CPSC 424/524 Fall 2023
Assignment 4

Due Date: Tuesday, December 12, 2023 by 11:59 p.m.

CUDA Matrix Multiplication

This assignment uses matrix multiplication to introduce you to GPU programming and performance issues. In many ways it's an exploratory project: in addition to what is required by the assignment, you may well want to experiment with various changes and optimizations in your code to learn more about programming GPUs.

Task 1 (20 points)

Implement a CUDA kernel to compute the product of two random **rectangular** matrices:

$$C = A * B$$

where A is $n \times p$, B is $p \times m$, C is $n \times m$, and n, m, and p are positive integers whose values are provided as command-line arguments. For Task 1, use GPU global memory to hold the matrices and carry out the calculation. The directory **`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment4`** contains a sample code for square matrices (**`matmul.cu`**), a makefile (**`Makefile`**), other files and instructions on how to run on the GPUs (**`README`**). **Please start with the `README` file, which is especially important!**

Algorithmically, the sample code is generally similar to the codes discussed in chapter 4 of Kirk & Hwu (3rd Edition), which is available online at no cost through the Yale Library at this URL:

https://www.sciencedirect.com/book/9780128119860/programming-massively-parallel-processors.

A similar code is also discussed on pages 28-32 of the *CUDA C Programming Guide* posted in the CUDA Documentation folder in the Files section of Canvas.

Each GPU thread in the sample code computes a single element of C, and the sample code checks to be sure it has a sufficient number of threads to carry out the computation. The sample code also includes a CPU version of the calculation, **which you should use ONLY for debugging purposes on small problems (e.g., Run 1 below)**. **Please be sure you comment it out or disable it the rest of the time! For large problems, it may run for hours!**) The CPU version is clearly marked with comments, to make it easy to comment it out.

In addition to the GPU codes for this assignment, you will be asked to create a simple serial CPU code that implements and times the "kij" variant of matrix multiplication described on slide 8 in the **`matrix_multiplication.pdf`** attachment to this assignment. Use this code (running on one CPU core of the node, not the GPU) to obtain serial CPU times requested below. Except as noted for tasks 1(b) and 2(b), all codes should be single precision (**`float`**), not double precision (**`double`**). (Note that the first line of the sample code is a **`#define`** statement to make it easy to switch between single precision and double precision.)

a. Run your kernel in **single precision** (using **`float`** variables) and produce a timing table for the five cases (matrix dimensions) shown in the following table:

	Run 1	Run 2	Run 3	Run 4	Run 5
n	1,024	8,192	1,024	8,192	8,192
m	1,024	8,192	1,024	8,192	1,024
p	1,024	8,192	8,192	1,024	8,192

Timings depend on the dimensions of the thread blocks and grid. For each case, try **a few** different block and grid dimensions and report only the best result for each case. (Make sure your table shows the block

and grid dimensions used to obtain each result.) Use the "kij" code to obtain comparable CPU times.

b. Run the case $n = m = p = 8{,}192$ using double precision on the GPU. In your report, compare the performance for this run with the corresponding single precision run. (You may wish to use block and grid dimensions for this run that differ from those used for Run 2 of part (a).)

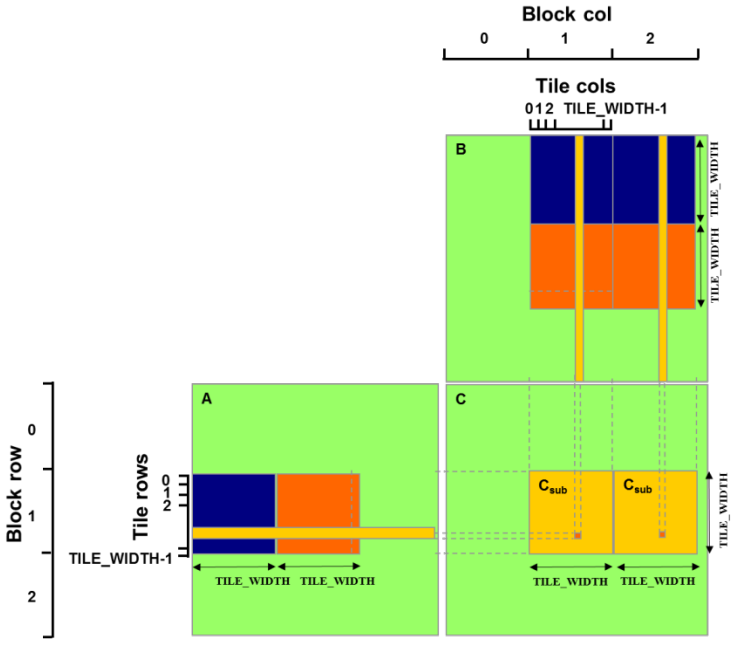
c. For square matrices ($n = m = p$), determine the largest matrix size for which you can run your Task 1 kernels on the GPU in single precision, and print GPU timing results for that size matrix. **[Note: Do not even try to run any CPU matrix multiplication code for this part.]**

Task 2 (40 points)

Modify your CUDA kernel to exploit shared memory on the GPU using a tiled matrix-multiplication code similar to those discussed in Chapters 4 & 5 of Kirk & Hwu and on pages 29-32 (illustrated in Figure 8) of the *CUDA C Programming Guide*. (**Please do not revise your code to use the data structures used in the book or *Programming Guide*. Instead, understand those codes and translate the ideas to your code from Task 1.**) In your new kernel, each thread block will compute a single tile of C, so each thread will compute one entry of C. You may assume that the tiles and blocks are square and of the same size, but do not assume that any of the matrix dimensions is necessarily a multiple of the tile width. Try to exploit memory coalescing in your new kernel, where possible. Repeat parts (a) and (b) from Task 1 using your new kernel. With the timing results, you should report the best tile size, and best block and grid dimensions, that you found in each case. **Except for debugging, please do not run serial calculations for Task 2. (That means, don't use the kij code at all, except for debugging.)**

Task 3 (30 points)

An important algorithmic decision in performance tuning is the granularity of thread computations. It may sometimes be advantageous to put more work into each thread and use fewer threads—for example, when some redundant work exists between threads. The figure below illustrates a situation in the tiled matrix multiplication kernel (cf., chapter 5 of Kirk & Hwu) where this *might* be the case (*your mileage may vary*).



In your Task 2 kernel, each thread block computes one tile by forming the dot products related to one tile-row of A and one tile-column of B. Each tile of C requires a separate set of tile loads (into shared memory) of the required tiles from global memory. As shown above, the calculations of two adjacent tiles requires the same set of tiles from A, so it seems inefficient/redundant to load those tiles separately for each tile of C. The redundant tile loads could be reduced if each block computed multiple adjacent tiles. For example, if each block computed two adjacent tiles

(instead of just one), then each thread could compute corresponding entries in each tile, and the number of tiles loaded from global memory into shared memory would be reduced by ~25%. Unfortunately, this approach does increase the usage of registers and shared memory, so it may not always be feasible or pay off. But it's worth a try.

Modify your **single-precision kernel from Task 2** to implement the strategy described here. (The number of adjacent tiles should be a parameter.) Then run the new kernel for the case $n = m = p = 8{,}192$, varying the number of adjacent tiles handled at one time to try to find the optimum number. (Use the same tile size as you used for the corresponding test case in Task 2.) Report the timing results obtained for successful trials, and indicate the optimum one. **For Task 3 only, you may assume that the matrix dimensions are multiples of the tile width.**

Task 4 (10 points)

Modify your Task 3 code so that it works when the matrix dimensions are not multiples of the tile width. Demonstrate that your new code works using $n = m = p = 1{,}100$ with a 16×16 tile size and 3 adjacent tiles.

Notes and Special Procedures

1. In the directory **`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment4`**, there is a **`README`** file containing important information on how to access GPUs for this assignment.

2. We have reserved eight Nvidia RTX 2080 Ti GPUs for class use in the **`cpsc424gpu`** reservation (4 GPUS on each of 2 nodes). That should be plenty for approximately 20 students provided everyone cooperates. You can limit contention for GPUs by using a batch submission as described in the **`README`** file. You should be able to do all your editing, job submissions and other tasks in a non-GPU session obtained via OOD or ssh.

3. If you must run interactively, please stay within reasonable limits on wallclock time, number of cores, memory, etc. to give everyone a fair chance to use the GPUs both for this assignment and, in some cases, for final projects. See the **`README`** file for more information.

Procedures for Programming Assignments

For this class, we will use the Yale Canvas website to submit solutions to programming assignments.

Remember: While you may discuss the assignment with the instructor, a ULA or your classmates, the work you turn in must be yours alone!

What should you include in your solution package?

1. **All source code files, Makefiles, and scripts that you developed or modified.** All source code files should contain proper attributions, as needed. In addition, make sure that your source code contains suitable documentation (e.g., comments) so that someone looking at your code can understand what it's trying to do. In part, your grade will depend on the documentation, code organization, and comments for your code.
2. **A report in PDF format** containing:
   a. Your name, the assignment number, and course name/number.
   b. Information on building and running the code:
      i. A brief description of the software/development environment used. For example, you should list the module files you've loaded.
      ii. Steps/commands used to compile, link, and run the submitted code. Best is to use a Slurm script that you submit using `sbatch`. The script could use a Makefile to build the code and would then run it multiple times so that you can report average timings. (If you ran your code interactively, then your report will need to list the commands required to run it.)
      iii. Outputs from executing your program(s).
   c. Any other information required for the assignment (e.g., answers to questions that may be asked in the assignment).

How should you submit your solution?

1. On the cluster, create a directory named "`netid_ps4_cpsc424`". (For me, that would be "`ahs3_ps4_cpsc424`".) Put into it all the files you need to submit, including your report.
2. Create a compressed tar file of your directory by running the following in its parent directory:

   ```
   tar -cvzf netid_ps4_cpsc424.tar.gz netid_ps4_cpsc424
   ```

3. To submit your solution, click on the "Assignments" button on the Canvas website and select this assignment from the list. Then click on the "Submit Assignment" button and upload your solution file `netid_ps4_cpsc424.tar.gz`. (We will set Canvas to accept only files with a "`gz`" or "`tgz`" extension.) You may add additional comments to your submission, but your report, including the plot, should be included in the attachment. You can use OOD, tools like scp or rsync, or various GUI tools (e.g., CyberDuck) to move files back and forth to Grace.

Due Date and Late Policy

Due Date: **Tuesday, December 12, 2023 by 11:59 p.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 85% credit

Up to 72 hours late: 50% credit

Up to 1 week late: 25% credit

More than 1 week late: No credit

General Statement on Collaboration

Unless instructed otherwise, all submitted assignments must be your own individual work. Neither copied work nor work done in groups will be permitted or accepted without prior permission.
However….
You may discuss questions about assignments and course material with anyone. You may always consult with the instructor or ULAs on any course-related matter. You may seek general advice and debugging assistance from fellow students, the instructor, ULAs, Canvas ED discussions, and Internet sites, including https://ask.cyberinfrastructure.org/c/yale/40 or similar sites.
However, except when instructed otherwise, the work you submit (e.g., source code, reports, etc.) must be entirely your own. If you do benefit substantially from outside assistance, then you must acknowledge the source and nature of the assistance. (In rare instances, your grade for the work may be adjusted appropriately.) It is acceptable and encouraged to use outside resources to *inform* your approach to a problem, but *plagiarism is unacceptable* in any form and under any circumstances. If discovered, plagiarism will lead to adverse consequences for all involved.
DO NOT UNDER ANY CIRCUMSTANCES COPY ANOTHER PERSON'S CODE—to do so is a clear violation of ethical/academic standards that, when discovered, will be referred to the appropriate university authorities for disciplinary action. Modifying code to conceal copying only compounds the offense.