THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

HYBRID CPU-GPU SORTING MECHANISMS FOR LARGE-SCALE OUT-OF-MEMORY DATA: DESIGN AND PERFORMANCE EVALUATION

LEAH CHOI
SPRING 2025

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degrees
in Bachelor of Science
with honors in Computer Engineering.

Reviewed and approved* by the following:

Anand Sivasubramaniam
Distinguished Professor of Computer Science and Engineering
Thesis Supervisor

Dong Xie
Assistant Professor of Computer Science and Engineering
Thesis Co-Advisor

Robert Collins
Associate Professor of Electrical Engineering and Computer Science
Thesis Honors Adviser

* Electronic approvals are on file.

ABSTRACT

In the modern technology landscape, efficient data sorting is crucial for the performance of numerous applications in fields ranging from machine learning to large-scale database management. As datasets continue to grow, traditional processing methods often struggle to keep pace, necessitating the development of more sophisticated technologies to handle these volumes efficiently. This thesis investigates the comparative performance of GPU-heavy versus hybrid CPU-GPU sorting methods, pivotal for addressing the challenges posed by large datasets. Our findings reveal that while a GPU-centric approach provides high throughput for smaller datasets less than 25GB, it faces significant performance decline when processing larger datasets due to memory constraints and increased data transfer overheads. A critical memory threshold, around 25GB, is identified, beyond which GPU performance deteriorates sharply due to excessive page faults and data transfer overheads associated with Unified Virtual Memory (UVM). Conversely, a balanced hybrid approach, utilizing a 50/50 CPU to GPU load factor, offers a strategic solution by distributing workloads to avoid GPU memory saturation and minimizing data movement. This method not only leverages the strengths of both the CPU and GPU but also addresses their limitations, enhancing overall system performance and stability across varying data sizes. The research underscores the importance of strategic workload distribution and memory management in large-scale data environments, providing a sustainable solution to the complexities of sorting large datasets.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my co-advisors, Professor Anand Sivasubramaniam and Professor Dong Xie, for their invaluable guidance and support throughout this project. I also extend my appreciation to my collaborators, Aayush Bhala and Alec Ming-Ta, for their valuable contributions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

In the contemporary technology landscape, the burgeoning volume of data necessitates advanced processing capabilities, particularly driven by the escalating demand for big data analytics, cloud computing, and real-time applications. Sorting, a fundamental operation in computing, emerges as a critical challenge amidst these demands. Its efficiency significantly impacts other algorithms, including search and merge algorithms, making it crucial in fields like machine learning and large-scale database systems. Traditionally, the Central Processing Unit (CPU) has been utilized for complex arithmetic operations but exhibits limitations in parallel sorting tasks. Despite the CPU's ability to handle multiple portions of sorting in parallel, its inherent parallelism restrictions often lead to suboptimal performance. Conversely, the Graphics Processing Unit (GPU) excels in high-speed graphic calculations and parallel tasks due to its architecture, which allows multiple streaming multiprocessors to execute tasks simultaneously. The optimized version of Odd-Even Merge Sort illustrates this point well. The approach minimizes decision-making by restructuring the network to ensure each step involves only compare-and-swap operations, enabling all GPU threads to execute the same instruction simultaneously, significantly enhancing parallelism and efficiency [1]. However, GPU-based sorting encounters significant setbacks when data sizes surpass the GPU's memory capacity. Performance degradation is primarily due to the necessity of data transfer between the CPU's host memory and the GPU's device memory across the slower PCIe bus, creating bottlenecks and increasing delay [2].

Various strategies have been proposed to ameliorate these bottlenecks. Gowanlock et al. suggest overlapping data movement with actual sorting processes using cudaStream, thereby enhancing the throughput of sorting operations [2]. Furthermore, data transfers between CPU and GPU can be managed more effectively through advanced techniques like prefetching or Unified Virtual Memory (UVM), which will be explored in later sections [3].

This thesis aims to demonstrate an effective workload distribution strategy between the CPU and GPU for handling datasets that exceed GPU memory limits while maximizing performance. The subsequent sections will discuss the hybrid CPU/GPU sorting technique, evaluate its performance, and propose future developments.

Chapter 2 Background

2.1 Parallel Computing

Traditionally, software has been written for serial computation. This method involves a processor executing instructions one after the other, with each step depending on the completion of the previous one. This linear approach is straightforward but often inefficient for tasks that are computationally intensive or data-heavy. Parallel computing addresses these inefficiencies by dividing a problem into discrete parts that can be solved simultaneously. Each part executes in parallel across different processors or cores, drastically reducing the time required to reach a solution. This method leverages the collective power of modern computing architectures, such as multi-core processors and graphic processing units (GPUs).

2.2 Parallelization Architecture

This section discusses different types of Parallelization Architecture.

2.2.1 OpenMP (Open Multi-Processing)

OpenMP is an API that supports multi-platform shared memory multiprocessing in CPU, making it invaluable for developing multi-threaded applications. OpenMP uses a set of compiler directives, library routines, and environment variables that influence run-time behavior. It is particularly effective in environments where applications need to scale dynamically according to available hardware resources. OpenMP's model of thread executing enables easy division of tasks across multiple processors.

2.2.2 GNU Parallel

GNU Parallel is a command-line tool designed to execute jobs in parallel on one or multiple machines. Unlike OpenMP, GNU Parallel is not embedded within the code but is used to orchestrate the execution of shell commands or scripts in a parallel fashion. It simplifies workload management by ensuring that every CPU core is tasked without manual intervention, handling the distribution and output collection seamlessly. The main advantage of GNU Parallel is its ease of use and its ability to speed up processes without requiring changes to existing scripts, making it a go-to for system administrators and researchers working with pipeline processing in Unix-like environments.

2.2.3 CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Unlike OpenMP and GNU Parallel, which are primarily aimed at CPU-based parallelism, CUDA gives developers direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of kernels. The approach is fundamentally different because it involves programming in a variant of C, C++, or Fortran with extensions, and it requires a good understanding of underlying hardware to optimize performance. CUDA is particularly suited for applications that can leverage thousands of small, independent, and simultaneous threads to perform massively parallel operations such as matrix multiplication, deep learning, and scientific simulations. The strength of CUDA lies in its ability to offer high throughput for tasks involving large blocks of data, which can be processed much faster compared to CPU-only code, due to the parallel nature of GPUs.

2.2 CPU and GPU comparison

CPUs and GPUs serve distinct roles in computing, optimized for different types of tasks based on their architecture. CPUs, with fewer cores capable of executing complex and serial instructions, excel in tasks requiring decision-making and intensive data handling due to their sophisticated hierarchical memory system, including multiple levels of caches and system RAM [4]. This design facilitates rapid task switching and efficient execution of general-purpose applications. In contrast, GPUs are designed for parallel processing, boasting hundreds to thousands of smaller cores that work simultaneously to handle large data blocks, making them ideal for tasks like graphics rendering and scientific simulations. While CPUs are versatile and excellent for a wide range of applications, GPUs offer unparalleled speed in parallel computational scenarios.

2.3 CUDA Memory Architecture and Constraints

The memory hierarchy of GPUs plays a pivotal role in determining the performance by influencing memory bandwidth and latency. Memory bandwidth, which measures the volume of data that can be processed per unit of time, is crucial for tasks involving large datasets, enabling rapid data transfers that enhance overall processing speed. On the other hand, latency, defined as the delay before the start of data transfer, critically affects algorithms that require frequent access to memory. For instance, in sorting algorithms, where quick and efficient data retrieval and storage are necessary, optimal utilization of the GPU's memory capabilities, such as those found in NVIDIA's high-bandwidth DRAM and the swift access provided by the L2 cache, can dramatically influence execution speeds. CUDA GPU architecture is designed for efficient parallel computing that includes a hierarchy involving Streaming Multiprocessors (SMs). Each SM is a crucial component of the GPU allowing it to handle multiple operations simultaneously. Inside each SM, there are multiple threads that execute the same or different instructions concurrently. This capability is fundamental to the GPU's ability to process vast amounts of data in

parallel, speeding up computations. Threads within an SM share both instructions and memory. The shared memory available to threads in the same SM is designed for data that multiple threads need to access quickly and simultaneously. Architecture allows for dynamic allocation and management of threads and memory, making GPUs adaptable to a wide range of computational challenges. [5]

Chapter 3 Data Transfer between Host-Device

Transferring data between main memory and the GPU is a critical bottleneck in achieving optimal performance in heterogeneous CPU/GPU systems. As modern CPUs have access to significantly larger main memory capacities (several TiBs) compared to the relatively smaller global memory of GPUs (less than 50 GiB), frequent data transfers become necessary for processing large datasets. In this section we provide some techniques to mitigate this bottleneck.

3.1 Pipelining Data Transfer

Pipelining is a technique used to overlap communication and computation, crucial in large-scale data processing to maximize resource utilization. Work by Gowanlocka et al. [2] demonstrates how to handle this technique effectively. In the context of hybrid sorting, pipelining involves initiating data transfer to the GPU while it is still executing computations on previously loaded data. This subsection provides a detailed analysis of pipelining strategies, including asynchronous data transfers using CUDA's cudaMemcpyAsync and the use of double-buffering to continuously feed the GPU without idle time. The challenges of maintaining efficient pipeline stages and handling variable data transfer rates are also explored.

3.1.1 Pinned Memory

Optimizing host-GPU memory transfers is crucial for performance in data-intensive applications. Pinned memory is key here, as it allows direct data transfers between the host and GPU without intermediate copying, reducing latency. Functions like cudaMallocHost() and cudaHostAlloc() are used to allocate pinned memory, ensuring faster data access for the GPU. CUDA streams enhance this process by

enabling asynchronous operations. They allow simultaneous memory transfers and computations, which is particularly effective in managing large datasets. [2] However, integrating CUDA streams with pinned memory requires careful management. While cudaHostRegister() can quickly pin existing pageable memory, its setup time might outweigh the benefits if not properly synchronized with computational tasks.

3.1.2 CUDAStrem

CUDA streams offer a mechanism to achieve greater concurrency and overlap between computation and data transfer. By allowing multiple streams to operate independently, different tasks (such as data loading, processing, and unloading) can be performed simultaneously on a single GPU. This subsection delves into the practical implementation of CUDA streams in hybrid sorting systems, detailing how streams can be utilized to parallelize operations and reduce total processing time. Issues related to synchronization, the potential for deadlock, and the efficient management of stream dependencies are discussed to optimize stream usage.

3.2 Unified Virtual Memory

Unified Virtual Memory (UVM) in CUDA dramatically simplifies memory management across CPUs and GPUs by allowing seamless data access without the need for manual data transfers. By using cudaMallocManaged(), memory is allocated in a way that both the CPU and GPU can easily access it, eliminating the complexities and performance costs associated with previous manual transfer methods [3]. When a page fault occurs—indicating that the required data is not in the local memory, the UVM system efficiently handles this by dynamically migrating the page to the accessing processor's memory. This process includes allocating a new page, unmapping and freeing the old page, and updating the page tables

accordingly. UVM's intelligent handling of data accessibility and page migration reduces overhead, enhances performance, and streamlines programming in heterogeneous computing environments.

3.3 Memory Prefetching

Memory prefetching in CUDA is a technique that reduces memory access latency, increasing bandwidth utilization. Prefetching involves asynchronously transferring data from host to device, or between different GPU memories before it is utilized for computation. In our implementation of sorting, we prefetch the portion of the input array based on the GPU load factor, before GPU starts sorting batch. This approach ensures data is available in the GPU's faster local memory when required, minimizing the delays caused by on-demand memory transfers during execution. The primary advantage of memory prefetching is the reduction in GPU stall time, where a GPU waits for data to be loaded into its memory. By moving data in advance, the GPU can maintain a higher level of computational throughput. CUDA provides explicit support for prefetching through APIs like `cudaMemPrefetchAsync()`, which allows developers to specify which data to move and to which memory (host or device). This functionality is particularly beneficial in systems equipped with NVIDIA's Unified Memory system we discussed, which manages a shared memory space between the CPU and GPU. However, memory prefetching contains drawbacks. Effective implementation requires a thorough understanding of the application's memory access patterns, as incorrect prefetching can lead to increased overhead and reduced performance. Moreover, the manual effort required to optimize prefetching can increase the complexity of program development and maintenance. Despite these challenges, memory prefetching is a powerful tool in the CUDA optimization arsenal, particularly when dealing with large datasets and memory-intensive operations [8].

Chapter 4 Methodology

In this chapter we outline our approach to optimizing sorting performance across CPU and GPU through hybrid dynamic workload distribution. We split the input array based on a CPU/GPU workload ratio, sorting each part on the respective processors. The sorted batches are then merged using merge sorts, with all operations facilitated by Unified Virtual Memory (UVM) to streamline data sharing and improve efficiency. Our method capitalizes on the CPU's handling of complex tasks and the GPU's parallel processing capabilities, aiming to overcome the common bottleneck of memory transfer rates as data scales.

4.1 UVM Data Allocation

Initially, we allocate the entire dataset into Unified Virtual Memory (UVM) to facilitate seamless data sharing between the CPU and GPU.

4.1.1 Workload distribution

Efficient workload distribution between the CPU and GPU is crucial for maximizing performance in hybrid sorting architectures. This section outlines our dynamic workload allocation strategies, which adapt to the system's current load and the characteristics of the data. We explore algorithms capable of dynamically adjusting the distribution based on runtime metrics, such as data complexity and processing times. These include heuristic-based approaches and machine learning models designed to predict the optimal division of tasks. The objective is to utilize the CPU for its strengths in complex decision-making and data preprocessing, while delegating highly parallelizable sorting tasks to the GPU.

4.1.2 Memory Prefetch

After determining the workload distribution, we calculate the indices for data sorting: starting from index 0, the portion that will be sorted on the CPU and the remainder on the GPU. Following this, we prefetch the data to the GPU to optimize access times and enhance sorting performance.

4.2 CPU Merge Sort Implementation

Mergesort is renowned for its efficiency and stability, making it ideal for sorting large datasets, particularly when adapted for multi-threaded execution on modern multi-core CPUs [11]. In this section, we detail the implementation of mergesort designed to capitalize on the parallel processing capabilities of CPUs. This approach is grounded in techniques described by Miller and Stout, who explore the efficiencies of parallel computing architecture [9]. Our implementation uses the pthread library to manage multiple threads, enabling the algorithm to significantly reduce the total sorting time by processing different segments of the dataset concurrently.

Each thread operates on its portion of the array, receiving necessary parameters such as array boundaries and recursion depth through a struct named ThreadArgs. Controlled recursion within the mergeSort function prevents the overhead associated with excessive thread creation. Following the sorting of individual segments, a two-way merging technique combines these segments into a single sorted array. This process, while efficient, incurs overhead from frequent dynamic memory allocations during the merge phase, suggesting room for optimization such as preallocating memory pools. This CPU-based mergesort serves as the initial phase in our hybrid sorting approach, preparing data for

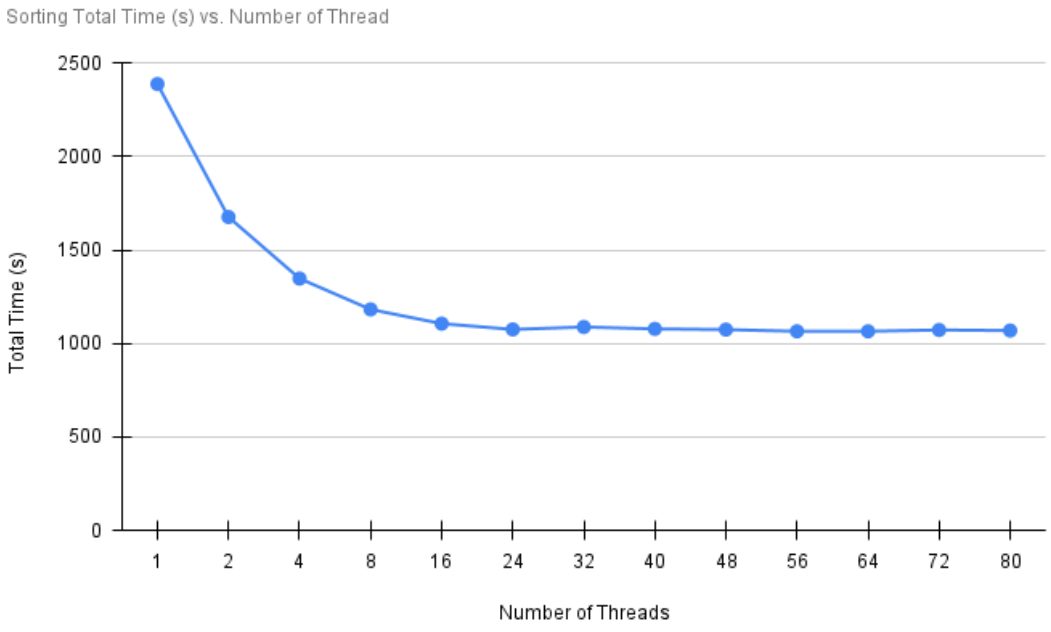

Figure 1 Merge Sort Time of CPU with Different Number of Threads.

subsequent processing and acceleration on the GPU. Leveraging the CPU for coarse-grained sorting tasks and the GPU for smaller, more manageable datasets ensures optimal utilization of each component's strengths. Future work will aim to refine this implementation to reduce overhead and enhance integration with GPU-based sorting phases, improving performance in large-scale data environments.

4.2.1 Merge Sort Performance with Various Thread Numbers

In our research, we investigated the performance of a multithreaded CPU mergesort algorithm applied to a 40GB dataset of doubles. Our initial findings indicated that the single-threaded execution was notably inefficient, requiring approximately 2400 seconds to sort 5300 million keys. As we increased the number of threads, we observed significant performance improvements, highlighting the benefits of parallel processing. Mergesort, by its nature, lends itself well to parallelization because different segments of the

data can be sorted independently before merging. This characteristic was evident as performance enhancements were observed with each doubling of the thread count.

Upon reaching 24 threads, we achieved optimal performance, clocking in at 1075 seconds. This marked improvement suggests that we effectively utilize the maximum capabilities of the CPU, likely aligning with the optimal use of physical and logical cores provided by the hardware. Beyond this point, further increases in the thread count did not yield any performance gains. This plateau can be attributed to several factors: the overhead associated with managing an excessive number of threads, the inefficiencies from context switching, and resource contention among threads, particularly for CPU cache and memory bandwidth.

Given these observations, we concluded that utilizing 24 threads for the CPU mergesort portion of our study was the most efficient approach. This setup allowed us to maximize computational efficiency and resource utilization of our hardware without incurring the diminishing returns associated with higher thread counts. Consequently, this configuration was adopted as the remainder of our research to ensure consistency and optimal performance in our sorting operations. This decision was also supported by system profiling tools that provided insights into CPU, memory, and thread utilization, confirming that 24 threads were optimal under our specific testing conditions.

4.3 GPU Sorting Implementation

4.3.1 Radix Sort for Primitive Keys

For sorting on the GPU, we chose a radix sort algorithm, which is well-suited for primitive data types (e.g. integers or floats) [6][7]. Radix sort is a non-comparative sorting method that operates by grouping keys by portions of their binary representation rather than comparing elements pairwise. On modern GPUs, this approach is extremely efficient for primitive keys – it can achieve linear-time complexity

relative to the number of items and make excellent use of parallelism. By contrast, traditional comparison sorts like mergesort or quicksort involve data-dependent branching and random memory access patterns, which are less GPU-friendly. Mergesort on GPU has higher complexity (O(N log N)) and generally slower throughput for primitive keys. Thus, using radix sort provides a performance advantage for large arrays of primitive keys. This performance benefit led us to the choice of radix sort as the core of the GPU sorting implementation.

4.3.2 Library Choice: CUB vs. Thrust for GPU Sorting

We implemented the GPU radix sort using NVIDIA's CUB library, rather than the higher-level Thrust framework, due to important performance and memory management considerations. Thrust offers convenient sorting routines, but it abstracts away memory allocation and algorithm selection in ways that proved problematic for our needs. Thrust's default sorting behavior sometimes falls back to a mergesort implementation – for instance, when a custom comparator or non-primitive key type is used, Thrust will use a stable mergesort instead of radix sort [12]. This was an issue in our case because the mergesort routine in Thrust requires additional working memory on the GPU. Thrust's mergesort (and even its radix sort implementation) requires O(N) auxiliary memory during sorting [13]. Thrust automatically allocates this temporary storage in GPU device memory, which quickly becomes a bottleneck for very large datasets. If the input dataset and thus the required temporary space approaches or exceeds the GPU's local memory capacity, Thrust's allocation can fail or crash the program [13]. We encountered this limitation with large inputs – the GPU simply did not have enough free memory to accommodate the sort's internal allocations.

The CUB library was selected to overcome these issues. First, CUB's device-level sorting primitive (cub::DeviceRadixSort) gives us manual control over memory allocation. Instead of the library implicitly allocating space on the GPU, CUB exposes an API where we first query the required temporary

storage size and then allocate it ourselves. We leverage this to allocate all necessary buffers in Unified Virtual Memory using cudaMallocManaged. Unified memory allows the allocation to reside in a managed pool that is accessible to both the CPU and GPU, with the CUDA runtime migrating pages between host and device as needed. Crucially, on modern systems, UVM supports memory allocated can exceed the physical GPU memory – the excess will reside in host memory and be paged in on demand. This approach prevented the crashes we saw with Thrust and allowed the handling of very large data sets, at the cost of some performance overhead when data paging was necessary.

4.3.3 Memory Management with Unified Memory

Using cudaMallocManaged for all data and temporary storage in our implementation was a deliberate strategy to handle large data sizes. Like we discuss Unified memory (UVM) provides a single address space shared between the CPU and GPU. In our context, it meant we could allocate a very large array of keys without worrying if it fit entirely in GPU memory. If the dataset is larger than GPU RAM, the driver will page data in and out from system memory. This capability, introduced in CUDA 8 and newer GPUs, allows a managed allocation to exceed the physical GPU RAM [14]. The benefit is that our sort can handle an input of arbitrary size without failing due to GPU memory exhaustion. The trade-off is that if the GPU must continuously swap data from host memory, the effective throughput drops due to PCIe transfer overheads.

4.3.4 Implementation Details of the gpu_merge Sorting Function

The key steps in the GPU sorting function are as follows

- Estimating Temporary Storage: CUB’s DeviceRadixSort::SortKeys function requires a temporary storage buffer to be provided. To determine the required size of this buffer, we call

cub::DeviceRadixSort::SortKeys with a nullptr storage pointer and a zero byte-length. This is a dry-run call that does not actually sort but returns the necessary size (in bytes) for the temporary storage. Usually, this radix sort implementation requires O(N) auxiliary memory during sorting. Internally, CUB computes how much memory it needs based on the number of items and the data type. This design allows the caller to allocate exactly the amount of memory needed for the sort.

- Allocating Temporary Storage: Using the size determined in the previous step, we allocate a buffer for temporary storage. We use cudaMallocManaged to allocate this buffer in unified memory, as discussed earlier. This buffer will be used by CUB's sorting algorithm for its internal bookkeeping and data swaps. Allocating in unified memory ensures the GPU can access it directly, and if the buffer is large, it can reside partly in host memory if necessary. By allocating once per sort call (or reusing an allocation across calls), we avoid repeated allocation overhead. If the allocation fails (which is unlikely with UVM unless the system truly runs out of memory), the code will handle the error accordingly (e.g., by returning an error status).
- Sorting the Data with CUB: With input data and temporary storage ready, we invoke cub::DeviceRadixSort::SortKeys. We pass the pointers to the input keys array and an output array if needed. Often CUB's radix sort operates out-of-place, ping-ponging between input and output buffers. If the algorithm requires, the input and output may swap roles between digit passes. We also provide the allocated temporary storage buffer and its size. CUB then launches GPU kernels to perform the radix sort. Because our keys are primitive types, CUB's highly optimized implementation will execute a radix sort that fully utilizes the GPU's parallel capabilities [15].

4.3.5 Performance Considerations and Limitations

Using CUB's radix sort in this way gives excellent performance for our use case, but there are a few considerations and limitations to acknowledge. First, the memory overhead for the sort is still in the order of the input size. By using unified memory, we mitigated the hard limit of GPU memory size, but if the dataset is extremely larger than GPU local memory, the sort may incur significant paging, which can slow it down. In practice, there is a trade-off between dataset size and sorting speed when using oversubscribed memory. Second, CUB's sorting routine is designed to automatically optimize thread usage and block sizes for the underlying GPU architecture. When CUB is called, it internally decides how many CUDA threads and blocks to launch, and how to partition the data among threads. This dynamic optimization is generally a strength because it yields near-optimal performance without user intervention. However, it also represents a limitation in flexibility. End-users do not have fine-grained control over these parameters at runtime. If one wanted to experiment with different thread/block configurations or optimizations, CUB does allow it through template parameters [16], but such changes require modifying the compiled code. In our methodology, we treated CUB as a black box that provides robust performance out-of-the-box, accepting its internal choices for thread-block sizing.

4.4 Final Merge Process

In the concluding phase of our hybrid sorting strategy, the final merging of sorted batches—one from the CPU and the other from the GPU—is efficiently managed using the __gnu_parallel::merge function from the GNU Parallel Mode library. The __gnu_parallel::merge, leverages parallel algorithms to efficiently combine the two sorted arrays [17]. By utilizing multiple cores, this method considerably reduces the time required for the final merge process compared to traditional single-threaded merge operations[10]. This merging process ensures that the data sorted independently by the CPU and GPU are seamlessly integrated into a single, consistently ordered array.

Chapter 5 Evaluation

In this chapter, we evaluate the performance of the various approaches discussed above and analyze the potential reasons behind their performance differences.

5.1 Experimental Methodologies

5.1.1 Hardware Configuration

The experiments were performed on a machine equipped with an NVIDIA Quadro RTX 8000 GPU, which boasts substantial computational capabilities suitable for intensive data processing tasks. The key specifications of the GPU include:

Table 1 Table of NVIDIA Quadro RTX 8000 GPU

Device Name	Quadro RTX 8000
Maximum Threads Per Block	1024
Maximum Threads Dimension	1024 x 1024 x 64
Maximum Grid Size	2147483647 x 65535 x 65535
Warp Size	32
Number of Multiprocessors	72
Total Global Memory	48GB

Additionally, the system utilized 24 CPU threads, which were determined to yield the fastest performance in preliminary testing discussed in chapter 4. This multi-threaded environment was crucial for the CPU components of the hybrid sorting algorithms.

5.1.2 Data Setup

The data used in the experiments consisted of double precision floating-point numbers (double, 8 bytes), aligned for x86 environments to ensure efficient memory access. The experiments started with datasets of 10 GB and were scaled up to approximately 100 GB to test the algorithms under varying data volumes.

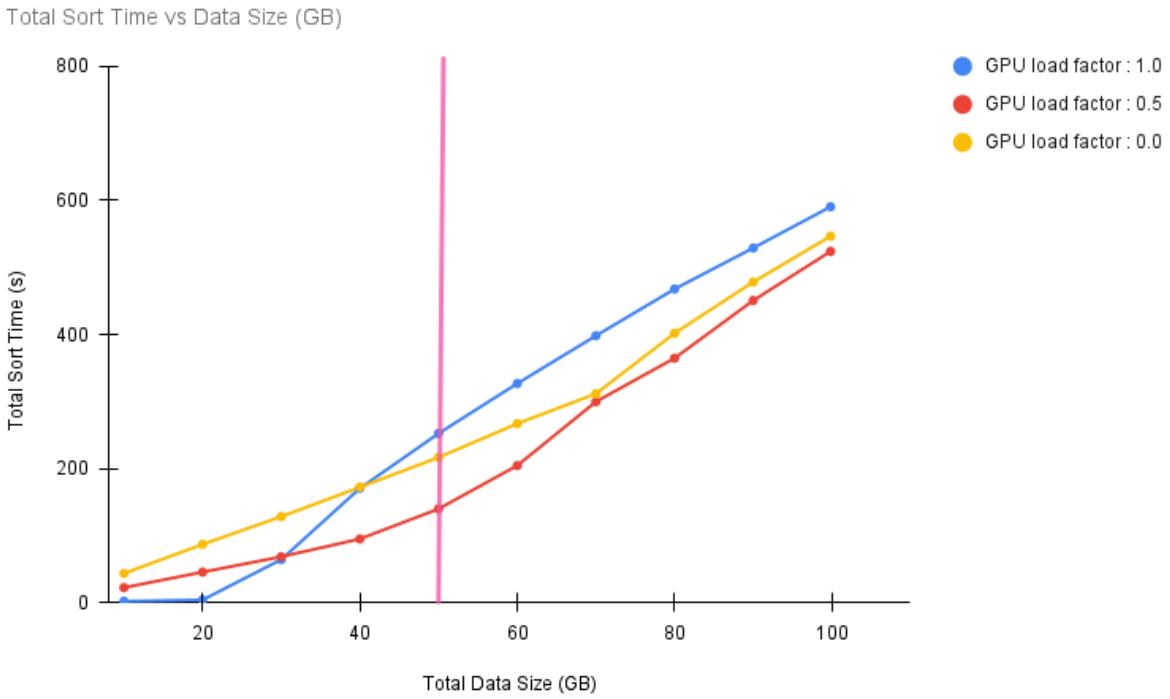

Figure 2 Sort Time of Different Workload Distribution – Pink line indicates GPU's local memory size

5.2 Workload Distribution

In our sorting method, we use a GPU load factor to denote the proportion of the sorting workload assigned to the GPU, with the remainder handled by the CPU. For instance, a load factor of 0.5 indicates an even split: the GPU sorts half of the data while the CPU sorts the other half. After both halves are sorted, a final merge step is performed on the CPU to produce the completely sorted output.

Figure 2 shows that at smaller data volumes, the GPU-only configuration (load factor 1.0) clearly outperforms the hybrid 0.5 and the mostly CPU 0.1 configurations. This is expected because with load factor 1.0 the highly parallel GPU handles the entire sort, leveraging its massive throughput and yielding the lowest execution times initially. However, we see that the throughput of the load factor 1.0 configuration starts surging once the dataset grows past roughly 25 GB. In contrast, the hybrid 0.5 configuration maintains a steadier performance up to a larger input size before eventually degrading. The load factor 0 (mostly CPU) case, while the slowest in absolute performance, exhibits a steady linear increase without the dramatic collapse. The key reason for the 1.0 configuration's degradation is the GPU memory limit. Datasets above 25–30 GB exceed the GPU's onboard memory capacity, causing the unified memory system to page data in and out between host and device. This is because radix sort requires an additional memory buffer of size O(N) for intermediate computations. Each time the GPU accesses a portion of the data that is not resident in its memory, a UVM page fault occurs, incurring a costly transfer from host memory. This frequent back-and-forth transfer of data overwhelms the benefits of GPU parallelism. As a result, beyond ~25 GB, the GPU-only strategy suffers severe performance penalties due to memory thrashing, whereas the hybrid strategy (0.5) avoids these issues by keeping half of the data on the CPU.

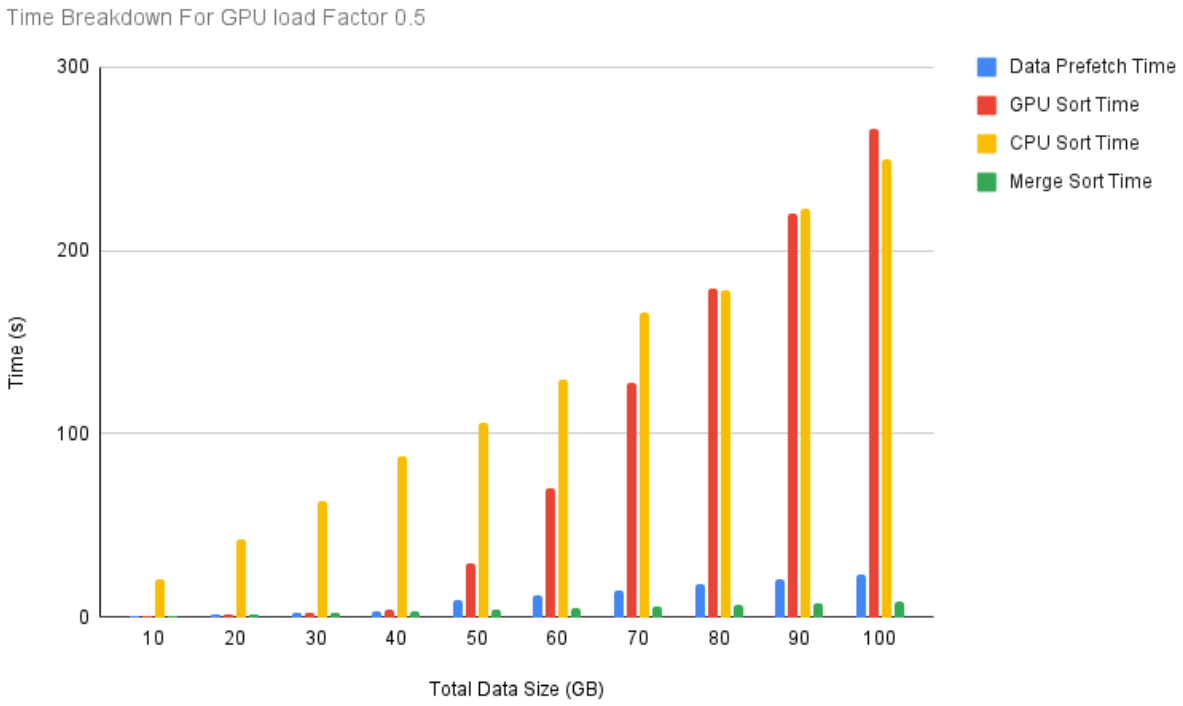

Figure 3 Time Breakdown of GPU Load Factor 0.5

5.2.1 Scaling Behavior in Hybrid Sorting

From Figure 3, we observe that the CPU merge sort time increases linearly with the input size. This linear growth is expected. The CPU's workload in the 0.5 scenario consists of sorting half of the data (roughly O(N log N) complexity for that portion) and then merging the two sorted halves (O(N) complexity). This overall workload leads to near-linear scaling as the total input size N increases. In contrast, the GPU sorting time remains comparatively low and nearly flat up to about a 25 GB share of data on the GPU, but it begins to rise sharply once the GPU's portion of the data crosses that ~25 GB memory threshold. The GPU's sorting kernel experiences severe slowdowns due to memory oversubscription and frequent page faults, since it must repeatedly fetch and evict pages of data through UVM. Essentially, the GPU threads spend most of their time waiting on data from memory rather than performing computation.

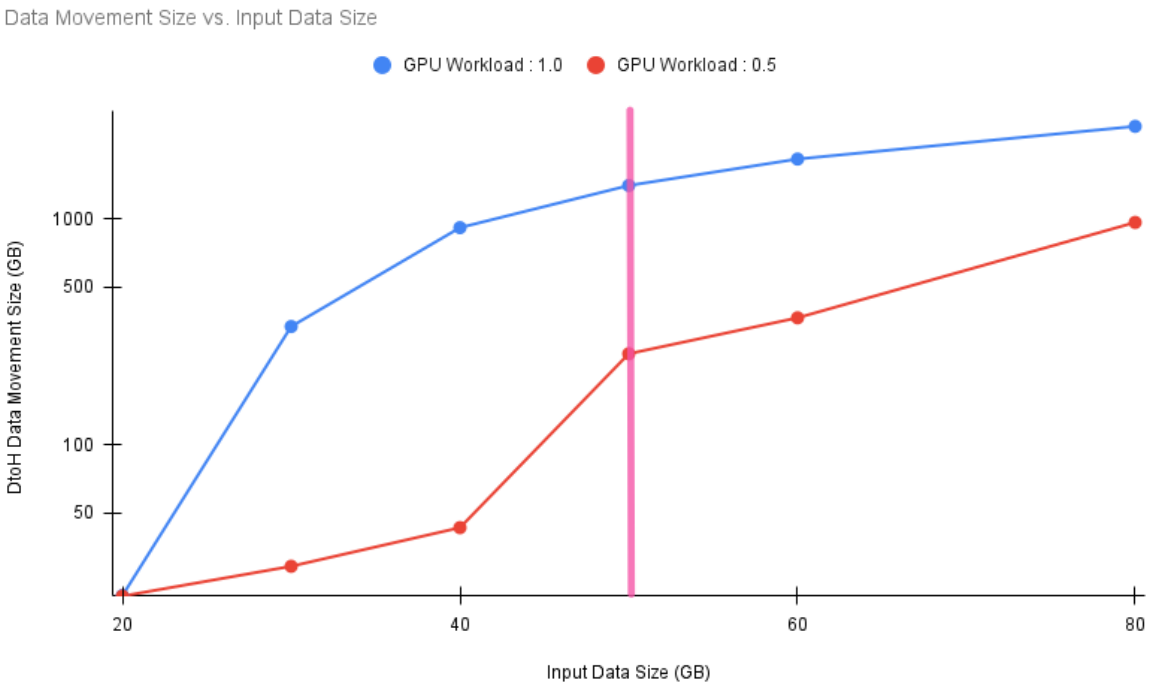

Figure 4 Device to Host (DtoH) Data Movement Size of GPU workload 1.0 and 0.5

5.3 Memory Management

Figure 4 shows the total amount of data transferred between device and host (GPU and CPU) as a function of input size, for different load factor configurations. In the GPU load factor 1.0 configuration, device-to-host (DtoH) data movement surges once the input size exceeds the GPU's capacity. Specifically, at an input of around 30 GB, the 1.0 configuration incurs an enormous surge in data movement: over 300 GB of cumulative data is transferred from the GPU to the CPU, which is in the order of ten times the original data set size. This staggering amount directly reflects the overhead of UVM page faulting and eviction. Every time the GPU's working set overflows its memory, previously evicted pages must be brought back from the CPU and other pages get evicted to make room, resulting in frequent back-and-forth data transfers. The unified memory system manages these transfers automatically but does so

reactively, causing repeated shuffling of data whenever the GPU accesses data that is not resident locally. These frequent page faults and migrations translate into hundreds of gigabytes of hidden data traffic, impacting performance as observed in the figure 2 and 3. Moreover, this spike is accompanied by excessive kernel-memory overhead: the GPU kernels spend much of their execution time stalled on memory operations (handling page faults) rather than doing useful work, which corresponds with the steep surge in sorting time for load factor 1.0 beyond 25–30 GB. Interestingly, the load factor 0.5 generates far less data movement under equivalent or even larger input sizes. Figure 4 indicates that a 60 GB dataset processed with a 0.5 load factor results in significantly less DtoH traffic than a 30 GB dataset processed with load factor 1.0. In the 0.5 case, even though the total dataset is twice as large, the GPU is only responsible for half of it (~30 GB), with the CPU handling the other half directly in host memory. Moreover, the final merge is performed on the CPU, which means the sorted GPU portion must eventually be transferred to the CPU – but this transfer can be done efficiently in bulk after the GPU has finished sorting, rather than causing continuous thrashing during the sort. Therefore, the total DtoH data moved in the 0.5 scenario grows much more slowly. The fact that the 60 GB input (with 0.5 load) incurs less traffic than the 30 GB input (with 1.0 load) underscores the benefit of avoiding full GPU oversubscription. This directly explains why, in Figure 2, the 0.5 configuration outperforms the 1.0 configuration for large inputs: it simply moves far less data to complete the sort.

Chapter 6 Conclusion and Future Work

6.1 Conclusion

A hybrid CPU-GPU approach for sorting large datasets demonstrated its effectiveness, emphasizing the importance of workload distribution and memory management. Our experiments revealed that a balanced workload, particularly a GPU load factor of 0.5, provides robust performance across various data sizes. This approach leverages the strengths of both the CPU and GPU, while mitigating the limitations imposed by the GPU's memory capacity. The findings indicate that while GPU-centric strategies excel for smaller datasets, their performance deteriorates significantly when processing data sizes that exceed the GPU's memory limits. This is primarily due to increased page faults and excessive data movement between the CPU and GPU. Conversely, a balanced approach reduces data movement and page faults, thereby maintaining higher efficiency and stability.

6.2 Future Work

Future work, we aim to fix the number of threads and grid sizes for GPU sorting for more accurate comparisons of performance across different configurations. Additionally, we could explore dynamic workload adjustments to optimize CPU-GPU distribution based on real-time performance metrics, ensuring optimal performance for a broader range of data sizes. Further development could also include advanced memory management techniques, such as explicit prefetching and memory advisement, which would mitigate the overhead associated with unified memory systems.

References

[1] Harris, M., & Sengupta, S. (2005). Improved GPU sorting. In M. Pharr (Ed.), GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (Chapter 46). Addison-Wesley. Retrieved from https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-46-improved-gpu-sorting

[2] Gowanlock, M., & Karsin, B. (2019). A hybrid CPU/GPU approach for optimizing sorting throughput. *Parallel Computing*, *85*, 45-55. https://doi.org/10.1016/j.parco.2019.01.004

[3] NVIDIA (2025), "Unified Memory in CUDA: A Beginner's Guide," Retrieved from https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[4]Rao, R. (2024, February 6). Understanding Nvidia CUDA Cores: A Comprehensive Guide. Wevolver. https://www.wevolver.com/article/understanding-nvidia-cuda-cores-a-comprehensive-guide

[5] NVIDIA. (2023), "GPU performance background: User's guide," Retrieved from https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-arch

[6] J. JáJá, An Introduction to Parallel Algorithms, Addison Wesley Longman Publishing Co., Inc., 1992.

[7] O. Polychroniou, K.A. Ross, A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort, in: Proc. of the 2014 ACM Intl. Conf. on Management of Data, 2014

[8] Van der Wijngaart, R., & Oh, F. (2022, March 23). Boosting application performance with GPU memory prefetching. *NVIDIA Developer Blog*. Retrieved from https://developer.nvidia.com/blog/boosting-application-performance-with-gpu-memory-prefetching/

[9] Miller, R., & Stout, Q. F. (1996). Parallel Algorithms for Regular Architectures: Meshes and Pyramids. MIT Press.

[10] Kitano, H., Cohen, P. R., & Belew, R. K. (Eds.). (1994). Parallel Processing for Artificial Intelligence 1. Elsevier Science Publishers.

[11] Marszałek, Z. (2017). Parallelization of modified merge sort algorithm. *Institute of Mathematics, Silesian University of Technology*. Published: September 1, 2017.

[12] Gevtushenko. (2023, October 17). Inquiry about the specific implementation of thrust::sort. *GitHub repository: NVIDIA/cccl*. Retrieved from https://github.com/NVIDIA/cccl/discussions/570

[13] Harrism. (2011, July 28). Thrust: sort_by_key slow due to memory allocation [Online forum comment]. *Stack Overflow*. https://stackoverflow.com/questions/6605498/thrust-sort-by-key-slow-due-to-memory-allocation

[14] obert_Crovella. (2014, August 29). Unified Memory vs Pinned Host Memory vs GPU Global Memory [Online forum comment]. *NVIDIA CUDA Forum*. https://forums.developer.nvidia.com/t/unified-memory-vs-pinned-host-memory-vs-gpu-global-memory/34640

[15] Adinets, A. (2020). *A faster radix sort implementation*. NVIDIA. Retrieved from https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf

[16] NVIDIA. (n.d.). CUB: Target architecture. In *cccl documentation*. Retrieved from https://nvidia.github.io/cccl/cub/#:~:text=,resources%20of%20their%20target%20architecture

[17] GNU Project. (n.d.). Using parallel mode. In *Libstdc++: Manual*. Retrieved from https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode_using.html