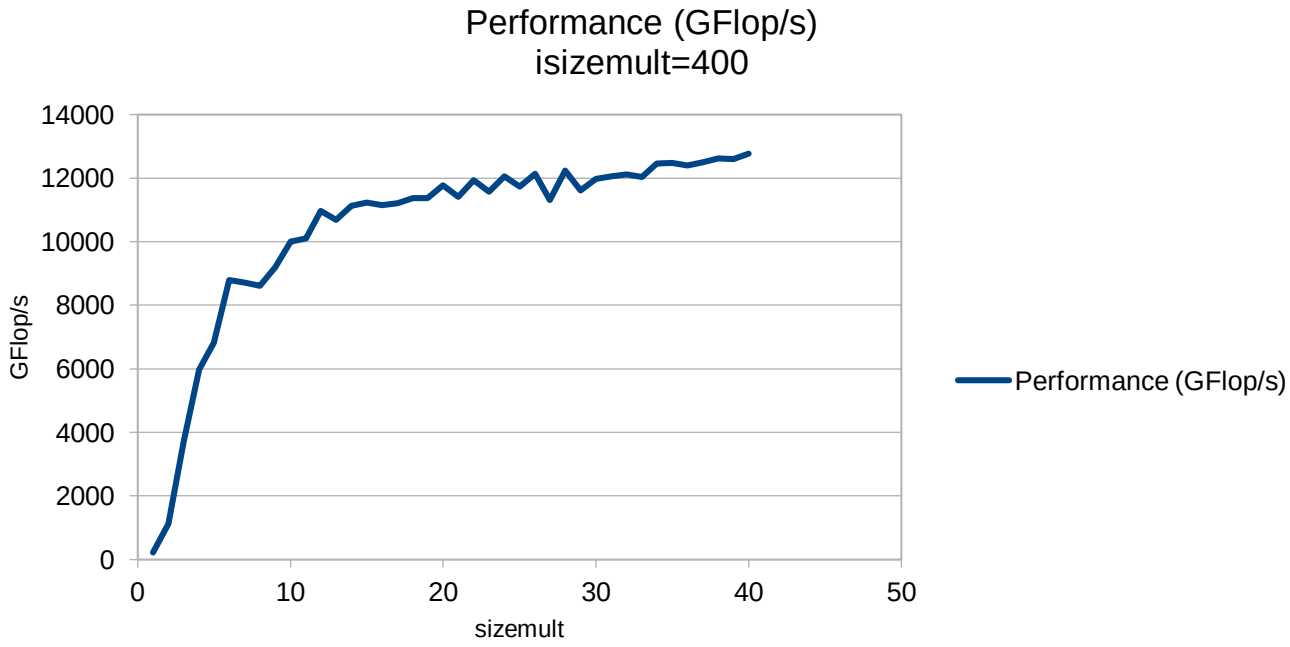
This code follows a good practice for performance measurement. For example, it verifies the computation result, performs a warmup operation before timing the execution, and times multiple iterations of matrix multiplication to make sure the execution time is long enough. Take note where the timing starts and stops.

Compile the code, and collect execution time for a wide range of matrix sizes. Show how performance changes with matrix size and explain why this trend is reasonable.



Here we have the timed runs for ./mmCUBLAS. There is an exponential increase in time (ms) as sizemult increases.

Judsen Hembree



Here is the Gflop/s performance of ./mmCUBLAS. This measures the floating point operations per second. It is evident that the GPU is underutilized until around sizemult 10-15. After this point Gflop/s has reached closer to its peak throughput. This correlates nicely with the time graph above it. As Gflop/s become more restricted the time per ms increases. It would appear that the ./mmCUBLAS implement has a theoretical gflop/s max around 13000-14000 gflop/s.

Judsen Hembree

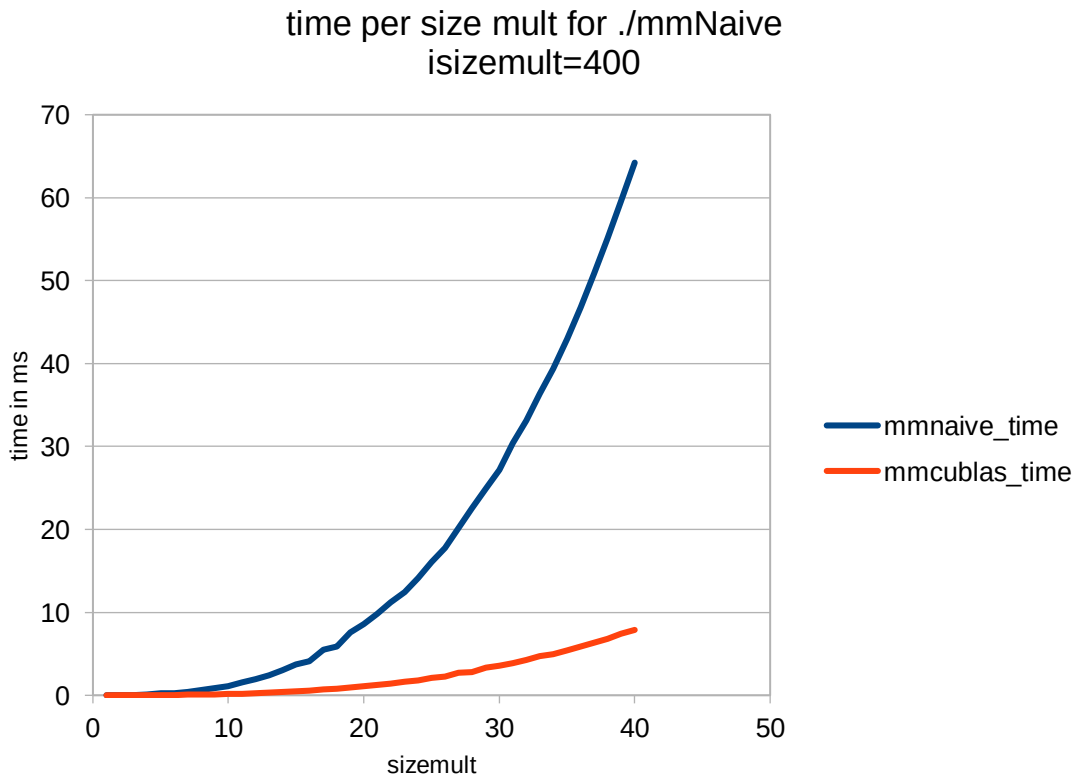
Part 2: Implement matrix multiplication with basic CUDA (20pts)

Instead of calling the cublasSgemm() function, implement a kernel function for matrix multiplication with basic CUDA, and timing its performance.

You should name your program mmNaive.cu. You can reuse the structure and code segments of mmCUBLAS.cpp, and replace cublasSgemm with your own implementation. Check to make sure the computation on the device is correct.

Use the similar timing method for your implementation. For example, perform a warmup operation before timing the execution, time multiple iterations of matrix multiplication to make sure the execution time is long enough.

Compile the code, and collect execution time for the same matrix sizes. Show how performance changes with matrix size and compare your performance with MMCublas. How slower is your implementation? Can you identify the reasons?



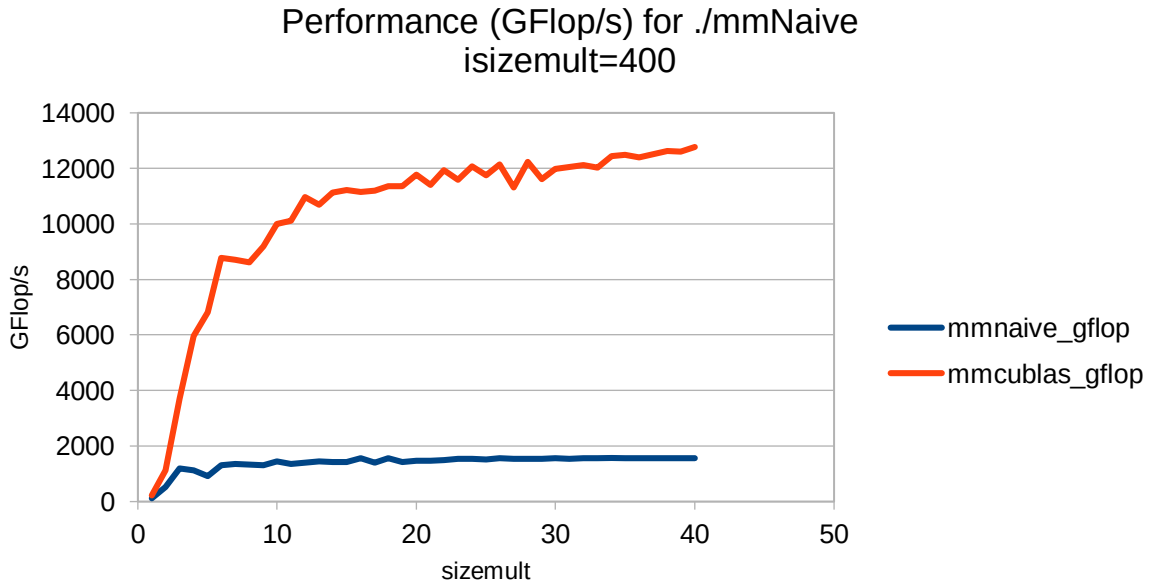
Here I have implemented a naive version of cuda matrix multiplication following the instructions for accurate performance measurement above. It shows just how optimized the CUBLAS library is. Optimizations that the CUBLAS library likely uses are matrix transposition, loop unrolling, and coalescing of memory using the shared memory L1 cache located on the gpu.

Judsen Hembree

My naive implement uses no transposition of matrices. The CUBLAS library transposes c matrices so that it can be placed into memory in a fashion that benefits cuda's preferred memory order. C stores information by rows whereas Cuda stores information in columns. By transposing matrices Cuda has faster memory access.

My naive implement uses no loop unrolling. Loop unrolling is a loop transformation technique that seeks to increase throughput at the expense of the codes binary size. By explicitly unrolling loops you decrease the loops iterations. By doing this you limit the operations that deal with the loops conditions. There are less conditional statements inside on the program.

My naive implement also uses no shared memory. GPU's have cache memory located on chip which allows them to run quicker than a CPU. The L1 cache is accessible to programmers via the __shared__ tag. However, in my naive implement I do not utilize the L1 cache and instead make all my calls to global memory. So any caching is done by the compiler.



The lack of the listed techniques above in my naive implement are likely the cause of my poor Gflop throughput. ./mmCublas's optimized version allows it to perform magnitudes more floating point operations than an unoptimized version.

Judsen Hembree

Part 3: Optimize your matrix multiplication (70pts)

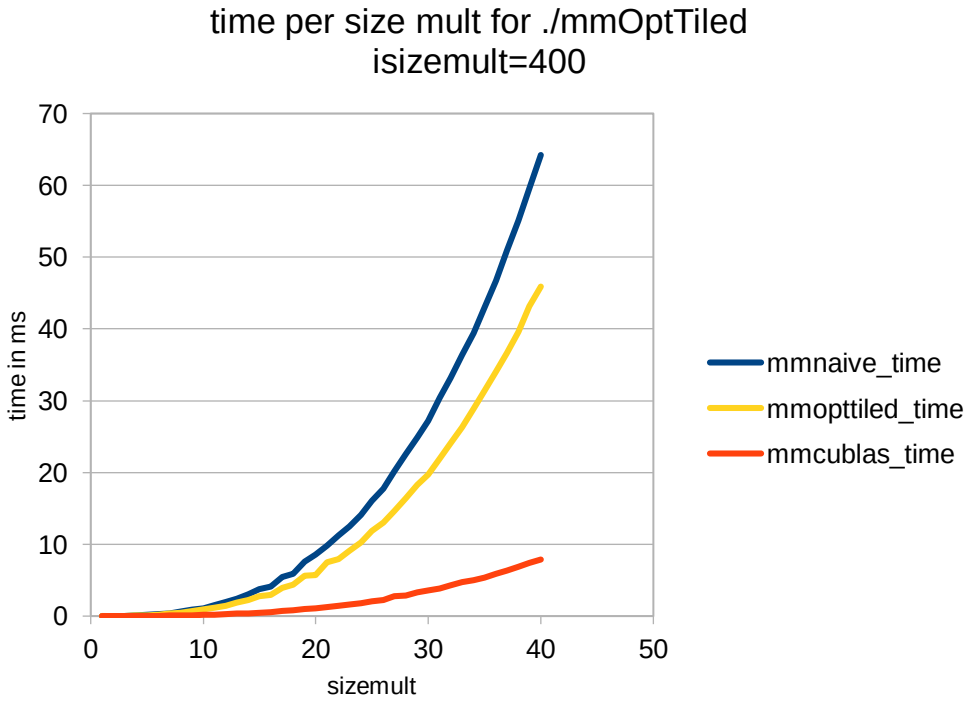
Optimize your code from Part 3. Name your program mmOpt.cu. You can reuse the structure and code segments of mmCUBLAS.cpp.

In your write up, describe in detail the techniques you have used and why you choose to use them. The final version should include several optimization techniques and has the highest performance. Present the speedup gained from each newly stacked technique. Your points will be based on the performance gain your programs achieve and your explanations, justifications, and reasoning.

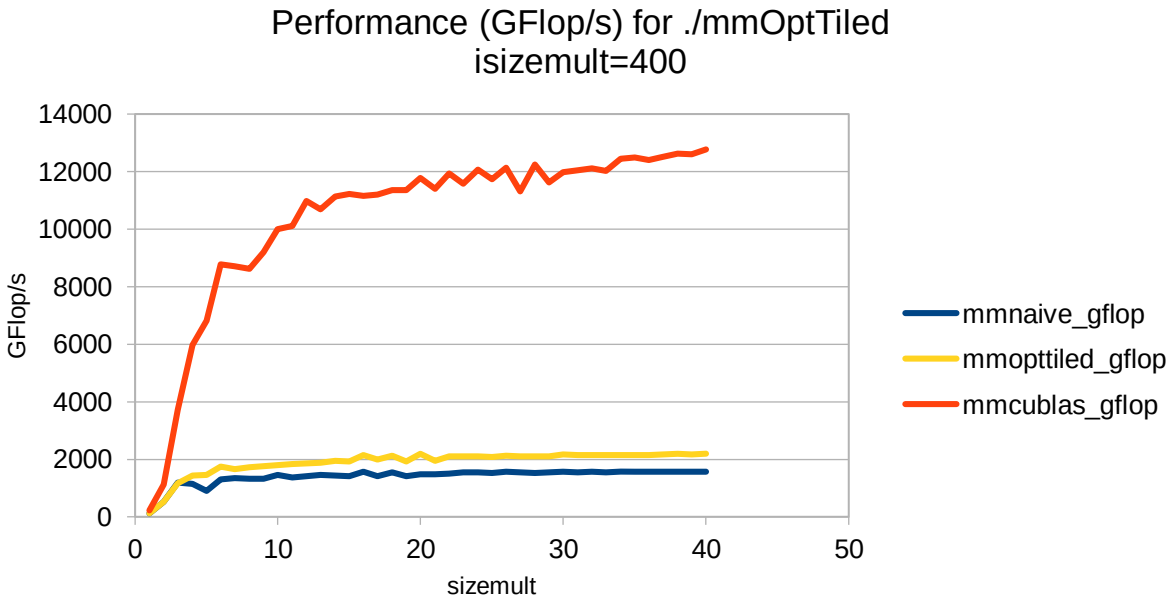
Matrix transpose is a common exercise in GPU optimization, so do not search for existing GPU matrix transpose code on the internet.

Optimization 1-2: shared memory and tiled partitioning

For my first and second performance improvement I utilized the L1 cache and a partitioning technique known as tiling. By allocating space for specific rows and columns of matrix A and B on the L1 cache, or shared memory, we can focus on a particular “tile” of matrix output C. Rows and columns of A and B loaded to the L1 cache reduces calls to global memory. Since the L1 cache is located directly on chip it allows for faster memory access and increases performanc
e.



We can see the impact of using tiling and shared memory above. I cut a significant chunk out of my naive implements operating time.

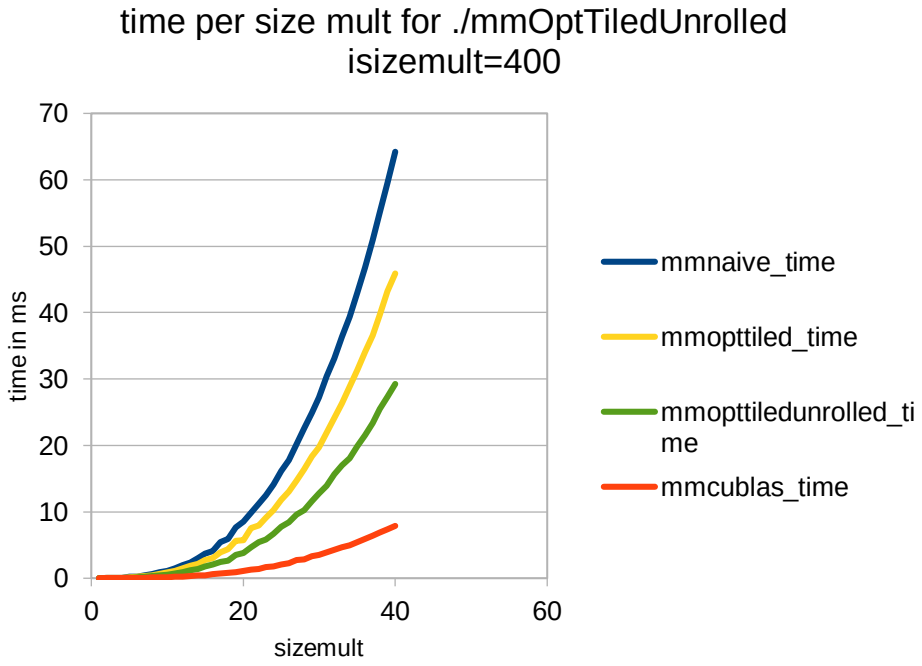


Here we can see how my gflop/s has increased. Most of this increase is likely from utilizing shared memory.

Judsen Hembree

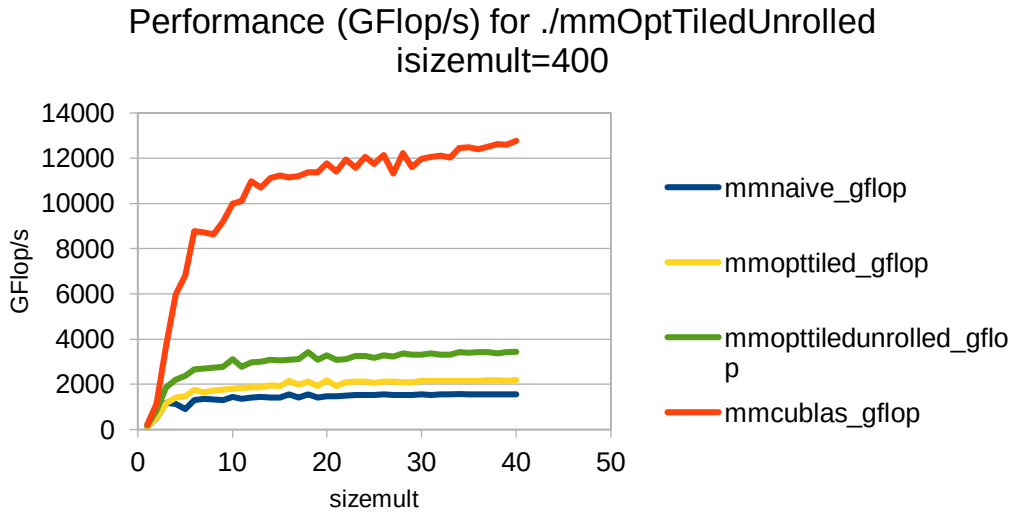
Optimization 3: Loop unrolling

Loop unrolling reduces conditional operations by lessening the amount of iterations in a for loop. NVCC, the compiler used for cublas, automatically unrolls inner loops, but by using the compiler directive #pragma unroll(x) you can force the compiler to unroll outer loops as well. I used a loop unroll factor of tile_size inside of my code. tile_size was a global variable set to 32. By unrolling the outer for loops we can mitigate the amount of conditional calls. However, this comes at the cost of increasing binary program size. So there is trade-offs.



The green line above has both optimization 1, 2, and 3 inside of its code. I have cut my time nearly in half from my naive program.

Judsen Hembree



My Gflop/s has increased as well. The increase in gflop/s was greater coming from loop unrolling than utilizing shared memory.

Judsen Hembree

Submission Instructions

Submission will be through canvas.

Please place the following files in your submission:

- Your writeup, in a file called `writeup.pdf`
- All your source codes in a single tar file. All code must be compilable and runnable!