Project 3

MATH 471/571 - Parallel Scientific Computing

Michal A. Kopera

First revision date: April 9th, 2021 (will give you feedback if you submit by then)

Final submission: April 21st, 2021 (no more redo's after that date - ideally you would complete before spring break - you need time to work on the final project after spring break)

Please complete the tasks outlined below and submit a PDF write-up along with the code and other files (Makefile, run script) you have used to obtain the results. You submit your solution by putting it in a folder named Project 2 in your Google Drive personal folder.

I will grade your project and provide you with some feedback if you submit by the first revision date. Please aim to complete everything by then. If you need more time, I will give you till after spring break to complete your project, but you will need time to work on the Final Project after spring break.

Please refer to the Syllabus regarding how the project points and mastery points affect your final grade. You can earn one project point for each complete task, and one mastery point for completing the mastery part. There is no partial credit, so you have to complete all the steps to earn a point.

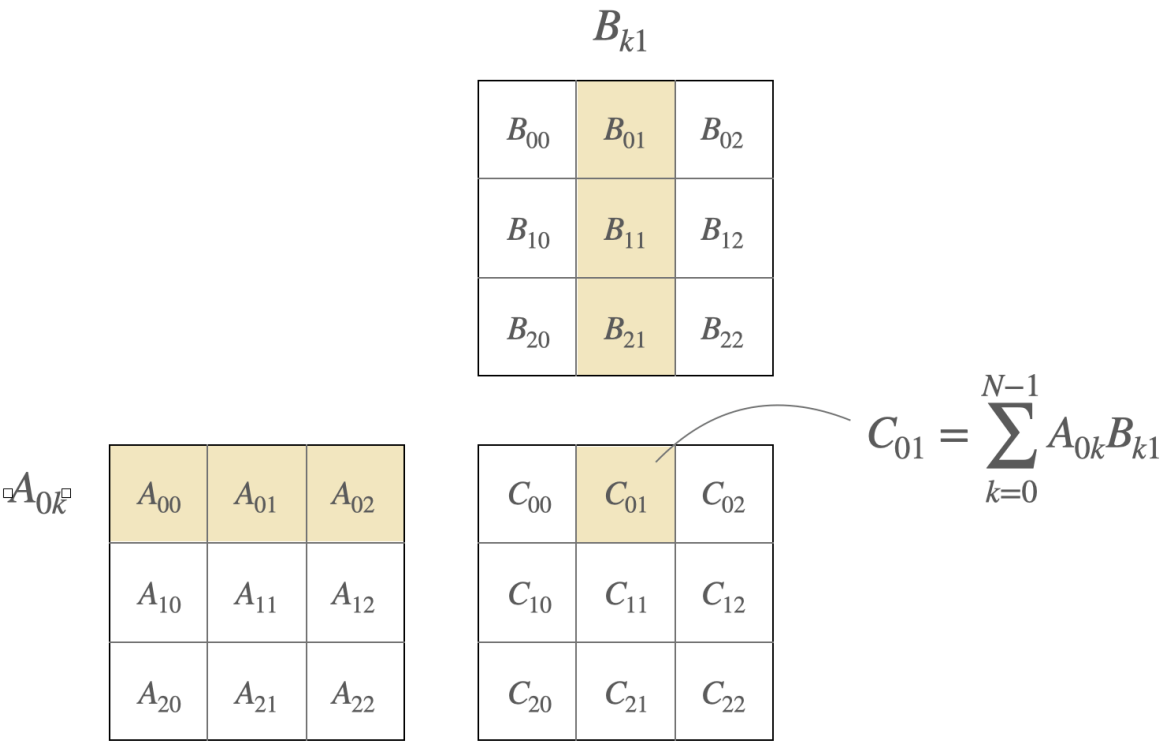
The projects are your individual work. I am happy to clarify some muddy points, but I would like to see your individual attempt to solve the project. If you got help from somebody, or used a resource outside class (i.e. website) please provide appropriate acknowledgement and/or reference. **You can** discuss your ideas on Slack, but **you cannot** share your codes.

Problem description

In this project you will explore different ways to compute matrix multiplication on a GPU using CUDA. Just to remind you, the multiplication of two matrices A and B (we assume they are square) is another square matrix C, such that:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

where N is the matrix size (i.e. $A, B, C \in R^{N \times N}$), and the first index in the subscript enumerates the row, and the second index the column of a matrix. This can be represented graphically as:



This process yields naturally to parallelization, as each C_{ij} is independent of another entries of that array. It also offers a few possibilities of implementation with shared memory parallelism.

Task 1 - Monolithic approach

The simplest approach to the implementation of the matrix matrix multiplication kernel in CUDA is the so-called monolithic approach, where we create one block of many threads, and each thread will compute one entry C_{ij}. We are limited, however, to only 1024 threads, which means our matrix cannot be larger than 32×32.

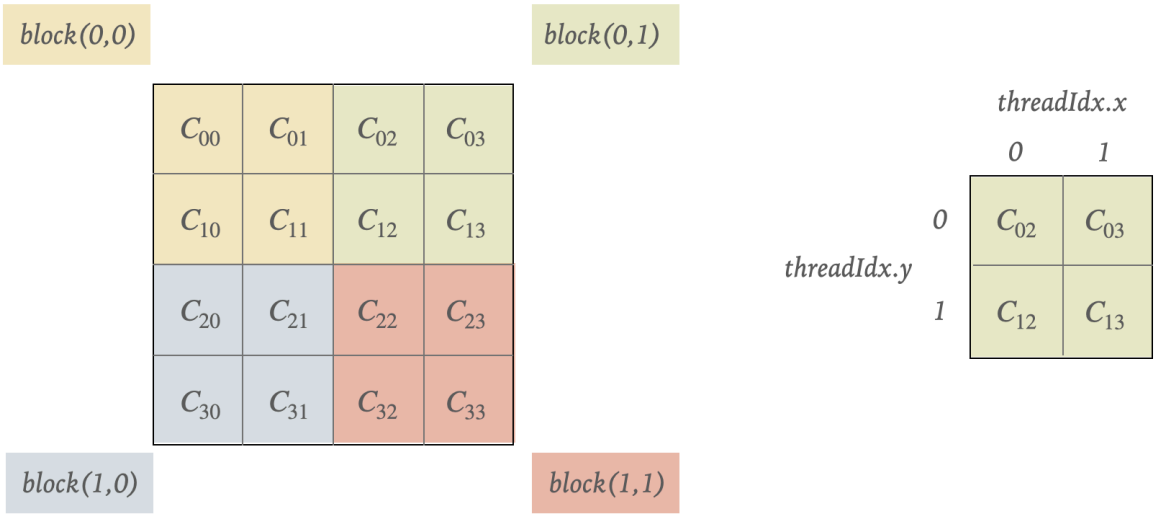
Complete the following steps:

1. Use the provided serial code for matrix-matrix multiplication (matmul.cu - cu extension just to make your life easier with compilation, etc) and create CUDA code which will follow the CUDA programing model (allocate arrays on the device, move data to the device, execute kernel, copy result to host, free memory on the device).

2. Use problem size $N = 32$ and one block of 1024 threads to run your kernel. Check whether your CUDA kernel produce the same results as the serial code.
3. Report what happens when you use larger problem size than $N = 32$ with one block and 1024 threads, and what happens when you increase the number of threads to match the new problem size (i.e. for $N = 40$ you would need $40 \times 40 = 1600$ threads).
4. Time the entire CUDA code (including memory transfers) and compare with the serial code. Does the device acceleration give you any benefit for a single small matrix multiplication?

Task 2 - 2D grid of 2D blocks decomposition

In class we discussed the organization of blocks in a 2D grid, and threads within a block in a 2D structure as well. Use that concept to implement a kernel taking advantage of that decomposition. Consider this simple example of a 4×4 matrix:



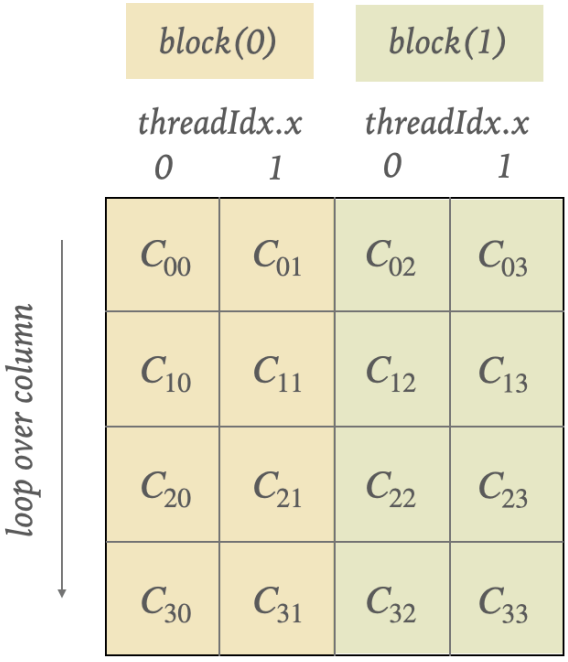
Here block form a 2D grid, and each block computes a tile of the matrix C. Inside each block, the threads are also organized in 2D grid, such that one thread computes only one entry of the matrix C. Mind that to compute C_{ij} each thread will have to access the entire row i of matrix A, and the entire column j of matrix B. Since we are using global memory, this won't be a problem, since all threads have access to all memory locations.

Complete the following steps:

1. Implement the 2D-2D matrix-multiplication algorithm in CUDA and check whether it gives you the same result as the serial code.
2. Use large N (i.e. $N = 2^{11}$ - the serial code should run for about a minute - for larger exponents it will be significantly longer) and time the kernel only (no memory transfers). Select a few block configurations (i.e. 32x32 threads, 16x16, 8x8, 4x4, 1x1) and time each run, making sure that each time you get a correct result. Present the times in a table, and compare against serial run. What is your conclusion?

Task 3 - 1D-1D decomposition

Another way to write the matrix multiplication kernel is to assume that each thread will compute one column of matrix C, and thus each block will compute several columns, equal to the number of threads chosen. This decomposition is illustrated in the image below:



Here blocks are organized linearly, and threads within a block are organized linearly as well. Mind that in this situation each thread will have to execute a loop over all elements of the column, i.e. C_{k2}.

Complete the following steps:

1. Implement the 1D-1D decomposition for CUDA matrix multiplication and show correctness by comparing with the serial code.

2. Use the same problem size as in Task 2 and time the kernel (only the kernel) for different configurations of threads per block (i.e. 1, 16, 64, 256, 1024). Compare the result with Task 2 by presenting both timings in a plot vs number of threads.
3. Write your conclusions from the comparison. Which approach and which configuration would you choose? Are there any general conclusions (i.e. valid regardless of the decomposition) that you can draw from this comparison?

Mastery

In scientific computing we can often represent a problem in terms of repetitive matrix-vector multiplication operation. Here we will examine how GPU acceleration can help with solution of such problems.

We will consider a power iteration of matrix A (you can use the same matrix as in previous tasks), given by:

$$x_{n+1} = \frac{A x_n}{||A x_n||},$$

where x_0 is an initial random vector (or vector of constant values, whichever you prefer), and

$$||b|| = \sqrt{\sum_{k=0}^{N-1} b_k},$$

is the Euclidian norm of a vector $b = A x$.

Complete the following steps:

1. Write a serial code implementing matrix-vector product $b = A x$. Create an analogous CUDA kernel and compare the results to make sure CUDA kernel is correct. Explain your choice of block-thread decomposition.
2. Write a serial code which computes the Euclidian norm of a vector. Write a matching CUDA kernel and make sure the result is correct. (we will discuss reduction operations with CUDA on Monday).
3. Write a normalization kernel, which divides each entry of vector b by its Euclidian norm.
4. Write serial power iteration code, which completes the following steps:
    1. Start with initial guess x_0
    2. Repeat 100 times:

2.1. Compute matrix-vector product: $b_n = A x_n$

2.2. Compute Euclidian norm $||b_n||$

2.3. Normalize $x_{n+1} = \frac{b_n}{||b_n||}$

5. Write a matching CUDA kernel. Make sure it gives the same result as the serial code, i.e. x_{100} are the same for both serial and CUDA codes. Time both the serial code and the CUDA code (including memory transfers, etc - remember that you do not have to move data back-and-forth every iteration, only at the beginning and the end of the process). Use the block-thread configuration which, in your opinion, is optimal for this kernel. Base your choice on your experience with Tasks 2-3 and explain your choice.
6. Does the CUDA acceleration help with this power-iteration problem? It it does not, can you think of what can be improved in the code?
7. (optional) if you want to create algorithm which is actually useful, you may want to stop the iteration after convergence is achieved, i .e. $||x_{n+1} - x_n|| < \epsilon$

Note: The power iteration problem computes eigenvectors (and eigenvalues) of matrix A, and as such is an essential component of many algorithms, for example the Google PageRank, which is a cornerstone to the success of Google Search.