Lab 3: Basic Matrix Multiplication

Objective

Implement a basic dense matrix multiplication routine. This lab introduces multi-dimensional thread blocks. Optimizations such as tiling and usage of shared memory are not required for this lab.

Instructions

Let `A` and `B` be two matrices. Assume that `A` is an `m × n` matrix, which means that it has `m` rows and `n` columns. Assume that `B` is an `n × k` matrix. Let the result of the multiplication `A * B` be matrix `C` of size `m × k`. That is, the number of rows in the resulting matrix equals the number of rows of the first matrix `A` and the number of columns of the second matrix `B`.

For example, the value of the entry $c_{1,1}$ of `C` after the operation `C= A * B` is the sum of all the element-wise multiplications of the entries of row `1` of `A` with the values in column `1` of `B`. In general, the $c_{i,j}$ entry of `C` will be the result of the sum of the element-wise multiplication of all the values in the `i`-th row in `A` and the `j`-th column in `B` or the inner/dot product of the `i`-th row of `A` and `j`-th column of `B`. . Figure 1 shows a visual illustration of the matrix-matrix multiplication operation.

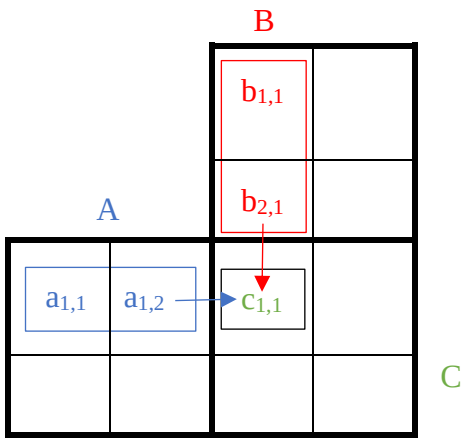

Figure 1. A visual representation of matrix multiplication.

Matrix-matrix multiplication is a good example of parallel computation. We must compute every element in `C`, and each of the computations are independent of the others, so we can efficiently parallelize. We will see different ways of achieving this. The goal is to add new concepts throughout this lab, ending up with a 2D kernel. In the next lab, we will use shared memory to efficiently optimize operations.

CPU Implementation

Implement a function that performs matrix-matrix multiplication on the CPU. The matrix multiplication function should multiply two matrices `A` and `B` to produce `C`. Allow the user to provide the size of matrices as input from the command line. If your executable is named `lab3` then the user should be able to run the program from the command line as follows:

`./lab3 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`. Initialize each element of `A` and `B` with `1`. Then multiply `A` and `B` and save the result into the matrix `C`. Sum up the elements of `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. Include a screenshot of the results of your CPU implementation (from the terminal) in your lab report. Use `5000`, `4000`, `3000` for the dimensions of matrices.

GPU Implementation

First, reset the contents of the host matrix C to be all `0`s. Print the sum of the entries of the host matrix `C` to confirm that its elements have been reset to `0`. Then convert your CPU matrix multiplication function to a CUDA implementation. Do not remove your CPU function implementation. The same correctness check mentioned above can be used to check if your implementation is correct after transferring the data from the device matrix `C` to the host matrix `C`. Note that the CUDA kernel should just multiply `A` and `B` and save the results on the device matrix `C`, i.e., the check for correctness should be performed after you have transferred the device matrix `C` to the host. Your implementation should work for matrices of any size, i.e. not only for square matrices. Include a screenshot of the results of your CUDA implementation (from the terminal) in your lab report. Use `5000`, `4000`, `3000` for the dimensions of matrices.

Now, implement two kernels for initializing the device matrices `A` and `B` (you can save this implementation in another .cu file). You should allocate device memory for `A`, `B`, and `C` but you shouldn't copy the data from the host to the device. Instead, the multiplication kernel should use the device data that were populated by the two initialization kernels.

Benchmarking

Regular CPU and GPU implementations: Time the CPU matrix multiplication function and the CUDA kernel matrix multiplication with matrices of size `10`, `100`, `1000`, `2000`, `3000`, `4000`, `5000`. Using your favorite plotting tool, plot the run times in milliseconds. Your plot should have matrix size in the x-axis (use squared matrices here) and run times for both CPU and GPU implementations on the y-axis. Also, record the speed-ups obtained, computed as ratios of CPU / GPU execution times. Run your benchmarks multiple times and report average run times.

Implementation with multiple kernels: Record the runtime of the CPU implementation including the memory allocations, matrix initializations, and the multiplication function invocation. Record the runtime of the CUDA implementation including the memory allocations,

matrix initializations (i.e. the invocation of the two initialization kernels), the multiplication kernel invocation, and the copying of data from the device to host. Also, record the speed-ups obtained. Plot the runtimes and speed-ups for matrices of size `5000`, `6000`, `7000`.

Questions

Answer the following questions.

1. How many floating operations are being performed in your matrix multiply kernel? Explain.
2. How many global memory reads are being performed by your kernel? Explain.
3. How many global memory writes are being performed by your kernel? Explain.
4. Describe what possible optimizations can be implemented to your kernel to achieve a performance speed-up.
5. Name three applications of matrix multiplication.

Submission Instructions

Upload a PDF of the answers to the above questions, plot of runtimes, the source code, and the binary executable to Canvas as part of a zipped folder named your_**msoeid-lab3.zip**.

Rubric

CPU implementation **_____ / 30**

correct result (code compiles and executes)	_____ / 10
implementation / correct coding	_____ / 15
profiling/timing results	_____ / 5

GPU implementation **_____ / 50**

correct result (code compiles and executes)	_____ / 10
implementation / correct coding	_____ / 35
profiling / timing results	_____ / 5

Documentation **_____ / 10**

Questions **_____ / 10**