Lab 4: CUDA Tiled Matrix Multiplication

Objective

Implement a tiled dense matrix multiplication routine using shared memory.

Instructions

In this lab, we will explore the memory hierarchy available on a GPU to optimize the matrix multiplication implementation from the previous lab. In our previous kernel implementation, each thread loads 2 x n elements from global memory, where n is the number of columns of matrix A. That is two elements for each iteration through the loop, one from matrix A and one from matrix B. Since accesses to global memory are relatively slow, this can leave the threads idle for hundreds of clock cycles for each access.

One way to reduce the number of accesses to global memory is to have the threads load portions of matrices A and B into shared memory, where we can access them much more quickly. Figure 1 shows a visual representation of this approach.

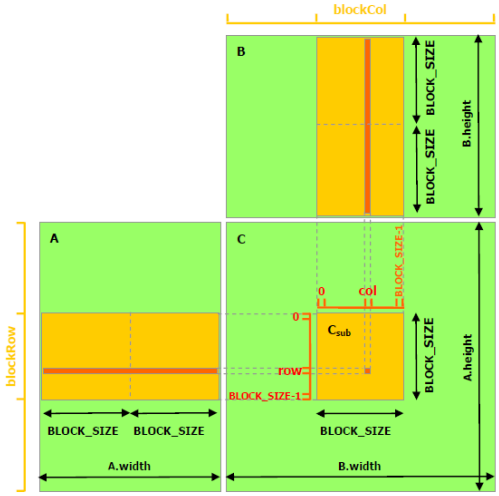

Figure 1. Multiplying two matrices using shared memory. From the NVIDIA CUDA C Programming Guide.

As in the previous lab implementation, each thread will be responsible for computing one element of the product matrix C. However, now each thread computes its corresponding element in C by multiplying together the red row shown in A and the red column shown in B in portions (Figure 1).

Pseudocode steps:

- Each thread block computes one sub-matrix C_{sub} of C.

- Each thread computes one element of C_{sub} accumulating results of tile inner products into a temporary variable.
- Loop over all the sub-matrices of A and B required to compute C_{sub} – these correspond to the tile phases discussed in lectures.
    - Load A_{sub} and B_{sub} from device memory to shared memory using all threads in coordination. Each thread loads one element of each sub-matrix.
    - Multiply A_{sub} and B_{sub} together. Each thread computes the corresponding element of C_{sub} by accumulating partial inner products into a temporary variable, which continues to be updated until all the tiles have been processed.
- Each thread writes one element of C_{sub} to memory.

In your kernel make use of the `__syncthreads()` call after each tile has been loaded and after the inner products of rows and columns of the loaded tiles have been computed. You can reuse your CUDA kernel for regular matrix-matrix multiplication from lab 3 in this lab. However, the device matrices A and B should be initialized using only one kernel. Initialize each element of A and B with `1`. Since the matrices are initialized in device memory then you should not copy data from host to device. Allow the user to provide the dimensions of matrices as input from the command line. If your executable is named `lab4` then the user should be able to run the program from the command line as follows:

```
./lab4 m n k,
```

where `m` is the number of rows of matrix A, `n` is the number of columns of matrix A, and `k` is the number of columns of matrix B. Sum up the elements of the host matrix C and print the result divided by `m * k`. The output should be `n`. Your implementation should work for matrices of any size, i.e. not only for square matrices. Both the kernels with and without shared memory should be used to compute the matrix multiplications. After the first multiplication kernel finishes computing and the data are copied to the host then reset the elements of the device matrix C to 0 (using `cudaMemset ( void* devPtr, int value, size_t count )`) before invoking the second multiplication kernel.

Benchmarking

Time both your GPU matrix multiplication implementations **with** and **without** shared memory with matrices of size `8192`, `16384`, and `32768`. Use square matrices, i.e. for size 8192, the command line input should be 8192, 8192, 8192. Using your favorite plotting tool, plot the runtimes in milliseconds. Your plot should have matrix size in the x-axis and runtimes for both the without and with shared memory implementations on the y-axis. Also, record the speedups obtained, computed as ratios of `(without shared memory) / (with shared memory)` execution times. Run your benchmarks multiple times and report average runtimes.

In your lab report, include a screenshot containing the output of correctness check and timing in milliseconds for both the CUDA implementations with and without shared memory for

command-line input of 32768, 16384, 8192 (matrix A dimensions: 32768x16384, matrix B dimensions: 16384x8192).

Questions

Include answers to the following questions in the lab report.

1. How many floating operations are being performed in the matrix multiply kernel that uses shared memory? Explain.
2. How many global memory reads are being performed by the kernel that uses shared memory? Explain.
3. How many global memory writes are being performed by the kernel that uses shared memory? Explain.
4. Describe what further optimizations can be implemented to the kernel that uses shared memory to achieve a performance speedup.
5. Compare the implementation difficulty of the kernel that uses shared memory compared to the kernel without shared memory. What difficulties did you have with this implementation?
6. Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform the multiplication in this case.
7. Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform the multiplication out of place.

Submission Instructions

You should upload a PDF of the answers to the above questions, plot of runtimes, the source code, screenshots of running your implementations from the terminal, and the binary executable to Canvas as part of a zipped folder named as **your_msoeid-lab4.zip**.

Rubric

GPU implementation **_____ / 80**

correct result (code compiles and executes)	_____ / 20
implementation / correct coding	_____ / 50
profiling / timing results	_____ / 10

Documentation **_____ / 10**

Questions **_____ / 10**