Histogram Lab Report

The goal of this lab is to implement and optimize an efficient histogramming algorithm using CUDA for large input arrays.

The program leverages GPU's parallel computation capabilities to calculate histograms with 8-bit counters, ensuring performance optimization.

This lab also focuses on optimizing memory management, thread/block configurations, and verifying the correctness of results through GPU and CPU comparisons.

Optimizations and Performance

1. Optimizing Memory Management

- Objective: To enhance the efficiency of memory transfers between the host and device, reducing the time spent on copying data and ensuring coalesced reads for improved access speed.

- Challenges Faced: One of the main difficulties was ensuring the data transfer was optimized for larger input sizes without causing memory access issues. Balancing memory access and avoiding bottlenecks while dealing with large datasets was another challenge.

- Impact on Execution Time:

  - Before optimization: Copying data took approximately 0.244691 s.

  - After optimization: Reduced to 0.195822 s.

  - This reduction in data transfer time directly improved overall performance by speeding up the process of moving data to and from the GPU.

- Reasoning Behind Performance Improvement: By optimizing memory transfer and ensuring the memory accesses were coalesced, the time spent on data copying was minimized, allowing the kernel to execute more efficiently and reduce the overall processing time.

2. Optimizing Thread/Block Configuration

- Objective: The goal was to fine-tune the thread and block configurations to maximize GPU

resource utilization, ensuring a balanced workload across threads and minimizing idle time.

- Challenges Faced: The primary challenge was selecting the ideal block size that would optimize GPU usage without causing excessive overhead from idle or under-utilized threads. Determining the optimal configuration required multiple attempts and experimentation.

- Impact on Execution Time:

  - Before optimization: Kernel launch took around 0.000569 s.

  - After optimization: Reduced to 0.000228 s.

  - By adjusting the configuration, we were able to improve parallelization and decrease the idle times, thus optimizing the kernel launch process.

- Reasoning Behind Performance Improvement: Adjusting the thread/block configuration optimized the GPU's parallel execution capabilities. A better configuration meant that the GPU could process more tasks in parallel, reducing the overall execution time.

3. Verification of Results

- Objective: Ensured that the results computed by the GPU were correct by comparing them with CPU-based reference implementations, confirming the accuracy of the histogram calculation.

- Challenges Faced: One of the difficulties was ensuring that the GPU implementation correctly mirrored the CPU calculation, especially considering boundary conditions and large input data sizes.

- Impact on Execution Time: The verification process did not significantly impact execution time as it was a relatively quick check.

- Reasoning Behind Performance Improvement: While verification did not reduce execution time, it was essential to ensure that the optimizations applied did not affect the correctness of the results. Confirming the correctness of the GPU implementation helped prevent any potential performance degradation due to incorrect processing.

Testing and Results

Test Case 1:

- Input Size: 1,000,000
- Number of Bins: 4096
- Execution Times:
  - Setup: 0.027418 s
  - Allocation: 0.244691 s
  - Data Copy: 0.003378 s
  - Kernel: 0.000569 s
  - Verification: Test Passed

Test Case 2:

- Input Size: 50,000
- Number of Bins: 4096
- Execution Times:
  - Setup: 0.001267 s
  - Allocation: 0.224315 s
  - Data Copy: 0.002646 s
  - Kernel: 0.000228 s
  - Verification: Test Passed

Test Case 3:

- Input Size: 50,000
- Number of Bins: 1024
- Execution Times:
  - Setup: 0.001025 s
  - Allocation: 0.195822 s
  - Data Copy: 0.000261 s
  - Kernel: 0.000180 s
  - Verification: Test Passed

Conclusion

The CUDA-based histogram computation was successfully optimized for performance by improving memory management,
tuning thread/block configurations, and ensuring result correctness through verification. Further optimizations can focus on
using shared memory and exploring more advanced techniques for optimizing larger datasets.

Future work may include refining memory access patterns, experimenting with different block/grid configurations,
and investigating more complex histogram computation strategies to further improve execution times for larger input sizes.