CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 4

Total number of points: 100 (+ 15 bonus points).

In this homework, you will implement different strategies for matrix multiplication on the GPU. Given matrices $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$, and $C \in \mathbb{R}^{m\times p}$, the **GEMM** (**Ge**neral **M**atrix **M**ultiply) operation is defined as $D = \alpha AB + \beta C$, where α and β are scalars. Explicitly, the elements of D are given by

$$d_{ij} = \alpha \sum_k a_{ik} b_{kj} + \beta c_{ij} \tag{1}$$

Some BLAS libraries perform in-place computation that saves the result D in the memory space of C, as cuBLAS does. This is possible if C is no longer needed after the computation. In this homework, we will perform in-place computation.

Please read the relevant sections of the "CUDA C Programming Guide" written by NVIDIA: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

Problem 1 Basic GEMM

The simplest implementation is to have one thread calculate a single element in the result D. Each thread reads the required row of A, column of B, and an element of C to compute the output element in D. This is equivalent to having each thread calculate the entire sum of k for a given (i, j) in Equation 1. This is equivalent to Figure 8 on page 36 (CUDA PG, Release 12.4). There are two ways to organize the threads: row-major and column-major.

- In row-major, consecutive threads calculate consecutive elements in a row of D.
- In column major, consecutive threads calculate consecutive elements in a column of D.

~~Question 1.1~~

(15 points) Implement the `basicMatMulRowMajor` kernel and the wrapper `basicGEMMRowMajor`.

~~Question 1.2~~

(15 points) Implement the `basicMatMulColumnMajor` kernel and the wrapper `basicGEMMColumnMajor`.

Question 1.3

(25 points) Explain the difference in performance between the row-major and column-major implementations. Also comment on how the size of problem affects the performance of the two implementations.

Problem 2 Shared Memory

The main bottleneck in the basic GEMM kernel is the memory access pattern which is not cache-friendly since a single warp reads along a row of B over the accumulating iterations. This leads to less cache reuse between warps. We can use shared memory to recitfy this by loading blocks of A and B into shared memory, and then having each thread block compute a sub-matrix of D.

In one iteration, each thread in a block loads one element of the corresponding submatrices of A and B into shared memory. Then, each thread accumulates the result for one element in the subamtrix of D. The final result is obtained after accumulating all the small block products. See pages 35 to 39 and Figure 9 on page 39 of the CUDA Programming Guide for a detailed illustration of this approach.

Question 2.1

(45 points) Implement the `SharedMemoryMatMul` kernel and the wrapper `sharedMemoryGEMM`. The starter code has a template kernel with template parameters to determine the size of the block of threads.

Problem 3 Hierarchical Tiling

We can further improve the performance of the shared memory approach by using hierarchical tiling. This is similar to the shared memory approach, but the block computation is broken down into a hierarchy of thread-block tiles, warp tiles, and thread tiles. Each thread also no longer computes a single element of the output matrix, but a tile of the output matrix. A detailed exposition of this approach is given in the CUTLASS Nvidia blog post.

Question 3.1

(*Bonus* 15 points) Implement the `TiledMatMul` kernel. The stater code has a template kernel with template parameters to choose the sizes of the loaded blocks and the number of threads per block. In the provided kernel wrapper `tiledGEMM`, each block of threads computes a 32×32 submatrix of D, and the number of threads per block is set to 128.

A Running the Code

1. We have provided a script `hw4.sh` that compiles the code and runs GEMM tests for each kernel with three different scales of matrix sizes. You can run it on the cluster using `sbatch hw4.sh`.

2. You can add additional tests with different size configurations to make sure your code runs correctly, but we only require `gpu_func.cu` for submission.

B Submission instructions

1. For all questions that require answers besides source code, put those answers in a separate PDF file and upload this file on Gradescope.

2. Make sure your code compiles on `cme213-login.stanford.edu` and runs. To check your code we will run `make` on `cme213-login.stanford.edu`, and this should produce an executable `main`.

3. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files while debugging, but make sure you test with the default files before submitting. Do not use external libraries, additional header files, etc. that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   `gpu_func.cu`

4. To submit, execute

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw4 <directory with your submission files>
```

5. You can submit up to 10 times before the deadline and each submission will replace the previous one.

The final list of items that we expect is:

1. `gpu_func.cu` in the code submission.
2. A screenshot of the output of the provided tests for each question in your PDF submission on gradescope.
3. A written answer to Question 1.3 also in your PDF submission on gradescope.