CPE 810 Lab 02

Zirui Wen

Question 1

Actually, I tried 4 methods that compare with the basic method; I will list them below:

- **Using Atomic Contention:** Atomic add refers to the use of intrinsic function, performing thread-safe addition on a shared variable without requiring locks. It ensures atomicity, preventing race conditions in multi-threaded environments.
- **Local Histogram in Shared Memory (Privatization)**: Initially, each thread performed atomic updates directly on the global histogram, which led to high contention and overhead. Switching to a strategy where each thread block accumulates a local histogram in shared memory, then later merges these results into the global histogram using atomic operations, significantly reduced global atomic contention. This change proved to be the most beneficial optimization.
- **Coarsening**: We can reduce the overhead of privatization via thread coarsening. In other words, we can reduce the number of private copies that are committed to the public copy by reducing the number of blocks and having each thread process multiple input elements.
- **Aggregation**: for each thread to aggregate consecutive updates into a single update if they are updating the same element of the histogram . Such aggregation reduces the number of atomic operations to the highly contended histogram elements, thus improving the effective throughput of the computation.

Here are the results:

We can see that, if we do not use atomic operation, the result may be wrong due to the conflict.

```
=== Performance Statistics ===
Total Operations: 0.30 billion operations
CPU Performance: 1.37 GFLOPS
Base Version Performance: 4.09 GFLOPS
Atomic Version Performance: 8.25 GFLOPS
Privatization Version Performance: 53.22 GFLOPS
Coarsened 1 Performance: 53.22 GFLOPS
Coarsened 2 Performance: 53.22 GFLOPS
Aggregation Version Performance: 53.21 GFLOPS
```

Figure 1: Algorithm performances

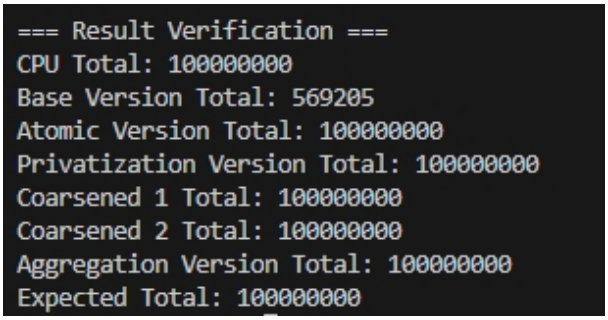

Figure 2: Result Verification

Question 2

Global Memory Reads:

- Each input element is read exactly once from global memory during kernel execution.

Global Memory Writes:

- The global memory writes for histogram bins occur only during the final combination step. For each block, there are at most M global writes (where $M$is the number of bins). So the number of global memory writes for histogram data is M/(threadsPerBlock) in total.

Question 3

Shared Memory Atomic Operations: For each input element, a corresponding atomic update is performed on the local (shared memory) histogram. This results in N atomic operations (where N is the number of input elements).

Global Memory Atomic Operations: At the end of kernel execution, each thread block updates the global histogram. If there are B thread blocks and M bins, then there are $B \times M$ atomic operations performed in the reduction phase.

In total: The kernel performs $N + (B \times M)$ atomic operations.

Question 4

If every element in the input array has the same value, every atomic update (both in shared and global memory) targets the same bin. This scenario leads to extremely

high contention, as all threads are attempting to update the same memory location concurrently.

With a random distribution of values, updates are spread more evenly across the available bins. If there are M bins, on average each bin receives about N/M updates, greatly reducing the likelihood of contention. The atomic operations become more distributed, which leads to improved performance.

Question 5

Theoretically, with fewer bins, many updates concentrate on a small number of memory locations, which results in severe contention and lower effective GFLOPS. Increasing the number of bins (to, say, 8, 16, or 32) spreads the updates over more bins, reducing atomic contention and generally improving performance. The optimal point is typically found when the balance between reduced contention and additional overhead in the reduction phase is achieved. Although contention is minimized when updates are spread very thinly, the overhead increases due to the larger shared memory allocation and more global atomic operations during the reduction phase. Additionally, resource usage (e.g., shared memory limits) might negatively impact occupancy. In practice, performance often peaks at a moderate bin count (e.g., around 64 or 128) and then degrades for very high bin numbers. In my test, when using the vectorsize of 1048576 and do the testing, I find something interesting. The picture shows the result:

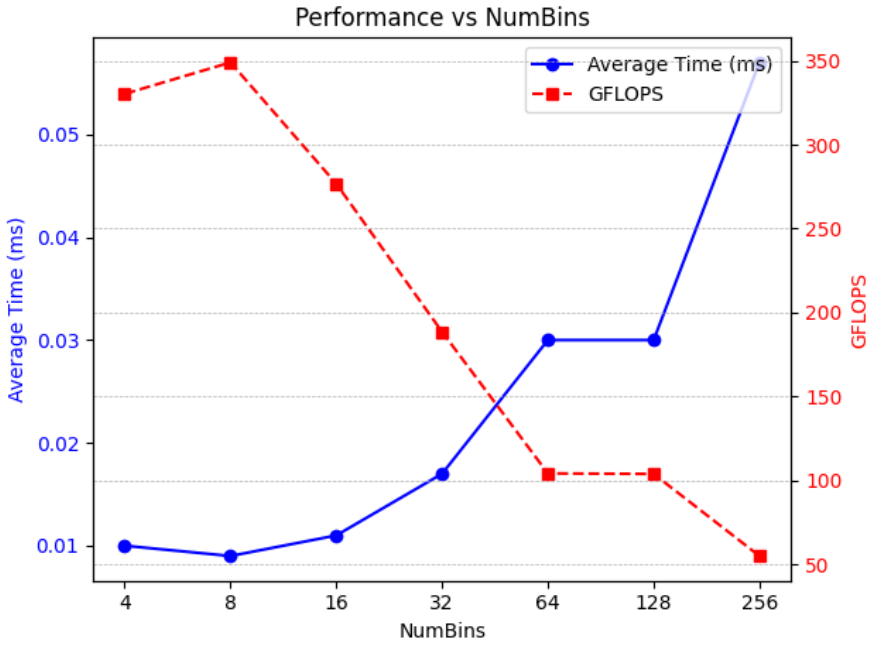

Figure 3: How the numbin influence the GFLOPS

As we can see, when using the numbin of 8, the GFLOPS comes to 350. But as the

number of bins becomes bigger, the GFLOPS decrease, which may not perform as we expected.

Question 6

Here are some methods that may solve the problem:

- Divide the massive dataset into smaller chunks that each fit within a single GPU's memory.
- Distribute the data chunks evenly across multiple GPUs. After processing, each GPU produces a local histogram. These histograms need to be merged to form the final result.
- Utilize CUDA streams to overlap data transfers and kernel execution. While one chunk is being processed, the next chunk can be loaded into memory, thus maximizing resource utilization and throughput.