Programming 2

Due: 12:29 PM, Oct 9, submit the report and source codes to canvas

100 points total

Overview

You will practice using programming models (abstractions): data parallelism and shared address space, in this assignment. Particularly, you will use basic CUDA, CUDA library, unified virtual memory supported CUDA, and OpenMP. Through the assignment, you will develop an understanding of the parallel execution on general purpose GPU architectures.

References:

1. [An Easy Introduction to CUDA C and C++](https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/)
2. [CUDA C/C++ Basics -- Nvidia](https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf)
3. [GPU-Accelerated Libraries for Computing](https://developer.nvidia.com/gpu-accelerated-libraries)
4. [An Even Easier Introduction to CUDA](https://devblogs.nvidia.com/even-easier-introduction-cuda/) (with unified memory, supported by Pascal and Volta GPUs)
5. OpenMP for GPU

Make sure to go through examples, understand them, and run them.

Experiment Platform

You will need to run the codes on a GPU system in Palmetto cluster. You can find user’s guide at https://www.palmetto.clemson.edu/palmetto/userguide\_howto\_control\_job\_placement.html. Make sure that you place your job on a node with Pascal or Volta GPUs.

Make sure to load cuda modules when you compile and run the codes. Submit each of your programs to the same machine.

**Part 1: Compute matrix multiplication with CUDA library (20pts)**

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. Show how performance changes with matrix size and explain why this trend is reasonable.

**Part 2: Implement matrix multiplication with basic CUDA (30pts)**

Instead of calling the cublasSgemm() function, implement a kernel function for matrix multiplication with basic CUDA, and timing its performance. You can reuse the structure and code segments of MMCublas.cpp, and replace cublasSgemm with your own implementation if you like, or you can write from scratch for everything including main, matrix init, etc. 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. Show how performance changes with matrix size and compare your performance with MMCublas. How slower is your implementation? Can you identify the reasons? Can you improve the performance of your implementation? If you have tried to improve the performance, what techniques have you used and what is the speedup?

**Part 3: Implement matrix multiplication with UVM supported CUDA (20pts)**

Make a copy of your program for part 2, and change it to use UVM-supported easier CUDA programming to implement matrix multiplication.

Use the similar timing method for your implementation. For example, perform a warmup operation before timing the execution and 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. Show how performance changes with matrix size and compare the performance with those from Part 1 and Part 2. Which implementation gives the best performance for different matrix sizes? Can you explain why?

**Part 4: Implement matrix multiplication with OpenMP target for GPU (30pts)**

Make a copy of your program for part 2, and change it to OpenMP target to offload matrix multiplication to GPU.

Use the similar timing method for your implementation. For example, perform a warmup operation before timing the execution and 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. Show how performance changes with matrix size and compare the performance with those from Part 1, 2 and 3. Among these implementations, which one is the easiest to think (abstract)? Which one is the easiest to program? What one achieves the best performance? If a different problem from matrix multiplication is assigned to you in your future career, which one do you likely select?

**Submission Instructions**

Submission will be through canvas.

Please place the following files in your submission:

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