CPSC 424/524 Spring 2020
Assignment 5

Due Date: April 29, 2020 by 11:59 p.m.

GPU Programming

This assignment uses matrix multiplication to introduce you to GPU programming and performance issues.

Task 1 (30 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 $p \times m$, and 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 **`/home/cpsc424_ahs3/assignments/assignment5/`** contains a sample code for square matrices (**`matmul.cu`**), along with a makefile (**`Makefile`**) and instructions on how to run on the GPUs (**`README`**). Note that the Makefile contains the list of module files you will need. Algorithmically, the sample code is generally similar to the codes discussed in chapter 4 of Kirk & Hwu (3rd Edition, available online through the Yale Library, and on pages 23-28 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 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 runs for hours!**) In addition to the GPU codes for this assignment, please 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 (1) core of the node, not the GPU) to obtain the serial CPU times requested below. Except for tasks 1(b) and 2(b), all codes should be single precision.

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

	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. Experiment with **a few** different block and grid dimensions and report only the best result for each case. (Make sure your table shows what block and grid dimensions you 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. (Note that the sample code includes a **`#define`** statement to make it easy to switch between single precision and double precision.) 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 in Run 2 of part (a).)

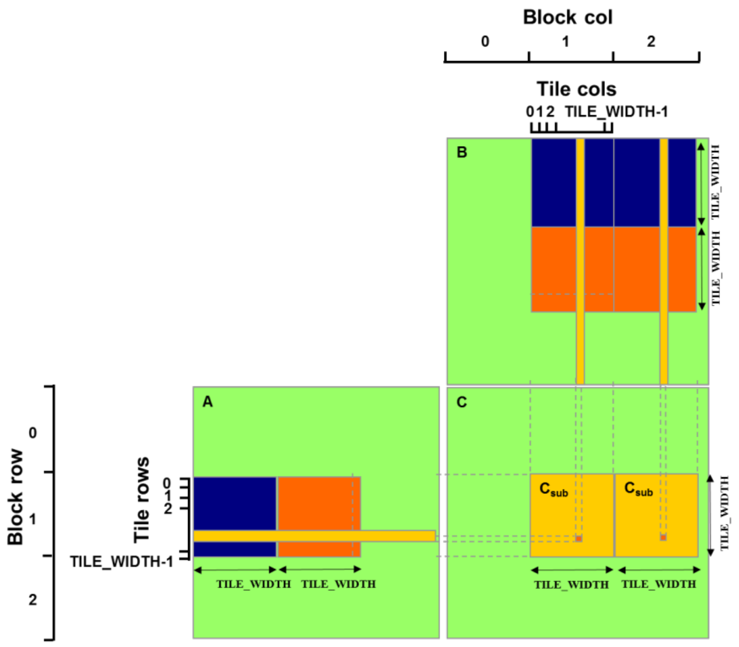
c. For square matrices ($n = m = p$), determine the largest matrix size for which you can run your Task 1 kernels on the GPU in single precision, and print GPU timing results for that size matrix. **[Note: Do not even try to run any CPU code for this part.]**

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 27-29 (illustrated in Figure 10) of the *CUDA C Programming Guide*. (**Please do not revise your code to use the data structures used in the text 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 is often 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 such a situation in the tiled matrix multiplication kernel (cf., chapter 5 of Kirk & Hwu).



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.

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: Extra Credit (10 points)

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

Notes and Special Procedures

1. As described in the **README** file on the cluster, two nodes (c22n05 and c22n06) have been set aside for class use for the balance of the semester. Each node contains 2 K80 GPUs that we have configured as 4 assignable GPUs. You may find it best to use interactive sessions for this assignment. To create an interactive session on a suitable node, use the following **srun** command:

   ```
   srun --pty -c 5 --mem-per-cpu=6100 -p cpsc424_gpu -t 2:00:00
           --gres-flags=enforce-binding --gres=gpu:1 bash
   ```

   You may add the **--x11** option to this command if you wish to make use of X Windows. The nodes are set up for shared access by up to 4 students at once. Please restrict your sessions to 5 cores, ~30GB of CPU memory, and 1 GPU. In order for all students to have a chance to do the assignment, please limit your to 2 hours if there are students waiting. (To see waiting jobs, run "**squeue -p cpsc424_gpu**".)

   If your laptop or desktop machine has a CUDA-capable GPU, you are free to install the CUDA Toolkit on that machine, so that you can do much of your code development without using the cluster at all. (You'll still need to make all the final timing runs on the cluster, however.) If you wish to do that, you can download the Toolkit for your machine from nvidia.com. (You may need to sign up as a developer and then download from the CudaZone. For the class, we will use version 10.1.x or 10.2.x of the Toolkit.)

2. The **README** file in **/home/cpsc424_ahs3/assignments/assignment5/** contains information on how to build GPU codes for this assignment. Additional information is available in the Cuda C Programming Guide and other documents that will be in the Files section of the class Canvas site.

Procedures for Programming Assignments

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

Remember: While you may discuss the assignment with me, a ULA, or your classmates, the source code you turn in must be yours alone and should not represent collaborations or the ideas of others!

What should you include in your solution package?

1. **All source code files, Makefiles, and scripts that you developed, used, or modified.** All source code files should contain proper attributions and suitable comments to explain 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. In this assignment, you may well want to run your codes interactively, in which case you'll need to list the commands required to build and run it. (You can make your timing runs via a batch script, if you prefer.)
      iii. Outputs from executing your program.

c. Any other information required for the assignment, including any questions you were asked to answer.

How should you submit your solution?

1. On the cluster, create a directory named "**NetID_ps5_cpsc424**". (For me, that would be "**ahs3_ps5_cpsc424**". Put into it all the files you need to submit.
2. Create a compressed tar file of your directory by running the following in its parent directory:

```
tar -cvzf NetID_ps5_cpsc424.tar.gz NetID_ps5_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_ps5_cpsc424.tar.gz**. (Canvas will only accept files with a "**gz**" or "**tgz**" extension.)You may add additional comments to your submission, but your report should be included in the attachment. You can use scp or rsync or various GUI tools to move files back and forth to Grace.

Due Date and Late Policy

Due Date:	**Wednesday, April 29, 2020 by 11:59 p.m.**
Late Policy:	On time submission: Full credit
	Up to 24 hours late: 90% credit
	Up to 72 hours late: 75% credit
	Up to 1 week late: 50% credit
	More than 1 week late: 35% 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, Piazza conversations, and Internet 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.