Subject: CAAM 420/520 – Homework 5
Date Posted: April 14, 2023
Date Due: 7:00pm, April 21, 2023 via Canvas

Total number of points available: 72 (CAAM 420 and 520)

GPU Accelerated Matrix Column Scaling *(pts)*
Suppose we are writing a kernel for right-multiplying a dense matrix by a diagonal matrix. Put mathematically, for some matrix $M \in \mathbb{R}^{n_{rows} \times n_{cols}}, D \in \mathbb{R}^{n_{cols} \times n_{cols}}$ we would like to compute:

$$AD = B$$

where

$$D_{ij} = \begin{cases} d_{ii} & i = j \\ 0 & otherwise \end{cases}$$

Consider the three different ways to impose a CUDA grid and thread block shown in Figures 1 - 3.

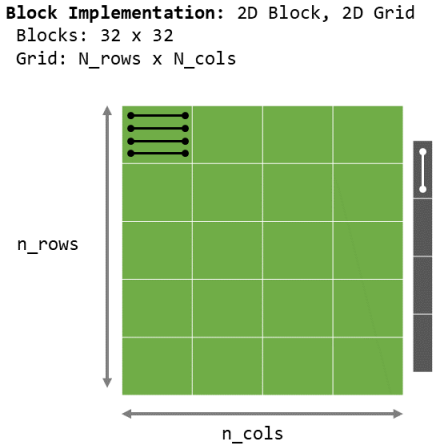

Figure 1. Block implementation: 2D CUDA grid with 2D thread blocks.

Assume we are using a regular row-major memory layout for the matrices. For the 2Dx2D/block setup the light green blocks in Figure 1 show the memory a thread block will act on. For the 2Dx1D/wrapped line and 1Dx1D/line setups, the light green areas in Figures 2 and 3 show the memory where it each thread block will *start* processing the matrix while the darker green areas shows the other elements that each thread block will process.

For all the problems note/assume that:

- `N_rows = n_rows / 32` and `N_cols = n_cols / 32`.
- The diagonal matrix can be stored as a vector, with `D[j]` $= d_{jj}$. The element `D[j]` scales the entire j^{th} column of the matrix M.
- Only a portion of the vector `D` is used by each thread block and each entry will be accessed more than once. As a result, the array for D should be stored in shared memory to reduce the costs of accessing each of its elements more than once.

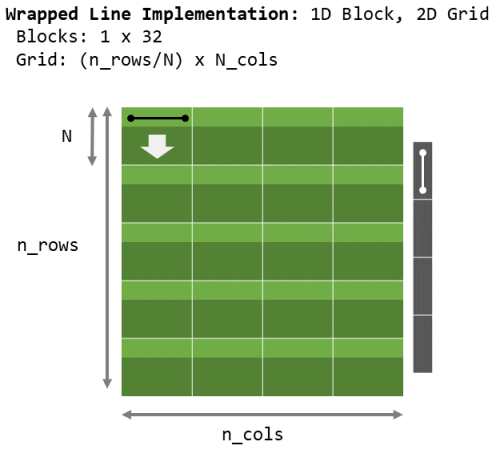

Figure 2. Wrapped line implementation: 2D CUDA grid with 1D thread blocks.

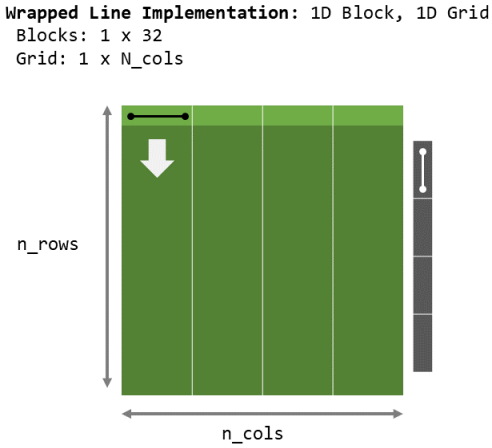

Figure 3. Line implementation: 1D CUDA grid with 1D thread blocks.

- Data is stored as floats, which are 4 bytes per.

Problem 1 Implement the Wrapped Line Kernel *(36 pts)*

Associated Files: `template.cu`

Change the name of the template file to: `<netID>_gpu_scaling.cu`

Expected compile/run command:

Commands/script for compiling and running your code on the NOTS cluster coming soon.

DO NOT RUN YOUR CODE ON THE LOGIN NODE.

Rice may ban you from the cluster if you do.

Implement the wrapped line setup kernel `matrix_scale` *(12pts)* and the function `launch_gpu_matrix_scale` *(24pts)*. Make sure to follow the outline provided for the launch function.

Within your CUDA kernel, make sure to load the array for `D` into shared memory. For this homework you will work with a single file. Download this file, ranem it using your NetID, and make your changes to it.

Problem 2 Kernel Characteristics *(12 pts)*

For each kernel setup, report in a table of the form:

Table 1. Example table with the desire formatting.

Setup:	Block	Wrapped	Line
Elements / thread			
Blocks/SM (thread-based)			
Global memory transactions per block			
Global memory transactions, entire grid			

(a) *(3 pts)* The number of elements a single thread will process.

(b) *(4 pts)* Suppose you were going to launch these kernels on a GPU with 32×64 threads per SM. Assuming there is no maximum number of thread blocks per SM (which is not true in real life) and shared memory is not an issue, what is the most thread blocks that can be run at the same time on a single SM for each setup based on thread usage? Note for the wrapped line case you will have to choose an N that maximizes the number of blocks per SM.

(c) *(5 pts)* The number of **global memory** transactions (reads) for both a single thread block and the ENTIRE grid (i.e. all thread blocks) for just the array `D`. Note that 32 elements (one block's worth) can be read in a single global memory transaction.

Problem 3 Shared Memory Limits *(8 pts)*

Suppose you loaded the appropriate chunk of ALL the arrays into shared memory; i.e. each thread block loaded the chunk from `A, B` and `D` that it needs to read from/write to into shared memory. There is no reason to do this in real life; assume this only for this problem.

Assume the GPU has 64KB of shared memory per SM. Assuming there is no issue with the number of threads per thread block versus threads per SM, what is:

(a) *(3 pts)* **Block setup:** The most blocks that could run on a single SM based on shared memory usage?

(b) *(3 pts)* **Wrapped line setup:** The most and fewest blocks per SM based on shared memory usage and the corresponding value of `N`?

(c) *(2 pts)* **Line setup:** The largest value of `n_rows` that will still allow a single block's shared memory to fit on a SM?

Problem 4 Concurrency and Portability *(16 pts)*

Suppose `n_rows = 1280 = 32 * 40` and `n_cols = 320 = 32 * 10` and your GPU has 26 SMs (like the Nvidia Tesla K80s). Assume you are using shared memory only for `D` again; in this case shared memory use will not be an issue.

(a) *(2 pts)* **Block setup:** How many thread blocks would the kernel use?

(b) *(2 pts)* **Wrapped line setup:** What is the most and fewest thread blocks the kernel could use?

(c) *(2 pts)* **Line setup:** How many thread blocks would the kernel use?

(d) *(2 pts)* If you needed to launch two of the kernels concurrently (i.e. launch each kernel in its own CUDA stream), which setup(s) is/are the most likely to actually allow two of its kernels to be run at the same time given SM usage?

(e) *(2 pts)* If you were able to upgrade to a new GPU, say the Nvidia Tesla P100, with more SMs (56 for the P100), which setup(s) will/can see the biggest improvement in concurrency and runtime?

(f) *(8 pts)* Launching a kernel costs overhead as does assigning thread blocks to SMs. The wrapped line setup allows you to tune the amount of memory processed by a single thread block via its parameter `N` which also controls the number of thread blocks launched. `N` also influences the number of global memory transactions needed for processing the entire grid.

What factors would you want to consider when choosing `N` if you are planning on running its kernel concurrently with other kernels and why?