Row-wise and Column-wise Matrix Sort Using GPUs

Sina Hakimzadeh

December 2024

1 Introduction

In this problem we are going to implement a cuda program to sort a matrix row-wise and column-wise and profile the program in different situations.

1.1 The algorithm

A row-wise and column-wise sorted matrix looks as follows:

$$M[i][j] \leq M[i][j+1]$$

$$M[i][j] \leq M[i+1][j]$$

It means each element of the matrix must be less than or equal to its right and left elements. The algorithm first algorithm that works well is that to flatten the matrix and sort it in usual ways, but we are not allowed to use this algorithm. Instead we use this algorithm:

```
REPEAT UNTIL (matrix is sorted){
    Sort each row of the matrix.
    Get transpose of the matrix.
    Again sort each row of the matrix.
    Again get transpose of the matrix.
}
```

1.2 Why multiThreaded design

A multithreaded design can theoretically improve performance by enabling concurrent execution of multiple tasks, which allows for better utilization of system resources, particularly when multiple cores or processors are available.
A multithreaded design in GPUs can improve performance for several reasons:

- **Parallel Execution:** GPUs are designed to handle many threads simultaneously. By distributing tasks across multiple threads, they can execute them in parallel, significantly speeding up computation, especially for tasks like matrix operations, image processing, or simulations.
- **Maximizing Hardware Utilization:** GPUs have many cores, and multithreading helps keep these cores busy, reducing idle time. This efficient use of available resources leads to better overall performance.
- **Hide Latency:** GPUs experience high latency due to memory accesses and other operations. By running multiple threads, one thread can be waiting on data while others continue to execute, reducing the time spent waiting for operations to complete.
- **Scalability:** As the problem size grows (e.g., processing larger datasets), multithreading allows the workload to be distributed across more threads, ensuring the GPU can scale efficiently and maintain performance.

1.3 Analysis Factors

The program will undergo a thorough analysis, taking into consideration several key factors that may influence its performance and behavior. These factors include variations in the size of the matrices being processed, as well as differences in the types of matrices used. Additionally, the analysis will examine the impact of different thread block and grid sizes, which play a crucial role in parallel processing efficiency. Finally, the program's performance will be evaluated based on the application of two distinct streaming approaches—Breadth-First Search (BFS) and Depth-First Search (DFS)—to assess how these algorithms influence the overall execution and scalability of the program.

1.4 Different Memory allocation

cuda has multiple funcitons to allocation memory on GPU or even CPU that listed below:

- `cudaMalloc` allocates memory on the device (GPU). The allocated memory is only accessible from the GPU, so if data needs to be transferred between the host (CPU) and device, explicit memory copies using cudaMemcpy are required. This function is efficient for GPU-only operations.
- `cudaMallocHost` allocates memory that is page-locked (pinned) on the host. This means the memory is physically locked in RAM and cannot be swapped out to disk. It allows faster transfers between host and device memory, as the memory is directly accessible for transfers with minimal overhead. However, it should be used carefully to avoid consuming too much pinned memory, as it is a limited resource.
- `cudaMallocManaged` allocates unified memory, which is accessible by both the host and device. CUDA automatically handles the memory migration between the host and device based on the access patterns, allowing easier data sharing between the CPU and GPU. This function simplifies memory management but can introduce some overhead due to the automatic synchronization of memory transfers.

`cudaMallocHost` is used to pin the data in host memory, enabling the use of streams for asynchronous operations. `cudaMalloc` is employed to allocate memory on the GPU. `cudaMallocManaged` is not used because memory management is handled manually, giving more control over data transfers and stream management.

1.5 Profiling result of sequential program

Table 1 presents the results of sorting a 1024×1024 matrix consisting of integer elements. The program takes 9 seconds to complete the sorting task for this matrix size. This dataset, with its moderate complexity and size, provides a solid benchmark for comparing the performance of the CPU-based implementation with the GPU-based alternative. By evaluating the time taken for the CPU version, we can gain insights into the computational load and performance bottlenecks associated with sorting larger datasets. The comparison with the GPU implementation will allow us to assess how leveraging GPU acceleration enhances the program's speed, parallel processing efficiency, and overall scalability, particularly as the matrix size grows..

Metric	**Value**
Task Clock	9,921.34 msec
CPU Utilization	1.005 CPUs utilized
Context Switches	188 (18.949 /sec)
CPU Migrations	11 (1.109 /sec)
Page Faults	1,088 (109.663 /sec)
Cycles	39,327,106,936 (3.964 GHz)
Instructions	79,460,132,849 (2.02 insn per cycle)
Branches	4,479,519,693 (451.504 M/sec)
Branch Misses	195,219,370 (4.36% of all branches)
Time Elapsed	9.875018323 seconds
User Time	9.911497000 seconds
System Time	0.010062000 seconds

Table 1: Performance Statistics

2 Sorting using CUDA

1. Read a matrix of a specific type mentioned in the input file.
2. Set the grid size and block size.
3. Allocate memory on GPU and copy data from host to device.
4. apply the algorithm in a while loop until the matrix is sorted.
5. synchronize the device and host.
6. copy data from device to host.
7. free the allocated memories.
8. write the matrix in output file.

2.0.1 Sorting Row & Checking

Each GPU thread is responsible for sorting a row using a serial algorithm such as merge sort or bubble sort. This kernel sorts all rows simultaneously on GPU while accessing neighbor element to maintain locality. Checking the matrix to ensure it is sorted also using the same method as sorting it, but in the middle of checking if one unsorted element is found the entire kernel stop.

2.0.2 Transposing

The matrix is divided to 32×32 matrix chunks and each chunk pass into a thread block. Each thread block saves its data in a shared memory, that accessible from threads of a thread block, to avoid overwriting data by other threads. Using `__syncThreads()` to ensure all threads have read their data. Then using the proper indexing to find the new position of each chunk.

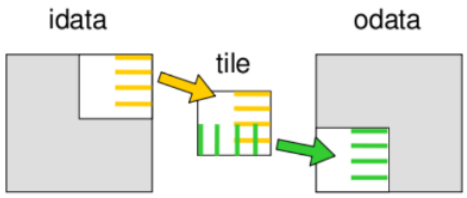


Figure 1: Parallel Matrix Transposing

2.1 Profiling And Optimizing

The initial profiling results(Table 2) reveal poor throughput across all kernels.

Function Name	No Optimization	Optimize sortRows	Optimize transpose	Optimize checkRow
checkSortedRowWiseInt	14.48%	8.9 %	8.2%	25.38%
sortRowsKernelInt	47.78 %	89.48 %	89.93 %	89.49%
transposeKernelInt	29.95 %	35.52 %	64.58 %	63.25%

Table 2: Program Throughputs

By configuring the `sortRowsKernelInt` function to operate within a single thread block and leveraging shared memory, significant improvements in memory throughput are achieved. Additionally, introducing a one-element padding to the shared memory in the `transposeKernelInt` function leads to a remarkable twofold increase in memory throughput, effectively reducing memory access conflicts. Finally, optimizing the `checkSortedRowWise` function by restricting it to a single thread block and strategically accessing even and odd elements separately further enhances performance, demonstrating the impact of tailored memory and thread management strategies on kernel efficiency.

2.1.1 Thread Configuration Impacts

In our approach, a single thread block handles row sorting, maximizing the use of shared memory for improved memory and compute throughput. Each thread copies a portion of the row from global memory to shared memory, where the data can be accessed with lower latency. Threads then work in parallel to sort the row efficiently. Once sorted, the data is copied back from shared memory to global memory.

This method reduces global memory accesses, which are slower, and ensures faster execution by leveraging the GPU's shared memory for both input and output. By using shared memory and parallel processing within a thread block, we optimize performance and fully utilize the GPU's resources.

2.2 Different Sorting Algorithm

There are several parallel sorting algorithms that can be utilized in GPU programming and multi-threaded applications. However, the focus of this course is on writing and optimizing GPU code, rather than designing parallel algorithms. The choice of sequential sorting algorithms can significantly impact performance and throughput, as they influence the access patterns to elements. For instance, the bubble sort in `sortRowsKernelInt()` achieves 90% compute throughput and memory throughput, while the L1 cache throughput is around 99.27%. This kernel takes 179.42ms per execution. In contrast, merge sort delivers significantly better performance, with 99.38% compute throughput and 99.38% memory throughput, taking only 50.52ms per launch. However, merge sort must be executed more times than bubble sort due to the way it sorts the rows and the relationship between sorting the rows and columns.(These statistics were obtained by running the kernel on a 1024 x 1024 matrix with integer elements.)

2.3 Final Result For Integer

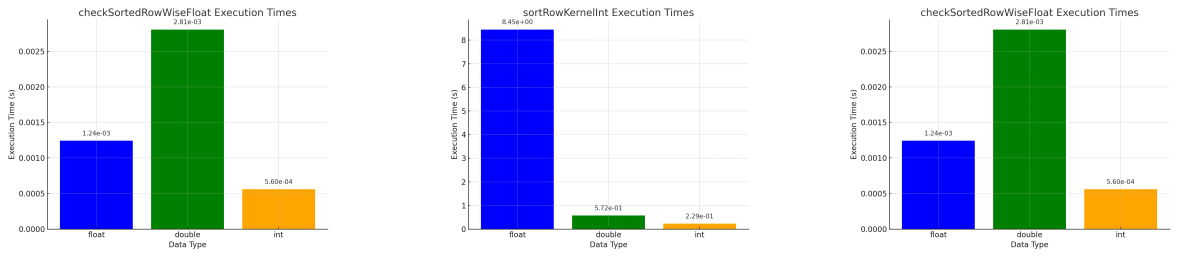
The final results for all kernels are presented in the table 3.

Kernel Name	Avg. Compute Throughput	Avg. Memory Throughput	Avg. Time Elapsed
checkSortedRowWiseInt	10.3%	38.6%	33.45 us
sortRowsKernelInt	93.2%	93.2%	8.58 ms
transposeKernelInt	24.6 %	75.3 %	96.86 us

Table 3: Average Kernel Performance Data (integer, 1024×1024)

2.4 Data Type & Data Size Impacts

The profiling results demonstrate that the data type of matrix elements significantly influences the number of kernel launches required during execution. Specifically, matrices with integer elements required 30 kernel launches, while those with double precision elements required 24, and matrices with float elements required only 2. This variation highlights how the choice of data type affects computational workload and memory operations, likely due to differences in data size, precision requirements, and memory access patterns.

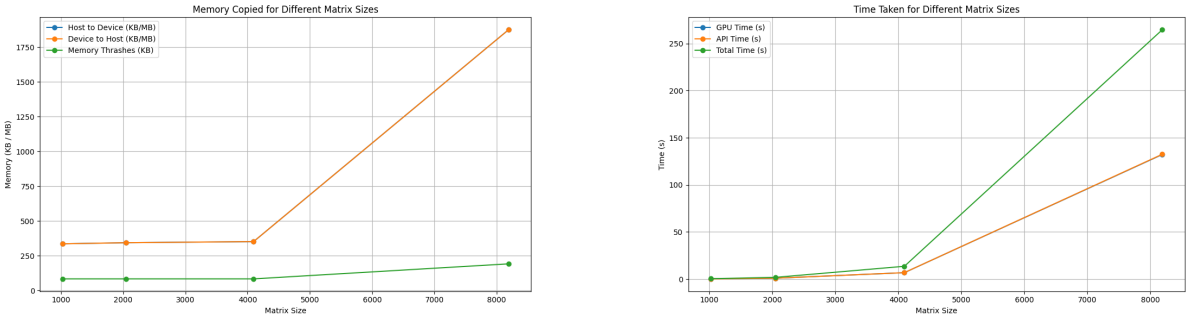


(a) checkSortedRowWiseFloat (b) sortRowKernelInt (c) checkSortedRowWiseFloat

Figure 2: **Kernels Profiling Results:** The profiling involved 30 kernel launches for integers, 24 for doubles, and 2 for floats.

Based on the results in figure 3, it can be inferred that as the matrix size increases, both the time taken for operations and the memory requirements grow exponentially. The total time spent on GPU activities and API calls rises significantly for larger matrix sizes, with a particularly sharp increase seen between the 4096 and 8192 sizes. This suggests that the system may face performance bottlenecks as the matrix size grows, possibly due to the increased computational load and memory transfer overhead.

Additionally, while the memory transferred between host and device remains relatively stable for smaller matrices, there is a marked jump at the 8192 matrix size, indicating that the system is strained when handling larger data sizes. This highlights the challenges of scaling up matrix operations, with the increasing time and memory costs posing potential limitations for large-scale computations.
The program is unable to handle a matrix size of 16,384 due to the L1 cache size limitation of 48KB. Since the row size is $4 \times 16{,}384$ bytes, it exceeds the available cache capacity. To process this larger dataset, shared memory must be utilized to manage smaller chunks of data efficiently.



(a) Memory

(b) Time

Figure 3: Profiling Results for Different Matrix Sizes

3 Overlapping Data Transfer And Computation(Streams)

In GPUs and CUDA, utilizing streams and overlapping computations with communication significantly enhances performance. Streams enable concurrent execution of multiple operations, such as kernel launches and memory transfers, on the GPU. This overlap hides the latency associated with data movement, allowing the GPU to remain actively processing data while waiting for transfers to complete. Furthermore, overlapping computations with communication minimizes idle time, maximizing the utilization of the GPU's processing power. By strategically organizing tasks into different streams and carefully managing dependencies, developers can optimize their CUDA applications for maximum throughput and efficiency. CUDA streams allow overlapping operations, which maximizes GPU utilization by keeping both the host and device busy. The following are examples of overlapping tasks:

- **H2D and Kernel Execution:** While transferring data for one graph component, the kernel for previously transferred data can execute in parallel.
- **Kernel Execution and D2H:** As a kernel completes execution in one stream, results can be transferred back to the host while another kernel runs in a separate stream.

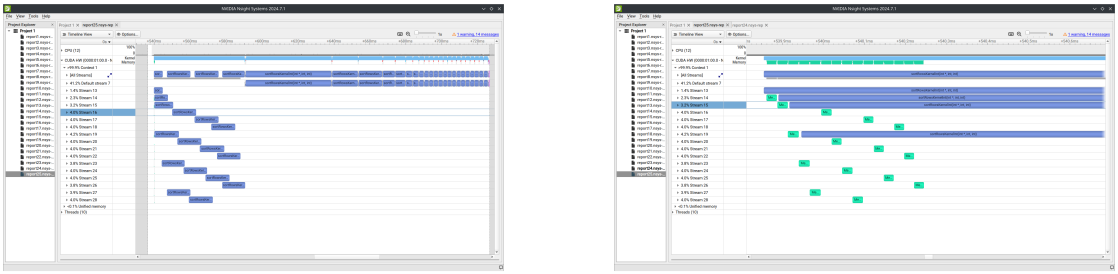
This overlapping reduces idle time and improves the throughput of the overall application.

3.0.1 DFS Approach

In the Depth-First Scheduling (DFS) approach, we issue a sequence of operations consisting of a Host-to-Device (H2D) data transfer, followed immediately by the kernel execution for the corresponding data chunk. This pipeline allows each kernel execution to overlap with the data transfer of the subsequent chunk, effectively utilizing the GPU's ability to handle computation and data transfer concurrently.

By overlapping these operations, the idle time between tasks is minimized, leading to a more efficient use of available resources. For instance, while the GPU is executing a kernel on one data chunk, the next chunk is being transferred from the host to the device in parallel. This overlapping not only reduces latency but also ensures that the GPU is continuously performing useful work, thereby improving overall performance.

This strategy is particularly beneficial for applications dealing with large datasets, where the overhead of sequential data transfer and kernel execution can become significant. By adopting DFS, the workflow achieves better throughput and reduced execution time, leveraging the full potential of the GPU's asynchronous processing capabilities.



(a) Program stream flow

(b) Program data transfer flow

Figure 4

3.0.2 What is the loss of using too many streams?

Increasing the number of GPU streams can initially boost performance by overlapping tasks, but excessive streams introduce overhead from scheduling and resource management. This overhead, combined with limited GPU resources, can lead to contention and reduced performance instead of further improvement. Optimizing the number of streams is essential to balance parallelism and efficiency.

As shown in Figure 5, the execution time of the `sortRowKernelInt` increases after 8 streams. This indicates that the overhead of managing additional streams outweighs the benefits of parallelism, leading to reduced performance as the number of streams grows.

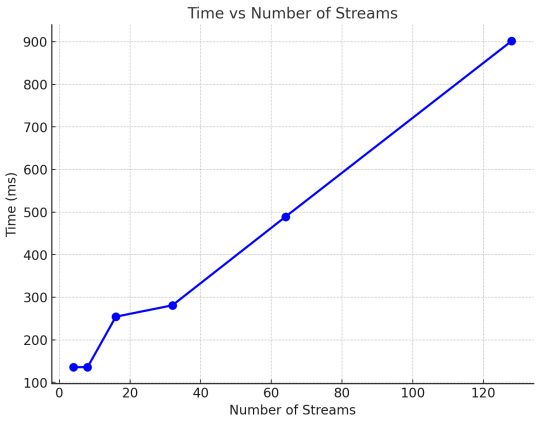


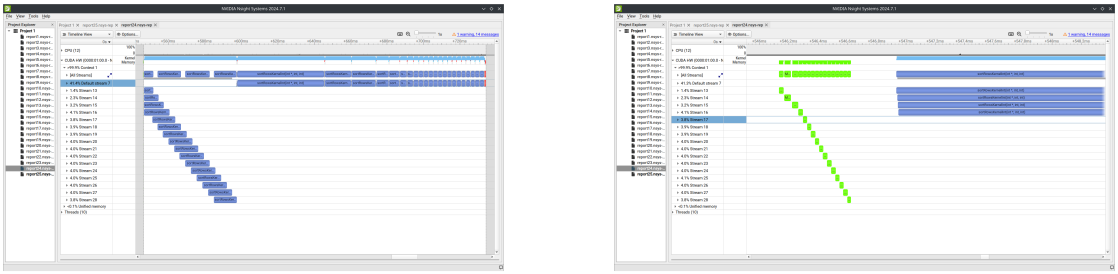
Figure 5: Effect of different stream sizes

3.0.3 BFS approach

In contrast to Depth-First Scheduling (DFS), the Breadth-First Scheduling (BFS) approach involves transferring multiple data chunks from the host to the device in parallel, followed by staggered kernel executions. This allows for broader overlap of operations, keeping the GPU busy with both data transfer and computation. BFS maximizes the use of GPU resources by handling several data chunks simultaneously, which improves throughput and reduces execution time.
BFS and DFS differ in how they handle memory retrieval on Fermi-based GPUs due to the GPU's blocking policies for Host-to-Device (H2D) and Device-to-Host (D2H) data transfers. In these GPUs, kernel executions must complete before a D2H transfer can begin, as the system requires that previous kernel calls finish to allow the copy to start. This synchronization limitation impacts performance, particularly when attempting to copy data back to the host, as it introduces delays between kernel execution and data retrieval. Consequently, this blocking behavior can reduce efficiency, especially in workflows where frequent data transfers to and from the host are required.

Based on the previous explanation, we understand that due to the nature of the problem, we cannot return the data during its first processing phase. This is because the data may require further processing or multiple sorting steps. As a result, there is no overlap between data transfer and computation, meaning that both BFS and DFS do not significantly impact the program's performance.

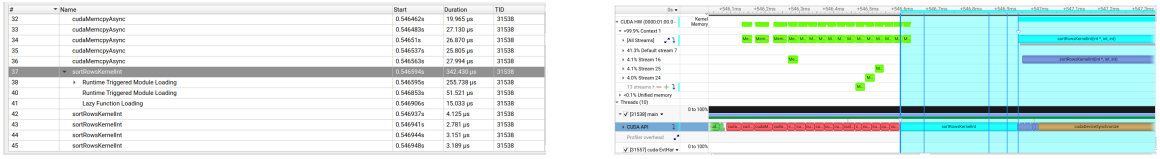
However, in the BFS approach, we observe that data transfers occur before computation begins. This raises the question: **why is this the case?**
This occurs due to the `Runtime Triggered Module Loading` mechanism, which initializes during the first kernel call of the program.



(a) Program stream flow

(b) Program data transfer flow

Figure 6



(a)

(b)

Figure 7: Illustration of runtime-triggered module loading and lazy function execution, showcasing how modules and functions are loaded or executed only when they are first called, optimizing resource usage and startup time.

The delay caused by `Runtime Triggered Module Loading` during the first kernel call introduces a one-time overhead as the GPU initializes its modules. While this delay is relatively small, it can impact performance in applications where minimizing idle time is critical. Depth-First Scheduling (DFS) can help mitigate this issue by overlapping the data transfer of the first chunk with the initialization process, effectively hiding the delay. This approach ensures concurrent execution of initialization and data transfer, leading to better utilization of available resources and improved throughput.

However, the choice between DFS and Breadth-First Scheduling (BFS) should consider workload characteristics and GPU architecture. For larger data chunks, BFS may be advantageous as it preloads multiple data chunks before starting computation, reducing the number of context switches and ensuring continuous processing. Nevertheless, BFS requires careful memory management to avoid resource contention or memory bandwidth saturation, which could offset its benefits.

Ultimately, the decision to use DFS or BFS depends on the size of the data chunks, the complexity of the kernel, and the need for overlapping operations. For smaller chunks, DFS is typically more effective in hiding initialization delays, while BFS may excel in scenarios where preloading larger chunks is feasible without straining memory resources.

4 Result And Discussion

The CUDA-based sorting algorithm shows significant performance gains by utilizing GPU parallelism. Profiling the sequential program revealed a 9-second runtime for a 1024×1024 matrix. After porting to CUDA, optimizations like shared memory and improved thread block configurations led to better performance.

Row sorting with bubble sort achieved 90% memory throughput, while merge sort performed better but required more kernel launches. The data type impacted performance, with floats benefiting from fewer kernel launches than integers and doubles.

As matrix size increased, performance decreased due to computation load and memory overhead, particularly above 8192. Efficient memory management, including shared memory, was critical. DFS and BFS scheduling methods performed differently based on chunk size, with DFS hiding delays and BFS managing large chunks well.

In conclusion, optimizing memory usage, thread blocks, and overlapping data transfers can greatly enhance GPU-based sorting, though scalability is affected by matrix size, data type, and stream number.