CS 610 Semester 2025–2026-I: Assignment 4

28th October 2025

Due Your assignment is due by Nov 14, 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`).

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>
-- -- -- <rollno-prob1.cpp
-- -- -- <other-source-files>
-- -- <problem2-dir>
-- -- -- <rollno-prob2.cpp
-- -- -- <other-source-files>
-- -- <problem3-dir>
-- -- -- <rollno-prob3.cpp
-- -- -- <other-source-files>
-- -- ...
```

- We encourage you to use the LaTeX typesetting system for generating the PDF file. You can use tools like Tikz, Inkscape, or Drawio 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 will evaluate the implementations with our inputs and test cases, so remember to test thoroughly.

Suggestions

- Refer to the CUDA C++ Programming Guide for documentation on the CUDA APIs and their uses.
- Use a workstation in the KD lab for performance evaluation of CPU-only code, and include the name of the workstation in your report. It will help reproduce the performance results.

Problem 1

[10+20+10+10 marks]

Consider the following 7-point 3D stencil code.

```
#define N 64
double in[N][N][N], out[N][N][N];
for (int i=1; i<N-1; i++) {
  for (int j=1; j<N-1; j++) {
    for (int 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]);
    }
  }
}
```

In the stencil pattern, the value of a point is a function of the six neighboring points (two each in the i, j, and k planes).

Implement four versions of the above stencil pattern. Your goal is to strive for getting as much speedup as possible, especially with the second version. 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)`).

Compare the performance of the different kernel versions, and explain your observations. Use `cudaEvent` APIs for timing CUDA kernels. Use the Nsight Systems or Nsight Compute profilers for analyzing the results.

You can implement the aforementioned versions in different files.

Problem 2

[20+20 marks]

Implement an inclusive prefix sum algorithm with CUDA for unsigned integer type for large input sizes that overflow the GPU memory. You are free to adapt any known parallel prefix sum algorithm for CUDA, but you are not allowed to use library implementations (e.g., Thrust).

You will implement two versions: (i) with the "copy-then-execute" model without using UVM features, and (ii) use UVM support with `cudaMallocManaged()`. The "copy-then-execute" model

should perform better than UVM for input sizes that fit in the GPU. Increase the value of the input to oversubscribe the GPU memory for the UVM implementation.

Use memory advises (e.g., `cudaMemAdvise()`) and prefetch hints (e.g., `cudaMemPrefetchAsync()`) to improve the performance. Usually, you have to try out all possible variations of the hints and check which is the best empirically.

Problem 3

[20+20+20+20 marks]

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

(i) 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.

(ii) Improve the performance of version (i) using different possible optimizations (e.g., shared memory tiling, launch multiple kernels, data prefetching and memadvise).

(iii) Implement the C code with UVM support with CUDA. Use memory advises and prefetch hints to improve the performance.

(iv) 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. You are allowed to partition the work across multiple kernel launches given the large iteration space.

Problem 4

[20+30 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 are 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 implements (a) tiling using shared memory, (b) constant memory for the filter, and (c) loop unrolling. Compare and report the performance improvements.