CPE810 – GPU And Multicore Programming

Lab Report #1: Homework 1

Tiled Based Matrix Multiplication

Instructor: Zhou Feng

TA: Ali Aghdaei

James Fong

I pledge my honor that I have abided by the Stevens Honor System

Table of Contents

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.

Required Functions and Implementation Details

1. Allocate Device Memory

```
//Allocating Device Memory.

unsigned int Size_A = (dimsA.y * dimsA.x);
unsigned int Size_B = (dimsB.y * dimsB.x);
unsigned int Size_C = (dimsC.y * dimsC.x);

unsigned int mem_size_A = Size_A * sizeof(float);
unsigned int mem_size_B = Size_B * sizeof(float);
unsigned int mem_size_C = Size_C * sizeof(float);

//Allocating HOST memory for the matrices.
float* h_A, * h_B, * h_C;

//Matrix A
cudaMallocHost((void**)&h_A, mem_size_A);
checkCudaErrors(cudaMallocHost((void**)&h_A, mem_size_A));

//Matrix B
cudaMallocHost((void**)&h_B, mem_size_B);
checkCudaErrors(cudaMallocHost((void**)&h_B, mem_size_B));

//Matrix C
cudaMallocHost((void**)&h_C, mem_size_C);
checkCudaErrors(cudaMallocHost((void**)&h_C, mem_size_C));

// initialize the matrixes
initializeMatrix(h_A, Size_A);
initializeMatrix(h_B, Size_B);

//Allocating DEVICE memory for the matrices.
float* a_D, * b_D, * c_D;
cudaMalloc(&a_D, mem_size_A);
cudaMalloc(&b_D, mem_size_B);

cudaMalloc(&c_D, mem_size_C);
```

2. Copy Host Memory to Device

```
cudaMemcpy(a_D, h_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(b_D, h_B, mem_size_B, cudaMemcpyHostToDevice);
```

3. Initialize Thread Block and Kernel Grid Dimensions

```
dim3 threads(TILE_WIDTH, TILE_WIDTH);
dim3 grid((dimsB.x + threads.x - 1) / threads.x, (dimsA.y + threads.y - 1) /
threads.y);
```

4. Invoke CUDA Kernel

```
    CUDAMatrixMultiply <<<grid, threads>>> (a_D, b_D, c_D, dimsA.y, dimsA.x, dimsB.y,
dimsB.x, dimsC.y, dimsC.x);
```

5. Copy Results from Device to Host

```
    cudaMemcpy(h_C, c_D, mem_size_C, cudaMemcpyDeviceToHost);
```

6. Deallocate Device Memory

```
    cudaFreeHost(h_A);
    cudaFreeHost(h_B);
    cudaFreeHost(h_C);
    cudaFree(a_D);
    cudaFree(b_D);
    cudaFree(c_D);
```

7. Implement The Matrix-Matrix Multiplication Routine Using Shared Memory and Tiling Algorithm

```
    dim3 dimsA;
    dim3 dimsB;
    dim3 dimsC;

    if (argc == 4) {
        assert(atoi(argv[1]) <= 0);
        dimsA.y = atoi(argv[1]);

        assert(atoi(argv[2]) <= 0);
        dimsA.x = atoi(argv[2]);

        assert(atoi(argv[3]) <= 0);
        dimsB.x = atoi(argv[3]);

        cout << "Command Lines accepted" << endl;
    }

    else {

        cout << "Enter row dimensions for matrix A: " << endl;
        cin >> dimsA.y;

        cout << "Enter column dimensions for matrix A: " << endl;
        cin >> dimsA.x;

        cout << "Enter row dimensions for matrix B: " << endl;
        cin >> dimsB.x;
    }

    dimsB.y = dimsA.x;
    dimsC.y = dimsA.y;
    dimsC.x = dimsB.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.

Using the following table below, I recorded the total amount of operations for using Tiled Dense Matrix Multiplication and graphed it on a chart on Graph 1.

The data used for the table is also listed on Table 1.

Given that we are working on Matrix-Matrix Product multiplication, it requires M N L multiplications, where M N L are derived from vectors c ∈ C^M, a ∈ C^N , b ∈ C^N, and where L is a lower triangular N × N matrix. Since $1 \leq i \leq L$, the matrix-matrix product has L fold complexity of a Matrix-Vector product, representing the following equation for FLOP calculation: 2*M*N*L − ML FLOPs. [1] [2]

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

Global memory reads depend on the size of the matrices being multiplied. This can be measured are being performed in the kernel using the operations:

```
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];
```

The operations are being done within a for loop, which is listed below with the following operation:

```
for (int k = 0; k < BLOCK_SIZE; ++k) {
        Csub += As[ty][k] * Bs[k][tx];
```

My personal machine is a Lenovo Legion Y540-15, which runs the GPU NVIDIA GeForce GTX 1660 Ti Mobile. This GPU has a memory bandwidth of 288.0 GB/s [3]. Due to using single-precision floating point value, the maximum loadable amount of operands is 72 GB/s.

There are two global memory accesses being performed; once for floating point multiplication and one floating point addition; fetching the A[] and B[] operation respectively; one operation multiplies the two elements, and the second operation accumulates the product for the P value. By the CGMA ratio, the ratio is 2:2, or 1.0. In addition, given that we are performing matrix multiplication, the global memory accesses are reduced by the factor of TILE_WIDTH, which is given a factor of 32 tiles. Thus, the CGMA ratio is increased from 1 to 32. [4]

Given that there are 4 bytes in a single-precision floating-point value, we can load a maximum of 72 GB/s operands, and a CGMA ratio of 32, the matrix multiplication kernel of my machine has a theoretical maximum of 2,304 GFLOPS per read. [5]

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

The result of the operations being performed by the kernel are written using the following operation:

```
C[c + wB * ty + tx] = Csub;
```

This operation is being performed with one global memory access; once for floating point multiplication on C[]. The operation is written onto the C[] element using linearized index, calculated from the given Row and Columns, derived from the matrix multiplication. By the CGMA ratio, the ratio is 1:1, or 1.0. In addition, given that we are performing matrix multiplication, the global memory accesses are reduced by the factor of TILE_WIDTH, which is given a factor of 32 tiles. Thus, the CGMA ratio is increased from 1 to 32. [4]

Given that there are 4 bytes in a single-precision floating-point value, we can load a maximum of 72 GB/s operands, and a CGMA ratio of 32, the matrix multiplication kernel of my machine has a theoretical maximum of 2,304 GFLOPS per write in all matrices, assuming they clear the final if statement condition; this also is equal to the number of elements found in matrix C. [5]

4. Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

Further optimizations that can be implemented from my kernel to achieve a performance speed up is to get the original A and B matrices to be read only memory, instead of being accessed from global memory. This way you are reducing the amount of calls done to the global memory and moving calls to the read only memory, which does not take as much 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 this case.

An algorithm that would perform matrix multiplication, that would perform the multiplication with matrices that have dimensions bigger than the maximum thread dimensions is already performed within the practice code. Given that the maximum thread dimensions within the implementation is 1024 thread limit per block, using multiple blocks for tiled dense matrix multiplication surpasses that limit by adding more blocks to work with for more than 1024 thread usage. This method uses multi-threading, which utilizes more cores in order to perform the multiplication.

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.

An algorithm that would perform matrix multiplication, that would perform the multiplication with matrices that exceed the size limit in global memory, would be to create a separate thread for each element in the resulting matrix, and avoid using calls to global variables. You can partition the data into subsets, so that the data fits into the shared memory, where the kernel computations are performed independently but done in parallel. As global memory is much slower, but much larger than shared memory, you would write the result into the host memory and later combine outputs of the data.

Experimentation

I changed the block size to be from 32 to 16, limiting the amount of operations that can happen. The listed table can be found for Table 2.

A graph for Table 2 can be found on Graph 2, with the same measurements as Graph 1.

Compared to Table 1, you can see the data listed with operations start to pick up to it's highest speed at 480 x 480 as compared to the first matrix with a block size of 32, reaching it's speed of GFLOPS at around 640 x 640. Increasing the amount of block size seems to allow for more computations at the same speed, thus it could be inferred that the higher the block size, the greater amount of computations possible within the same amount of time.

Conclusion

Going through Tiled Multiplication was incredibly difficult for myself, as I did not understand CUDA very well. By going through examples that are shown in class, extensive testing with Kerim and Matt, and constant trial and error, I better understand how CUDA functions. The implementation of using Matrix Multiplication through CUDA has extended my understanding and ability to perform calculations through the GPU, through different calls and allocations to HOST and Memory. I still need a good amount of practice but I understand the basis of how to implement certain functions.

Tables

Table 1

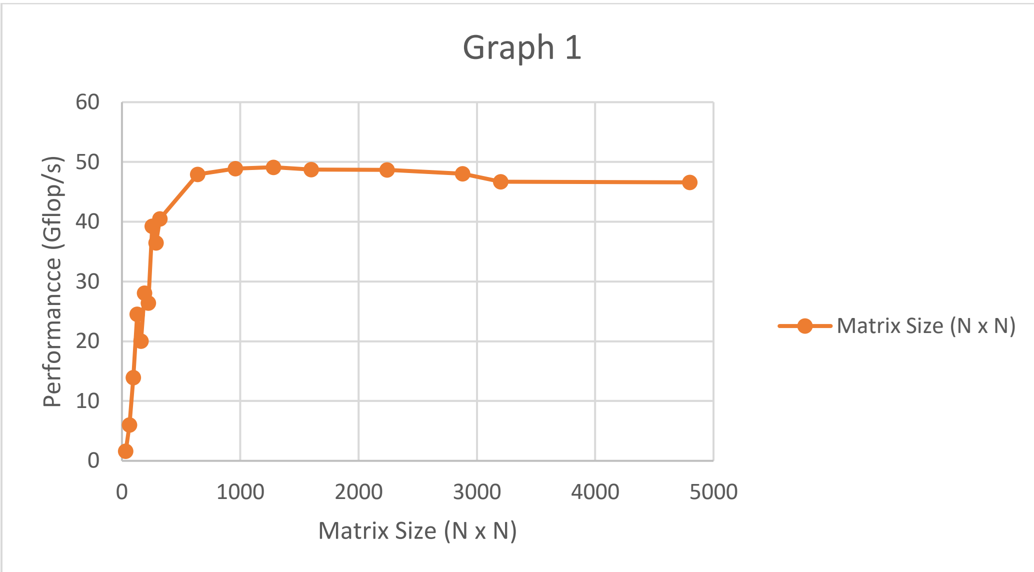
Block Size	N Multiple	Matrix Size (N x N)	Performance (Gflop / s)	Time (m/sec)	Size (Ops)
32	1	32	1.6	0.041	65536
32	2	64	6	0.087	524288
32	3	96	13.92	0.127	1769472
32	4	128	24.51	0.171	4194304
32	5	160	20.06	0.408	8192000
32	6	192	28.04	0.505	14155776
32	7	224	26.39	0.852	22478848
32	8	256	39.23	0.855	33554432
32	9	288	36.45	1.311	47775744
32	10	320	40.5	1.618	65536000
32	20	640	47.95	10.935	524588000
32	30	960	48.89	36.191	1769472000
32	40	1280	49.12	85.39	4194304000
32	50	1600	48.71	168.188	8192000000
32	70	2240	48.68	461.726	22478848000
32	90	2880	48.04	994.505	47775744000
32	100	3200	46.67	1404.152	65536000000
32	150	4800	46.58	4748.795	2.21184E+11

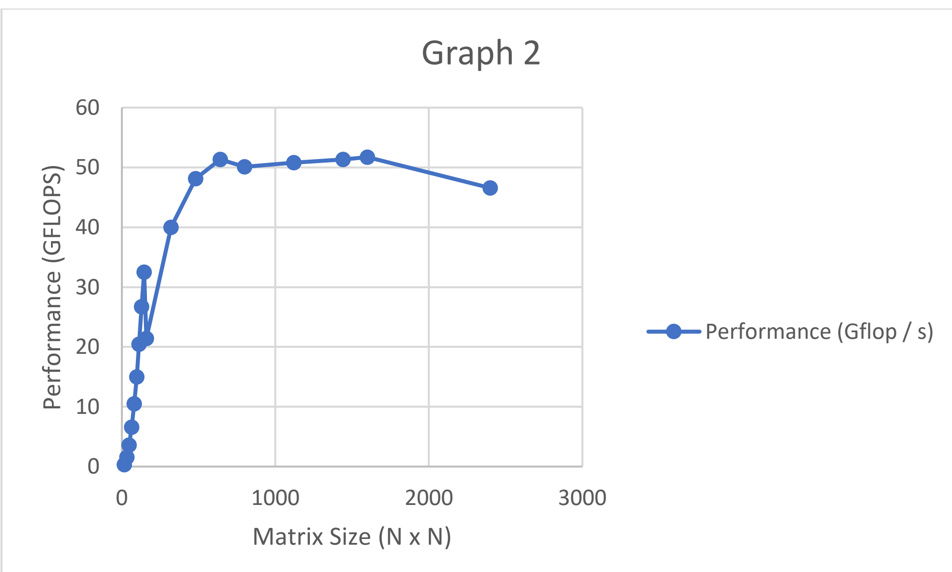
Table 2

Block Size	Matrix Size (N x N)	Performance (Gflop / s)	Time (m/sec)	Size (Ops)	Work Group Size
16	16	0.33	0.025	8192	256
16	32	1.56	0.042	65536	256
16	48	3.56	0.062	221184	256
16	64	6.6	0.079	524288	256
16	80	10.48	0.098	1024000	256
16	96	15.01	0.118	1769472	256
16	112	20.42	0.138	2809856	256
16	128	26.7	0.157	4194304	256
16	144	32.5	0.184	5971968	256
16	160	21.39	0.383	8192000	256
16	320	39.97	1.64	65536000	256
16	480	48.13	4.595	221184000	256
16	640	51.35	10.21	524288000	256
16	800	50.09	20.445	1024000000	256

Graphs

Graph 1.





References

[1]: https://hal.inria.fr/hal-03117491/document

[2]: https://www.stat.cmu.edu/~ryantibs/convexopt-F18/scribes/Lecture_19.pdf

[3]: https://www.techpowerup.com/gpu-specs/geforce-gtx-1660-ti-mobile.c3369

[4]: https://engineering.purdue.edu/~smidkiff/ece563/NVidiaGPUTeachingToolkit/Mod4/Mod4.pdf

[5]: https://www.sciencedirect.com/topics/computer-science/global-memory-access