Module 8 Lab: Histogram

Sajina Pathak

April 2025

1 Introduction

Example

Let's take an example of this situation:

Input:

```
input[] = [5, 5, 5, 5, 5, 5, 5, 5, ... lots of 5s]
```

After `histogram_kernel` (in `bins_temp[]`):

```
bins_temp[5] = 312
```

(The count of '5' is 312.)

After `saturate_kernel` (in `bins[]`):

```
bins[5] = 255
```

2 Plain Naïve (Unoptimized) CUDA Histogram

Initially, I implemented a **naïve** histogram kernel where each thread directly performed an `atomicAdd()` operation on the global `bins[]` array. The implementation used a type cast:

```
atomicAdd((unsigned int*)&bins[bin], 1);
```

Problems Faced

This approach caused **misaligned memory access** errors because the `bins[]` array was declared as `uint8_t` (8-bit integers), whereas `atomicAdd()` operates safely only on 4-byte (32-bit) aligned memory.

The kernel crashed with the error:

CUDA Error: misaligned address

Thus, it became clear that casting types and using atomic operations without careful memory alignment is unsafe.

3 Optimizations Attempted

3.1 Trial 1: Temporary Unsigned Integer Bins with Saturation

Idea: Instead of working directly on `uint8_t bins`, I allocated a temporary `unsigned int` array on the device. After the kernel execution, results were copied back to the host and saturated (clamped) at 255 before converting to `uint8_t`.

Result:

- `Setting up the problem... 0.020047 s`
- `Input size = 1,000,000, Number of bins = 4096`
- `Allocating device variables... 0.288025 s`
- `Copying data from host to device... 0.002727 s`
- `Launching kernel... 0.000356 s`
- `Copying data from device to host... 0.000025 s`
- `Verifying results... TEST PASSED`

3.2 Trial 2: Shared Memory per Block

Idea: Each block maintained a private copy of the histogram in `__shared__` memory to minimize global memory contention. After processing, each block merged its results into global memory with atomic operations.

Result:

- `Setting up the problem... 0.020564 s`
- `Input size = 1,000,000, Number of bins = 4096`
- `Allocating device variables... 0.197116 s`
- `Copying data from host to device... 0.001047 s`
- `Launching kernel... 0.000419 s`
- `Copying data from device to host... 0.000023 s`
- `Verifying results... TEST PASSED`

4 Observations and Analysis

During the experiments, I ran the histogram computation with an extremely large input size of approximately 1.78 billion elements. The number of bins was increased to 8192 to observe how it impacts various stages of execution.

4.1 Execution Timings

- **Problem Setup (input initialization)**: 38.10 seconds
- **Device Memory Allocation**: 0.18 seconds
- **Host to Device Memory Copy**: 1.52 seconds
- **Kernel Execution (Launching kernel)**: 0.57 seconds
- **Device to Host Memory Copy**: 0.000035 seconds
- **Verification**: TEST PASSED

4.2 Key Observations

1. Problem Setup The problem setup phase, where the input array is initialized on the host, took the majority of the time, about 38 seconds. This is expected because generating random integers for nearly two billion elements is a CPU-bound task and scales linearly with input size. Importantly, this step is independent of the number of bins used.

2. Device Memory Allocation Allocating memory for the input array and bins on the GPU was very fast, taking only about 0.18 seconds. Even though the number of bins was doubled from 4096 to 8192, the bins array remains relatively small (approximately 8 KB), so memory allocation did not become a bottleneck.

3. Host to Device Memory Copy Transferring the large input array from host to device took around 1.52 seconds. This copy time is primarily determined by the size of the input array and is not significantly affected by the number of bins.

4. Kernel Execution When using 8192 bins, the kernel execution time was approximately 0.57 seconds. Compared to previous runs with fewer bins, there was a noticeable but modest increase in execution time. This is understandable because a higher number of bins slightly increases the overhead of atomic operations and shared memory management inside the kernel. However, the increase was not drastic, and the kernel still performed efficiently even at this large bin count.

5. Device to Host Memory Copy Copying the final bins array back to the host remained extremely fast (essentially instantaneous). Since the bins array is small (only a few kilobytes), this phase does not contribute significantly to total runtime.

4.3 Conclusions

Overall, increasing the number of bins from 4096 to 8192 caused a slight increase in kernel execution time but did not meaningfully impact the other stages of the program. The dominant factor affecting overall runtime remains the input initialization on the host. The CUDA histogram kernel demonstrated excellent scalability and robustness even when handling very large datasets and a high number of bins. It can be concluded that using 8192 bins is both feasible and efficient for large-scale histogram computations on modern GPUs.

5 Conclusion

Through careful memory management and shared memory optimization, the histogram performance improved significantly. Avoiding misaligned memory operations was critical for correctness, and using shared memory further reduced the latency of atomic operations.

6 Resources explored

- F. Dehne, A. Fabri, and A. Rau-Chaplin, "Histogram Construction on GPUs: A Survey of Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 560-574, Mar. 2019. Available: https://ieeexplore.ieee.org/document/8067397

- **YouTube Video:** Efficient Histogram Calculation on GPU - CUDA Programming. Available: https://www.youtube.com/watch?v=FlmytqGjcrU

- **Article:** How We Built an AI CUDA Engineer - Sakana AI. Available: https://pub.sakana.ai/ai-cuda-engineer/