Parallel Computing Project Proposal

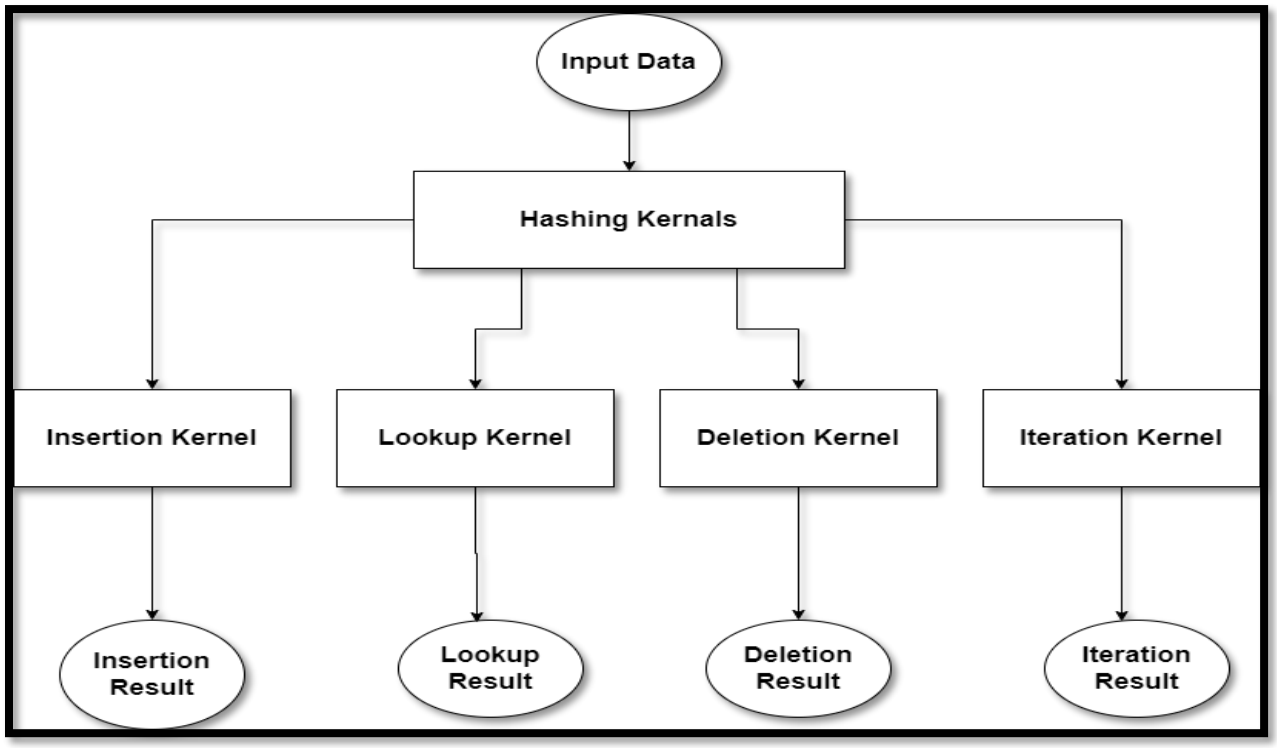
Name	Sec	B. N	ID
Karim Mahmoud Kamal	2	12	9203076
Mustafa Mahmoud Hamada	2	25	9203519

Submitted To:

Dr: Dina Tantawy

Eng: Muhammed Abdullah

Overview:



Insert Kernal:

This CUDA kernel function inserts key-value pairs into a hash table using multiple threads for parallel processing. Each thread loads a key-value pair and computes its insertion location using a hash function. The thread then attempts to insert the key into the calculated location using an atomic compare-and-swap (CAS) operation to ensure thread safety. If the location is empty or already contains the same key, the value is inserted or updated. If the location is occupied by a different key, the thread uses **linear probing**, incrementing the insertion location until it finds an empty slot. This process ensures efficient handling of collisions in the hash table.

Lookup Kernal:

This CUDA kernel function performs parallel searches for key-value pairs in a hash table. Each thread processes one key from the target array, using a hash function to compute the initial search location. The thread checks if the key at this location matches the target key. If it does, the corresponding value is retrieved and stored in the target array. If the location is empty, it means the key is not present in the hash table, and the target value is set to indicate this. If the key does not match and the slot is occupied, the thread uses linear probing to check subsequent locations until it either finds the key or confirms its absence. This approach efficiently handles collisions and ensures accurate search results.

Deletion Kernal:

This CUDA kernel function removes key-value pairs from a hash table. Each thread processes one key from the array of pairs that we need to remove. The thread searches for the key, and if it finds a match, it sets the corresponding value to a special empty value, indicating deletion. If the slot is already empty, the key is not present, and the thread exits. If the slot contains a different key, the thread uses linear probing to check subsequent slots until it finds the key or an empty slot. Deleted keys are not removed from the table; only their values are set to -1 to indicate deletion.

Iteration Kernal:

This CUDA kernel function fetches all non-empty pairs from the hash table. Each thread examines one slot in the table. If the slot contains a key that is not marked as empty and a corresponding value that is also not empty, the thread increments a shared counter atomically to ensure thread safety and assigns the key-value pair to the result array at the position indicated by the counter. This way, the function collects all valid pairs from the hash table while maintaining correct indexing despite concurrent access by multiple threads.

Experiments and results

We conducted several experiments comparing the performance of a hash table on both CPU and GPU with the same data size. The results showed a significant reduction in time when using the GPU. For example, when inserting 8 million items, the hash table operation took approximately 1075 ms on the CPU, whereas it only took 27 ms on the GPU. This represents a 96% reduction in time when using the GPU.

	Test size	CPU	GPU
Inserting	500K	50.2296 ms	1.007712 ms
	1 M	115.968 ms	2.657984 ms
	2 M	241.133 ms	6.077312 ms
	4 M	838.339 ms	12.815264 ms
	6 M	920.558 ms	20.253023 ms
	8 M	1074.77 ms	26.302879 ms
Searching	500K	62.3925 ms	0.276448 ms
	1 M	109.873 ms	0.719136 ms
	2 M	244.819 ms	1.657504 ms
	4 M	923.818 ms	3.620416 ms
	6 M	899.3 ms	5.987744 ms
	8 M	1032.13 ms	7.559488 ms
Removing	500K	50.7158 ms	0.556704 ms
	1 M	126.48 ms	1.524832 ms
	2 M	240.978 ms	3.462880 ms
	4 M	317.728 ms	7.364352 ms
	6 M	902.104 ms	11.278336 ms
	8 M	1078.5 ms	15.118496 ms
Fetching	500K	36.1485 ms	0.083968 ms
	1 M	73.2689 ms	0.157472 ms
	2 M	167.999 ms	0.307200 ms
	4 M	533.854 ms	0.589792 ms
	6 M	455.583 ms	0.817120 ms
	8 M	659.422 ms	1.175648 ms

Performance analysis

Theoretical CPU benchmarks

Hardware Specifications

CPU

Model: Intel(R) Xeon(R) CPU @ 2.20GHz

Cores: 2

Clock Speed: 2.20 GHz (2199.998 MHz)

Cache Size: 56320 KB

GPU (Colab Free Edition)

Typically, the free edition of Google Colab provides access to NVIDIA Tesla K80 GPUs.

CUDA Cores: 2496

Base Clock Speed: 560 MHz

Memory Bandwidth: 240 GB/s

CPU Code:

Our CPU code for hash table operations (insert, search, remove, and fetch) uses linear probing for collision handling. The average-case complexity for each operation is O(1), assuming a good hash distribution and load factor. However, in the worst-case scenario, the complexity can degrade to O(n).

GPU Code

Our GPU implementation follows the same approach but leverages parallelism, the time complexity remains O(1) for average-case operations but can significantly benefit from parallel execution.

CPU Theoretical Performance:

- **Clock Speed**: 2.20 GHz
- **Operations per Cycle**: Let's assume the CPU can perform 4 operations per cycle.
- **Total Operations per Second**: 2.20 GHz × 2 cores × 4 operations per cycle = 17.6 GFLOPS

For inserting 8 million items:

Operations: Assuming each insertion involves a few operations (hashing, comparison, assignment), let's estimate around 20 operations per insertion.

Total Operations: 8 × 10^6 × 20 =160×10^6 operations

Time: (160×10^6 operations) / (17.6×10^9 operations/second)=0.009 seconds=9 ms

GPU Theoretical Performance

- **CUDA Cores:** 2496
- **Clock Speed:** 560 MHz
- **Operations per Cycle per Core:** assume 1 operation per cycle per core.
- **Total Operations per Second:** 560 MHz×2496 cores=1.397 TFLOPS

For inserting 8 million items:

- **Operations:** Assuming the same number of operations per insertion (20 operations).
- **Total Operations:** 8×10^6×20=160×10^6
- **Time:** (160×10^6 op) / (1.397×10^12 op/second) ≈0.00011 seconds=0.11 ms

Actual benchmarks

From our provided experiments:

CPU Insertion Time for 8 million items: 1075 ms

GPU Insertion Time for 8 million items: 27 ms

Speedup: 1075 / 27 ≈ 39.81

Comparison to Theoretical Speedup

Based on the theoretical estimates (9 ms for CPU and 0.11 ms for GPU), the theoretical speedup could be around (9 /0.11) ≈ 81.82

If your speedup is below the theoretical one (this is mostly the case), how do you explain this, what could be changed to achieve a better one?

The actual speedup (39.81) is below the theoretical speedup (81.82) due to several factors:

1. **Memory Transfer Overheads**: Transferring data between the CPU and GPU can be time-consuming.
2. **Kernel Launch Overheads**: There is an inherent overhead in launching GPU kernels.
3. **Hardware Utilization**: Ensuring full utilization of the GPU cores can be challenging.

Potential Improvements

1. **....**
2. **....**

Summary of the performance analysis:

Theoretical CPU Insertion Time: 9 ms

Actual CPU Insertion Time: 1075 ms

Theoretical GPU Insertion Time: 0.11 ms

Actual GPU Insertion Time: 27 ms

Actual Speedup: 39.81

Theoretical Speedup: 81.82