CS 610 Semester 2023–2024-I: Assignment 4

11th October 2023

Due Your assignment is due by Oct 25, 2023, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not copy 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 Canvas.
- Submit a PDF file with name "<roll-no>.pdf". We encourage you to use the LaTeX typesetting system for generating the PDF file.
- Briefly explain your implementations, results, and other issues of interest (if any). Include the exact compilation instructions for each programming problem. For example, 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.
- Name each source file "<roll-no-probno>.cpp" where "probno" stands for the problem number (e.g., "23111545-prob2.cpp").
- 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.
- Write your code such that the EXACT output format is respected.
- We will evaluate the implementations with our inputs and test cases, so remember to test thoroughly.
- We may deduct marks if you disregard the listed rules.

Problem 1 **[20+30+10+10+20 marks]**

Consider the following code.

```c
#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 referred to as *stencil* pattern. In the stencil pattern, the value of a point is a function of the eight neighboring points (three in row $i-1$, two in row i, and three in row $i+1$). In the above listing, the access pattern has reuse on the array `in` in 3 dimensions.

You will implement five versions of the above stencil pattern. Your goal is to strive for getting as much speedup as possible with the second optimized kernel. Compare the performance of the different kernel versions, and explain your observations. Initialize the elements of `in` with random values.

(i) Implement a naïve CUDA kernel (i.e., no optimizations are required) that implements the above code.

(ii) Use shared memory tiling to improve the memory access performance of the code. Investigate and describe how the size of the block/tile computed by each thread block influences the performance. Experiment with blocks with sides comprising values from the set $\{1, 2, 4, 8\}$,

(iii) Implement other loop transformations like loop unrolling and loop permutation over version (ii). Explain your optimizations and highlight their impact.

(iv) Implement version (iii) using pinned memory (e.g., `cudaHostAlloc(..., cudaHostAllocDefault)`.

(v) Implement version (iii) using unified virtual memory (e.g., `cudaMallocManaged()`).

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

Problem 2 **[30+10 marks]**

Implement the exclusive prefix sum algorithm with CUDA. You are allowed to port any known parallel algorithm to CUDA. NVIDIA provides a "parallel algorithms" library called Thrust that is similar in spirit to C++ STL. Write a function to use Thrust's prefix sum algorithms to check for the correctness of your implementation.

We will test your code for correctness and performance on random input arrays.

Problem 3 **[30+30+30+30 marks]**

Parallelize the attached C code (`problem3-v0.c`) using CUDA. You will implement three versions.

1. The first version will be a vanilla port of the C code. Your goal in this version is to ensure correctness. The challenges are in mapping the large iteration space of the 10D loop nest and writing back the output data from the device to the host. Implementing optimizations is optional.
2. Improve the performance of version (i) using different possible optimizations (e.g., shared memory tiling, launch multiple kernels, data prefetching and memadvise).
3. Implement the C code with UVM support with CUDA.
4. Implement a version using different transformations provided by Thrust. A few Thrust transformations are well-suited for the given problem.

Compare the performance of the four versions.