Optimizing Performance: Unveiling the Power of Single-Threaded, Multi-Threaded and GPU Programming in C/C++

A Project By:

Dhruv Thanawala

Preface:

This Project Demonstrates how Parallelization of different data structures and Algorithms can affect a program. By the end of this, you will be able to understand the basics of threads and will know where and where not to implement it. I have implemented data structures and algorithms to clearly distinguish performance. All Codes have been timed and the effects of parallelism can clearly be seen.

Parallelism might be strong but comes with some problems. The code becomes complex. There is an overhead in launching every thread and in some cases it might just be better to just use a single thread. While writing parallel code, we have to keep memory safety and thread synchronization in mind. While there are some problems that come with writing multithreaded code, it can significantly increase the speed of your code, if used correctly. Parallelization can be achieved using a CPU and GPU with multiple cores. In this project I have used the ***CPU:'Intel 12th Gen Intel(R) Core(TM) i7-12700H'*** *and* ***GPU:'NVIDIA GeForce RTX 3050 Laptop GPU'****. The programming languages used are:* ***C****,* ***C++*** *and* ***CUDA****.*

Introduction:

What is concurrency?

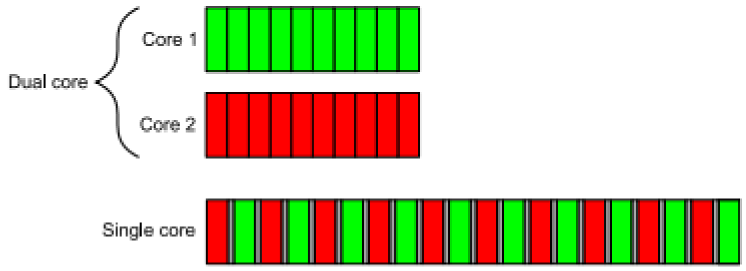
At the simplest and most basic level, concurrency is about two or more separate activities happening at the same time. We encounter concurrency as a natural part of life; we can walk and talk at the same time or perform different actions with each hand, and of course we each go about our lives independently of each other—you can watch football while I go swimming, and so on.

Concurrency in computer systems:

When we talk about concurrency in terms of computers, we mean a single system performing multiple independent activities in parallel, rather than sequentially, or one after the other. It isn't a new phenomenon: multitasking operating systems that allow a single computer to run multiple applications at the same time through task switching have been commonplace for many years, and high-end server machines with multiple processors that enable genuine concurrency have been available for even longer. What is new is the increased prevalence of computers that can genuinely run multiple tasks in parallel rather than just giving the illusion of doing so.

Historically, most computers have had one processor, with a single processing unit or core, and this remains true for many desktop machines today. Such a machine can really only perform one task at a time, but it can switch between tasks many times per second. By doing a bit of one task and then a bit of another and so on, it appears that the tasks are happening concurrently. This is called task switching. We still talk about concurrency with such systems; because the task switches are so fast, you can't tell at which point a task may be suspended as the processor switches to another one. The task switching provides an illusion of concurrency to both the user and the applications themselves. Because there is only an illusion of concurrency, the behavior of applications may be subtly different when executing in a single-processor task-switching environment compared to when executing in an environment with true concurrency.

Computers containing multiple processors have been used for servers and high performance computing tasks for a number of years, and computers based on processors with more than one core on a single chip (multicore processors) are common as desktop machines too. Whether they have multiple processors or multiple cores within a processor (or both), these computers are capable of genuinely running more than one task in parallel. We call this hardware concurrency



Two approaches to concurrency: parallel execution on a dual-core machine versus task switching on a single-core machine

GPU and CUDA:

A GPU, or Graphics Processing Unit, is a specialized processor designed to accelerate the rendering of graphics and visual effects. It excels at parallel processing, handling multiple tasks simultaneously, making it particularly efficient for graphics-related computations. Beyond graphics, modern GPUs are increasingly used for general-purpose computing tasks, such as scientific simulations and machine learning, thanks to their parallel processing capabilities.

In comparison to the central processor's traditional data processing pipeline, performing general-purpose computations on a graphics processing unit (GPU) is a new concept. In fact, the GPU itself is relatively new compared to the computing field at large. However, the idea of computing on graphics processors is not as
new as you might believe.

CUDA, or Compute Unified Device Architecture, is a parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the computational power of NVIDIA GPUs for general-purpose computing tasks. CUDA allows programmers to write code that can be executed on NVIDIA GPUs, facilitating parallel processing and accelerating applications in fields like scientific computing, artificial intelligence, and data analysis. It is a language based on C and C++ and follows their conventions.

The CUDA Architecture, in contrast to its predecessors, departed from the conventional partitioning of computing resources into vertex and pixel shaders. It introduced a unified shader pipeline, enabling each arithmetic logic unit (ALU) on the chip to be orchestrated by a program for general-purpose computations. NVIDIA, envisioning these graphics processors for broader computational use, designed the ALUs to adhere to IEEE standards for single-precision floating-point arithmetic. Unlike previous architectures tailored primarily for graphics, the ALUs in the CUDA Architecture were equipped with an instruction set optimized for general computation. Notably, the GPU's execution units were granted unrestricted read and write access to memory, alongside the utilization of a software-managed cache known as shared memory. These enhancements aimed to transform the GPU into a powerhouse for both general computation and traditional graphics tasks.

Understanding Threads:

1. ***Concurrency vs. Parallelism:***
    - Concurrency involves multiple tasks making progress without necessarily executing simultaneously.

    - Parallelism is the simultaneous execution of multiple tasks.

2. ***Threads:***
    - Threads are the smallest units of execution in a program.
    - They share the same memory space but have their own registers and stacks.

3. ***Thread Safety:***

    - Ensuring data structures and algorithms can be safely accessed and modified by multiple threads is crucial.
    - Synchronization mechanisms (locks, semaphores) prevent data corruption.

Parallelization of Data Structures:

1. ***Arrays and Lists:***
    - Parallelizing operations on arrays and lists is straightforward.
    - Element-wise operations, filtering, and mapping can be effectively parallelized.

2. ***Linked Lists:***
    - Due to their sequential nature, parallelizing operations on linked lists is challenging.
    - Insertion, deletion, and traversal may not benefit much from parallelization.

<u>Parallelization of Algorithms:</u>

1. ***Sorting:***
    - Parallel sorting algorithms, like parallel quicksort or mergesort, can significantly improve performance.
    - Load balancing is crucial to ensure each thread has an equal workload.

2. ***Searching:***
    - Binary searches and divide-and-conquer search algorithms can be parallelized.
    - Managing shared data structures, like search trees, requires careful consideration.

3. ***Graph Algorithms:***
    - Parallelization of graph algorithms depends on the algorithm's nature.
    - Parallel breadth-first search or components of algorithms like Dijkstra's can be effective.

4. ***Machine Learning Algorithms:***
    - Iterative processes in machine learning algorithms can be parallelized.
    - Parallelizing gradient descent or matrix operations in neural networks can lead to significant speedups.

Considerations for Parallelization:

1. ***Task Dependency:***
   - Identify dependencies between tasks to avoid conflicts and ensure tasks are completed in the correct order.

2. ***Granularity:***
   - Determine the appropriate level of granularity for parallel tasks to balance overhead and resource utilization.

3. ***Load Balancing:***
   - Distribute the workload evenly among threads to optimize performance.

4. ***Data Sharing and Synchronization:***
   - Use synchronization mechanisms to ensure thread safety and minimize the need for locks.

Where Not to Implement Parallelization:

1. ***Sequential Algorithms:***
   - Some algorithms inherently have a sequential nature and may not benefit from parallelization.

2. ***Small Datasets:***
   - For small datasets, parallelization may introduce more overhead than performance gains.

3. ***Limited Resources:***
   - On systems with limited resources, parallelization may not yield significant improvements.

<u>**Project Implementation:**</u>

1. Binary Search Tree and Hash Table:

a. Binary Search Tree (BST):

Insertion:

Algorithm: Recursive insertion in a BST.

Time Complexity: O(log n) on average, O(n) in the worst case (for unbalanced trees).

Implementation: CPU-based recursive approach.

Searching:

Algorithm: Recursive search in a BST.

Time Complexity: O(log n) on average, O(n) in the worst case (for unbalanced trees).

Implementation: CPU-based recursive approach.

b. Hash Table:

Insertion:

Algorithm: Using a hash function to determine the index and handling collisions.

Time Complexity: O(1) on average (constant time), but may vary with collisions.

Implementation: CPU-based hash table implementation.

Searching:

Algorithm: Using a hash function to determine the index and handling collisions during search.

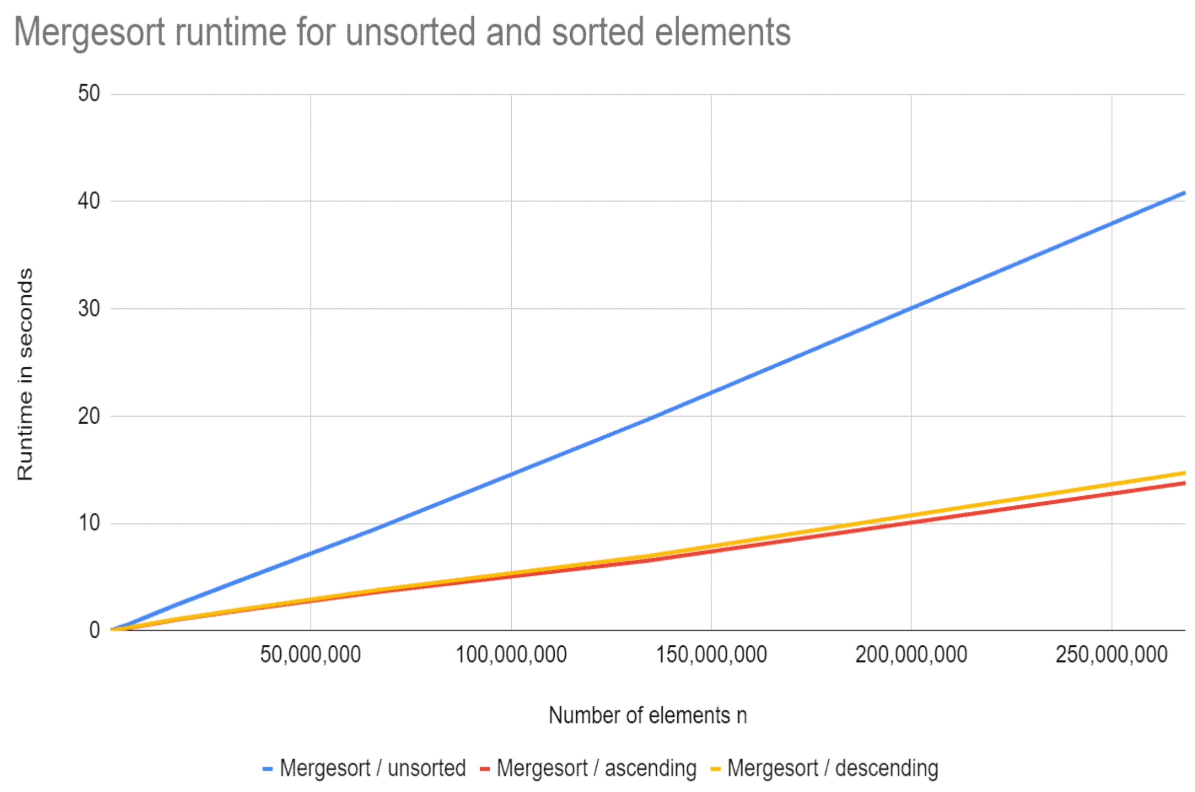
Time Complexity: O(1) on average (constant time), but may vary with collisions.

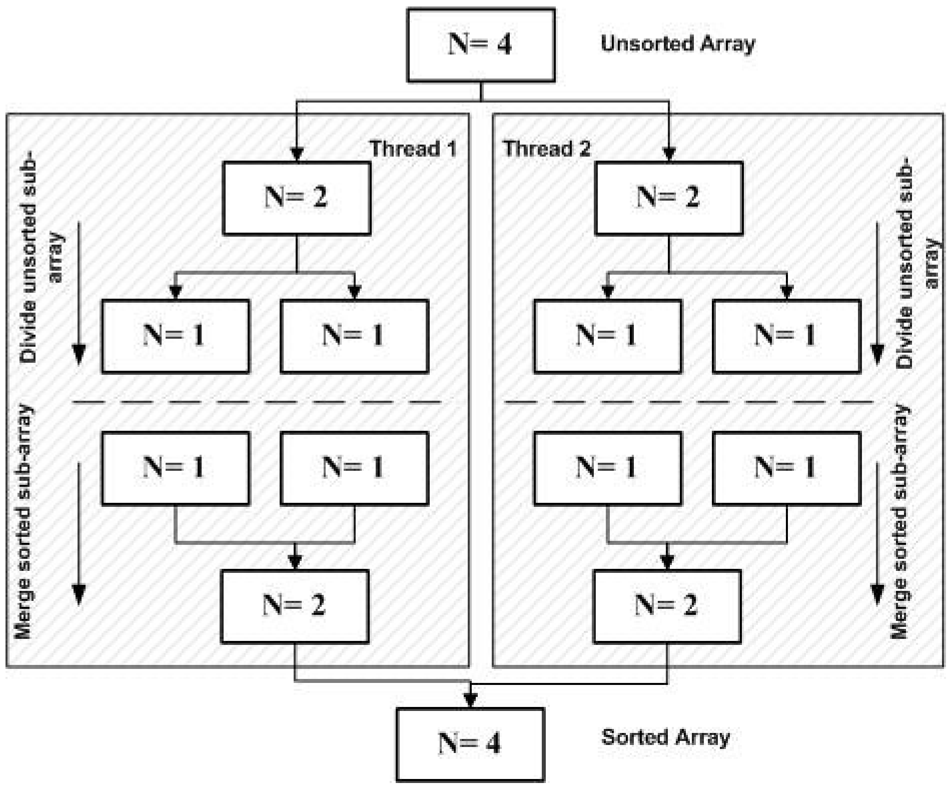
Implementation: CPU-based hash table implementation.

2. **Merge Sort Single Threaded vs Multi Threaded**

Aspect	Single-threaded Merge Sort	Multi-threaded Merge Sort
Time Complexity	O(n log n)	O(n log n)
Performance	Similar for small to moderately sized arrays.	Potential improvement on multi-core systems for large arrays.
Factors Influencing Performance	Array size, characteristics of data, hardware architecture.	Array size, number of available cores, thread management overhead.
Parallelization	Not applicable.	Utilizes parallelism on multi-core systems.
Potential Speedup	Limited by single core.	Depends on available cores and array size.
Overheads	Minimal	Thread synchronization, management, potential contention

Trade-offs	Simplicity of implementation	Potential speedup on large arrays vs. overhead and complexity of parallelization





Merge sort implementation in multiple threads

3. *Merge Sort vs Bitonic Sort*

a. Merge Sort:

Merge Sort is a popular divide-and-conquer sorting algorithm that efficiently sorts an array by recursively dividing it into two halves, sorting each half, and then merging the sorted halves to produce the final sorted array.

b. Bitonic Sort:

Bitonic Sort is an efficient parallel sorting algorithm that requires the input size to be a power of 2. It is based on the concept of bitonic sequences, which are sequences that first monotonically increase and then monotonically decrease or vice versa. The algorithm recursively builds a bitonic sequence, and then repeatedly merges bitonic sequences to achieve sorting.

Data Structure Used: Arrays
Time Complexity:
*Merge Sort:*_ O(nlogn)
Bitonic Sort: O(log^2 n)

Implementation:

- *CPU Merge Sort:* This is a traditional CPU-based implementation of the Merge Sort algorithm using a recursive approach.

- *GPU Merge Sort:* The GPU version of Merge Sort that uses CUDA to achieve parallelism. It utilizes CUDA kernels to perform sorting operations on the GPU.

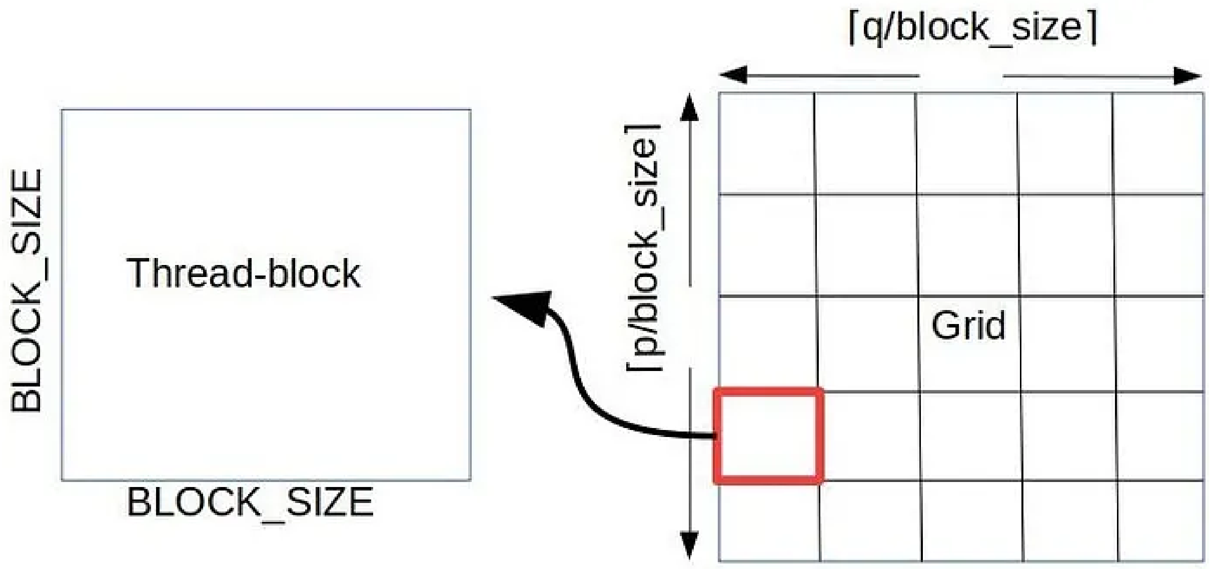
- *CPU Bitonic Sort:* A CPU-based implementation of the Bitonic Sort algorithm. It requires the input size to be a power of 2.

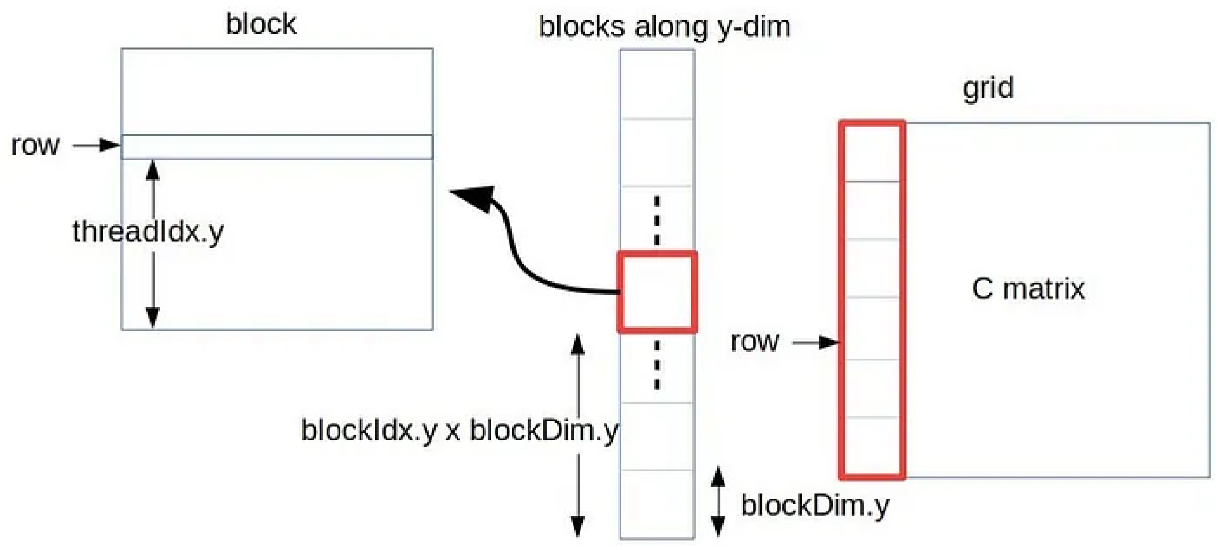
- *GPU Bitonic Sort:* The GPU version of Bitonic Sort that takes advantage of CUDA parallelism. Like the GPU Merge Sort, it uses CUDA kernels for sorting on the GPU.

4. Matrix Multiplication

In CUDA programming model threads are organized into thread-blocks and grids. Thread-block is the smallest group of threads allowed by the programming model and grid is an arrangement of multiple thread-blocks. If you are unfamiliar with thread-blocks and grid, refer to this. A thread-block or grid can be arranged in 1-D, 2-D or 3-D.

Sine we are multiplying 2-D matrices it only makes sense to arrange the thread-blocks and grid in 2-D. In most modern NVIDIA GPUs one thread-block can have a maximum of 1024 threads. Therefore we can use a 32 x 32 2-D thread-block (Let's assume that our thread-block size is BLOCK_SIZE x BLOCK_SIZE from here). Now how should we arrange our grid? Since the output matrix is p × q, we need to have at least ⌈p/32⌉ number of thread-blocks in y-dimension and ⌈q/32⌉ number of thread-blocks in x-dimension.



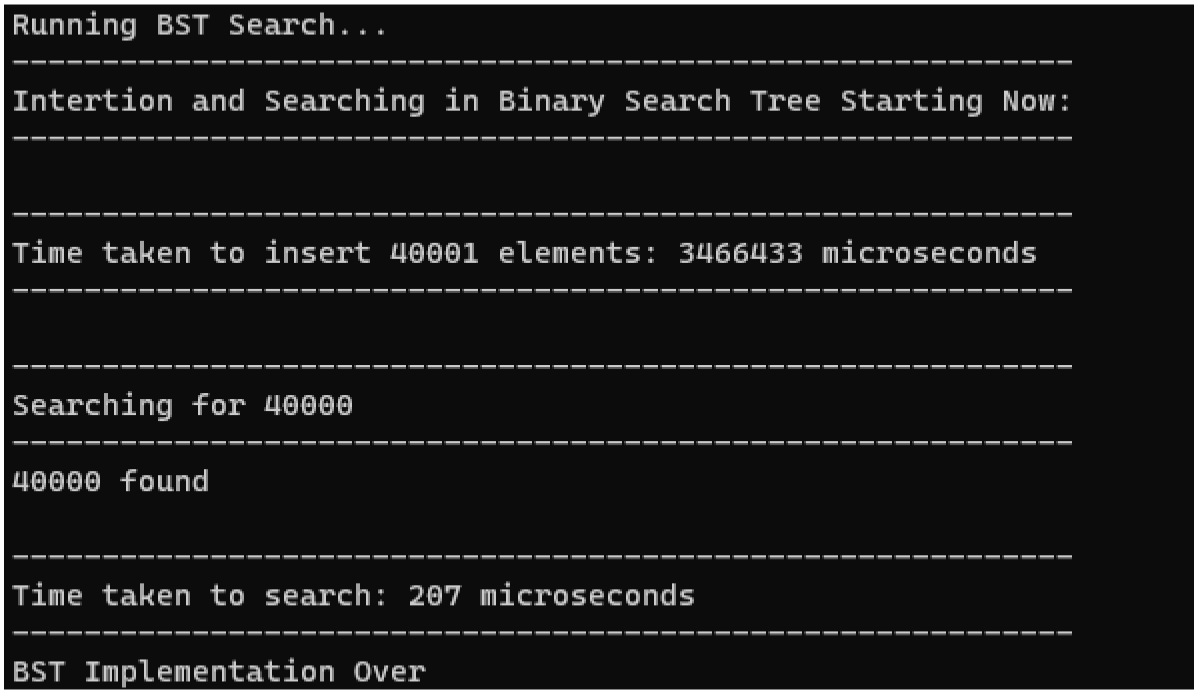


So block and grid dimension can be specified as follows using CUDA. Here I assumed that columns in the matrix are indexed in ***x***-dimension and rows in ***y***-dimension. So ***x***-dimension of the grid will have **⌈*q/32*⌉** blocks.

Speed Test Results:

1. Binary Search Tree and Hash Table:

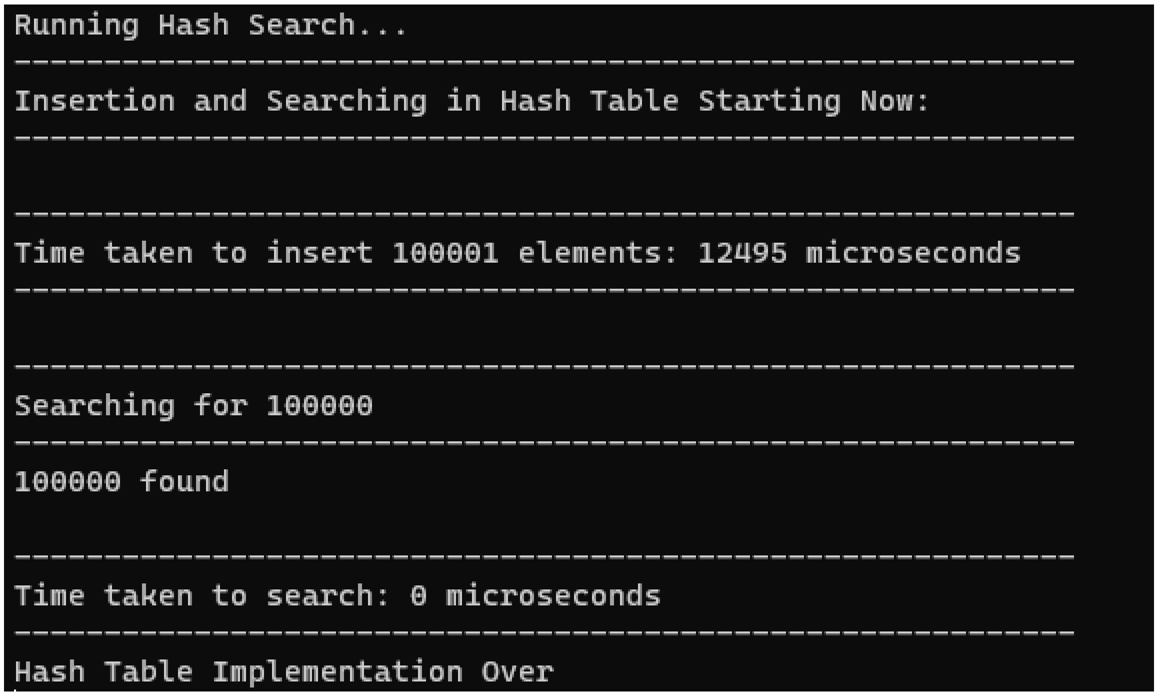
Binary Tree:
40001 random numbers have been inserted in the binary tree.
A search operation is being performed for the number 40000.
Insertion and Searching have been timed.



```
Running BST Search...
-----------------------------------------------------------
Intertion and Searching in Binary Search Tree Starting Now:
-----------------------------------------------------------


-----------------------------------------------------------
Time taken to insert 40001 elements: 3466433 microseconds
-----------------------------------------------------------


-----------------------------------------------------------
Searching for 40000
-----------------------------------------------------------
40000 found

-----------------------------------------------------------
Time taken to search: 207 microseconds
-----------------------------------------------------------
BST Implementation Over
```

Hash Table:

100001 random numbers have been inserted in the Table with the hey as numbers and value as boolean ‘TRUE’.
A search operation is being performed for the number 100000.
Insertion and Searching have been timed.



```
Running Hash Search...
------------------------------------------------------------
Insertion and Searching in Hash Table Starting Now:
------------------------------------------------------------


------------------------------------------------------------
Time taken to insert 100001 elements: 12495 microseconds
------------------------------------------------------------


------------------------------------------------------------
Searching for 100000
------------------------------------------------------------
100000 found

------------------------------------------------------------
Time taken to search: 0 microseconds
------------------------------------------------------------
Hash Table Implementation Over
```

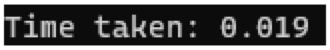
Result:
Hash Table came out to be faster in searching and Insertion operations.

Reason:
Hash Table uses indexing for searching and insertion operation while in Binary tree, we will have to traverse tree nodes till we find the value.

2. ***Merge Sort Single Threaded vs Multi Threaded***

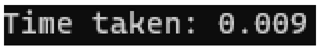
A 100 thousand random numbers have been inserted in an array using a single thread for both cases.

Single Threaded Merge Sort:



```
Time taken: 0.019
```

Multi Threaded Merge Sort:



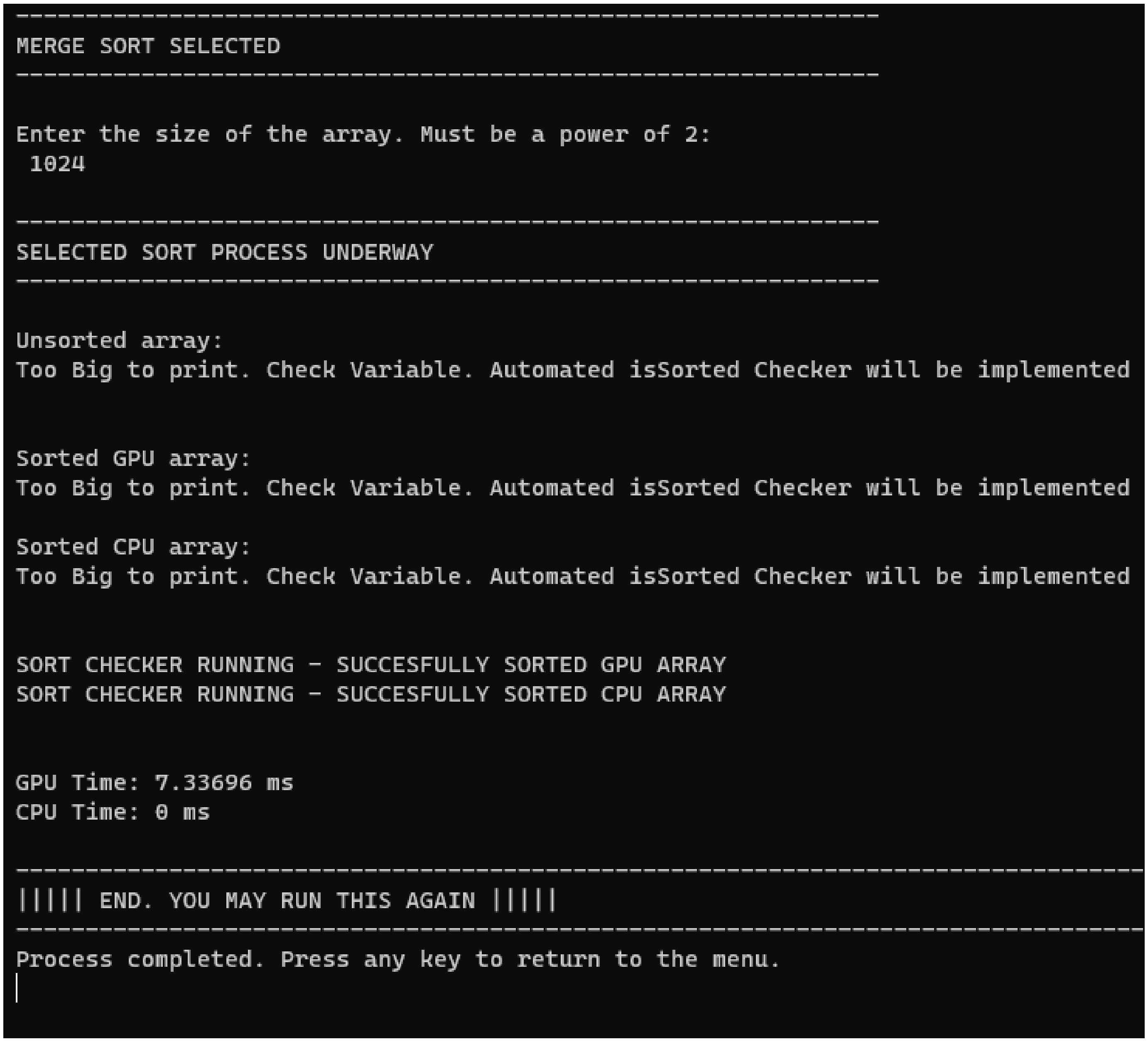
```
Time taken: 0.009
```

Result:
Multi Threaded Merge Sort is Significantly Faster than Single Threaded Merge Sort.

Reason:
Merge Sort is an embarrassingly parallel algorithm. More the threads, the faster it gets. If there were lesser values, the overhead cost would not be worth it and single threaded merge sort would have been faster.

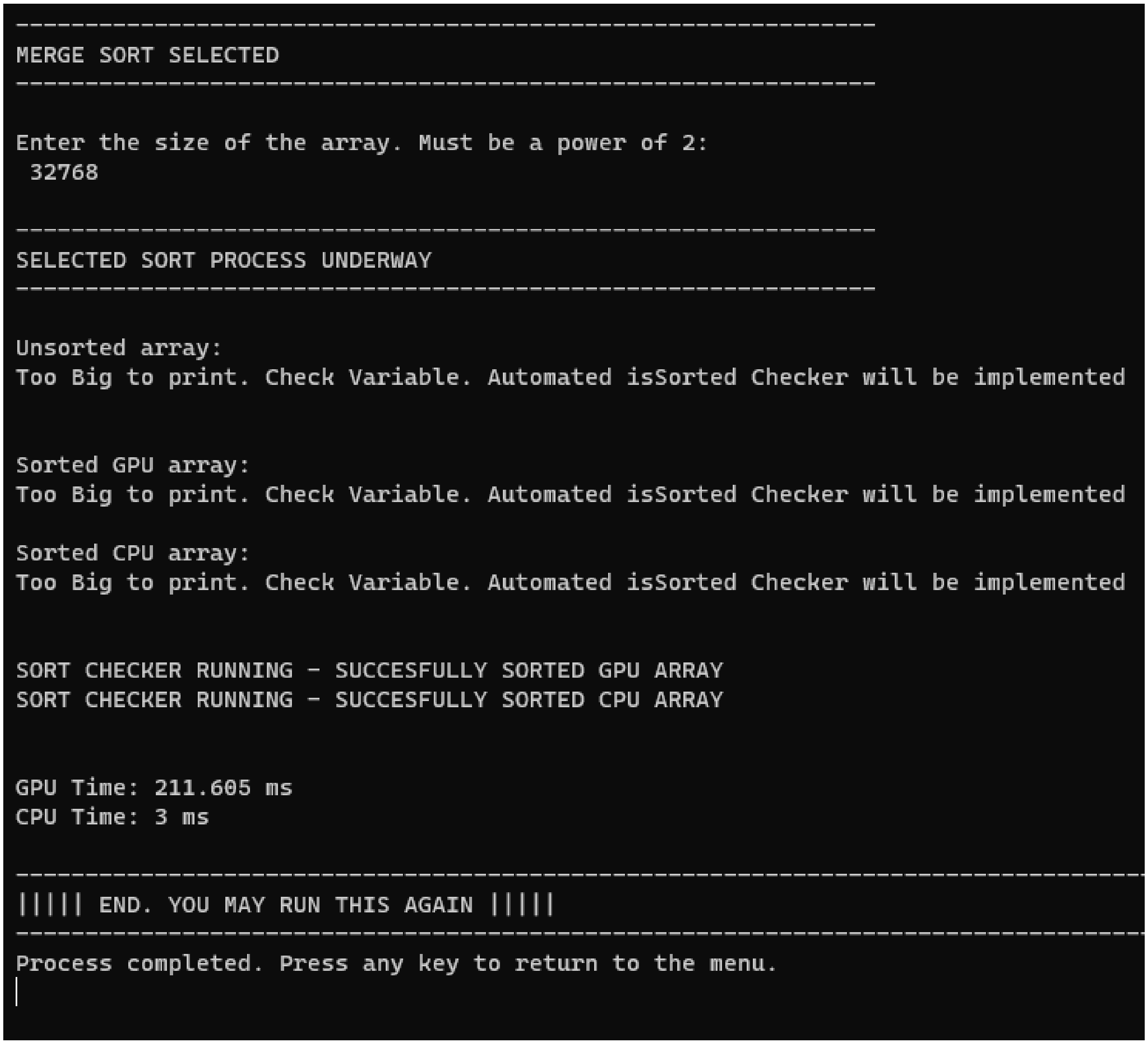
3. Merge Sort vs Bitonic Sort

Merge Sort Case 1:



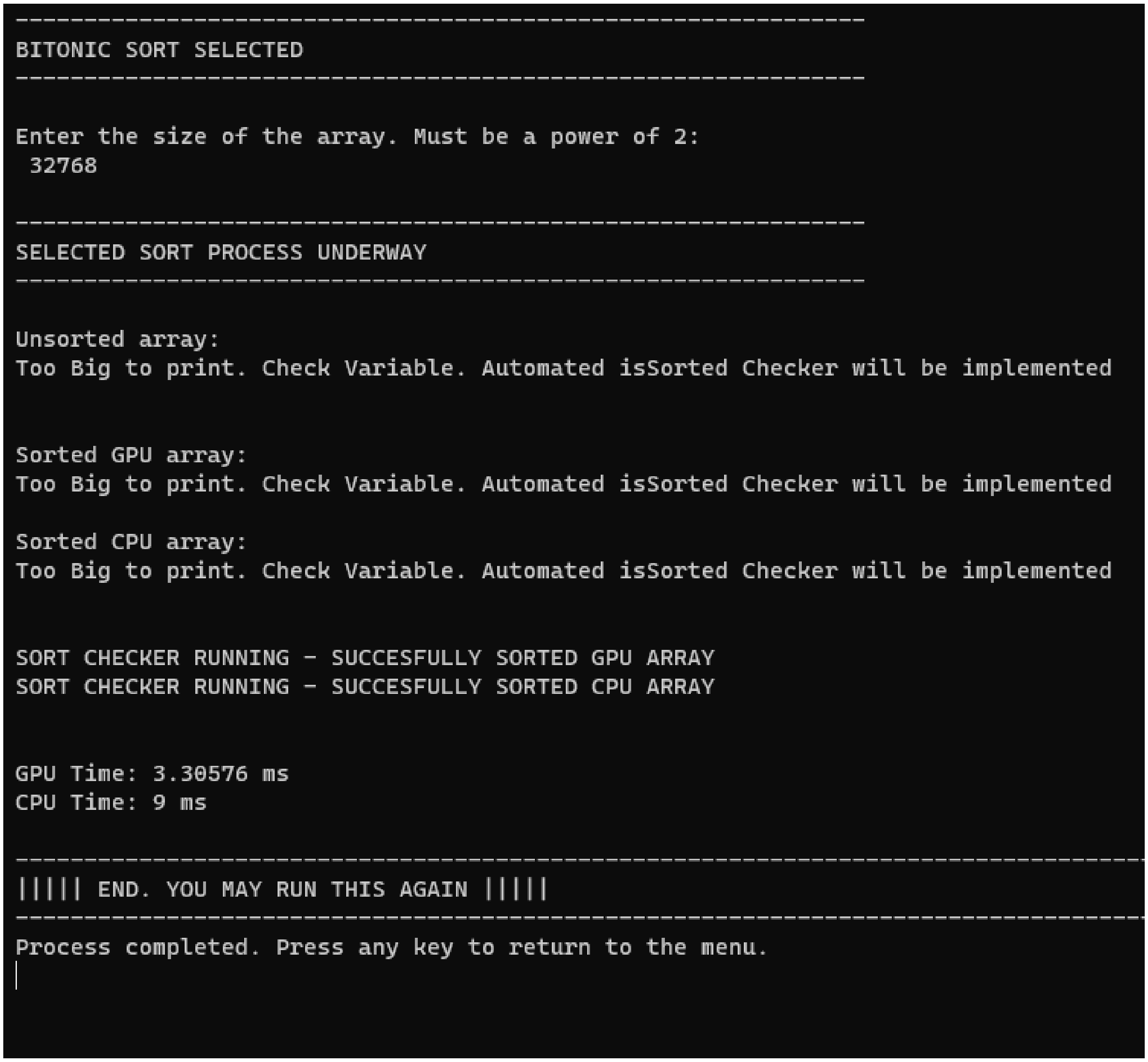
```
--------------------------------------------------------------
MERGE SORT SELECTED
--------------------------------------------------------------


Enter the size of the array. Must be a power of 2:
 1024

--------------------------------------------------------------
SELECTED SORT PROCESS UNDERWAY
--------------------------------------------------------------


Unsorted array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


Sorted GPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented

Sorted CPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


SORT CHECKER RUNNING - SUCCESFULLY SORTED GPU ARRAY
SORT CHECKER RUNNING - SUCCESFULLY SORTED CPU ARRAY


GPU Time: 7.33696 ms
CPU Time: 0 ms

------------------------------------------------------------------------------------------
||||| END. YOU MAY RUN THIS AGAIN |||||
------------------------------------------------------------------------------------------
Process completed. Press any key to return to the menu.
```

Merge Sort Case 2:



```
-------------------------------------------------------------
MERGE SORT SELECTED
-------------------------------------------------------------

Enter the size of the array. Must be a power of 2:
 32768

-------------------------------------------------------------
SELECTED SORT PROCESS UNDERWAY
-------------------------------------------------------------

Unsorted array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


Sorted GPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented

Sorted CPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


SORT CHECKER RUNNING - SUCCESFULLY SORTED GPU ARRAY
SORT CHECKER RUNNING - SUCCESFULLY SORTED CPU ARRAY


GPU Time: 211.605 ms
CPU Time: 3 ms

--------------------------------------------------------------------------------
||||| END. YOU MAY RUN THIS AGAIN |||||
--------------------------------------------------------------------------------
Process completed. Press any key to return to the menu.
|
```

Bitonic Sort Case 2:



```
--------------------------------------------------------------
BITONIC SORT SELECTED
--------------------------------------------------------------

Enter the size of the array. Must be a power of 2:
 32768

--------------------------------------------------------------
SELECTED SORT PROCESS UNDERWAY
--------------------------------------------------------------

Unsorted array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


Sorted GPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented

Sorted CPU array:
Too Big to print. Check Variable. Automated isSorted Checker will be implemented


SORT CHECKER RUNNING - SUCCESFULLY SORTED GPU ARRAY
SORT CHECKER RUNNING - SUCCESFULLY SORTED CPU ARRAY


GPU Time: 3.30576 ms
CPU Time: 9 ms

---------------------------------------------------------------------------------
||||| END. YOU MAY RUN THIS AGAIN |||||
---------------------------------------------------------------------------------
Process completed. Press any key to return to the menu.
```

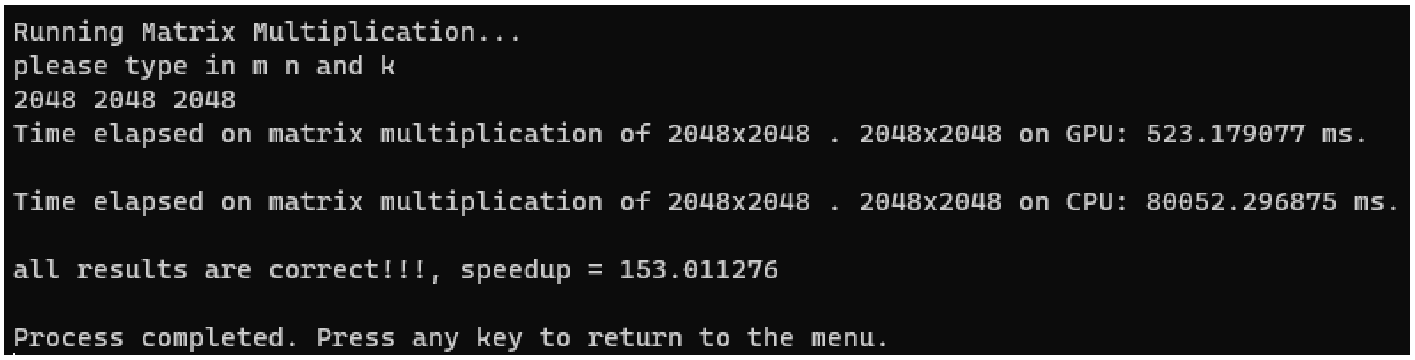
Result:
In cases 1 and 2 of the merge sort algorithm, the overhead cost incurred by the algorithm may not justify its benefits, rendering the CPU's native processing speed more advantageous. In scenarios where the computational gains achieved by the algorithm are overshadowed by the additional time required for its implementation, the inherent efficiency of the CPU becomes a more favorable option.

In the context of Bitonic sort, it is the GPU demonstrated significantly superior performance compared to merge sort, when applied to the same random values.

Reason:
The notable performance improvement observed in the GPU implementation of Bitonic sort compared to merge sort for the same dataset can be attributed to the parallel processing capabilities inherent to GPUs. Bitonic sort, designed to exploit parallelism efficiently, aligns well with the parallel architecture of GPUs. This allows the GPU to handle multiple elements simultaneously, resulting in accelerated sorting times. In contrast, the sequential nature of merge sort may lead to a higher overhead cost, making it less suitable for the parallel processing power offered by GPUs. The superior performance of the GPU in this context underscores the importance of selecting sorting algorithms that align with the strengths of the underlying hardware architecture.

4. Matrix Multiplication

Random elements have been inserted into the Matrix A of dimensions m*n and Matrix B of dimensions n*k. The same matrices will be multiplied in both CPU and the GPU. The results will be stored in a CSV file in the current directory.



```
Running Matrix Multiplication...
please type in m n and k
2048 2048 2048
Time elapsed on matrix multiplication of 2048x2048 . 2048x2048 on GPU: 523.179077 ms.

Time elapsed on matrix multiplication of 2048x2048 . 2048x2048 on CPU: 80052.296875 ms.

all results are correct!!!, speedup = 153.011276

Process completed. Press any key to return to the menu.
```

Result:
Performance of the GPU is significantly better than the CPU

Reason:
Matrix multiplication involves a large number of parallelizable computations, and GPUs are well-suited for handling such tasks concurrently.
Unlike CPUs, which are optimized for sequential processing, GPUs consist of numerous cores capable of performing parallel computations simultaneously. This parallelism allows the GPU to efficiently handle the matrix multiplication operations, leading to

faster execution times compared to the CPU. In summary, the parallel architecture of the GPU is better aligned with the parallel nature of matrix multiplication, enabling it to outperform the CPU in terms of computational speed for this specific task. The observed performance difference underscores the advantage of leveraging specialized hardware, such as GPUs, for parallelizable workloads like matrix operations.

Conclusion:

In conclusion, the exploration of single-threaded, multi-threaded, and GPU programming in C/C++ has provided valuable insights into optimizing performance across various computing scenarios. Single-threaded programming remains crucial for simpler tasks and resource-constrained environments, emphasizing efficiency in sequential execution. Multi-threaded programming has showcased its prowess in enhancing performance by parallelizing tasks, particularly in scenarios where concurrent processing is advantageous. The introduction of GPU programming, exemplified by platforms like CUDA, has revolutionized high-performance computing, unlocking immense parallel processing capabilities for tasks ranging from graphics rendering to scientific simulations. Each approach has its strengths and applicability, and a nuanced understanding of when to leverage single-threaded, multi-threaded, or GPU programming is essential for maximizing computational efficiency in diverse computing environments. The collective comprehension of these programming paradigms empowers developers to tailor their strategies, ensuring optimal performance across a spectrum of applications and computing architectures.