

CS 623 Semester 2026–2027-I: Assignment 2

5th October 2026

Due Your assignment is due by Oct 26, 2026, 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 GPU architecture. Include the updated **Makefile** with exact compilation instructions for each programming problem (e.g., **arch** and **code** flags to **nvcc**).

Submission

- Email me a compressed file called “`<roll>-assign2.tar.gz`”. The compressed file should have the following structure.

```
-- roll-assign2
-- -- roll-assign2.pdf
-- -- Makefile
-- -- <problem1-dir>
-- -- -- <rollno-prob1.cu>
-- -- -- <other-source-files>
-- -- <problem2-dir>
-- -- -- <rollno-prob2.cu>
-- -- -- <other-source-files>
-- -- ...
```

- We encourage you to use the L^AT_EX 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 Release 12.9 for documentation on the CUDA APIs and their uses.

Problem 1

[10 + 10 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. Implement a CUDA kernel to compute 3D convolution for each element of an input 3D array by averaging the values of its neighboring elements keeping the element under consideration at the center.

- Use single-precision floating point type.
- Assume that the sides in 3D arrays have the same dimensions and are a multiple of eight.
- The convolution filter is a 3D cube with dimensions $K \times K \times K$, where $K = 5$. The filter is initialized in the template file.
- Assume the missing boundary elements (i.e., ghost cells) are zero.
- The output array has the same dimensions $N \times N \times N$ as the input array, where $N \in \{64, 128, 256\}$.
- Implement a single-threaded CPU convolution and assert that the your CPU and GPU results are same (a helper function is included in the template file).

Implement two versions of the 3D kernel. The first one should be a basic version that uses global memory while the second one is your optimized version that implements (a) tiling using shared memory, (b) constant memory for the filter, and (c) loop unrolling. Compare and report the performances both end-to-end and just the kernels. It is always a good practice to take the average of multiple runs.

Problem 2

[15+15+15 marks]

Parallelize the attached C code (`problem2-v0.c`) using CUDA. You will implement three versions.

- (i) The first version will be a vanilla port of the C code. Your goal in this version is to ensure correctness. The output file `results-v0.txt` the number of points should match.

The challenge is in mapping the large iteration space of the 10D loop nest and writing back the output data from the device to the host. You are allowed to launch multiple kernels.

- (ii) Improve the performance of version (i) using different possible optimizations (e.g., shared memory tiling, launch multiple kernels, and data prefetching).
- (iii) Implement a version using different transformations provided by Thrust (e.g., `thrust::transform`). A few Thrust transformations are well-suited for the given problem.

Compare the performance of the three versions. You are allowed to partition the work across multiple kernel launches given the large iteration space.

Problem 3

[30 + 30 + 5 + 20 marks]

A Bloom filter is a probabilistic data structure used to efficiently test whether an element is a member of a set. It uses a bitmap (bit array) and multiple (say k) hash functions to represent the inserted elements. While inserting a key x , the k hash functions are evaluated with x , and the corresponding bit positions in the bitmap are set. A lookup for key y proceeds similarly, and returns “present” only if all the hashed locations are set. The key is not stored in the filter.

A Bloom filter has no false negatives: if it reports that a key is absent, then the key is guaranteed to not be in the set. It is obvious that a vanilla Bloom filter can generate false positives. That is, the Bloom filter may report that a key exists even though the key was never inserted. The advantage of a Bloom filter is that it provides fast insertion and lookup operations while using relatively little memory.

The above design cannot support deletion because resetting a bit in the bitmap from 1 to 0 may affect the correctness for other keys that hash to the same location. You have to design a “counting Bloom filter” that supports deletion by instead maintaining counters. For a counting Bloom filter, each position contains a counter instead of a single bit.

- (i) Implement the three data structure operations, `insert()`, `delete()`, and `search()` as separate kernels. Each kernel will take a batch of keys to be operated on. The three kernels will be invoked separately from the host code, e.g., `insert()` followed by `search()` followed by `delete()` and again followed by `search()`. That is, an `insert(x)` cannot overlap with a `delete(y)`. This constraint helps in checking the correctness of your implementations.
- (ii) Implement a “blocked” variant for the counting Bloom filter. The blocked version will use $k + 1$ hash functions. On an input x , the first hash function will select a GPU L1 cache block, and the next k hash functions will operate in the given block. This improves locality, but increases the rate of false positives.
- (iii) Write a test routine `check_result()` on the host that will validate the correctness of your GPU kernels. A simple idea is to perform the same operations on a `std::set` or `std::unordered_map` on the host and check for equivalence with the GPU Bloom filter data. Include the test routine in your submission.

This also allows you to (imperfectly) test for false positives for search operations. Report the false positive rate (ratio of false searches over all searches) for the two variants, given an input trace.

- (iv) Profile your `insert()` and `search()` kernels (ignore `delete()`) with NVIDIA Nsight Compute, and report the following metrics as output by the profiler: memory bandwidth utilization, arithmetic intensity, achieved occupancy, warp efficiency, L2 cache hits, and long scoreboard stalls. List the top three performance bottlenecks listed by the profiler (e.g., uncoalesced memory access, short or long scoreboard stalls, and atomics) in your PDF report.

Here are more details on the implementations.

- Your host code should take four arguments as follows.

```
./problem3-vanilla.out N <abs-path-insert-trace> <abs-path-search-trace> <abs-path-delete-trace>
```

N represents the number of elements/counters in the Bloom filter array.

The two variants could be in different files.

- The key data type is uint32, and you can use 8-bit counters. Ignore counter overflows. Deleting a non-existent value should do nothing.

Note that CUDA does not provide 8-bit CASes.

- Your implementation should read the keys to be operated on from a binary input file containing uint32 numbers. There will be different trace files for each operation. The traces can have duplicates, which have to be ignored (i.e., the counters will not be incremented).

```
// Read trace file insert1
insert_host(insert1); // Calls the GPU kernel insert() with insert1 sequence
// Read trace file search1
search_host(search1); // Calls the GPU kernel search() with search1 sequence
// Read trace file delete1
delete_host(delete1); // Calls the GPU kernel delete() with delete1 sequence
```

Generate a few input traces yourself and include the binary files in your submission (should be of reasonable size). You can assume each operation (i.e., kernel call) will process a maximum of 10^7 keys.

- You can assume that the input array to be operated on (i.e., passed to the kernels) will fit in GPU memory. You are allowed to preprocess the input key array.
- Set k to three, and use any reasonable (not too expensive) hash function of your choice that generates hashes in the range $[0, N)$. Define the hash functions in separate functions: `hash1()`, `hash2()`, `hash3()`, and `hash4()`.
- You can use any mapping of threads to data, e.g., one thread per key or one cooperative group per key.
- You should implement all the functionality from scratch, i.e., you are not allowed to use any third-party libraries. Your map on the GPU should be a raw array, and not a container like hash tables.
- Ensure that you use atomic operations to avoid data races while updating the map of counters.