CS 610 Semester 2022–2023-I: Assignment 5

30th October 2022

Due Your assignment is due by Nov 13, 2022, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.
- Refer to the CUDA C++ Programming Guide for documentation on the CUDA APIs and their uses.
- It is always a good idea to take the average of multiple runs while reporting performance.

Submission

- Submission will be through mooKIT.
- Name each source file as "<roll-no-probno>.cpp" (e.g., "22111090-prob2.cpp").
- Submit a PDF file with name "<roll-no>.pdf". Describe your solutions to the problems in the PDF, and explain your implementations. Describe any challenges (if any) that you may have faced.
- Provide exact compilation instructions (e.g., specify `arch` and `code` flags to `nvcc`). We will use the GPU servers in the department for our evaluation.
- We may provide template code for a few problems. Modify the template code where instructed and perform evaluations.
- We encourage you to use the LaTeX typesetting system for generating the PDF file.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Your solutions should only use concepts that have been discussed in class.
- Show your computations where feasible and justify briefly in the PDF.
- Write your code such that the EXACT output format is respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our OWN inputs and test cases (where applicable), so remember to test thoroughly.
- We may deduct marks if you disregard the listed rules.

Problem 1

[10+20+20 marks]

(a) Create a parallel CUDA kernel for the following code.

```
#define SIZE (8192)
double F_in[SIZE][SIZE], F_out[SIZE][SIZE];

for (int k=0; k<100; k++)
 for (int i=1; i<SIZE-1; i++)
  for (int j=0; j<SIZE-1; j++)
   F_out[i][j+1] = F_in[i-1][j+1]+F_in[i][j+1]+F_in[i+1][j+1];

```

(b) Now run the same kernel with `SIZE` equal to 8200 without modifying the code. Compare and report the performance of the two kernels, and explain the performance difference (if any).

(c) Try all valid optimizations to improve the performance for the second sub-problem. Explain your steps in the report.

Make sure to time the same calls across the two versions.

Problem 2

[10+20+20 marks]

1. Implement a naïve matrix multiplication kernel with CUDA. The naïve version directly implements the standard ijk $\mathcal{O}(n^3)$ algorithm.
2. Implement an optimized matrix multiplication kernel. Your kernel should include shared memory optimizations (like tiling) for improving performance. In addition, you are encouraged to use other valid tricks like loop interchange and loop unrolling.

Assume that the matrices are square, and the size is a power of two. Initialize the matrix with random contents.

Compare the correctness of your results with the serial version, and report speedups with the two CUDA implementations. Your goal is to strive for getting as much speedup as possible with the second optimized kernel.

Your report should investigate and describe how each of the following factors influences the performance of the kernels:

- The size of matrices to be multiplied, using the following sizes: 1024, 2048, and 4096.
- The size of the block/tile computed by each thread block. Experiment with blocks of sizes 8, 16, and 32.

Problem 3

[10+10+10+10 marks]

We will build on the optimized matrix multiplication kernel that you came up with for Problem 2.

Compare the performance impact of replacing calls to `cudaMemcpy()` using the pinned memory variant `cudaHostAlloc(..., cudaHostAllocDefault)`, zero-copy memory variant `cudaHostAlloc(..., cudaHostAllocMapped)`, and unified virtual memory `cudaMallocManaged()`.

Thus, there will be four kernel versions for this problem. Compare the performance of the different kernels, and justify your observations. Assume that the size of matrices are 1024. The matrix dimensions should work with the available pinned memory.

Use `cudaEvent` APIs for timing kernels and different CUDA functions. You should use tools like nvprof to justify your results. Nvprof should suffice, although Nvidia now recommends using Nsight Compute or Nsight Systems.

Problem 4

[10+20+10 marks]

Consider the following code.

```
#define N 64
float in[N][N][N], out[N][N][N];
for (i=1; i<N-1; i++) {
 for (j=1; j<N-1; j++) {
  for (k=1; k<N-1; k++) {
   out[i][j][k]=0.8 * (in[i-1][j][k] + in[i+1][j][k] + in[i][j-1][k] +
              in[i][j+1][k] + in[i][j][k-1] + in[i][j][k+1]);
  }
 }
}

```

The above computation pattern is also referred to as the stencil computation pattern. In this pattern, the value of a point is a function of the neighboring points. The access pattern has reuse on the array `in` in 3 dimensions.

(i) Implement a naïve CUDA kernel that implements the above code.

(ii) Use tiling to exploit locality in shared memory to improve the memory access performance of the code.

You are free to try other valid optimizations (like unrolling) for improved performance.

Initialize the elements of `in` with random values. Report and compare the performance of the two versions. Explain your optimizations and highlight their impact in the second version.