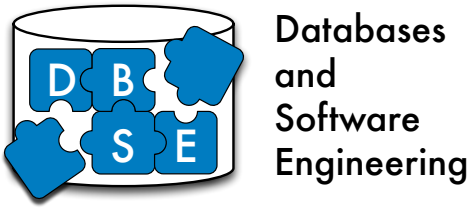
University of Magdeburg

Faculty of Computer Science



Master's Thesis

Extending CUDPP Sorting for In-Situ Aggregation for Larger Than Memory Data

Author:

Vigneshwaraa Sarangapani

December 03, 2024

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dr.-ing. Balasubramanian Gurumurthy

Department of Computer Science
Institute for Technical and Operational Information Systems
Otto von Guericke University

Sarangapani, Vigneshwaraa:
Extending CUDPP Parallel Sorting for In-Situ Aggregation for Larger Than Memory Data
Master's Thesis, University of Magdeburg, 2024.

Abstract

Large-scale data processing continues to challenge computational resources, particularly when dealing with datasets that exceed available memory in the underlying device. This research addresses critical shortcomings with respect to GPU with larger-than-memory sorting techniques by extending and improving existing approaches, specifically building upon the CUDA Parallel Primitives Library (CUDPP) and incorporating Thrust library optimizations.

The primary objective of this study was to develop an enhanced methodology for sorting and aggregating large-scale datasets using Graphics Processing Units (GPUs), overcoming traditional memory limitations. By systematically modifying existing sorting algorithms, we introduced novel techniques that improve data handling efficiency and computational performance.

Our approach implemented two primary sorting strategies: an extended CUDPP-based implementation and a modified approach utilizing Thrust's radix sort algorithm. Through comprehensive experimental evaluation, we conducted detailed performance comparisons between these implementations, analyzing their respective strengths in managing out-of-core data sorting and on-the-fly aggregation.

Key contributions include improved memory management techniques, enhanced sorting algorithms optimized for large-scale datasets, and a comparative analysis that provides insights into GPU sorting methodology. The research demonstrates significant potential for accelerating data processing tasks in high-performance computing environments, particularly in domains requiring rapid sorting and aggregation of extensive datasets.

The experimental results reveal substantial performance improvements over existing methodologies, with particular efficiency gains in handling datasets that traditionally challenge GPU memory constraints. These findings not only advance GPU sorting techniques but also provide a framework for future research in scalable data processing strategies.

Acknowledgements

I am profoundly grateful to Prof. Dr. rer. nat. habil. Gunter Saake, for giving me the opportunity to collaborate on a research project that ignited my interest and ultimately inspired my selection of this thesis topic.

I am especially grateful to Dr.-ing. Balasubramanian Gurumurthy, whose exceptional guidance, humility, and dedication deeply influenced my approach to research and concept development. He always found time to provide unwavering support, generously sharing his expertise in query processing. His insights and approach to problem-solving have been an inspiration, and I feel privileged to have learned so much from him.

Thank you for your steadfast support and encouragement. Your mentorship has been pivotal in shaping my academic and professional journey, and I am truly grateful for the opportunity to learn from someone as talented and dedicated as yourself.

Lastly, I would like to express my heartfelt gratitude to my friends and family for their unwavering support and encouragement. Their belief in me has been a constant source of strength throughout this journey, and I am deeply thankful for their love and understanding.

Contents

List of Figures xi

List of Code Listings xiii

List of Acronyms xv

1 Introduction 1

2 Background 5
2.1 GPU Architecture . . . 5
2.1.1 Graphics Processing Cluster . . . 5
2.1.2 Streaming Multiprocessors . . . 6
2.1.3 Texture Processing Cluster . . . 6
2.1.4 Warp . . . 7
2.1.5 SIMT . . . 7
2.1.6 Block . . . 8
2.1.7 Grid . . . 8
2.1.8 Cuda Cores . . . 9
2.1.9 Tensor Cores . . . 9
2.1.10 Register File . . . 9
2.1.11 Texture Unit . . . 10
2.1.12 L1/Shared Memory . . . 10
2.2 GPU Memory Architecture . . . 10
2.2.1 GDDR6 SDRAM . . . 11
2.2.2 L2 Cache System . . . 11
2.3 Sorting . . . 11
2.3.1 Radix Sort . . . 12
2.3.2 Merge Sort . . . 13

3 Parallel Processing and CUDPP Implementation of Merge Sort 15
3.1 Parallel Processing . . . 15
3.2 CUDPP . . . 16
3.2.1 Parallel Reduction . . . 17
3.2.2 Split Operation . . . 17
3.2.3 Parallel Prefix Sum . . . 17
3.2.4 simpleCopy . . . 18
3.2.5 Sorting Stages . . . 19
3.3 Binary Search . . . 20

3.4 blockWiseSort 20
3.5 compareSwapVal 21

4 Implementation **23**
4.1 Sorting Larger than Memory Data Sizes 23
4.2 In Situ Aggregation 28
4.2.1 Exclusive Aggregation 28
4.2.2 By-Product Aggregation 29

5 Evaluation **33**
5.1 Evaluation Setup 33
5.1.1 Dataset Distribution 33
5.1.2 Environment Setup 34
5.2 Data Grouping 34
5.3 Hard Hitter 35
5.4 Unique Values 37
5.5 Aggregation 38
5.6 Evaluation Results 40

6 Related Work **41**

7 Conclusion and Future Work **43**

Bibliography **45**

List of Figures

2.1 Turing GPU Architecture 6
2.2 Turing Streaming Multiprocessor 7
2.3 Thread Block 8
2.4 SIMT Branching as explained in [Cor24] 8
2.5 Tensor Core 9
2.6 GPU Memory Architecture 10
2.7 Radix Sort 12
2.8 Merge Sort 13

3.1 CUDPP Layers 16
3.2 Split Operation 17
3.3 Prefix Sum 18

4.1 Window Value Comparison 26
4.2 Exclusive Aggregation 28
4.3 By Product Aggregation 30

5.1 Data Grouping 35
5.2 Hard Hitter 36
5.3 Hardhitter Sorting Stages 36
5.4 Unique Values 37
5.5 Unique Values Sorting stages 38
5.6 Unique Values 38
5.7 Data Grouping 39
5.8 Hard Hitter Values 39

List of Code Listings

3.1 Simple Copy 19
3.2 Binary Search Lower 20
3.3 Binary Search Higher 20
4.1 Initial Sort 24
4.2 Finding Position in Window 24
4.3 Windowing Condition 25
4.4 Copying Values from first array 25
4.5 Copying Values from second array 25
4.6 Copying Default Values 26
4.7 Sort Loading 27
4.8 Placing based on position 27
4.9 Validating the sorting 27
4.10 Exclusive Aggregation 29
4.11 By Product Aggregation Zero Location 30
4.12 By Product Aggregation First Location 31

List of Acronyms

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CTA	Cooperative Thread Array
CUDA	Compute Unified Device Architecture
CUDPP	CUDA Data Parallel Primitives
FPU	Floating Point Unit
Gbps	Gigabits per second
GPC	Graphics Processing Cluster
GPU	Graphical Processing Unit
RAM	Random Access Memory
SDRAM	Synchronous dynamic random-access
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple threads
SM	Streaming Multiprocessor
SPMD	Single program, multiple data
SSD	Solid State Drive
TPC	Texture Processing Cluster
VRAM	Video Random Access Memory

1. Introduction

In computer science, sorting is a basic function that is necessary for many applications, such as databases, search engines, and data analysis. Sorting algorithms have historically been run on Central Processing Units (*CPUs*), which sequentially process instructions. Due to the sequential architecture, CPUs are inherently limited by the speed of a single core, making sorting operations time-consuming. Although there are CPUs with multiple cores, each core still typically handles only a few threads in parallel. High levels of parallelism are necessary for effectively sorting large datasets, which presents a problem for CPUs because the cores have to precisely coordinate their jobs, which can slow down the process thus making it less efficient.

Large dataset sorting also requires frequent data access, which frequently results in a memory bandwidth bottleneck. CPUs generally have limited memory bandwidth relative to the amount of data in large sorting tasks, causing delays as cores wait for data from memory. While CPUs are general-purpose processors, they lack specialized hardware like dedicated sorting accelerators which are specifically designed to handle sorting of large datasets efficiently.

Due to those limitations, sorting on CPUs can become difficult and resource-intensive as data sizes increase. For large-scale sorting tasks, alternative architectures such as GPUs are more viable.

Sorting in GPU

Advances in their architecture have influenced the evolution of Graphical Processing Units (*GPUs*) for sorting, making them more effective at managing large-scale, parallelizable activities like sorting. At first, GPUs were mainly made for handling visual input and displaying graphics, with an emphasis on parallel pixel processing. However, over time, their highly parallel structure began to appeal to researchers and engineers looking to accelerate non-graphics tasks, especially in data-intensive operations like sorting.

Early on, researchers saw that the GPUs parallel capabilities could be useful for sorting tasks. Bitonic and Radix sort are two examples of sorting algorithms that

were modified for GPUs, utilizing their parallel nature to attain improved performance compared to CPUs for large datasets. Radix-based sorting techniques have also been refined, with implementations achieving over one billion keys per second, a substantial improvement over CPU-based methods.[SJ17]

Sorting on GPUs presented memory access issues because, in general, GPU memory management was less flexible than CPU memory management. In order to solve this, glsplGPU developed additional shared memory, which allowed for quicker data transfers between processing cores while sorting. Sorting efficiency was further increased by lowering bottlenecks thanks to these memory optimizations.

Tensor Cores and better cache architectures are examples of recent GPU innovations that, while primarily aimed at machine learning, also increase sorting task performance by speeding up arithmetic and data handling. These features of architecture enable GPUs to handle larger amounts of data and carry out complex tasks needed by sophisticated sorting algorithms.

Goal of this Thesis

The CUDA Data Parallel Primitives (*CUDPP*) is highly significant for GPU sorting due to its role in enabling efficient, scalable, and high-performance sorting algorithms on GPUs. However, One early study addressed the limitations of GPU memory and the need for external sorting when data exceeds the GPUs memory capacity, introducing methods to reduce transfer overhead and balance load across GPUs.[PSHL10]. The primary goal of this thesis is to develop efficient techniques for processing large-scale data that exceeds the memory capacity of standard processing units, with a particular focus on sorting and in-situ aggregation on GPUs. This research aims to address the challenges of handling datasets larger than available memory, minimizing transfer overhead, and optimizing load balancing across GPUs. By exploring both sorting and aggregation, the thesis seeks to improve the efficiency and scalability of data processing on GPUs

As part of the research, we will explore the following research questions:

1. How can efficient sorting algorithms be designed to handle datasets larger than available GPU memory?

2. How can in-situ aggregation be optimized on GPUs for large-scale datasets, maintaining memory efficiency and reducing the amount of data that must be transferred during computation?

To address the above-mentioned challenges, the research proposes a windowing mechanism. In this mechanism, the dataset is partitioned into smaller chunks that fit the GPU's memory and processed. After processing, the chunks are merged back together.

Structure of the Thesis

In Chapter 2, we discuss the architecture of the Graphical Processing Unit and the sorting techniques used. In Chapter 3, we discuss how parallel processing is carried out in GPU. Further, we also explain the implementation of the sorting methods in CUDPP. We present the implementation details of sorting larger-than-memory data sizes and in situ aggregation in Chapter 4. The comparison results of sorting and aggregation are presented in Chapter 5. We provide details of related work in Chapter 6 and our conclusion in Chapter 7. Finally, we detail the future work that can be done in Chapter 8.

2. Background

The process of sorting on a Graphics Processing Unit (GPU) involves multiple components working together to efficiently manage data organization. In this chapter, we explore the architecture of GPU and how they are responsible for processing data. In Section 2.1, we explain the intricate components present in GPU. Following that we delve into the mechanism of various sorting algorithms in Section 2.2

2.1 GPU Architecture

Sorting is one of the primary algorithms used in many operations. Most sorting algorithms are performed in the CPU. Parallel processing can be executed in a multi-core CPU but requires a more complex implementation. With the influence of GPU, parallel processing is made more efficient. The number of threads used in GPU is comparatively higher than those used in CPU. Even though GPU offers higher thread, other obstacles include limited memory capacity, data transfer overhead, and Implementation Complexity. In this section, the hardware unit of RTX 2080Ti following Turing Architecture (as shown Figure 2.1)is explained. A GPU consists of Graphics Processing Clusters (*GPCs*), Texture Processing Clusters (*TPCs*), and Streaming Multiprocessors (*SMs*) [NVI18].

2.1.1 Graphics Processing Cluster

Graphics Processing Clusters (GPCs) are components that are in control of the processing power of the GPU. The GPC is the dominant high-level hardware block with all of the key graphics processing units residing inside the GPC.[NVI20] It is responsible for organizing the process of carrying out graphics rendering jobs like rendering a 3D game scene, processing visual effects for a movie, or generating real-time graphics for simulations. They have committed hardware blocks for computing, rasterization, texturing, and shading.

Figure 2.1: Turing GPU Architecture

2.1.2 Streaming Multiprocessors

Streaming Multiprocessor (SM) is one of the fundamental components in the GPU architecture. They play an important role in the execution of parallel computation. They are essential for attaining the high computational throughput and efficiency that GPUs are renowned for in a variety of fields, including artificial intelligence, scientific research, gaming, and computer graphics. Task scheduling and resource allocation are handled by SMs, which also distribute threads among available cores for best performance. They assign resources like registers and shared memory and oversee how each thread's instructions are carried out. SMs are arranged into groups, frequently referred to as "warps," that carry out instructions concurrently. Once a block of threads is executed parallelly, they begin executing the next thread block in sequence. Specialized processing units that are optimized for mathematical and arithmetic operations, like Arithmetic Logic Units ($ALUs$) and Floating Point Units ($FPUs$), are found in SMs and can be used for a variety of computational tasks.[SGNB17]. SMs can effectively control data access and reduce the amount of time spent waiting for data from the main memory since they have their memory structure, which includes local registers, shared memory, and cache as indicated in Figure 2.2.

2.1.3 Texture Processing Cluster

Texture Processing Clusters (TPCs) are a group of Streaming Multiprocessors (SMs) and a polymorph engine. The polymorph engine is a specialized unit present in each TPC and is in control of handling Vertex Fetch, Tessellation, viewport transform, attribute setup, and stream output.[NVI21]. Texture Processing Cluster contributes to the overall processing power of GPU. They are used in managing and distributing the workload across the Streaming Multiprocessors. The foundational terms and concepts will be thoroughly explained in the sections that follow, providing

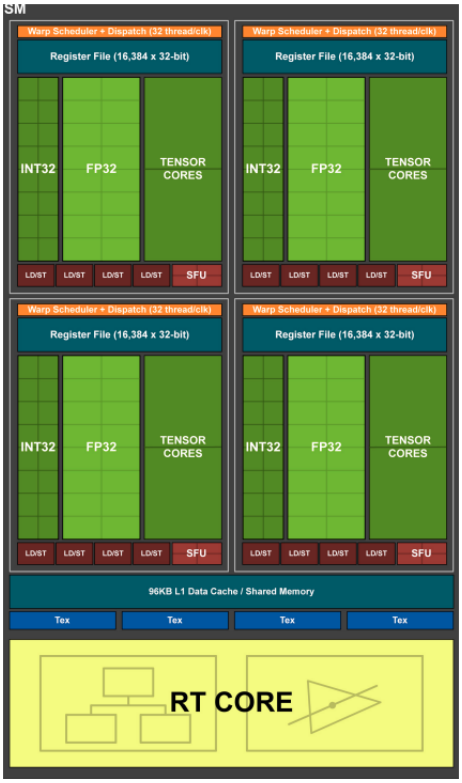


Figure 2.2: Turing Streaming Multiprocessor

a comprehensive understanding of the essential terminology used throughout this discussion.

2.1.4 Warp

A warp is a group of 32 threads that execute the same instruction when placed within a thread block as indicated in Figure 2.3. A thread block will be further separated into a series of warps for execution after it has been assigned to an SM. Each thread block contains a fixed number of warps. A thread will never be split between two warps. The task of scheduling the warps to the SM's underlying cores falls to the warp schedulers found in each SM. This hardware scheduler reduces the impact of instruction latency by allocating more warps than the number of cores on a GPU.

2.1.5 SIMT

Single instruction, multiple threads (*SIMT*) is an execution model used in parallel computing where Single instruction, multiple data (*SIMD*) is combined with multithreading. It is different from Single program, multiple data (*SPMD*) in that all instructions in all "threads" are executed in lock-step.[WIK24]A single instruction in pure SIMD affects every piece of data in precisely the same way. This limitation is somewhat relaxed in SIMT: certain threads can be activated or deactivated, allowing instructions and data to be handled only on active threads while leaving local data unaltered on idle threads.

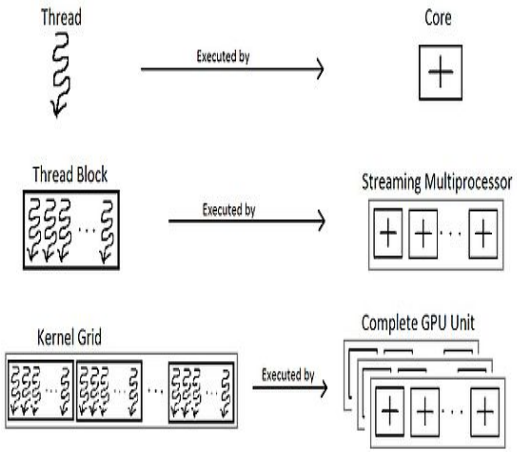


Figure 2.3: Thread Block

As a result, SIMT can accommodate branching(as shown in Figure 2.4), though not very efficiently. Given an if-else construct beginning with if (condition), the threads for which condition==true will be active when running statements in the if clause, and the threads for which condition==false will be active when running statements in the else clause. The results should be correct, but the inactive threads will do no useful work while they are waiting for statements in the active clause to be completed. Branching within SIMT is illustrated in the figure below.

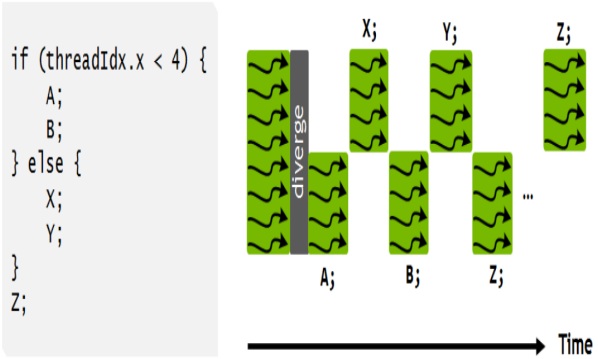


Figure 2.4: SIMT Branching as explained in [Cor24]

2.1.6 Block

A group of concurrently running threads that are located on the same shared memory (SM), share its resources, and communicate with one another through various hardware techniques is known as a thread block. Within its grid, every thread block has a block ID. A thread block can be one, two, or three-dimensional.

2.1.7 Grid

A grid is a collection of thread blocks that a kernel launches and uses to read data from global memory, write results to the global memory, and ensure that all nested kernel calls are synchronized.

2.1.8 Cuda Cores

The processors underlying GPU computing are called Compute Unified Device Architecture (*CUDA*) cores, which enable parallel processing and boost performance across a range of applications. Within a graphics processing unit (GPU), CUDA cores are distinct processing units. They operate similarly to tiny processors, able to handle several calculations at once. CUDA cores split up computational workloads into smaller, independent parts and process each one separately. Parallel processing enables the speedier execution of data-intensive tasks and complex algorithms. Organizations can save a lot of money on hardware upgrades and electricity consumption by using CUDA cores in GPU computing workflows rather than just CPUs or clusters.

2.1.9 Tensor Cores

Figure 2.5 shows the Tensor Cores which enable dynamic calculation and mixed precision computing. They are powerful cores that accelerate overall performance while preserving accuracy. To speed up the computation, they combine lower-precision and higher-precision data types. A tensor is a data type which considered a container to store multi-dimensional datasets. The tensor core can perform multiple operations in one clock cycle. The fused multiply-addition algorithms are employed by tensor cores. The matrix multiplications are logically simple, but because each calculation needs registers and caches to store intermediate results, the entire process is computationally demanding. For these types of calculations, the Tensor cores are particularly well-suited.

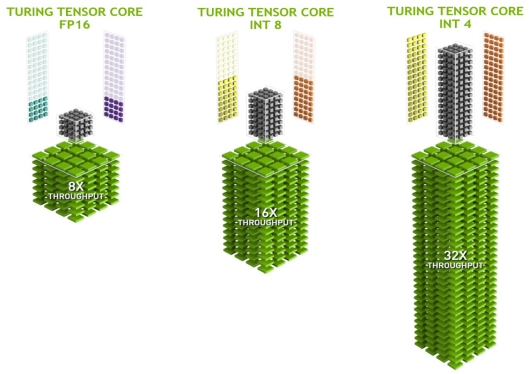


Figure 2.5: Tensor Core

2.1.10 Register File

The memory region that feeds directly into the CUDA cores is referred to as the register file. It is thus arranged into 32 banks, which correspond to the 32 threads in a warp. The register file is a large matrix with 32 columns and many rows made up of 4-byte elements. A warp works on complete rows, with each thread (CUDA core) working on a distinct column (bank) within a given row.

2.1.11 Texture Unit

Texture units are responsible for handling texture fetches. Textures are 2D or 3D data arrays that are frequently utilised in applications related to graphics and parallel computing. Texture units are specialised parts made to effectively sample and retrieve data from these textures while GPU kernels are running. Anisotropic filtering, bilinear interpolation, and managing memory access patterns appropriate for textures are among the functions performed by texture units. [RD17]

2.1.12 L1/Shared Memory

The standard on-chip storage region known as L1 offers quick access to data that has recently been read from or written to main memory (Random Access Memory (*RAM*)). L1 acts as the overflow region when "register spilling" occurs—a situation in which an SM's register file cannot contain all of the active data. The shared memory area is situated in the same physical memory as the L1 cache; however, it is not identical to L1 in that any thread within a thread block can access all of its data. Thread communication and data sharing are made possible as an outcome.

2.2 GPU Memory Architecture

GPU memory is arranged in a hierarchical structure according to its speed, capacity, and use. Registers, on-chip caches, and global memory (Video Random Access Memory (*VRAM*)) are commonly included in the hierarchy. The hierarchy's levels each have a distinct function in speeding up computations. Memory bandwidth, or the speed at which data can be read from or written to memory, is another factor that affects a GPU's performance in addition to memory size. Sufficient memory bandwidth is essential for supplying GPUs' enormous parallel processing power. Figure 2.6 shows the memory system of Turing architecture consisting of GDDR6 SDRAM, L2 Cache, and local memory which are specially mapped regions of main memory that are allocated to each SM.

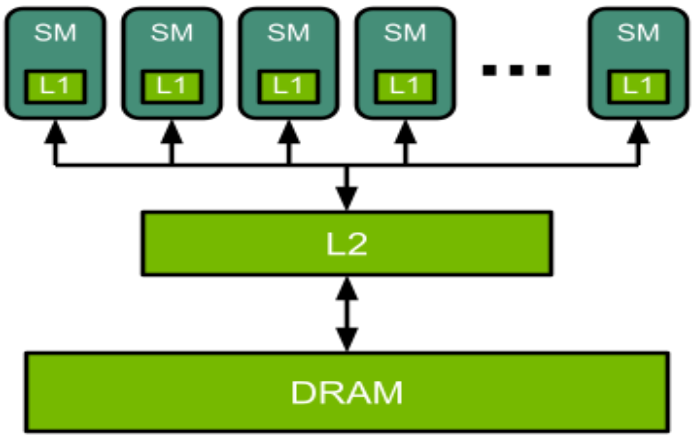


Figure 2.6: GPU Memory Architecture

2.2.1 GDDR6 SDRAM

Global Memory constitutes the majority of the device's primary memory. This is the primary storage space for the different kinds of data needed by the GPU, such as textures, frame buffers, shaders, and other resources for rendering and parallel computing. The GPU's Global Memory can store a significant amount of data because it usually has a larger capacity than other memory types. In general, access to global memory is slower than that of registers and caches. GDDR6 is a high-speed Synchronous dynamic random-access (*SDRAM*) memory intended to accommodate both new applications that require even more memory bandwidth and legacy applications that support high-bandwidth applications like graphic cards, game consoles, and high-performance compute systems. Multiple data chunks (words or bits) can be transferred simultaneously in a single clock cycle employing GDDR6's parallel architecture. The higher overall bandwidth is a result of this parallelism. With performance demonstrated at 18 Gigabits per second (*Gbps*) and a per-device bandwidth of 72 GB/s, GDDR6 is a cutting-edge solution for graphics memory. Turing is the first GPU architecture to support GDDR6 memory.[NVI18]

2.2.2 L2 Cache System

Thousands of threads that need to access data simultaneously must be managed concurrently, which presents a significant challenge in GPU architecture. This complexity exceeds the demands that are typically found in CPU environments, where a few dozen threads or fewer can run concurrently. An L2 cache is a crucial component for overcoming this difficulty and providing optimal data access across multiple threads. A GPU's on-chip memory system known as the L2 cache (Level 2 cache) is located between the off-chip global memory and the on-chip L1 cache. The purpose of the L2 cache is to accelerate further reloads. They increase throughput by assembling multiple small transfers into larger, more efficient memory requests. The main function of the L2 cache is to serve as a bigger, more shared cache across multiple Streaming Multiprocessors (SMs). It lessens the need to retrieve data directly from the slower off-chip global memory and helps lower memory access latency. L2 Cache's shared nature makes it easier to communicate between the GPU's various processing units and improves cache space utilization. Coherency and consistency in the data stored across the various SMs are upheld by the L2 cache. By guaranteeing that every processing unit sees the same representation of memory, it avoids problems caused by inconsistent data.

2.3 Sorting

Sorting is a fundamental operation that is a time-critical component of various applications such as databases and search engines.[WS10] Because of its significance, sorting is one of the most studied algorithms and a fundamental computational building block. The effectiveness of sorting procedures is a fundamental component of many algorithms' efficiency. The size of datasets users are dealing with is constantly increasing and it calls for sorting algorithms that can run efficiently on large-scale computing systems.[SSD+16]

2.3.1 Radix Sort

An algorithm for non-comparative sorting is radix sort. It is a linear sorting algorithm. The time complexity of Radix sort is O(nd), where n is the size of the array and d is the number of digits in the largest number, and space complexity is O(n + k) where n is the size of the array and k is the range of each digit, typically the base. It is considered to be a stable sort because it maintains the relative order of elements with equal values. The Radix sort algorithm is given in Algorithm 1 Generating and allocating elements into buckets based on their radix prevents comparison. This bucketing procedure is repeated for each significant digit in elements with multiple significant digits, maintaining the order from the previous step, until all digits are taken into account. Radix sort has also been referred to as bucket sort and digital sort for this reason. The sort works by ordering each digit from least significant to most significant. Radix sorting in base 10 would proceed by sorting the digits in the one's place, ten's place, and so forth. To sort the values in each digit place, Radix sort employs counting sort as a subroutine. An example of Radix sort is explained in Figure 2.7

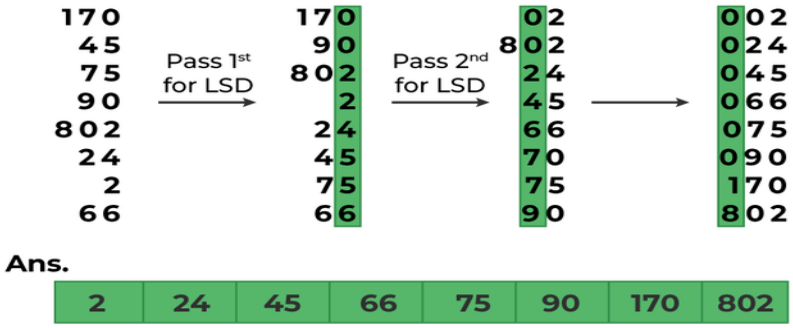


Figure 2.7: Radix Sort

Algorithm 1 Radix Sort

```
function RadixSort(Array, p)                          ▷ p is the number of passes
    for j ← 1 to p do
        int count_array[10] ← {0}
    for i ← 0 to n do
        count_array[key of(Array[i]) in pass j] + +
    for k ← 1 to 10 do
        count_array[k] ← count_array[k] + count_array[k − 1]
for i ← n − 1 downto 0 do
        result_array[count_array[key of(Array[i])]] ← Array[j]
count_array[keyof(Array[i])] − −
for i ← 0 to n do
        Array[i] ← result_array[i]                    ▷ Update the main array
end function
```

2.3.2 Merge Sort

A popular, reliable, and effective sorting algorithm that adheres to the divide-and-conquer strategy is merge sort. The Merge sort algorithm is given in Algorithm 2. The sort is carried out by dividing the array into halves, further dividing each halves into equal halves, and so on until they are small enough as shown in Figure 2.8. Then each chunk is sorted and then merged to form the previous length, recursively until it forms a single array of actual size. A list can be effectively sorted in O(n*log(n)) time using merge sort. Without requiring additional space, merge sort can be utilized with linked lists. Although it's efficient, the merge sort requires additional space of O(n) for the temporary array. The merge sort goes through the entire process even though the array is sorted, which is a drawback.

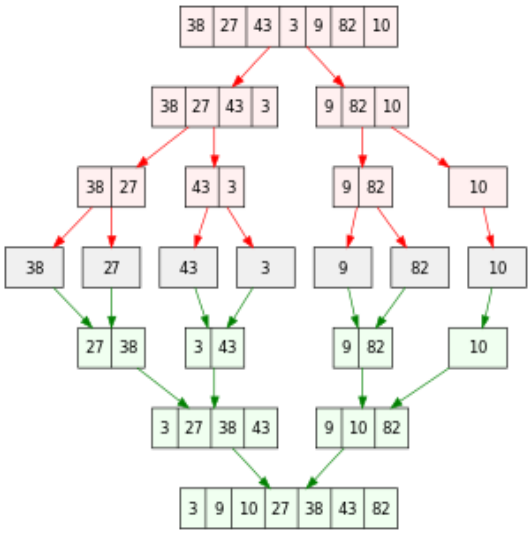


Figure 2.8: Merge Sort

Algorithm 2 Merge Sort

```
function MERGE(A, p, q, r)        ▷ Where A - array, p - left, q - middle, r - right
    n_1 = q − p + 1
    n_2 = r − q
    Let L[1 ... n_1 + 1] and R[1 ... n_2 + 1] be new arrays
    for i = 1 to n_1 do
        L[i] = A[p + i − 1]
    for j = 1 to n_2 do
        R[i] = A[q + j]

L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r do
        L[i] < R[j]
A[k] = L[i]
i = i + 1
L[i] > R[j]
A[k] = R[j]
j = j + 1 else
        A[k] = −∞          ▷ We mark the duplicates with the largest negative integer
j = j + 1
```

end function

So far, in this chapter, we explored the efficiency of GPUs in performing high-speed sorting operations, showcasing their capability to handle large-scale data with remarkable performance improvements. Building on this foundation, the next chapter delves into the configuration and utilization of CUDPP, a powerful library designed to harness the potential of GPUs for various parallel computing tasks.

3. Parallel Processing and CUDPP Implementation of Merge Sort

In the last chapter, we explained the architecture of GPU and the working of the sorting algorithm. In this chapter, let's delve into Parallel Processing in Section 3.1 and the nuanced implementation intricacies of CUDPP in Section 3.2. Then we discuss Parallel Reduction in Section 3.2.1, Split Operation in Section 3.2.2, and Parallel prefix sum in Section 3.2.3. Then, we will discuss how CUDPP has implemented multi-stage such as simpleCopy in Section 3.2.4, sorting stages in Section 3.2.5. Finally, we will discuss how binary search is carried out in Section 3.3 and block-wise sort in Section 3.4

3.1 Parallel Processing

GPU is a data-parallel processor, that has many cores runs thousands of parallel threads, and processes thousands of data. All the data are processed by the same program, as it uses the SIMT computational model. In GPU data parallelism is achieved by executing the same program on multiple data elements concurrently, which contrasts task parallelism (executing different tasks parallelly) and Instruction level parallelism (exploiting parallelism within a single thread of execution by reordering or executing instructions in parallel when possible).

To facilitate efficient data parallelism, it is imperative to utilize foundational primitives meticulously crafted with a profound understanding of parallel algorithmic intricacies and optimization strategies. These primitives are tailored to address prevalent scenarios encountered in parallel computations such as generating a single value from many parallel threads, partitioning of data from many parallel threads, and output of many parallel threads with variable output per thread.

Within this paradigm, the CUDA Data Parallel Primitives (CUDPP) library emerges as an invaluable toolset. CUDPP offers a comprehensive suite of optimized parallel primitives meticulously engineered for NVIDIA CUDA-enabled GPUs.

In this chapter, let's delve into the nuanced implementation intricacies of CUDPP, elucidating the design principles, optimization methodologies, and architectural considerations underlying its efficacy in harnessing the immense parallel processing capabilities inherent in GPU architectures and how merge sort is implemented in CUDPP.

3.2 CUDPP

CUDPP, the acronym for the CUDA Data Parallel Primitives Library, stands as a foundational resource encompassing a diverse array of data-parallel algorithmic primitives. These encompass critical functionalities such as parallel-prefix-sum (often referred to as "scan"), parallel sorting, and parallel reduction. These primitives serve as indispensable architectural elements, forming the bedrock for an extensive spectrum of data-parallel algorithms. Their utility extends across essential computational tasks, ranging from sorting and stream compaction to the construction of intricate data structures like trees and summed-area tables. It is written in C for CUDA.[HSO07] CUDPP is designed to exclusively operate on GPU data within GPU memory, rendering it incapable of supporting data transfer or memory allocation operations. This design choice underscores its specialization in leveraging the computational power of the GPU while eschewing involvement in tasks related to data movement or memory management.

It is structured into four layers to provide a comprehensive framework for parallel computing tasks as explained in Figure 3.1.

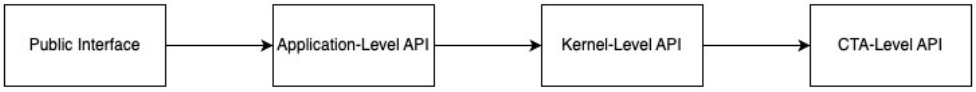


Figure 3.1: CUDPP Layers

- At the topmost layer is the Public Interface, serving as the external library interface and the primary entry point for most applications. This layer abstracts the underlying functionality and provides a user-friendly interface for interacting with CUDPP. It interfaces with the Application-Level API.
- The Application-Level API encompasses functions callable from CPU code. These functions facilitate the execution of code jointly on both the CPU (host) and the GPU. They serve as a bridge between the higher-level application logic and the lower-level parallel computation routines. The Application-Level API interacts with the Kernel-Level API.
- The Kernel-Level API consists of functions that run entirely on the GPU, leveraging the parallel processing capabilities of the device across an entire grid of thread blocks. These functions are responsible for executing core data-parallel algorithms efficiently. They may, in turn, call into the CTA-Level API.

- At the lowest level is the CTA-Level API, comprising functions that run exclusively on the GPU within a single Cooperative Thread Array (*CTA*), also known as a CUDA thread block. These functions operate at a lower level of abstraction and typically implement core data-parallel algorithms by processing data within shared memory, maximizing efficiency and performance.

3.2.1 Parallel Reduction

In data parallel computing, parallel reductions are basic operations that are often used to compress a vector of data into a single value based on a specified binary associative operator. Addition (+), multiplication (*), determining the minimum or maximum value (min/max), and carrying out logical operations like AND or OR are some examples of these operators. Because they enable parallelization without changing the outcome, binary associative operators are essential for parallel reductions. Reductions are usually realized by tree-based algorithms, in which the hierarchical organization of computations allows for the effective use of parallelism. This method guarantees that the processing resources that are available are used as efficiently as possible, which speeds up the computation of the final reduced value.

3.2.2 Split Operation

The "Split Operation" entails segregating elements in an array based on a specified criterion, typically with true elements grouped together at the beginning. This operation is particularly relevant in scenarios where data needs to be partitioned or sorted based on certain conditions. For instance, in sorting algorithms or constructing tree data structures, segregating elements based on a Boolean flag is a common requirement. In the provided example presented in Figure 3.2, the array consists of boolean flags denoting true or false values, accompanied by payloads associated with each element. The desired outcome is an array where all true elements are positioned at the beginning, followed by false elements, while maintaining the original order within each group. This operation is essential for various applications, such as sorting datasets based on specific criteria or organizing data for efficient tree construction.

Figure 3.2: Split Operation

3.2.3 Parallel Prefix Sum

The parallel prefix sum, also known as scan, is a foundational operation in parallel computing, utilized to efficiently compute prefix sums of elements within an array. Given an array, $A = [a_0, a_1, \ldots, a_{n-1}]$ and a binary associative operator $\bigoplus$ with an

identity element I, the scan operation generates a new array where each element represents the result of combining all elements up to that position in the original array using the operator $\oplus$. This process entails initializing the output array with the identity element at the first position, followed by iteratively computing the prefix sums for each subsequent position in the output array by combining the corresponding elements from the input array. The resulting array encapsulates the cumulative sums of elements up to each position in the original array, facilitating efficient computation of various algorithms in parallel computing contexts. For example, given the input array [2, 3, 7, 5], the scan operation yields the output set [2, 5, 12, 17] as shown in Figure 3.3.

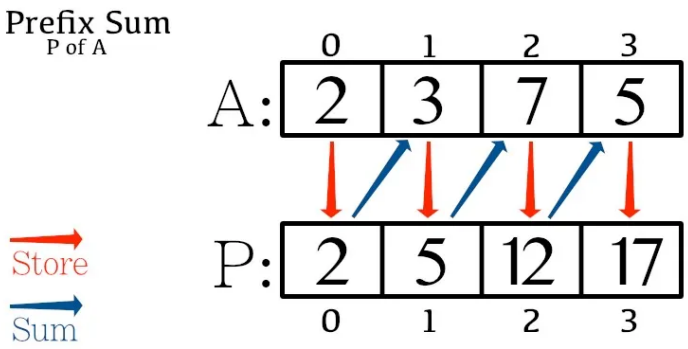


Figure 3.3: Prefix Sum

3.2.4 simpleCopy

Listing 3.1 explains the implementation of simple copy in CUDPP. The primary objective of this kernel is to optimize the process of copying key-value pairs in a parallel computing environment. Each thread within a block is assigned the task of copying a single key-value pair from input arrays to output arrays. This parallelization strategy leverages the computational power of the GPU by organizing threads into blocks and assigning them unique identifiers calculated from block and thread indices, facilitating efficient parallel processing across the device.

To ensure optimal performance, the copying process incorporates checks to ensure that each thread operates within the specified range of elements designated for copying. By adhering to these constraints, threads can independently execute the copying operation concurrently, thereby enhancing overall computational efficiency.

Upon completion of the copying operation, the sorting process proceeds in two stages: simpleMergeLower and simpleMergeHigher. In the first stage, simpleMergeLower, the lower (left) block of the partition undergoes sorting and merging operations. Subsequently, in the simpleMergeHigher stage, the upper (right) block of the partition is similarly sorted and merged. This staged approach to sorting allows for efficient organization and manipulation of data, optimizing performance and resource utilization within the parallel computing environment.

Listing 3.1: Simple Copy

```
__global__
void simpleCopy(T *A_keys_dev, unsigned int *A_vals_dev,
T *A_keys_out_dev, unsigned int *A_vals_out_dev, int offset,
                int numElementsToCopy) {
    int myId = blockIdx.x * blockDim.x + threadIdx.x;

    if (myId == 0 && threadIdx.x == 0)
        __syncthreads();

    if (myId >= numElementsToCopy)
        return;
    A_keys_out_dev[offset + myId] = A_keys_dev[offset + myId];
    A_vals_out_dev[offset + myId] = A_vals_dev[offset + myId];

    if (myId == 0 && threadIdx.x == 0)
        __syncthreads();
}
```

3.2.5 Sorting Stages

simpleMergeLower

This CUDA kernel employs an efficient block-level merging strategy to sort and merge arrays, enhancing performance by utilizing shared memory and gracefully handling edge cases. Each thread within a block is responsible for loading a total of 6 keys and 6 values, with 2 keys and values stored in registers and 4 keys and values residing in shared memory. To optimize memory access, the maximum values from the register are stored in shared memory. Subsequently, a loop is initiated to iteratively insert values into the output array until all elements are processed. Synchronization among threads within the block ensures coherent execution. During each iteration, a binary search is conducted on the keys stored in shared memory to identify the insertion point for the keys stored in registers. Once determined, the keys and corresponding values are inserted into the output array. The process repeats, with subsequent sets of keys and values loaded into registers and shared memory until the loop meets the termination condition. Upon completion of this kernel execution, the left block of the partition is effectively sorted, contributing to the overall sorting and merging process.

simpleMergerHigher

The implementation of this CUDA kernel mirrors the structure of simpleMergeLower, as explained previously. However, it diverges in its approach to binary search: here, the search is directed towards finding the upper-bound insertion point, whereas simpleMergeLower seeks the lower-bound insertion point. Upon completion of this kernel execution, the sorting of the right block of the partition ensues. Ultimately, once both kernel executions conclude, the entire partition undergoes sorting, culminating in the successful completion of the sorting and merging process

3.3 Binary Search

In the lower version of the binary search as shown in Listing 3.2, the algorithm partitions the search space into fragments with progressively decreasing offsets. It dynamically adjusts the mid-point index based on whether the current value is greater than or equal to the target value. Conversely, in the higher version(Listing 3.3), the mid-point index is adjusted based on whether the current value is greater than the target value. Through iterative refinement, the binary search process continues until the correct insertion position is identified. This dual approach enables the efficient determination of insertion points for both lower and upper bounds within the sorted array, contributing to the overall effectiveness of the sorting and merging process.

Listing 3.2: Binary Search Lower

```
inline void binSearch_fragment_lower(T* binArray, int offset,
int &mid, T testValue)
{
    mid = (binArray[mid] >= testValue ? mid-offset : mid+offset);
}
```

Listing 3.3: Binary Search Higher

```
inline void  binSearch_fragment_higher(T* binArray, int offset,
int &mid, T testValue)
{
mid = (binArray[mid] > testValue ? mid-offset : mid+offset);
}
```

3.4 blockWiseSort

The CUDA kernel function blockWiseSort orchestrates a sophisticated block-wise sorting operation on arrays comprising keys and their corresponding values. This operation employs a blend of merge sorting and register sorting techniques to efficiently organize the data. Initially, the function is initialized by defining necessary constants and loading data into registers. Thread and block indices are then mapped out to facilitate parallel processing across the GPU.

The sorting process commences within registers, where a sequence of compareSwapVal calls is executed to sort the elements efficiently. Once sorted, the function combines the sorted blocks, ensuring coherence in the overall sorting process. To determine the optimal insertion points for elements from registers into the shared memory array scratchPad, a binary search is conducted. Subsequent refinement through linear searches further enhances the precision of the insertion points.

Once the elements are sorted and properly positioned within scratchPad, they are written back to memory. Specifically, the sorted elements are coalescedly written back to the global memory arrays containing the keys and values. Throughout the

execution of this kernel function, optimization techniques such as coalesced memory access, shared memory efficiency, and manual loop unrolling are employed to maximize performance. These strategies ensure that the sorting operation is executed with optimal speed and efficiency, harnessing the full potential of the GPU architecture.

3.5 compareSwapVal

This device function facilitates the comparison and swapping of values within an array, requiring two values, A1 and A2, along with their respective indices, index1 and index2. Additionally, it takes into account a constant array named stringVals, the total number of elements, and pointers to the arrays scratch and stringVals2. Initially, it compares A1 with A2, and if A1 is found to be greater than A2, it swaps both A1 and A2 as well as their corresponding elements in the scratch array.

However, in the event that A1 is equal to A2, the function conducts additional checks to resolve the tie. It begins iterating through the elements of the stringVals2 array, starting from a calculated offset based on the current index. This iteration continues until unequal elements are encountered or until the end of the array is reached.

If the resolved elements from stringVals2 indicate that A1 should precede A2, based on the sorting criteria, the function swaps A1 and A2 along with their corresponding elements in the scratch array. This meticulous comparison and swapping process ensures the accurate sorting and manipulation of elements within the array, maintaining integrity and orderliness throughout the sorting operation.

In this chapter, we detailed the configuration of CUDPP for sorting, highlighting its capabilities and setup process for efficient GPU-based Sort operations. The next chapter will extend these concepts to address more complex challenges, such as sorting datasets larger than the available memory and implementing in-situ aggregation.

4. Implementation

The previous chapter explains the CUDPP implementation of Merge Sort for aggregating offloaded data. In this chapter, we extend the existing implementation strategies to accommodate larger memories in Section 4.1. Further, we also include an additional function to support In-situ aggregation to enable faster-ordered aggregation, enhancing the efficiency and scalability in GPUs in Section 4.2.

4.1 Sorting Larger than Memory Data Sizes

The research expands the capability of the CUDPP merge sort, allowing for the sorting of values larger than the device memory size, which is supported by CUDPP which was previously feasible within GPU memory constraints.

Our implementation has two parts:

- CPU part, to chunk data to fit GPU memory which holds larger memory chunks in sorting
- GPU part, to sort the data chunks.

Initially, the larger memory in the CPU is divided into smaller chunks, each of size 2^{28}, which is the maximum number of elements that can be sorted using CUDPP in GPU. As mentioned in Listing 4.1, the chunks are passed on to the GPU for sorting and the results are transferred back to the CPU and stored in an array. The sorting is done by CUDPP, as explained in Chapter 3. We have also experimented with sorting the initial chunks using Thrust's Radix sort as explained in Section 2.3.1. Once all the chunks are sorted and stored in an array, then begins the next phase.

```
1  const int64_t numElement = 1LL << 31;
2  int numOfLessMemElement = numElements/(1<<28);
3  h_keysSorted = (unsigned int*)malloc((1<<28)*sizeof(int64_t));
4
5  for(int i=0;i<numOfLessMemElement;i++){
6      std::copy(h_keysWindowOld + (i * (1<<28)), h_keysWindowOld + ((i
           * (1<<28)) + (1<<28)), h_keysSorted);
7      std::copy(h_valuesWindowOld + (i * (1<<28)), h_valuesWindowOld +
           ((i * (1<<28)) + (1<<28)), h_values_window);
8      cudppMergeSortDispatch(h_keysSorted, h_values_window, (1<<28)
           ,5,0);
9      std::copy(h_keysSorted, h_keysSorted + (1<<28), h_keysWindow + (i
           *(1<<28)));
10     std::copy(h_values_window, h_values_window + (1<<28),
           h_values_sorted_window + (i*(1<<28)))
11 }
12 }
```

Listing 4.1: Initial Sort

The second phase happens in the CPU;

1. The initially sorted array is split into equal halves imaginarily.
2. From each half, a window of size 2^{27} is made, and the position of the window is calculated as mentioned in Listing 4.2. with the first window starting at the 0th position and the second window from half of the sorted array.

```
1  if (loopCount > 0){
2      if (numElementsFromWindowACovered < numElements/2){
3          startValueA = numElementsFromWindowACovered;
4          endValueA = startValueA + (1<<27) - 1;
5      }else if (numElementsFromWindowACovered == (numElements/2)){
6          startValueA = numElementsFromWindowACovered - (1<<27);
7          endValueA = numElementsFromWindowACovered - 1;
8      }
9      if (numElementsFromWindowBCovered < numElements/2){
10         startValueB = (numElements/2) + numElementsFromWindowBCovered
               ;
11         endValueB = startValueB + (1<<27) - 1;
12     }else if (numElementsFromWindowBCovered == numElements/2){
13         startValueB = (numElements/2) + numElementsFromWindowBCovered
               - (1<<27);
14         endValueB = (numElements/2)+ numElementsFromWindowBCovered -
               1;
15     }
16 }
```

Listing 4.2: Finding Position in Window

3. The values in the first and last positions of the windows are compared.
4. As mentioned in Listing 4.3, the split arrays are checked for merging conditions, until the end of both arrays is reached.

```
1 while ((pkeys[startValueA] < pkeys[startValueB] &&
2 pkeys[endValueA]<= pkeys[startValueB] &&
      numElementsFromWindowACovered < numElements/2)
3 ||(pkeys[startValueB]< pkeys[startValueA] &&
4 pkeys[endValueB]<= pkeys[startValueA] &&
      numElementsFromWindowBCovered < numElements/2)
5 ||(pkeys[startValueA]==pkeys[startValueB] && pkeys[endValueA]==pkeys[
      endValueB] && (int)pkeys[startValueA]==2147483647
6 && (int)pkeys[endValueA]==2147483647)){
7     ....
8
9 }
```

Listing 4.3: Windowing Condition

5. If the first and last values of the first window are less than or equal to the first value of the second window, the values within the window are copied to the resultant array, and the window is shifted to the next 2^{27} values(Listing 4.4).

```
1         std::copy(pkeys + startValueA, pkeys + endValueA,
             h_keysSortedWindow + positionInArray);
2         std::copy(pvals + startValueA, pvals + endValueA,
             h_valsSortedWindow + positionInArray);
3         startValueA = startValueA + (1<<27);
4         endValueA = startValueA + (1<<27);
5         positionInArray += (1<<27);
6         numElementsFromWindowACovered += (1<<27);
```

Listing 4.4: Copying Values from first array

6. Similarly, If the first and last values of the second window are less than or equal to the first value of the first window, the values within the window are copied to the resultant array, and the window is shifted to the next 2^{27} values (Listing 4.5).

```
1         std::copy(pkeys + startValueB, pkeys + endValueB,
             h_keysSortedWindow + positionInArray);
2         std::copy(pvals + startValueB, pvals + endValueB,
             h_valsSortedWindow + positionInArray);
3         startValueB = startValueB + (1<<27);
4         endValueB = startValueB + (1<<27);
5         positionInArray += (1<<27);
6         numElementsFromWindowBCovered += (1<<27);
```

Listing 4.5: Copying Values from second array

7. Also if windows of both arrays have only default values, the values are copied to the sorted array and the loop continues.(Listing 4.6)

   For example in Figure 4.1, in case 1, the first and last value of Window W1 is less than or equal to the first value of Window W2 and in case 2, the first

```
1  else if (pkeys[startValueA]==pkeys[startValueB] && pkeys[endValueA]==
       pkeys[endValueB] && (int)pkeys[startValueA]==2147483647
2  && (int)pkeys[endValueA]==2147483647){
3          std::copy(pkeys + startValueA, pkeys + endValueA,
               h_keysSortedWindow + positionInArray);
4          std::copy(pvals + startValueA, pvals + endValueA,
               h_valsSortedWindow + positionInArray);
5          positionInArray += (1<<27);
6          std::copy(pkeys + startValueB, pkeys + endValueB,
               h_keysSortedWindow + positionInArray);
7          std::copy(pvals + startValueB, pvals + endValueB,
               h_valsSortedWindow + positionInArray);
8
9          positionInArray += (1<<27);
10
11         startValueA = startValueA + (1<<27);
12         endValueA = startValueA + (1<<27);
13         numElementsFromWindowACovered += (1<<27);
14         startValueB = startValueB + (1<<27);
15         endValueB = startValueB + (1<<27);
16         numElementsFromWindowBCovered += (1<<27);
17     }
```

Listing 4.6: Copying Default Values

value of Window W1 is less than or equal to first and last value of Window W2. Therefore the resultant array becomes W1 + W2.

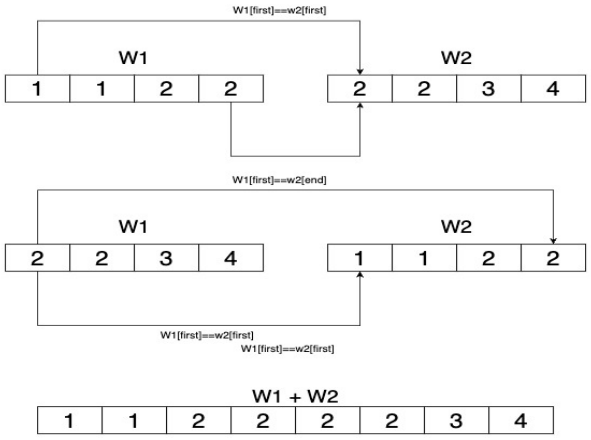


Figure 4.1: Window Value Comparison

8. If the comparison conditions are not met, then the two windows are merged to form a single array of size 2^{28}(as mentioned in Listing 4.7) and transferred to GPU for sorting.

9. The sorted array is again transferred back along with the position of sorted elements and stored in the resultant array. The sorted elements are stored in the resultant array by their position obtained from the CUDPP sort. The position from CUDPP will range from 0 to 2^{28}, therefore to obtain the correct

```
if (numElementsFromWindowACovered < numElements/2){
    std::copy(pkeys + startValueA, pkeys + endValueA, h_keysWindow);
    std::copy(pvals + startValueA, pvals + endValueA, h_valuesWindow)
        ;
}
if (numElementsFromWindowBCovered < numElements/2){
    std::copy(pkeys + startValueB, pkeys + endValueB, h_keysWindow +
        (1<<27));
    std::copy(pvals + startValueB, pvals + endValueB, h_valuesWindow
        + (1<<27));
}
```

Listing 4.7: Sort Loading

position each time the position is multiplied by the iteration count(as mentioned in Listing 4.8). In this way, all sorted elements are positioned correctly in the resultant array.

```
for (size_t i=0;i<numElementsWindowOutput;i++){
    int index = h_keysIndexSorted[i];
    if (index!= -1 && index != -2){
        if (((numPartitions + 1)/2)<= swapPoint){
            h_keysSortedWindow[positionInArray + i] = h_keysWindow[
                index];
            h_valsSortedWindow[positionInArray + i] = h_valuesWindow[
                index];
        }
    }
}
```

Listing 4.8: Placing based on position

10. After each sort and copy operation, both the windows are moved along in their respective partition. This process continues iteratively until both the partitions have reached their end.

11. After the sorting, the algorithm validates whether the entire values are sorted(as mentioned in Listing 4.9). If not, it initiates another iteration of the second phase with the resultant array as input to further sort until the entire values are sorted.

```
bool breakout = true;
for (size_t i=1;i<numElements;i++){
    if((pkeys[i] < pkeys[i-1] && pkeys[i] !=0 && pkeys[i-1] !=0)){
        breakout = false;
        break;
    }
}
```

Listing 4.9: Validating the sorting

In addition to larger-than-memory sorting, another update to sorting is to improve ordered aggregation. To this end, we include an in-situ aggregation to support aggregation while sorting. We elaborate on our implementation in the next section.

4.2 In Situ Aggregation

ThesimpleMergeHigher function is extended to include in situ aggregation during the sorting process. This extension includes two types of aggregation: Exclusive Aggregation and By-Product Aggregation. These extensions provide efficient management of duplicate keys while maintaining the sorted order of the dataset.

4.2.1 Exclusive Aggregation

Listing 4.10 explains how exclusive aggregation is done. Exclusive Aggregation processes keys and their corresponding values by aggregating duplicate keys and the duplicate keys are replaced with a default value (INT_MAX).

While the dataset is sorted, the algorithm iterates over the dataset and examines the consecutive position at 'n' and 'n+1'. If these keys are the same, their corresponding value is aggregated and stored at the index of 'n'. The keys array from the 'n+2' index is shifted one position to the left, and the last position is modified into INT_MAX ensuring the end of the dataset. The values array is also shifted similarly and the last value is made 0 as shown in the example in Figure 4.2.

Unsorted

Key	1	1	2	2	1	1	2	2
Key	1	1	1	1	1	1	1	1

Sorted - Aggregated

Key	1	2	2147483647	2147483647	2147483647	2147483647	2147483647	2147483647
Key	4	4	0	0	0	0	0	0

Figure 4.2: Exclusive Aggregation

This process ensures that after sorting, each key is present only once in the dataset with its corresponding values implying the aggregated value of all duplicate entries.

```
1  //Updating aggregate for Zero location
2  if ((myKey[0] <= localMaxB || myKey[depth - 1] >= localMinB)) {
3      if (myKey[0] >= localMinB) {
4          if (myValue[0] && myKey[0] <= localMaxB) {
5              size_t targetIndex = (globalCAddress + zeroLoc) - (
                   prefixSum[tid] + equality_offset[0]) ;
6              atomicAdd(&A_values_out[targetIndex], myValue[0]);
7              myValue[0] = 0;
8              A_keys_out[targetIndex] = myKey[0];
9              A_index_out[targetIndex] = (myKey[0]==MAX_VAL ? -2 : (
                   myStartIdxA + aIndex + depth * tid + 0));
10         }
11     }
12
13 //Updating aggregate for first location
14 if ((myKey[1] >= localMinB)) {
15     if (myValue[1] && myKey[1] <= localMaxB) {
16         size_t targetIndex = ( (globalCAddress + firstLoc + 1) - (
               prefixSum[tid] + equality_offset[0]) );
17         atomicAdd(&A_values_out[targetIndex], myValue[1]);
18         myValue[1] = 0;
19         A_keys_out[targetIndex] = myKey[1];
20         A_index_out[targetIndex] = (myKey[1]==MAX_VAL ? -2 : (
               myStartIdxA + aIndex + depth * tid + 1));
21         }
22     }
```

Listing 4.10: Exclusive Aggregation

4.2.2 By-Product Aggregation

In contrast to Exclusive aggregation, the By-Product aggregation preserves all the keys in their sorted order without removing duplicates as indicated in Listing 4.11 and Listing 4.12. Each Unique key's first occurrence stores the aggregated values of the duplicate keys.

During the sorting process, as the dataset is being sorted, it simultaneously performs the By-product aggregation. The algorithm examines keys at position at 'n' and 'n+1'. If these keys are equal, their corresponding values are aggregated and stored at the index of 'n' in the values array and the value at 'n+1' is made 0. The rest of the keys and values are kept in the order they were originally sorted without having any duplicates removed as shown in the example in Figure 4.3.

Unsorted

Key	1	1	2	2	1	1	2	2
Key	1	1	1	1	1	1	1	1

Sorted - Aggregated

Key	1	1	1	1	2	2	2	2
Key	4	0	0	0	4	0	0	0

Figure 4.3: By Product Aggregation

```
1  //Updating aggregate for Zero location
2  {
3      binSearch_whole_merge_higher(BKeys, index, aggIndex, myKey[0]);
4      T cmpValue = BKeys[index];
5      if (cmpValue <= myKey[0] && index < INTERSECT_B_BLOCK_SIZE_simple
           )
6          cmpValue = BKeys[++index];
7      index = (cmpValue <= myKey[0] ? index + 1 : index);
8      //Search for aggregate position here
9      T searchKey = BKeys[aggIndex];
10     while (searchKey >= myKey[0] && aggIndex > 0)
11         searchKey = BKeys[--aggIndex];
12
13     if (aggIndex == 0 && BKeys[aggIndex] == myKey[0])
14         aggIndex = 0;
15     else if (BKeys[aggIndex + 1] == myKey[0])
16         aggIndex++;
17
18     if (!(BKeys[aggIndex] == myKey[0])) {
19         aggIndex = index;
20     }
21
22     //Updating aggregate
23     if (myKey[0] >= localMinB) {
24         if (myKey[0] != MAX_VAL && myValue[0] // && myKey[0] <=
               localMaxB
25             && (globalCAddress + aggIndex) < size ) {
26             atomicAdd(&A_values_out[globalCAddress + aggIndex],
                   myValue[0]);
27             myValue[0] = 0;
28         }
29             A_keys_out[globalCAddress + index] = myKey[0];
30     }
31 }
```

Listing 4.11: By Product Aggregation Zero Location

```
1  //Updating aggregate for the first location
2  {
3      while (BKeys[index] <= myKey[1] && index <
           INTERSECT_B_BLOCK_SIZE_simple)
4          index++;
5
6      //updating aggIndex -- using binary search method
7      aggIndex = index==0?index:index - 1;
8      searchKey = BKeys[aggIndex];
9
10     while (searchKey >= myKey[1] && aggIndex > 0)
11         searchKey = BKeys[--aggIndex];
12
13     if (aggIndex == 0 && BKeys[aggIndex] == myKey[1])
14         aggIndex = 0;
15     else if (BKeys[aggIndex + 1] == myKey[1])
16         aggIndex++;
17
18     if (!(BKeys[aggIndex] == myKey[1])) {
19         aggIndex = index;
20     }
21
22     if (
23     (myKey[1] <= localMaxB && myKey[1] >= localMinB) || (bIndex +
           index) >= (partitionSizeB)
24     && (aIndex + tid * depth + 1 < sizePerPartition)
25     ) {
26         if( (globalCAddress + aggIndex + 1) < size) {
27             atomicAdd(&A_values_out[globalCAddress + aggIndex + 1],
                   myValue[1]);
28             myValue[1] = 0;
29         }
30
31         if ((myKey[1] < localMaxB && myKey[1] >= localMinB) || (
               bIndex + index) >= (partitionSizeB))
32             A_keys_out[globalCAddress + index + 1] = myKey[1];
33         }
34     }
```

Listing 4.12: By Product Aggregation First Location

In conclusion, the introduction of these techniques to the existing algorithm elevates the functionality making it effective for ordered aggregation in larger datasets.

5. Evaluation

The efficiency of Merge Sorting and Sort-merge aggregation algorithms can be evaluated by analyzing their runtime performance. These techniques are assessed using datasets with various distributions to understand the in-depth behavior under different conditions. By measuring the runtime across these scenarios, we can determine the effectiveness and performance of merge sorting and sort-merge aggregation in handling various data patterns. In section 5.1, we explain the evaluation setup required for evaluating the algorithm. We also go over the various dataset distributions that were assessed. We provide our analysis of sorting over different data groups in section 5.2. Then we present our analysis of hard hitter distribution in section 5.3. Further, we also discuss the performance of the algorithm on the unique dataset in section 5.4. We also provide a comprehensive discussion of the performance of in-situ aggregation in section 5.5. Finally, We provide the conclusion in section 5.6.

5.1 Evaluation Setup

The dataset used for evaluation measure varies in size, ranging from 2^{27} to 2^{31} values. The datasets are evaluated for ten iterations and the average runtime is used for analysis. We measure the execution time taken for the techniques used in our evaluation. In section 5.1.1, we explain the different data distributions used in our evaluation, followed by section 5.1.2, details of the execution environment are discussed.

5.1.1 Dataset Distribution

The sorting and aggregating techniques are tested with three different data distribution sets. The distributions used are,

Dense unique random distribution

The data are unique and densely populated in this type of distribution. This random distribution of unique values leads to scattering of values. The values in this

distribution are generated using the theortical basis presented in [WHO02]. For generating a value in the distribution, we use a generation defined by formula

$$a[i] = i\%\text{upperbound} + 1$$

, where i ranges from 0 to *datasize* and upperbound sets the maximum possible value that the generator can produce. This ensures that the values are unique within the range.

Heavy Hitter Distribution

In a Heavy Hitter distribution as presented in [VPCE20], the dataset is initially populated with unique values that are generated using a modulo operation, and beyond a certain threshold, all subsequent values are replaced with a single repeated value. In our analysis, we used 1 as the repeating value.

The values are generated using the formula:

$$a[i] = \begin{cases} i\%128 + 1 & \text{if } i < 0.9 \times \text{data size} \\ 1 & \text{otherwise} \end{cases}$$

Uniform random values

In a uniform random distribution, each value has an equal probability of being selected. The generation of each value is independent of the previously generated values. The values are produced using the formula:

$$\text{value}(i) = \text{rand}()$$

where rand() generates a random value within a specified range. Each value in this series is randomly chosen, ensuring that every possible value has the same probability of occurrence.

5.1.2 Environment Setup

We conducted our performance evaluation on a system equipped with an NVIDIA RTX 2080 Ti GPU and running Ubuntu 20.04.6. The RTX 2080 Ti, known for its high-performance CUDA cores and 11GB of GDDR6 memory, provides substantial computational power for our tests. Our machine includes a 512GB NVMe Solid State Drive (*SSD*) with 3200MB/s read speed and 1300MB/s write speeds, ensuring fast data access and storage, as well as 32GB of DDR4 RAM at 2666MT/s.

5.2 Data Grouping

The dense unique random distribution values with a constant number of groups show sorting time for both Thrust + CUDPP and CUDPP alone showing some variation as data grouping size increases as shown in Figure 5.1. In general, Thrust + CUDPP shows a more stable sorting time, whereas CUDPP alone shows a slightly higher variance, especially with very large data sizes. It also shows that for smaller

data groups, the sorting time for both Thrust + CUDPP and CUDPP alone are relatively close. As the data grouping increases to 10,000 and beyond, Thrust + CUDPP consistently outperforms CUDPP, showing lower sorting times. Notably, as the data size scales up to 100,000,000, Thrust + CUDPP maintains a more consistent sorting time, with only a modest increase, while CUDPP alone shows a more significant increase in sorting time, particularly when the data grouping reaches 10,000,000.

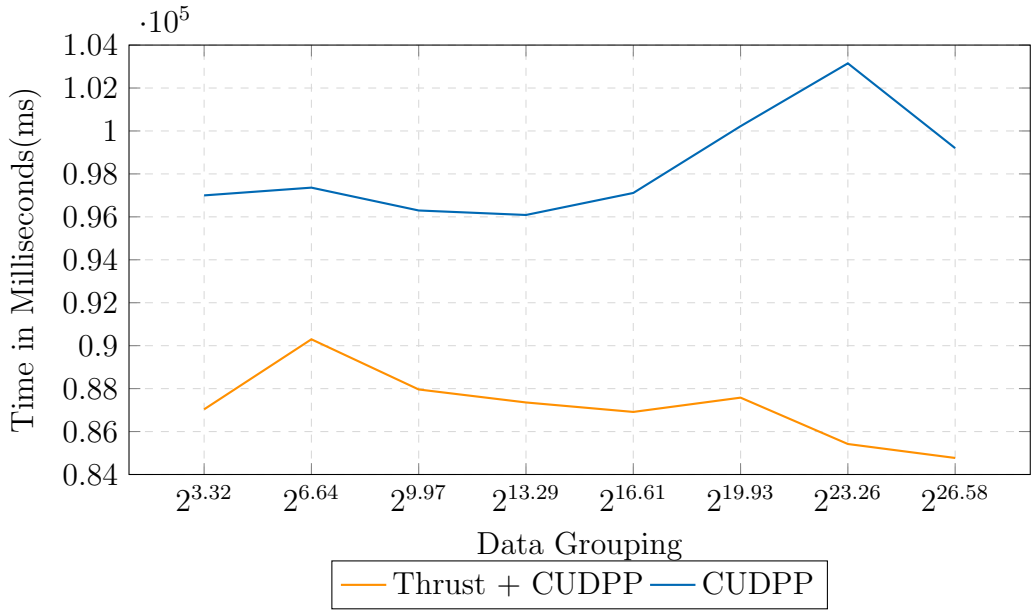


Figure 5.1: Data Grouping

5.3 Hard Hitter

Figure 5.2 details the sorting of Hardhitter values. For smaller data sizes, the sorting time for Thrust is significantly lower than CUDPP. This data shows that Thrust maintains a stable sorting time, while CUDPP's time has risen sharply. As the data size increases over larger than memory, both methods exhibit a significant increase in sorting times. However, it is also to be noted that Thrust + CUDPP consistently outperforms CUDPP. The difference in performance expands as the data size increases further.

In both methods, the first seven chunks of 2^{28} values are sorted in a reasonably consistent manner. The sorting time increases noticeably in the eighth chunk as shown in Figure 5.3. This spike indicates that the concentration of "hard-hitter" values—more difficult or complex for the sorting algorithm to process—is probably higher in this iteration. Once all the chunks are sorted, the values are sent to the windowing algorithm. Here the sorting time shows a dramatic increase, as it divides and sorts.

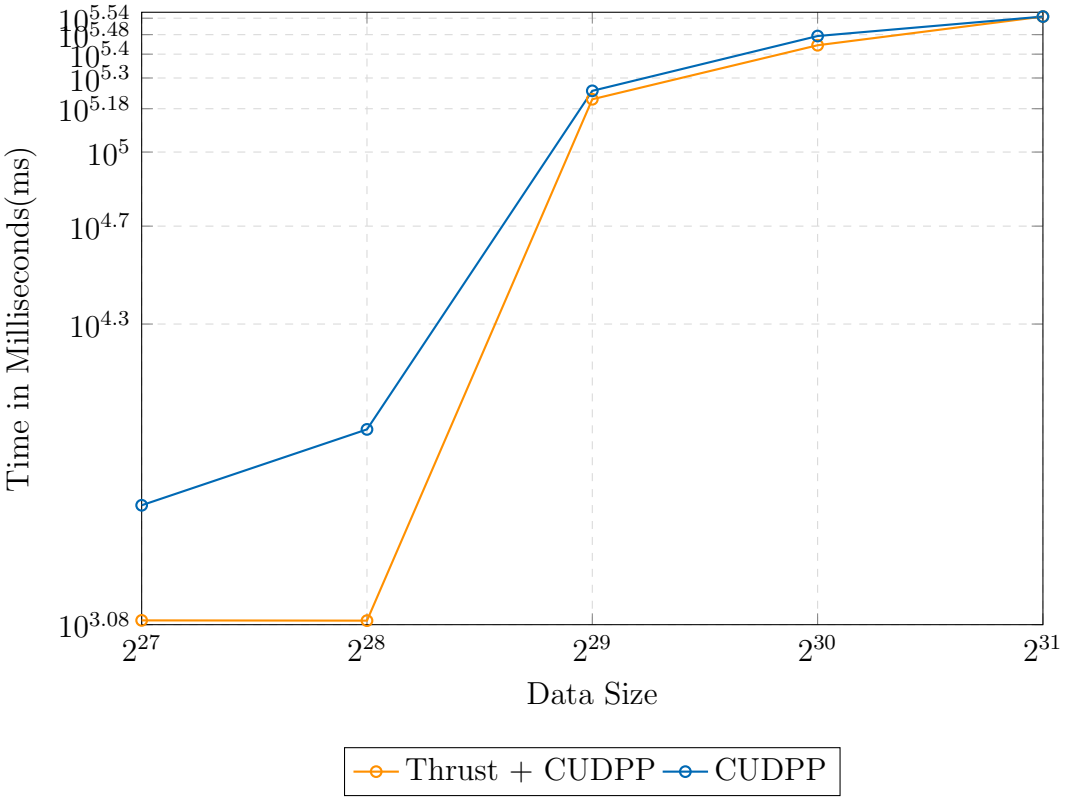


Figure 5.2: Hard Hitter

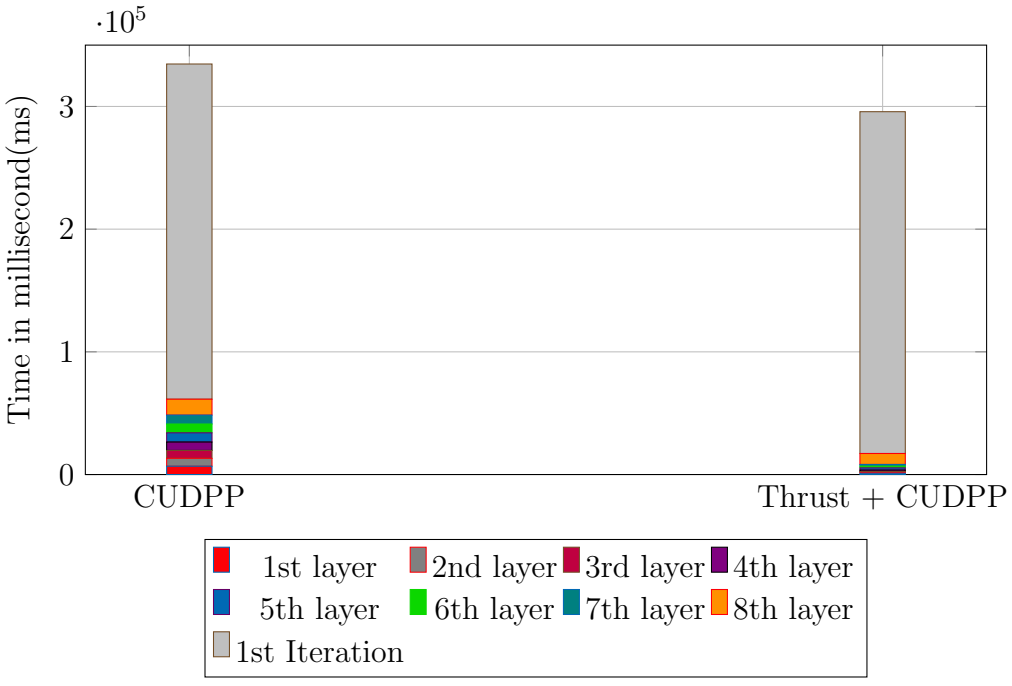


Figure 5.3: Hardhitter Sorting Stages

5.4 Unique Values

The performance of sorting unique values, as shown in Figure 5.4, has a pattern resembling that of hardhitter values. Thrust has significantly faster sorting times for smaller data volumes than CUDPP, keeping performance consistent while CUDPP's sorting time rises noticeably. Sorting times for Thrust + CUDPP and CUDPP both significantly increase as the data size exceeds the available memory. Nonetheless, Thrust + CUDPP continuously beats CUDPP alone; the difference in performance grows as the amount of data increases.

Interestingly, there is a minor drop in the average performance time for unique values, even though the overall performance trends for these values are comparable to those for hard-hitter values. This implies that compared to hard-hitter values, unique values might be a little simpler or easier to sort, leading to a somewhat quicker sorting process. Even with this decline, the overall trend is still the same: Thrust + CUDPP offers a more scalable and effective solution than CUDPP alone, especially when data quantities increase.

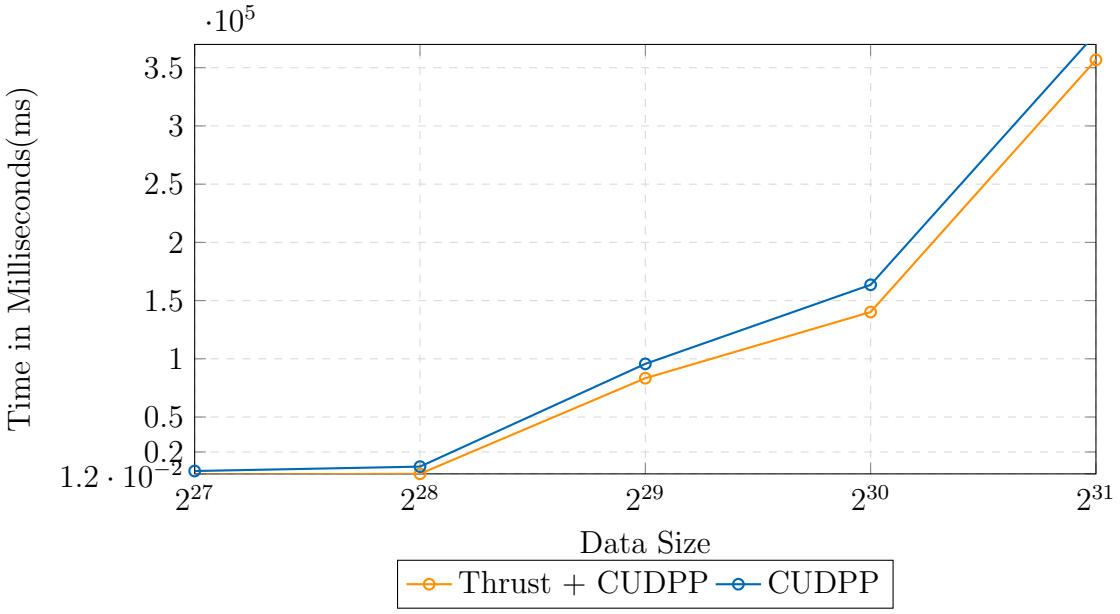


Figure 5.4: Unique Values

The distribution of processing times over several repeats for unique values exhibits an outline resembling that of hard-hitter values when the split of stages is analyzed but with generally significantly shorter times as indicated in Figure 5.5.

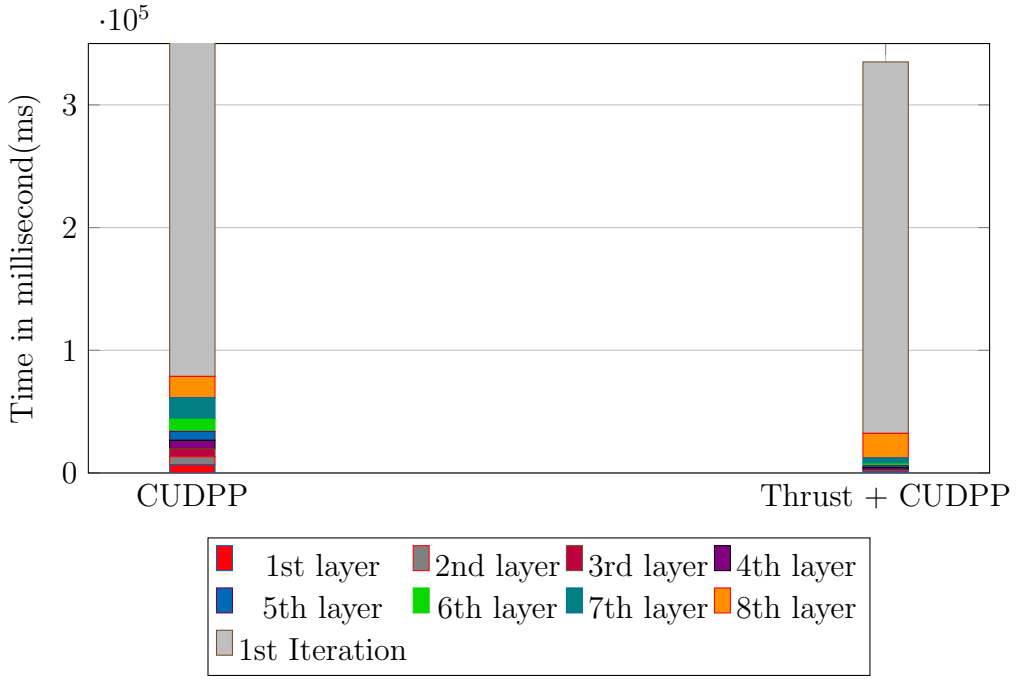


Figure 5.5: Unique Values Sorting stages

5.5 Aggregation

The in-situ aggregation results for unique values, data grouping, and hard-hitter values all show a similar pattern to the initial sorting operations, but with an expected increase in processing time due to the added complexity of aggregation.

Even as processing durations increase with bigger data quantities, Thrust + CUDPP consistently beats CUDPP for unique values (as shown in Figure 5.6), highlighting the overhead caused by aggregation operations.

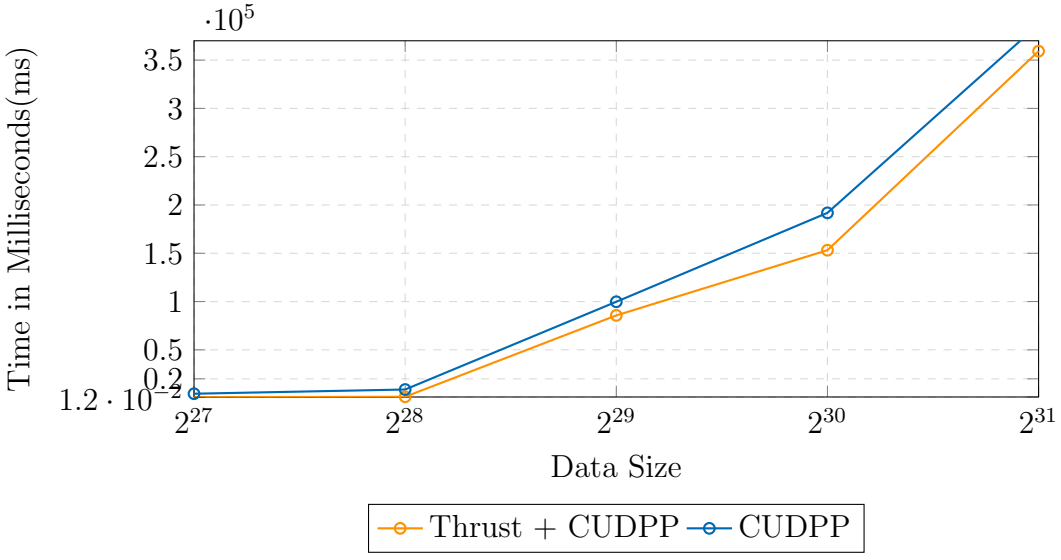


Figure 5.6: Unique Values

This pattern is reflected in data grouping (as shown in Figure 5.7), where despite the time increase from aggregation, Thrust + CUDPP stays faster across different group sizes.

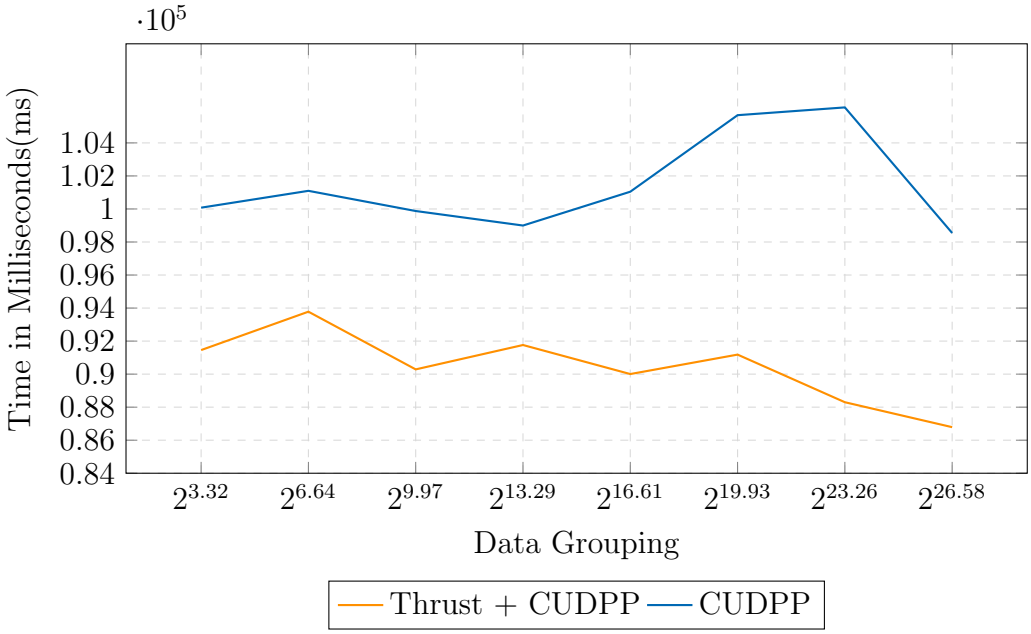


Figure 5.7: Data Grouping

The effect is more noticeable in hard-hitter values (as shown in Figure 5.8)because of the data's intrinsic complexity, which causes even larger time increases. Even so, Thrust + CUDPP performs better than CUDPP alone, managing the bigger and trickier datasets with greater efficiency. Thrust + CUDPP constantly performs better than aggregation, despite the fact that aggregation lengthens processing times for all types of data. This illustrates the robustness and efficiency of Thrust + CUDPP in intricate data processes.

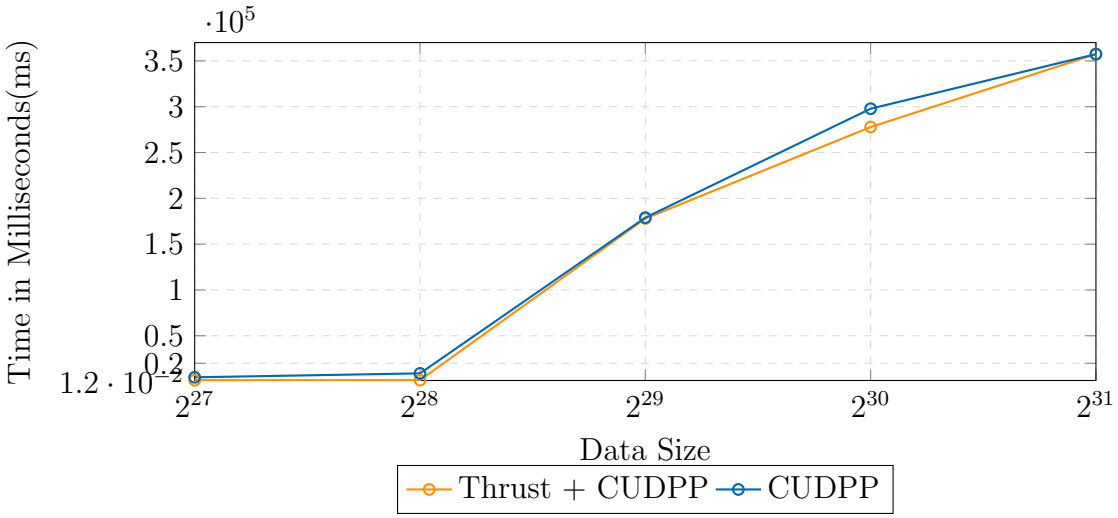


Figure 5.8: Hard Hitter Values

5.6 Evaluation Results

In this section, we provide our overall inference on the results of our experiments. The findings demonstrate that there is a processing time overhead associated with the in-situ aggregating method, especially as data sizes grow. In spite of this, Thrust + CUDPP works better than CUDPP alone in all kinds of data, including hard-hitter values, unique values, and data groupings. Due to the added aggregation complexity, Thrust + CUDPP's performance advantage is visible even as processing times increase, demonstrating how well it handles large-scale datasets. The aggregation step lengthens the processing time for unique values, yet Thrust + CUDPP is still able to perform better. In the same way, Thrust + CUDPP continues to outperform CUDPP in data grouping even when processing times increase with increasing group sizes. Thrust + CUDPP continues to perform better, but the effect is particularly noticeable with hard-hitter values where the data's complexity significantly increases processing time. Overall, the Thrust + CUDPP technique offers a reliable and effective solution across a range of data kinds and sizes, demonstrating its appropriateness for complicated data operations even when aggregation increases the processing burden.

6. Related Work

In this chapter, we compare our approach with other studies that address similar sorting operations in databases, particularly those optimized for parallel processing on GPUs. Numerous studies have introduced innovations in multi-GPU systems, optimized memory management, and comparison-based sorting techniques to enhance the performance and scalability of sorting on GPUs.

One of the early advancements in GPU-based sorting was presented by Green et al.[GMB12] who developed the GPU merge path algorithm, which partitions sorted arrays into non-overlapping sub-arrays, independently merged by GPU cores. Their approach demonstrated 20x and 50x speedups for integer and floating-point merges, respectively, over sequential CPU-based methods, and achieved a 10x speedup compared to the CUDA Thrust library's parallel merge sort.

Extending this, Tanasić et al.[TVJ+13] proposed a multi-GPU merge sort technique, which addresses the issue of dispersed data sorting across multiple GPUs. They used a novel pivot selection mechanism for inter-GPU communication, yielding a 1.9x speedup on two GPUs and 3.3x on four GPUs, demonstrating the scalability and efficiency of their method.

In comparison-based sorting,Leischner et al.[LOS10] proposed the GPU sample sort outperforms the GPU Thrust merge sort by 25% to 68%, as well as the GPU quicksort and radix sort. Their findings demonstrate that multi-way sorting algorithms, such as sample sort, can dramatically outperform classic two-way merge sort on GPUs.

Further refinements were made by Karsin et al.[KWC+18] who proposed GPU-MMS (Multiway Merge Sort), which reduces memory access bottlenecks while balancing hardware-specific limiting constraints. Their technique surpasses cutting-edge GPU library implementations by an average of 21%, making it an essential enhancement in comparison-based GPU sorting.

Ye et al.[YFL+10] presented a high-performance comparison-based sorting algorithm optimized for many-core GPUs. By assigning independent pairs of sequences to

GPU warps and utilizing coalesced global memory accesses, they effectively reduced synchronization bottlenecks and optimized bandwidth usage. The GPU-Warpsort algorithm demonstrated up to a 30% performance improvement over previous optimized GPU sorting algorithms, particularly for large input sequences, highlighting the importance of hardware-aware algorithm design for efficient sorting on GPUs

In order to demonstrate high data throughput and effective resource utilisation, Zhou et al.[ZZF20] created a pipelined, comparison-based iterative sort algorithm especially for FPGAs. According to Zhou et al., this method works especially well for real-time applications that need dependable and quick sorting on FPGAs. Marcelino et al.[MNC08] further contributed by designing FPGA-specific hardware sorting units optimized for embedded systems, achieving a performance improvement of up to 25 times over traditional software-based sorting methods.Their method demonstrates the performance improvements possible with FPGA-embedded sorting units that are specifically designed to satisfy the demands of limited surroundings (Marcelino et al.)

7. Conclusion and Future Work

In this thesis, we explored the innovative approaches to GPU sorting, with a specific focus on in-situ aggregation and the challenges of extending sorting capacity beyond the confines of available memory. The primary objective was to enhance sorting efficiency and scalability, thereby enabling the processing of larger datasets that are increasingly common in contemporary applications.

Our research revealed that GPU-based sorting algorithms can significantly outperform traditional CPU methods, particularly when leveraging in-situ aggregation techniques. This approach not only optimizes memory usage but also reduces data transfer overhead, leading to substantial improvements in processing speed. Furthermore, we successfully demonstrated methods for sorting datasets larger than memory by implementing partitioning strategies that facilitate efficient data handling and retrieval.

To perform larger than data size sorting, we used radix sort and merge sort. We detailed the mechanism of radix sorting and merge sorting and also the runtime efficiency of them. To perform efficient in-situ aggregation, we experimented it with exclusive aggregation and by-product aggregation. We explained the merging mechanism and its memory usage. We also demonstrated the algorithm's efficiency when implemented with radix sort and merge sort.

The results in this thesis are as follows:

- We argued that the sorting can be done for larger than memory size using windowing mechanism. We provided in the work a novel way to expand the capabilities of CUDPP's sorting functionality.
- We provide in this work, the aggregation of values while the keys are being sorted. We explained that based on need of preservation of duplicate keys, by-product or exclusive aggregation can be performed.
- We also provide ways to combine both the proposed mechanism to sort and aggregate the values for larger than data sizes.

We conducted experiments using three different data distributions: dense, unique random, and sequential. Each distribution exhibited distinct impacts on processing times. The results indicated that, while the aggregation step increased processing time for unique values, Thrust + CUDPP consistently outperformed CUDPP across all three distributions. We identified a processing time overhead associated with the in-situ aggregating method, particularly as data sizes increase. Despite this overhead, the Thrust + CUDPP technique consistently outperforms CUDPP alone across various data types, including hard-hitter values, unique values, and data groupings.

Future Work

This work opens several directions for future research, particularly in extending the implementation of in situ aggregation within sorting algorithms. One such interesting approach is to incorporate in situ aggregation into radix sort, expanding on methods such as those described by Stehle et all.[SJ17].

The sorting of larger-than-memory data sizes and in situ aggregation can be performed in other hardware. The approach can be modified to be executed in FPGAs and Intel's Many Integrated Core (MIC) and optimized for efficiency by exploiting hardware-related instructions. Future implementations could significantly increase speed and efficiency by utilizing hardware-specific instructions as discussed by Peter et all. [PSHL09]

The in situ aggregation can also be considered for cross-platform implementation on FPGAs and MICs. Research by Govindaraju et al. [GMS+13] demonstrated cache-efficient sorting on GPUs which might be expanded for equivalent aggregation performance improvements on FPGA or MIC systems. This paves the path for creating efficient and scalable solution for complex data processing.

Bibliography

[Cor24] Cornell University. Single Instruction, Multiple Threads (SIMT) Warp, 2024.

[GMB12] Oded Green, Robert McColl, and David A. Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, page 331–340, New York, NY, USA, 2012. Association for Computing Machinery.

[GMS+13] M.B. Giles, G.R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing op2 for gpu architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451–1460, 2013. Novel architectures for high-performance computing.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John Owens. *Parallel prefix sum (scan) with CUDA*, volume 39, pages 851–. 08 2007.

[KWC+18] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. Analysis-driven engineering of comparison-based sorting algorithms on gpus. In *Proceedings of the International Conference on Supercomputing*, pages 86–95. Association for Computing Machinery, 6 2018.

[LOS10] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Gpu sample sort. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[MNC08] Rui Marcelino, Horácio Neto, and João M.P. Cardoso. Sorting units for fpga-based embedded systems, 2008.

[NVI18] NVIDIA Corporation. NVIDIA Turing GPU Architecture: Graphics Reinvented. White paper, 2018.

[NVI20] NVIDIA Corporation. NVIDIA Ampere GA102 GPU Architecture. White paper, 2020.

[NVI21] NVIDIA Corporation. NVIDIA Jetson TX2 NX System-on-Module. White paper, 2021.

[PSHL09] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, page 403–410, Berlin, Heidelberg, 2009. Springer-Verlag.

[PSHL10] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Parallel external sorting for cuda-enabled gpus with load balancing and low transfer overhead. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1–8, 2010.

[RD17] A. N. Romanyuk and O. O. Dudnyk. Ways to improve performance of anisotropic texture filtering. In *2017 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–6, 2017.

[SGNB17] Sruthikesh Surineni, Ruidong Gu, Huyen Nguyen, and Michela Becchi. Understanding the performance-accuracy tradeoffs of floating-point arithmetic on gpus. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 207–218, 2017.

[SJ17] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, page 417–432, New York, NY, USA, 2017. Association for Computing Machinery.

[SSD+16] Hideyuki Shamoto, Koichi Shirahata, Aleksandr Drozd, Hitoshi Sato, and Satoshi Matsuoka. Gpu-accelerated large-scale distributed sorting coping with device memory capacity. *IEEE Transactions on Big Data*, 2:57–69, 1 2016.

[TVJ+13] Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, and Wen-mei Hwu. Comparison based sorting for systems with multiple gpus. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, page 1–11, New York, NY, USA, 2013. Association for Computing Machinery.

[VPCE20] Federica Ventruto, Marco Pulimeno, Massimo Cafaro, and Italo Epicoco. On frequency estimation and detection of heavy hitters in data streams. *Future Internet*, 12(9), 2020.

[WHO02] Pei-Chi Wu, Kuo-Chan Huang, and Shih-Ting Ouyang. Bit-parallel random number generation for discrete uniform distributions. *Computer Physics Communications*, 144(3):252–260, 2002.

[WIK24] WIKIPEDIA contributors. Single instruction, multiple threads — Wikipedia, The Free Encyclopedia, 2024.

[WS10] Jan Wassenberg and Peter Sanders. Faster radix sort via virtual memory and write-combining. 8 2010.

[YFL+10] Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and Paolo Ienne. High performance comparison-based sorting algorithm on many-core gpus. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[ZZF20] Jingyang Zhou, Xiongkui Zhang, and Jiameng Fan. Pipelined implementation of serial comparison based iterative sort on fpga. In *ACM*

International Conference Proceeding Series, pages 26–30. Association for Computing Machinery, 10 2020.

I hereby declare that I am writing the accompanying thesis independently and did not use sources and tools other than those indicated.

Magdeburg, den 03. December 2024