Parallel Computing Project

Parallel Selection on GPUs

Name: Shredan abdullah kamal
ID: 9202707
Sec: 1
BN: 33

Report

INTRODUCTION:

Selection, also known as stream compaction or filtering, is a common programming concept that has a wide range of applications in area of statistics, database software, artificial intelligence, image processing, and simulations. It produces a smaller output array, containing only wanted elements from the input array.

Implementation Details:

The implementation of a GPU-based filtering and compaction algorithm using CUDA and the Thrust library. The primary goal is to filter elements in an array based on a predicate condition and compact the results into a final output array. Additionally, the implementation utilizes CUDA streams for efficient data transfer and execution overlap.

CUDA Kernels:

filter_kernel

The filter_kernel function is responsible for applying the predicate condition to each element in the input array. It uses shared memory to load data in tiles for efficient access.

compact_kernel

The compact_kernel function compacts the filtered results by removing invalid entries (marked as -1). This kernel also uses shared memory to store valid elements temporarily and then writes them to the final output array using atomic operations.

Streaming: Two CUDA streams are created for managing asynchronous operations.

The input data is copied from host to device asynchronously using stream1. This allows the GPU to perform other tasks concurrently with this data transfer.

Thrust is used to sort the data on the GPU. The sorting operation is associated with stream1.

The filter_kernel is launched to filter data based on a predicate condition. This kernel is executed in stream1.

The dev_count variable, used to keep track of the number of valid elements, is initialized to 0 using stream2.

The compact_kernel is launched to compact the filtered results. This kernel is executed in stream2, potentially overlapping with other operations in stream1.

The final output array and the count of valid elements are copied back to the host asynchronously using stream2.

In short

stream1 handles sorting and filtering operations.

stream2 initializes the count and handles compacting the filtered results and copying the final data back to the host.

- For the CPU implementation.
  The CPU implementation uses the qsort function to sort the final output array, which has a time complexity of **O(n log n)**.
  Both filtering and compacting operations iterate through the input data once, which has a time complexity of **O(n)**.
  Overall Time Complexity: O(n log n) + O(n), which simplifies to O(n log n).
  The CPU implementation has a time complexity of **O(n log n).**
- For the GPU counterpart, the theoretical benchmark depends on the specific operations performed on the GPU and their respective complexities. Where the sorting operation using Thrust library has a time complexity of **O(n log n)**, and both filtering and compacting operations have a time complexity of **O(1).**

Therefore, the overall time complexity for the GPU implementation is also O(n log n) + O(1), which simplifies to **O(n log n)**. Similar to the CPU implementation

Input Size	CPU	GPU	Speedup(CPU/GPU)
100	0.00 milliseconds	970.168335 milliseconds	0
1 000 000	12.31 milliseconds	3.840416 milliseconds	3.205
2 000 000	17.22 milliseconds	4.639904 milliseconds	3.711
10 000 000	96.63 milliseconds	10.073024 milliseconds	9.592

Speedup= 10.073024 milliseconds/96.63 milliseconds≈ 10

the speedup of the GPU over the CPU is approximately 10, when the input size array is 10 000 000

Theoretical Speedup= N/M, where the GPU has N processing elements (cores) and the CPU has M cores.

Theoretical Speedup= 256/4 = 64

Comparison to Theoretical Speedup:
Theoretical Speedup= N/M, where the GPU has N processing elements (cores) and the CPU has M cores. the number of cores in the GPU=256, the number of cores in the CPU=4, so the theoretical maximum speedup would be 64x.
the observed speedup was found to be 10x, likely due to overheads such as memory
transfers and the fact that not all parts of the algorithm can be perfectly parallelized.
Explanation for Lower than Theoretical Speedup:
If a significant portion of the task cannot be parallelized, the overall speedup will be limited, regardless of the number of cores.

In this case, tasks such as data transfers between host and device, kernel launch overhead, and synchronization are inherently sequential and limit the achievable speedup.

Transferring data between the CPU and GPU memory is relatively slow compared to computation. The cudaMemcpyAsync calls to transfer data to and from the GPU introduce latency that does not benefit from parallel processing.

Atomic operations used in compact_kernel to count valid elements and to write them back to global memory can serialize parts of the computation, limiting parallel efficiency.