Assignment 6, Final Assignment

David Warren
University of Alabama
dlwarren2@crimson.ua.edu
CWID: 11703777

I. Introduction

This report reflects the observations and analyses of cuBLASSgemm parallel implementations of matrix multiplication and shared memory implementations of matrix multiplication using CUDA. In Part I, we explore parallelization using the cuBLAS library. In Part II, we explore parallelization using CUDA programming. In Part III, we compare our performance results to that of matrix multiplication in the previous Project 2. Both programs accept arbitrary matrix sizes.

II. Ease of Use

A. *GPU and Compiler Information*

GPU used was the NVIDIA Tesla V100 on the DMC cluster. Default settings were using in the run_gpu script. Compiler used was gcc (GCC) 7.3.0. Optimization flag was set to default -O0 least optimized for parts I-II.

B. *Using the Makefile*

- "Make" to compile *sgemm.c, matrix.c* in Part I and *shared_mm.cu* in Part II. Each program has its own makefile in their respective folders
- "Make clean" to remove binary and executable files
- To execute the programs, run them with the following arguments: "./mm file1.mtx file2.mtx file1dim1 file1dim2 file2dim1 file2dim2"

III. Part I Observations

cuBLASSgemm performed exceptionally well on large matrix sizes. Below you may see my results for given pairs of a & b matrices from the google drive folder.

Fig (1) cuBLASSgemm performance

Matrix Size	sgemm time (s)	sgemm performanc
[3x3] [3x2]	0.000496	0.0002 GFLOPS
[32x32] [32x32]	0.000681	0.2 GFLOPS
[1024x1024] [1024x1024]	0.001096	3.918 TFLOPS
[2048x2048] [2048x2048]	0.00196	17.53 TFLOPS

Due to low number of operations, the small matrices naturally have a low performance. You may really see the effectiveness of this function in the larger pairs of matrix multiplication. The 2048x2048 matrices problem scored very high on performance and held the highest performance metric out of both parts of this assignment with 17.53 TFLOPS. One interesting note on this implementation. Since cuBLASSgemm uses FORTRAN style column-major ordering for matrices, we had to trick the function into transposing our matrices into row-major ordering. While this took some time to achieve successfully, it makes our resulting matrix, output as "c.dat" after each run_gpu script instance, much easier to read and verify.

IV. Part II Observations

Shared memory matrix multiplication using CUDA programming performed quite well on large inputs, however did not perform as well as the cuBLASSgemm function. Peak performance for large inputs was with a block size of 16, with 1024x1024 at 5.187 TFLOPS and 2048x2048 at 5.91 TFLOPS. The 32x32 matrix peaked at a block size of 4. This makes sense as it allows for more parallelization, but decreases on the communication costs of say using a block size of 1, which really is not much of tiled matrix multiplication at all. With the shared memory implementation however, we were able to achieve better performance on smaller matrices, most likely due to being able to optimize block sizes for different inputs. You can see the full results of all a & b matrix pairs from the Google drive below, with block sizes in the range { 1, 4, 8, 12, 16 }.

Fig (2) Block Size 1 performance

Matrix sizes	Block Size 1 time (s)	Block Size 1 performance
[3x2] [3x3]	0.000044	0.002 GFLOPS
[32x32] [32x32]	0.000052	2.52 GFLOPS
[1024x1024] [1024x1024]	0.170575	25.18 GFLOPS
[2048x2048][2048x2048]	1.401474	24.52 GFLOPS

Fig (3) Block Size 4 performance

Matrix sizes	Block Size 4 time (s)	Block Size 4 performance
[3x2] [3x3]	N/A	N/A
[32x32] [32x32]	0.000046	2.85 GFLOPS
[1024x1024] [1024x1024]	0.004611	931.46 GFLOPS
[2048x2048][2048x2048]	0.037355	919.82 GFLOPS

Fig (4) Block Size 8 performance

Matrix sizes	Block Size 8 time (s)	Block Size 8 performance
[3x2] [3x3]	N/A	N/A
[32x32] [32x32]	0.000048	2.73 GFLOPS
[1024x1024] [1024x1024]	0.001257	3.42 TFLOPS
[2048x2048][2048x2048]	0.009487	3.62 TFLOPS

Fig (5) Block Size 12 performance

Matrix Sizes	Block Size 12 time (s)	Block Size 12 performance
[3x2] [3x3]	N/A	N/A
[32x32] [32x32]	0.001004	0.13 GFLOPS
[1024x1024] [1024x1024]	0.001008	4.26 TFLOPS
[2048x2048][2048x2048]	0.007431	4.62 TFLOPS

Fig (6) Block Size 16 performance

Matrix sizes	Block Size 16 time (s)	Block Size 16 performance
[3x2] [3x3]	N/A	N/A
[32x32] [32x32]	0.000827	0.15 GFLOPS
[1024x1024] [1024x1024]	0.000828	5.187 TFLOPS
[2048x2048][2048x2048]	0.005815	5.91 TFLOPS

V. Part III Observations

We now will compare the performance of the GPU matrix multiplication implementations to the CPU performance of Assignment 2. Recall, in Assignment 2 we used two methods for matrix multiplication: single register reuse, blocking, and a combination of both.

A. Single register reuse from Assignment 2

First see the performance from Assignment 2 seen below. The performance metrics used were the implementations with -O0 optimizations in the compiler.

Fig (7) Single Register Reuse performance

	ijk	jik	ikj	kij	jki	kji
GFLOPS	0.774847	0.614022	1.379492	1.370672	0.271218	0.23454

As you can see, the ikj method hit peak performance at 1.379292 GFLOPS in 10000x10000 matrix multiplication. For large matrix multiplication, our GPU performance greatly exceeds that of CPU performance.

B. Blocking from Assignment 2

See below the performance from Assignment 2's blocking implementation for matrix multiplication. These performance metrics were also gathered using the -O0 compiler optimizations.

Fig (8) Blocking performance in GFLOPS

	ijk	jik	ikj	kij	jki	kji
16	1.403387	1.351818	1.321562	1.326711	0.512471	0.630856
32	1.484794	1.357386	1.379682	1.363916	0.498399	0.506869
64	1.496316	1.402321	1.406499	1.382087	0.497388	0.501146
128	0.782911	0.774912	1.339661	1.275623	0.330302	0.475397
256	0.796609	0.753738	1.394318	1.346473	0.457817	0.542614
512	0.801143	0.782286	1.41336	1.408996	0.402167	0.533043

Peak performance for 16, 32 and 64 block size are held by *ijk* with around 1.4-1.5 GFLOPS. Peak performance for 128, 256, and 512 are held by *ikj* with 1.3-1.4 GFLOPS. Again, these were on 10000x10000 sized matrices. GPU performance again greatly exceeded CPU performance.

C. Combination from Assignment 2

See below the performance for combination of single register reuse and blocking for size matrices of 2048x2048. Performance was measured in GFLOPS.

Fig (9) Combination performance in GFLOPS

	ijk	jik	ikj	kij	jki	kji
16	3.591561	3.419653	3.749825	3.751255	2.695864	2.859195
32	3.778166	3.664244	3.861342	3.802993	2.629922	2.803311
64	3.771746	3.656441	3.92213	3.509196	2.443805	2.586338
128	2.391988	2.458115	3.857597	3.54987	1.718247	2.015485
256	2.514728	2.479785	3.904554	3.799822	2.006773	1.971327
512	2.502276	2.313585	3.890833	3.881294	1.79078	1.868282

In the -O0 optimized code, the peak performance achieved was the ikj implementation with a block size of 64 at 3.92213 GFLOPS. Again, GPU performance of large matrix multiplication greatly exceeds CPU performance. With matrices of size 2048x2048, the shared memory implementation outperforms the best combination by a factor 92x and the cuBLASSgemm outperforms the best combination implementation by a factor of 446x!

VI: Source Code And Results

Source code is attached in tar file. Sample outputs are included in each respective folder of /cuBLASSgemm and /sharedMM. Each run of the script produces a result matrix named "c.dat" in the respective folders mentioned before. Overall, cuBLASSgemm performed better than the shared memory implementation on very large inputs, but at times tiled matrix multiplication using shared memory did outperform the function on smaller input sizes. Both methods vastly outperformed CPU matrix multiplication.