HPCA Programming Assignment 2021-2022
Optimizing performance of Checkered Matrix Multiplication

In this assignment, your task is to implement and optimize the "Checkered Matrix Multiplication (CMM)" of two n*n matrices. The sample algorithm is pictorially depicted below. A sample single-threaded unoptimized implementation is also provided via GitLab — check the bottom of this document for the same.

Input: Two n*n matrices. n = 2^k, where k is a natural number and k >= 2.
Output: A matrix of size (n/2) * n

You will have to work on the programming assignment **individually.** The assignment should be submitted on gradescope.

Explanation and algorithm of CMM:

Checkered Matrix Multiplication (CMM) is a variant of matrix multiplication. Following is the explanation of the algorithm:

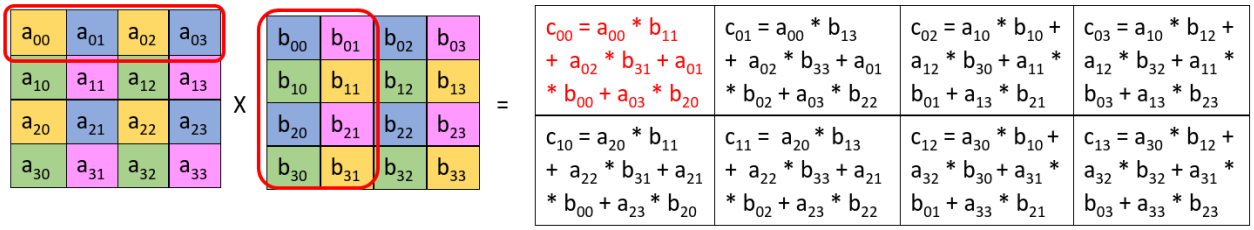
Let us define even and odd positions in matrices A and B. In matrix A, we traverse the elements row-wise, and an element A_{ij} is at even position if j is even and odd otherwise. In matrix B, we traverse the elements column-wise, and an element B_{ij} is even if i is even, odd otherwise.

Let us define columnset in matrix B as a pair of consecutive columns, e.g., 0^{th} and 1^{st} columns of B form the 0^{th} columnset, 2^{nd} and 3^{rd} columns of B form the 1^{st} columnset and so on.

We follow a pattern to multiply the matrices as described below. Note that a row in matrix A is multiplied with a columnset in B.

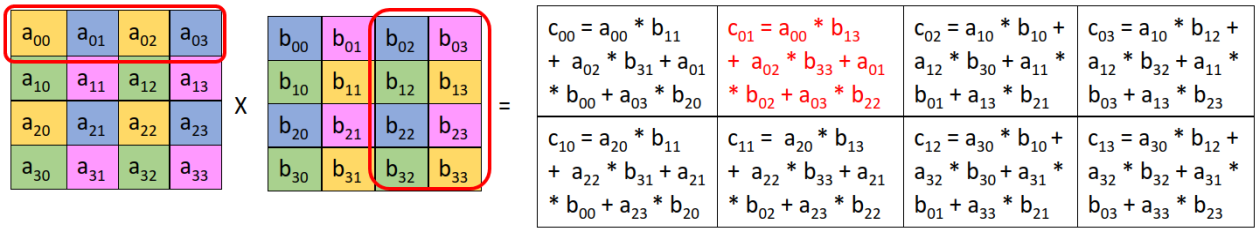
Step 1: Let us take the 0^{th} row of matrix A (say rowA0) and multiply it with 0^{th} columnset of matrix B (= {0^{th} column of B, 1^{st} column of B}) to get C_{00}.

The **even** elements rowA1 are multiplied with **odd** elements of 1^{st} column in the columnset (marked by yellow in figure). And the **odd** elements of rowA1 are multiplied with **even** elements of 0^{th} column in the columnset (marked by blue in figure).



Step 2: Next, we shift the columnset, it now consists of column 2 and 3. Let us take 0^{th} row of matrix A (rowA0) and multiply it the 1^{st} columnset of matrix B to get C_{01}.

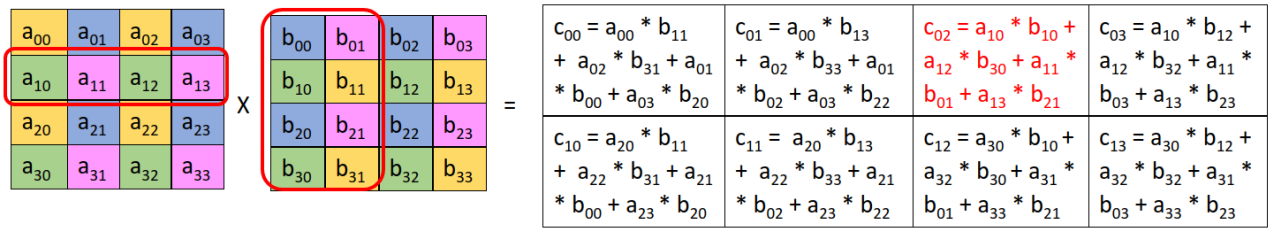
Similar to previous step, the **even** elements rowA0 are multiplied with **odd** elements of 1^{st} column in the columnset, and the **odd** elements of rowA0 are multiplied with **even** elements of 0^{th} column in the columnset.



Note that each row contributes to only half row elements in matrix C, now to obtain the other half elements in this row, we move to the 1st row in matrix A (say rowA1)

Step 3: We again start from the first columnset i.e., columnset = {0th column of B, 1st column of B}. We multiply the 1st row of matrix A (rowA1) with this columnset.

However, this time, the **even** elements rowA1 are multiplied with **odd** elements of 0th column in the columnset (marked by green in figure). And the **odd** elements of rowA1 are multiplied with **even** elements of 1st column in the columnset (marked by pink in figure).



Please note that, every row odd numbered row (o-based indexing), follows the same rule as A1. Whereas, every even numbered row, follows the same rule as rowA0 mentioned previously.
Also, the product of the ith row of matrix A and the jth column of matrix B will be placed in the cell C[floor(i/2)][j] if i is even and in C[floor(i/2)][j+N/2] if i is odd.

You can find the code for this algorithm in GitLab repository.

Details for submitting:

There are three major activities in this assignment, divided in two parts (PartA, PartB):

1. **[PartA-I] Optimize single-threaded CMM (CPU):** In this part, you will hand optimize the single-threaded implementation provided on GitLab.
   Tips: Identify performance bottleneck in your code using hardware performance counters (e.g., via perf) and optimize the source accordingly.
   Relevant file: PartA/header/single_thread.h
2. **[PartA-II] Implement and optimize multi-threaded CMM (CPU):** In this part, you w*ill implement a multi-threaded version of CMM. Test the scalability of your implementation by varying the number of threads and optimize it to achieve maximum scalability (an ideal implementation will achieve a speedup of 2x if the number of cores are increased by a factor of 2, and so on). It is mandatory to use "pthreads" for multi-threaded implementation. Do not use any other custom libraries.
   Relevant file: PartA/header/multi_thread.h

3. **[PartB] Implement and optimize CMM in CUDA (GPU):** In the final part, you are required to implement CMM for a GPU using CUDA.
   Relevant file: TBD

Deliverables: Submission will be in two parts—one for the CPU-based implementations (this will include both single threaded and multi-threaded implementations), and the other for GPU-based implementation. For each part, you are required to submit a report along with your code. Your report should contain details such as the runtime of unoptimized code, and the effect of each of your optimizations on the runtime over different input size (say, n = 4k, 8k and 16k). You should also include details such as how you identified the bottlenecks in the code (e.g., your analysis via hardware performance counters), and which optimizations you have implemented. For the multi-threaded implementation, your report must include an evaluation of the scalability of your implementation (e.g., how does the runtime change with different thread count), along with important details of your implementation.

What/how to submit:
Clone the GitLab repository on your machine and follow its README.md for instructions on how to compile and run the programs. All your code must be implemented in "header/". DO NOT MODIFY main.cpp.
You need to submit your assignment as a zip file named as per the last five digits of your SR number (e.g., 15964.zip).
It should contain two directories – i.e., "header/" that would contain your implementation and "reports/". Name your reports as "reports/Report_PartA.pdf" and "reports/Report_PartB.pdf".

WARNING: IF YOU DO NOT SUBMIT THE CODE AS SPECIFIED ABOVE THEN YOUR ASSIGNMENT WILL NOT BE EVALUATED.

Deadlines:
PartA: 3rd November, Wednesday, 11:59pm
PartB: TBD

Link to GitLab repository:
https://gitlab.com/shweta_pandey/hpca-assignment-2021

Tips for analyzing the performance of your software:
A little bit about the perf tool and hardware performance counters: Modern processors come with an extensive set of performance monitoring counters. It is also called the performance monitoring unit or PMU. These counters typically count occurrences of specific events, e.g., cache miss, TLB miss, etc. Tools like Linux's perf tools provide users with an easy-to-use command-line interface to program and measure these performance counters. There are other tools like Likwid that could do the same or even more.

To see what all performance counters are exposed through perf tool, you can run the following command
$ perf list

For measuring the number of L1-cache-load misses (from user side) for a command say, sleep 1 (sleeps for 1 second) you can use the following command
$ perf stat -e L1-dcache-load-misses:u sleep 1

There are tons of information and tutorials online on perf. A few examples are https://perf.wiki.kernel.org/index.php/Tutorial and http://www.brendangregg.com/perf.html

Performance measurement tips: Real hardware numbers are not deterministic. It changes from run to run. So we strongly suggest that run each measurement at least 5 times and report the mean (and standard deviation if you wish to).