Programming II

Due: submit the report and source codes to canvas

100 points total

Overview

You will practice CUDA programming in this assignment. Particularly, you will follow the basic CUDA programming and don’t use unified virtual memory. Through this assignment, you will develop an understanding of CUDA execution model, memory model, and techniques for performance optimization.

References:

1. CUDA C programmer’s guide <https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>
2. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA by Ryoo et al. <https://dl.acm.org/citation.cfm?id=1345220>

Environment platform

You will run the experiments on Palmetto GPU nodes. You should request the same node (or the nodes with the same GPU card, e.g., A100, V100, P100, K40, or K80) for your experiments. For Palmetto resource request, visit CCIT website <https://citi.sites.clemson.edu/infrastructure/>.

**Part 1: Compute matrix multiplication with CUDA library (10pts)**

mmCUBLAS (available on canvas) uses GPU device to compute matrix multiplication matrix\_result = A \* B for single precision matrices. Instead of implementing from scratch, this program invokes the cublasSgemm() function.

This code follows a good practice for performance measurement. For example, it verifies the computation result, performs a warmup operation before timing the execution, and times multiple iterations of matrix multiplication to make sure the execution time is long enough. Take note where the timing starts and stops.

Compile the code, and collect execution time for a wide range of matrix sizes from small to large. The largest size you attempt should exceed the GPU physical memory, which would cause an error. Show how performance changes with matrix size and explain why this trend is reasonable.

**Part 2: Implement matrix multiplication with basic CUDA (20pts)**

Instead of calling the cublasSgemm() function, implement a kernel function for matrix multiplication with basic CUDA, and timing its performance.

You should name your program mmNaive.cu. You can reuse the structure and code segments of mmCUBLAS.cpp, and replace cublasSgemm with your own implementation. Check to make sure the computation on the device is correct.

Use the similar timing method for your implementation. For example, perform a warmup operation before timing the execution, time multiple iterations of matrix multiplication to make sure the execution time is long enough.

Compile the code, and collect execution time for the same matrix sizes as Part 1. Show how performance changes with matrix size and compare your performance with MMCublas. How slower is your implementation? Can you identify the reasons?

**Part 3: Implement matrix multiplication with Unified Virtual Memory** cudaMallocManaged **(20pts)**

UVM allows programmers to oversubscribe the GPU memory, i.e., the kernel can access larger space than the GPU physical memory. A beginner tutorial can be found here <https://developer.nvidia.com/blog/unified-memory-cuda-beginners/>.

implement a kernel function for matrix multiplication with cudaMallocManaged, and timing its performance.

You should name your program mmMallocManaged.cu. You can reuse the structure and code segments of mmCUBLAS.cpp, and replace cublasSgemm with your own implementation. Check to make sure the computation on the device is correct.

Use the similar timing method for your implementation. For example, perform a warmup operation before timing the execution, time multiple iterations of matrix multiplication to make sure the execution time is long enough.

Compile the code, and collect execution time for the same matrix sizes as Part 1. In addition, collect execution time for three larger sizes that fail on part2 and part3. Show how performance changes with matrix size and compare your performance with Part1 and Part2. Is the time trend reasonable? Is it slower or faster? Can you identify the reasons?

**Part 4: Optimize your matrix multiplication (50pts)**

Optimize your code from Part 2. Name your program mmOpt.cu. You can reuse the structure and code segments of mmCUBLAS.cpp. Collect the results for the same problem sizes as Part2, and calculate the speedup for each problem size.

In your write up, describe in detail the techniques you have used and why you choose to use them. The final version should include several optimization techniques and has the highest performance. Present the speedup gained from each newly stacked technique. Your points will be based on the performance gain your programs achieve and your explanations, justifications, and reasoning.

Matrix transpose is a common exercise in GPU optimization, so do not search for existing GPU matrix transpose code on the internet.

**Submission Instructions**

Submission will be through canvas.

Please place the following files in your submission:

* Your writeup, in a file called writeup.pdf
* All your source codes in a single tar file. All code must be compilable and runnable!