CS 610 Semester 2020–2021-I: Assignment 5

17th November 2020

Due Your assignment is due by Dec 3, 2020, 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.
- We MAY check your submission(s) with plagiarism checkers.

Submission

- Write your programs in C++ and CUDA. You can IMPLEMENT MULTIPLE KERNEL VERSIONS TO EMPHASIZE THE INCREMENTAL IMPACT of your optimizations. Include relevant information in your report.
- You can use the department GPU servers or any other system with NVIDIA GPUs.
- Submission will be through mooKIT.
- For time measurements, you should ignore the time for memory allocation. For e.g., ignore `cudaMalloc()` but include the time for `cudaMemcpy()`.
- Submit a compressed file with name `"<roll-no>.zip"`. The compressed file can have the following structure.

```
-- roll-no
-- -- report.pdf
-- -- <problem1-dir>
-- -- -- roll-no.cu
-- -- <problem2-dir>
-- -- -- roll-no.cu
```

  Name your files as `"roll-no.cu"`. Submit a PDF file with name `"report.pdf"`. We encourage you to use the LaTeX typesetting system for generating the PDF file.

  The file should include your name and roll number, and RESULTS AND EXPLANATIONS for the following problems. Include the exact compilation instructions for each programming problem, for e.g., `nvcc -arch=sm_61 -std=c++11 assignment5-p1.cu -o assignment5-p1`. Include any other information and assumptions that you think will help with the evaluation.

- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Write your code such that the EXACT output format (if any) is respected.
- Do not change code templates that you are not supposed to touch (for e.g., correctness checks). You may be penalized.
- We will evaluate your implementations on the department GPU servers.

- We will evaluate the implementations with our OWN inputs (for e.g., different input array sizes) and test cases, so remember to test thoroughly.

Problem 1 **[50 marks]**

Consider the following code.

```
#define SIZE 8192
double F[SIZE][SIZE];

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

(i) Create a naïve parallel CUDA kernel. Make sure that the result is correct.

(ii) Try and run the same kernel with `SIZE` equal to 8200. You are free to try all valid optimizations to achieve good performance.

Report and compare the performance of the two kernels (kernel 1 with `SIZE=8192` and kernel 2 with `SIZE=8200`), and explain the performance difference.

Problem 2 **[50 marks]**

Consider an array $A = [a_0, a_1, \ldots, a_{n-1}]$. The *exclusive prefix sum* computation on A returns the array $[0, (a_0), (a_0 + a_1), \ldots, (a_0 + a_1 + \cdots + a_{n-2})]$. In other words, an exclusive sum implies each element j of the result array is the sum of all elements up to but *not* including j from the input array.

A prefix scan operation generalizes the above computation with a binary associative operation $\oplus$ and an identify element I. In an *inclusive* sum, all elements including j are summed.

Implement a parallel version of the exclusive prefix scan algorithm with CUDA. Apply tiling to exploit locality in shared memory. Assume the size of inputs are always a power of two.

Report the performance of your kernel(s), and explain the impact of your optimizations on the performance.

Problem 3 **[50 marks]**

Implement an *optimized* CUDA kernel for multiplying the transpose of a square matrix A with itself (i.e., $A^T A$). The sequential code for the computation is given below.

```
#define SIZE 4096
double F[SIZE][SIZE];
// C = (A^T)A
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    for (int k = 0; k < SIZE; k++)
      C[i][j] += A[k][i] * A[k][j];
```

Assume the matrix size is always a power of two and $1024 \leq$ `SIZE` ≤ 8192. Report the performance of the serial version and the parallel CUDA kernel(s). You are free to try all valid optimizations (for e.g., tiling and unrolling) to achieve good performance for this kernel.

Problem 4 [50 marks]

Implement two versions of matrix multiplication with CUDA: a naïve kernel and an optimized kernel. The naïve version directly implements the standard $\mathcal{O}\left(n^3\right)$ algorithm. You are free to use any valid trick like blocking/tiling and loop unrolling for the second optimized kernel. 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.

For this problem, assume that the matrices are square and the size is a power of two. Initialize the matrix with random contents.

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.
- Any other optimizations that you tried for the second CUDA kernel. For example, you may use `#pragma unroll` to unroll your loop or use manual unrolling.

Problem 5 [50 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 sometimes referred to as stencil computation. 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. You can also check whether other memory allocation functions in CUDA help improve the kernel performance (for e.g., `cudaMallocPitch()` and `cudaMalloc3D()`). Refer to the CUDA C++ Programming Guide for more information.

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