Matthew Lepis
CPE 810

10/10/2021
CUDA C/C++ Tiled Cached Matrix Multiplication

## Objective

Implement a tiled dense matrix multiplication routine (C=A*B) using shared memory in CUDA. A and B matrix elements can be randomly generated on the host. Your code should be able to handle arbitrary sizes for input rectangular matrices.

## Instructions

Make a new folder for your project. Copy and modify the code of matrixMul in CUDA sample code to include the following key functions:

1. Allocate device memory
2. Copy host memory to device
3. Initialize thread block and kernel grid dimensions
4. Invoke CUDA kernel
5. Copy results from device to host
6. Deallocate device memory
7. Implement the matrix-matrix multiplication routine using shared memory and tiling algorithm
8. Handle boundary conditions (thread divergence) when dealing with arbitrary matrix sizes

---

## Preface

My implementation was written inspired from the CUDA matrixMul sample, as well as from course-external sources of which will be cited explicitly for reference.

### Code Execution Guide

My code can be ran via two methods. Command Line argument following:

```
./myMatrixMul <rowDimA>  <colDimA>  <colDimB>
```

Or via console user prompt. To access this, exclude command line arguments when running the executable. The prompts will guide your operation.

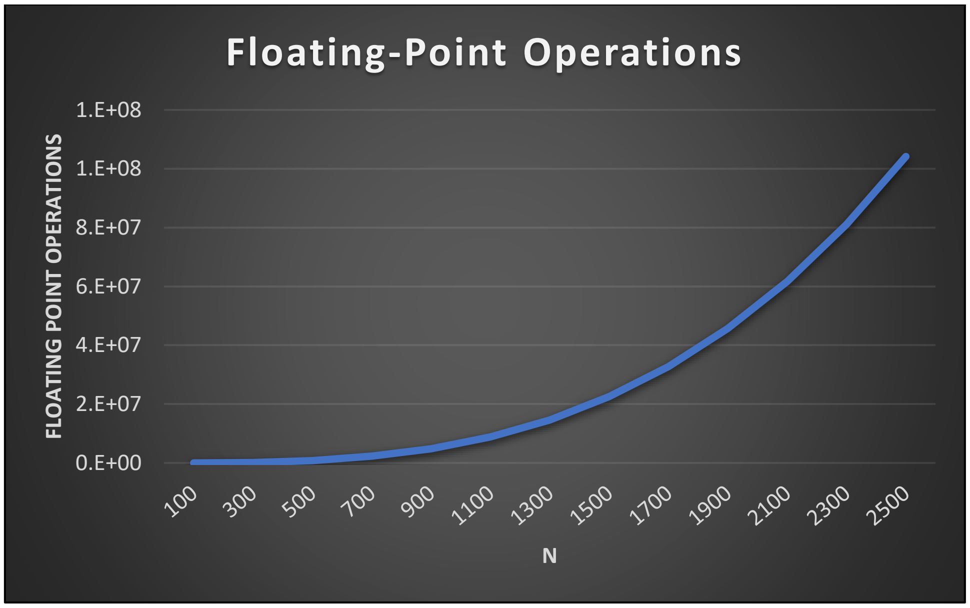
I have included an instruction checklist which allows for quick indexing of my code to ensure I followed the instructions outlined on the previous page.

**Instruction Checklist**

| | Line of Occurrence | Function/Kernel/Definition |
|---|---|---|
| 1 | 122 | cudaMalloc(&a_D, a_Bytes), … |
| 2 | 132 | cudaMemcpy(a_D, a_H, a_Bytes, cudaMemcpyHostToDevice), … |
| 3 | 136 | dim3 threads, dim3 grid |
| 4 | 147 | matrixMultiplyShared <<<grid, threads, 0, stream>>> (a_D, b_D, c_D, dimsA, dimsB, dimsC) |
| 5 | 175 | cudaMemcpy(c_H, c_D, c_Bytes, cudaMemcpyDeviceToHost) |
| 6 | 184 | cudaFree(a_D), … |
| 7 | 20 | __global__ void matrixMultiplyShared(float* a, float* b, float* c, dim3 dimsA, dim3 dimsB, dim3 dimsC) |
| 8 | 139 | blocks_x = (dimsB.x + threads.x - 1) / threads.x, … |

## Questions

(1) How many floating operations are being performed in your dense matrix multiply kernel if the matrix size is N times N? Explain.



[1]

After gathering performance metrics like GFLOPS and the execution time, calculating the total floating-point operations is rather straight forward.

$$Total\ FP\ Ops = \frac{(GFLOPS * 10^9) * Execution\ Time[s]}{300}$$

[1] Find the table of data for this graph in the appendix.

I divide by 300 here, because my code implementation runs the kernel 300 times to get a larger sample time of the operation. Therefore, dividing the number of GFLOPS found by my metrics would yield the GLOPS count for 1 kernel call.

---

(2) How many global memory reads are being performed by your kernel? Explain.

Looking at my kernel implementation:

"

```
__global__ void matrixMultiplyShared(float* a, float* b, float* c, dim3 dimsA, dim3
dimsB, dim3 dimsC)
{
	__shared__ float sharedA[TILE_WIDTH][TILE_WIDTH];
	__shared__ float sharedB[TILE_WIDTH][TILE_WIDTH];
	int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	int Row = by * TILE_WIDTH + ty;
	int Col = bx * TILE_WIDTH + tx;
	float Cvalue = 0.0;

	if (dimsA.x != dimsB.y) return;
	for (int i = 0; i < (int)(ceil((float)dimsA.x / TILE_WIDTH)); i++)
	{

		if (i * TILE_WIDTH + tx < dimsA.x && Row < dimsA.y)
			sharedA[ty][tx] = a[Row * dimsA.x + i * TILE_WIDTH + tx];
		else
			sharedA[ty][tx] = 0.0;


		if (i * TILE_WIDTH + ty < dimsB.y && Col < dimsB.x)
			sharedB[ty][tx] = b[(i * TILE_WIDTH + ty) * dimsB.x + Col];
		else
			sharedB[ty][tx] = 0.0;

		__syncthreads();

		if (Row < dimsA.y && Col < dimsB.x)
			for (int j = 0; j < TILE_WIDTH; j++)
				Cvalue += sharedA[ty][j] * sharedB[j][tx];

		__syncthreads();
	}

	if (Row < dimsC.y && Col < dimsC.x)
		c[Row * dimsC.x + Col] = Cvalue;
}
```

"

The lines that are colored red indicate a global memory read. This is true because the kernel is passed the memory address of the matrices, and when indexing these pointers we find the data stored at those locations. When assigning the shared memory locations with values from the original matrix pointers, we are reading in global memory to shared memory.

Since every thread is executing this code (the kernel code), we must multiple our analysis by the thread count. Digging deeper in, we encounter a for loop, we must multiple our analysis by each iteration. Furthermore, we must account for each matrix access, specifically, both the global memory access of matrix A and matrix B. The number of global memory reads is highly dependent on the size of each of the matrices. As such I will not explicitly state the amount. Another way to find this value could be to keep a count with an atomic operation and iterate that count every time you would transfer memory into shared memory. I had issue doing so and have not implemented this strategy due to time constraints.

(3) How many global memory writes are being performed by your kernel? Explain.

Please refer to my answer for the previous question for the code sippet. Specifically, the blue colored text. This text shows the code which define global memory writes. This kernel at that line is writing the 'c', matrix C, a float pointer. Every index assigned to is a global memory write. So this analysis would be thread count times time the amount of times that final 'if' statement is computed true. This happens to be equal to the number of elements found in matrix C This analysis is also highly dependent on the size of the matrices. My explanation for no explicit answer is as found in question (2).

(4) Describe what further optimization can be implemented to your kernel to achieve a performance speedup.

Sending the original A and B matrices to device constant memory (read-only memory), and only reading that data from there, instead of from standard global memory.

(5) Suppose you have matrices with dimensions bigger than the max thread dimensions allowed in CUDA. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in the case.

I would suggest that the algorithm in place already does this. This is why we are utilizing more than one block. They way I am interpreting this question is that "max thread dimension" is referring to the 1024 thread limit per block. And as we are using multiple blocks for our tiled approach, we are technically surpassing this max thread limit.

(6) Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

You could accomplish this by segmenting the matrices' memory into addressable chunks then calling a modified kernel for each segment. You must write each kernels' result into host memory, for the later purpose of piecing the parts together in the end. Each kernel must be called in an arbitrary serial order, as the memory capacity would be fully utilized by one kernel at a time. Unless you have multiple GPUs, in which case you could parallelize this operation as well as a higher tier of parallelization.

## Experimentation

For experimentation, I gathered a set of 8 different matrix configurations.

| | A | | B | |
|---|---|---|---|---|
| Matrix Config. | Row | Col | Row | Col |
| 1 | 1000 | 500 | 500 | 1000 |
| 2 | 500 | 1000 | 1000 | 500 |
| 3 | 1250 | 400 | 400 | 1250 |
| 4 | 400 | 1250 | 1250 | 400 |
| 5 | 2000 | 250 | 250 | 2000 |
| 6 | 250 | 2000 | 2000 | 250 |
| 7 | 625 | 800 | 800 | 625 |
| 8 | 800 | 625 | 625 | 800 |

These matrix configurations were then input into my program with varying thread tile width. Meaning each tile was TILE_WIDTH * TILE_WIDTH in size. I gauged the performance of each configuration by measure the GFLOPS and the execution time. The data is as on the next page.

| TILE_WIDTH | rowDimA | colDimA | rowDimB | colDimB | GFLOPS | Time(ms) |
|---|---|---|---|---|---|---|
| 32 | 1000 | 500 | 500 | 1000 | 1090.8 | 0.917 |
| 32 | 500 | 1000 | 1000 | 500 | 839.21 | 0.596 |
| 32 | 1250 | 400 | 400 | 1250 | 1113.02 | 1.123 |
| 32 | 400 | 1250 | 1250 | 400 | 822.11 | 0.487 |
| 32 | 2000 | 250 | 250 | 2000 | 1167.71 | 1.713 |
| 32 | 250 | 2000 | 2000 | 250 | 650.11 | 0.385 |
| 32 | 625 | 800 | 800 | 625 | 904.08 | 0.691 |
| 32 | 800 | 625 | 625 | 800 | 1007.31 | 0.794 |
| 16 | 1000 | 500 | 500 | 1000 | 958.94 | 1.043 |
| 16 | 500 | 1000 | 1000 | 500 | 779.76 | 0.641 |
| 16 | 1250 | 400 | 400 | 1250 | 1107.81 | 1.128 |
| 16 | 400 | 1250 | 1250 | 400 | 838.21 | 0.477 |
| 16 | 2000 | 250 | 250 | 2000 | 1155.43 | 1.731 |
| 16 | 250 | 2000 | 2000 | 250 | 763.96 | 0.327 |
| 16 | 625 | 800 | 800 | 625 | 905.74 | 0.69 |
| 16 | 800 | 625 | 625 | 800 | 990.32 | 0.808 |
| 8 | 1000 | 500 | 500 | 1000 | 690.88 | 1.447 |
| 8 | 500 | 1000 | 1000 | 500 | 706.81 | 0.707 |
| 8 | 1250 | 400 | 400 | 1250 | 731.25 | 1.709 |
| 8 | 400 | 1250 | 1250 | 400 | 553.04 | 0.723 |
| 8 | 2000 | 250 | 250 | 2000 | 750.94 | 2.663 |
| 8 | 250 | 2000 | 2000 | 250 | 543.06 | 0.46 |
| 8 | 625 | 800 | 800 | 625 | 657.36 | 0.951 |
| 8 | 800 | 625 | 625 | 800 | 643.35 | 1.243 |

I will sort this data into a few orders of interest. Specifically, (Chart 1): Greatest to Least GFLOPS, and (Chart 2): Least to Greatest Time.

Something to note: the performance of this code is heavily skewed by GPU Boost. In the future, I will seek a way to turn this feature off to obtain more representative data.

CHART 1: Greatest to Least GFLOPS

| TILE_WIDTH | rowDimA | colDimA | rowDimB | colDimB | GFLOPS | time(msec) |
|---|---|---|---|---|---|---|
| 32 | 2000 | 250 | 250 | 2000 | 1167.71 | 1.713 |
| 16 | 2000 | 250 | 250 | 2000 | 1155.43 | 1.731 |
| 32 | 1250 | 400 | 400 | 1250 | 1113.02 | 1.123 |
| 16 | 1250 | 400 | 400 | 1250 | 1107.81 | 1.128 |
| 32 | 1000 | 500 | 500 | 1000 | 1090.8 | 0.917 |
| 32 | 800 | 625 | 625 | 800 | 1007.31 | 0.794 |
| 16 | 800 | 625 | 625 | 800 | 990.32 | 0.808 |
| 16 | 1000 | 500 | 500 | 1000 | 958.94 | 1.043 |
| 16 | 625 | 800 | 800 | 625 | 905.74 | 0.69 |
| 32 | 625 | 800 | 800 | 625 | 904.08 | 0.691 |
| 32 | 500 | 1000 | 1000 | 500 | 839.21 | 0.596 |
| 16 | 400 | 1250 | 1250 | 400 | 838.21 | 0.477 |
| 32 | 400 | 1250 | 1250 | 400 | 822.11 | 0.487 |
| 16 | 500 | 1000 | 1000 | 500 | 779.76 | 0.641 |
| 16 | 250 | 2000 | 2000 | 250 | 763.96 | 0.327 |
| 8 | 2000 | 250 | 250 | 2000 | 750.94 | 2.663 |
| 8 | 1250 | 400 | 400 | 1250 | 731.25 | 1.709 |
| 8 | 500 | 1000 | 1000 | 500 | 706.81 | 0.707 |
| 8 | 1000 | 500 | 500 | 1000 | 690.88 | 1.447 |
| 8 | 625 | 800 | 800 | 625 | 657.36 | 0.951 |
| 32 | 250 | 2000 | 2000 | 250 | 650.11 | 0.385 |
| 8 | 800 | 625 | 625 | 800 | 643.35 | 1.243 |
| 8 | 400 | 1250 | 1250 | 400 | 553.04 | 0.723 |
| 8 | 250 | 2000 | 2000 | 250 | 543.06 | 0.46 |

GFLOPS. Here, it is clear that a tile width of 8 is not great for pushing floating point operations, they overall have a much lower value when viewing the chart.

CHART 2: Least to Greatest Execution Time

| TILE_WIDTH | rowDimA | colDimA | rowDimB | colDimB | GFLOPS | time(msec) |
|---|---|---|---|---|---|---|
| 16 | 250 | 2000 | 2000 | 250 | 763.96 | 0.327 |
| 32 | 250 | 2000 | 2000 | 250 | 650.11 | 0.385 |
| 8 | 250 | 2000 | 2000 | 250 | 543.06 | 0.46 |
| 16 | 400 | 1250 | 1250 | 400 | 838.21 | 0.477 |
| 32 | 400 | 1250 | 1250 | 400 | 822.11 | 0.487 |
| 32 | 500 | 1000 | 1000 | 500 | 839.21 | 0.596 |
| 16 | 500 | 1000 | 1000 | 500 | 779.76 | 0.641 |
| 16 | 625 | 800 | 800 | 625 | 905.74 | 0.69 |
| 32 | 625 | 800 | 800 | 625 | 904.08 | 0.691 |
| 8 | 500 | 1000 | 1000 | 500 | 706.81 | 0.707 |
| 8 | 400 | 1250 | 1250 | 400 | 553.04 | 0.723 |
| 32 | 800 | 625 | 625 | 800 | 1007.31 | 0.794 |
| 16 | 800 | 625 | 625 | 800 | 990.32 | 0.808 |
| 32 | 1000 | 500 | 500 | 1000 | 1090.8 | 0.917 |
| 8 | 625 | 800 | 800 | 625 | 657.36 | 0.951 |
| 16 | 1000 | 500 | 500 | 1000 | 958.94 | 1.043 |
| 32 | 1250 | 400 | 400 | 1250 | 1113.02 | 1.123 |
| 16 | 1250 | 400 | 400 | 1250 | 1107.81 | 1.128 |
| 8 | 800 | 625 | 625 | 800 | 643.35 | 1.243 |
| 8 | 1000 | 500 | 500 | 1000 | 690.88 | 1.447 |
| 8 | 1250 | 400 | 400 | 1250 | 731.25 | 1.709 |
| 32 | 2000 | 250 | 250 | 2000 | 1167.71 | 1.713 |
| 16 | 2000 | 250 | 250 | 2000 | 1155.43 | 1.731 |
| 8 | 2000 | 250 | 250 | 2000 | 750.94 | 2.663 |

Execution Time. With 4 values in the up third of values, 16 tile width seems to be the fastest execution on average. With 32 tile width follow in at 3 values.

## Conclusion

Finally, creating my own tiled cache matrix multiplication implementation has greatly increase my understand of CUDA. I have a solid understand of the basic paradigm when it comes to allocation and transferring of memory. I need more work on understand parallelization of algorithms, however. The experiments I ran, gave me insight on how exactly changing the tile width of my blocks is helpful for execution as well.

## Appendix

Question 1’s Graph data points.

| N | Floating Point Operations |
|---|---|
| 100 | 7.E+03 |
| 300 | 2.E+05 |
| 500 | 8.E+05 |
| 700 | 2.E+06 |
| 900 | 5.E+06 |
| 1100 | 9.E+06 |
| 1300 | 1.E+07 |
| 1500 | 2.E+07 |
| 1700 | 3.E+07 |
| 1900 | 5.E+07 |
| 2100 | 6.E+07 |
| 2300 | 8.E+07 |
| 2500 | 1.E+08 |