CS 610 Semester 2024–2025-I: Assignment 4

25th October 2024

Due Your assignment is due by Nov 3, 2024, 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.
- You can tweak the compilation commands in the `Makefile` to suit your compiler version and the target architecture. Include the updated `Makefile` with exact compilation instructions for each programming problem (e.g., `arch` and `code` flags to `nvcc`).
- Refer to the CUDA C++ Programming Guide for documentation on the CUDA APIs and their uses.

Submission

- Submission will be through Canvas.
- Submit a compressed file called "⟨`roll`⟩`-assign4.tar.gz`". The compressed file should have the following structure.

```
-- roll
-- -- roll-assign4.pdf
-- -- <problem1-dir>
-- -- -- <source-files>
-- -- <problem2-dir>
-- -- -- <source-files>
-- -- <problem3-dir>
-- -- -- <source-files>
-- -- <problem4-dir>
-- -- -- <source-files>
```

- We encourage you to use the LaTeX typesetting system for generating the PDF file. You can use tools like Tikz, Inkscape, or Draw.io for drawing figures if required. You can alternatively upload a scanned copy of a handwritten solution, but MAKE SURE the submission is legible.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Write your programs such that the EXACT output format (if any) is respected.
- We will primarily use the GPU3 department server for evaluation.
- We may evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.

Problem 1

[10+20+10+10+10 marks]

Consider the following code.

```
#define N 64
double 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

[20+10 marks]

Implement the exclusive prefix sum algorithm with CUDA for unsigned integer type. 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

[20+20+20+20 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.

Problem 4

[25+35 marks]

Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights are provided via an input array that we will call the convolution filter.

(i) Implement a CUDA kernel to compute 2D convolution for each element of an input 2D array by averaging the values of its neighboring elements.

(ii) Implement a CUDA kernel to compute 3D convolution for each element of an input 3D array by averaging the values of its neighboring elements in a cuboid of unit length keeping the element under consideration at the center.

- Use single-precision floating point type.
- Assume that the 2D and 3D arrays are squares and cubes. That is, sides have the same dimension and is a multiple of eight.
- The convolution filter will be a square with odd dimensions.
- Assume the missing boundary elements (i.e., ghost cells) are zero.

Implement two versions of the 2D and 3D kernels. The first one should be a basic version that uses global memory while the second one is your most optimized version that possibly exploits features like shared memory, constant memory, and loop tiling. Compare and report the performance improvement.