HETEROGENEOUS COMPUTING FOR ACCELERATING STRUCTURE QUERY PROCESSING

YONG KEH KOK

DOCTOR OF PHILOSOPHY IN ENGINEERING

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY
UNIVERSITI TUNKU ABDUL RAHMAN
DECEMBER 2025

HETEROGENEOUS COMPUTING FOR ACCELERATING STRUCTURE QUERY PROCESSING

By

YONG KEH KOK

A thesis submitted to the Department of Electronic Engineering,
Faculty of Engineering and Green Technology,
Universiti Tunku Abdul Rahman,
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Engineering
December 2025

ABSTRACT

Analytical structured query language (SQL) workloads increasingly run on commodity servers, yet it remains unclear when a single graphics processing unit (GPU) can replace or augment small clusters. Existing GPU based database systems often optimise isolated kernels, assume uncompressed data, or offload only selected operators, so they give limited insight into end-to-end behaviour and the cost of moving data between CPU and GPU.

This thesis establishes when a single GPU can accelerate SQL analytics. It develops a GPU structured query accelerator with a GPU-accelerated query pipeline that compresses, stores, schedules and processes data on the device as a first-class execution target. The design combines a compression-first pre-processing path based on a GPU PFOR-Delta variant, chunked columnar storage sized to device memory, an SQL front-end with a placement scheduler, and GPU centric hash and sort-merge joins that run directly on compressed chunks. It also integrates GPU implementations of exact string matching, histogram based string predicates and short-pattern approximate matching so that structured and text conditions execute within one framework.

The thesis implements this accelerator and evaluates it on synthetic joins of up to 512 million tuples, on TPC-H queries of up to 50 GB, and on healthcare and geospatial heat map workloads. The experiments use PostgreSQL on a multi-core CPU and an eight-node Hadoop/Spark cluster as baselines. PFOR-Delta reduces analytical data to 42–83% of its original size while it remains fast enough for pre-processing, and GPU joins reduce join times relative to the CPU at all tested scales. On TPC-H Query 2 at 50 GB, the single-GPU workstation

achieves end-to-end speedups of 21–25x over PostgreSQL after PCI Express (PCIe) transfer costs, and on the healthcare and geospatial workloads it matches or exceeds the eight-node cluster, including 4–18x speedups for heat map generation. The evaluation maps data sizes and workload shapes that benefit most from GPU acceleration and identifies cases where PCIe transfers and memory bandwidth limits erode these gains. The evaluation maps data sizes and workload shapes that benefit most from GPU acceleration and identifies cases where PCIe transfers and memory bandwidth limits erode these gains. Together, the architecture and results establish architecture aware criteria for treating a single GPU as the primary analytical engine and provide a blueprint for GPU accelerated SQL systems that balance single GPU accelerators against multi core CPUs and small clusters.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the kind supports offered by many mentors. I had owned a lot of thank you to them. I would like to acknowledge those provided selfless help throughout the journey.

I would like to express my gratitude to my supervisor team, Dr. Peh Chiong Teh. Their individuals have facilitated an environment conducive to my productivity, alleviating the typical stresses associated with graduate school. Their valuable encouragement, unwavering support, and constructive feedback have been instrumental to my success.

I wish to extend my gratitude to Dr. Vooi Voon Yap, Dr. Ettikan Kandasmy, Dr. Poh Kit Chong, Dr. Wai Kong Lee, Dr. Alex Ooi, Dr. Keeratpal Singh, Dr. Yew Chong Chew, and Dr. Liang Kim Meng for their unwavering friendship during my years in progressing my PhD in UTAR. Additionally, I would like to express my appreciation to my colleagues companions Alwyn Goh, Ng Kang Seong, Dr. Khoong Neng Choong and Dr. Hock Woon Hon, for their continuous support and encouragement in all my pursuits.

Finally, I would like to take this opportunity to express my heartfelt gartitude to my family members. My mother, Yoke Choo Chen has been giving unwavering support during my PhD journey. My wife, Michelle Hooi Ching Beh gives the love, care, and understanding. She were the emotional pillars supports for keeping me motivated and focused throughout the process. I proud to have Matthew Xue Zhang Yong as my son, and I will always cherish his contribution to my success.

APPROVAL SHEET

This dissertation/thesis entitled "**HETEROGENEOUS COMPUTING FOR ACCELERATING STRUCTURE QUERY PROCESSING**" was prepared by YONG KEH KOK and submitted as partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering at Universiti Tunku Abdul Rahman.

Approved by:

_____
(Ir. Dr. Teh Peh Chiong)
Date:……8/12/2025……..
Supervisor
Department of Electronic Engineering,
Faculty of Engineering and Green Technology,
Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 8/12/2025

SUBMISSION OF THESIS

It is hereby certified that **YONG KEH KOK** (ID No: **14AGD06590**) has completed this thesis entitled "**HETEROGENEOUS COMPUTING FOR ACCELERATING STRUCTURE QUERY PROCESSING**" under the supervision of Dr. Teh Peh Chiong (Supervisor) from the Department of Electronic Engineering, Faculty of Engineering and Green Technology.

I understand that University will upload softcopy of my thesis in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

_____

(YONG KEH KOK)

DECLARATION

I, YONG KEH KOK hereby declare that the thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

(YONG KEH KOK)

Date: 8/12/2025

TABLE OF CONTENTS

PAGE

ABSTRACT **ii**
ACKNOWLEDGEMENTS **iv**
LIST OF TABLES **1**
LIST OF FIGURES **2**
LIST OF PSEUDO ALGORITHMS **4**
LIST OF ABBREVIATIONS **5**

1 INTRODUCTION **6**
1.1 Background 7
1.1.1 The Rise of Heterogeneous Computing in Structured Query Processing 7
1.1.2 Identifying and Addressing Bottlenecks in Multi-core CPU 10
1.1.3 Operator-Level Acceleration and Programming Models 12
1.2 Embarking On The Research 14
1.2.1 Problem Statement 14
1.2.2 Research Questions 16
1.2.3 Research Objectives 18
1.3 Thesis Contribution 19
1.4 Outline of Thesis 21

2 LITERATURE REVIEW **23**
2.1 GPU-accelerated Structured Query Processing 23
2.1.1 From early GPU primitives to full database engines 23
2.1.2 GPU-accelerated database architectures 24
2.1.3 Cost models and heterogeneous query planning 26
2.1.4 Limitations and implications for this thesis 26
2.2 GPU Compression for Column Based Data Store 27
2.2.1 Role of compression in columnar and GPU settings 28
2.2.2 Heavyweight schemes on GPUs and their limitations 28
2.2.3 Lightweight schemes, vectorisation, and GPU-friendly designs 29
2.2.4 Column-wise partitioning and data movement 31
2.2.5 Summary and implications 32
2.3 Accelerated Execution for Join Operation 32
2.3.1 CPU-based join optimisation 33
2.3.2 Vector processors and early GPU-based joins 33
2.3.3 Modern GPU join algorithms and comparative studies 34
2.3.4 Summary and research gap 35
2.4 Accelerated Big Data Processing with Geospatial Computation 36
2.4.1 Geospatial analytics as a GPU workload 36
2.4.2 Hybrid CPU–GPU architectures and geospatial engines 37
2.4.3 Challenges and opportunities 38
2.5 Accelerated String Matching In GPU Parallel Streaming 38

2.5.1 String matching fundamentals 39
2.5.2 Hash match on GPU 39
2.5.3 Histogram optimisation with CUDA 40
2.5.4 Bit-parallel approximate matching and GPU streaming 41
2.5.5 Implications of GPU Compression Techniques for Column-Based Analytical Data Stores 43
2.6 Summary of Literature and Research Gaps in GPU-Accelerated Structured Query Processing 44

3 GPU STRUCTURE QUERY ACCELERATOR 47
3.1 Architecture Design of GPU Structure Query Accelerator 48
3.1.1 Research Questions to Architecture Design 49
3.1.2 Implementation of GPU Query Accelerator 52
3.1.3 Evaluation of GPU Query Accelerator on TPC-H Workloads 59
3.1.4 Discussion of Performance Gains and GPU Underutilisation 71
3.2 Data Partitioning for GPU Compressor 72
3.2.1 Implementation Pre-processor 73
3.2.2 Comparative Study of GPU Compression Schemes 80
3.2.3 Interpretation of Compression Trade-offs and PFOR-Delta Performance 87
3.3 GPU-accelerated Execution for Join Operation 88
3.3.1 Implementation accelerated GPU Query Processing 89
3.3.2 Evaluation of GPU Hash and Sort-Merge Joins 94
3.3.3 Analysis of Join Scalability and GPU–CPU Comparison 99
3.4 Summary of GPU Query Accelerator Architecture and Performance 100

4 PATHWAYS OF GPU IN BIG DATA APPLICATIONS 103
4.1 GPU and Big Data Application 104
4.1.1 Implementation 105
4.1.2 Performance Assessment of GPU and Hadoop Platforms for Healthcare Queries 107
4.1.3 Cost and Performance Implications for Healthcare Analytics 111
4.2 GPU-accelerated Geospatial Processing 116
4.2.1 Implementation 117
4.2.2 Performance Study of GPU-Accelerated Geospatial Heat Map Generation 122
4.2.3 Discussion of GPU Geospatial Pipeline and Spark Comparison 127
4.3 Summary and Implications for Single-GPU Big Data and Geospatial Workloads 127

5 STRING MATCHING QUERY 130
5.1 Hash Match on GPU 132
5.1.1 Implementation 133

5.1.2 Experimental Results for Hash Match on Fermi and Kepler GPUs 135
5.1.3 Discussion of Hash Match Behaviour and Thread Configuration 139
5.2 Histogram Optimization with CUDA 140
5.2.1 Implementation 140
Multi-core CPU with OpenMP 142
5.2.2 Experimental Results for Histogram Optimisation on CPU and GPU 147
5.2.3 Analysis of Streaming Histogram Design and GPU Occupancy 155
5.3 Accelerated Bit Parallel Approximate Matching 156
5.3.1 Implementation Techniques in GPU 156
5.3.2 Experimental Evaluation of Asynchronous Wu–Manber for Short Patterns 158
5.3.3 Implications of Approximate Matching for GPU Query Pipelines 163
5.4 Summary of GPU-Accelerated String Matching for Query Predicates 164

6 CONCLUSION 165
6.1 Summary of Contributions 166
6.2 Recommendations For Future Work 168
6.2.1 Multi-GPUs Acceleration for Query Processing Techniques 168
6.2.2 Enhanced Parallelism With Leveraging GPU Advancement 169
6.2.3 Integration of GPU-Accelerated String Matching With MPI Systems 169

7 BIBLIOGRAPHY 170

LIST OF TABLES

Table 3.1: Research Questions and Architectural Implications.....51
Table 3.2: Specification of Xeon and Kepler 60
Table 3.3: Query Complexity Comparison.....66
Table 3.4: Averaging of GPU Occupancy in % 68
Table 3.5: Averaging of CPU Usage in %.....68
Table 3.6: Impact of PCIe communication on GPU for Query 2 (50 GB).....69
Table 3.7: Compression Ratio 82
Table 3.8: Compression & Decompression Time (Seconds).....84
Table 3.9: Specification of CPU and GPU 95
Table 4.1: Specification of Hadoop Vs GPU Workstation.....108
Table 4.2: Queries Description for Healthcare Dataset.....111
Table 5.1: Specification of GPU K20c & GTX980Ti 147
Table 5.2: GPU Kernel Implementations Global Load Efficiency.....152
Table 5.3: Summary Of Performance Gain For Async Against Sync 163

LIST OF FIGURES

Figure 3.1: Architecture of GPU Query Accelerator 53
Figure 3.2: Query 1 - Shipping and Return Trends Analysis 63
Figure 3.3: Query 2 - Customer Purchasing Behaviour by Market Segment ... 64
Figure 3.4: Analysis of 'BUILDING' Market Segment 65
Figure 3.5: Three Queries Execution 65
Figure 3.6: Resources Usage for CPU & GPU 68
Figure 3.7: Essential Procedure of GPU Data Partition 74
Figure 3.8: Step of Memory & Process Optimisation 79
Figure 3.9: Compression Ratio of GPU Compression Algorithms 83
Figure 3.10: GPU Compression and Decompression Time 85
Figure 3.11: Evaluation of Hash Join on Generated Relations 97
Figure 3.12: Evaluation of Hash Join on TPC-H 98
Figure 4.1: Comparison of Query Execution 112
Figure 4.2: GPU Columnar File System 117
Figure 4.3: SOA CUDASET Structure 119
Figure 4.4: SQL for Heat Map Computation 121
Figure 4.5: Result of Data Pre-processing 124
Figure 4.6: Heat Map query execution 126
Figure 4.7: Visualization of Heat Map 126
Figure 5.1: Hash Match Workflow 132
Figure 5.2: Comparison Performance of String Matching 137
Figure 5.3: Hash Value Kernel Matching on K20c 138
Figure 5.4: Kernel Execution Time 139
Figure 5.5: The Speedup Calculation 148
Figure 5.6: Average Speedup for Both Global and Shared Memory 149
Figure 5.7: GPU Occupancy from NVIDIA Nsight 150
Figure 5.8: Atomic Requests Transactions 151
Figure 5.9: Average Speedup Of All Kernels On Both GPU Cards 153
Figure 5.10: Average Speedup for Non-Streaming And Streaming 154
Figure 5.11: Time Profile of The Streaming Implementation (4 streams) 154
Figure 5.12: Average Speedup of K20C and GTX 980Ti 155
Figure 5.13: Sync - Pattern Matching (GPU) 157

Figure 5.14: Async - Overlapping Pattern Matching (4 Streams) 158
Figure 5.15: Timing performance for APM in K40c with k=2 and 4. 160
Figure 5.16: Timing performance for APM in K40c with k=6 and 8. 160
Figure 5.17: Timing performance for APM in GTX980 with k=2 and 4. 161
Figure 5.18: Timing performance for APM in GTX980 with k=6 and 8. 161
Figure 5.19: Timing performance for APM in GTX1080 with k=2 and 4. 162
Figure 5.20: Timing performance for APM in GTX1080 with k=6 and 8. 162

LIST OF PSEUDO ALGORITHMS

Pseudo Algorithm 2.1: Wu-Manber Algorithm 42
Pseudo Algorithm 3.1: Pre-Processing of Data Partition 75
Pseudo Algorithm 3.2: PFOR-Delta Compression 77
Pseudo Algorithm 3.3: PFOR-Delta Decompression 78
Pseudo Algorithm 3.4: GPU Join 89
Pseudo Algorithm 3.5: Hash Table Partition 92
Pseudo Algorithm 5.1: Simple Computation for Sequential Histogram 141
Pseudo Algorithm 5.2: Global Memory Stride Access Kernel Code 144
Pseudo Algorithm 5.3: Shared Memory with Stride Access Kernel Code 145
Pseudo Algorithm 5.4: CUDA Stream Implementation 147

LIST OF ABBREVIATIONS

AOS	Array of Structure
API	Application Programming Interface
APM	Approximate Pattern Matching
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPI	Deep Packet Inspection
FFNs	Feed-Forward Neural Networks
FPGA	Field Programmable Gate Arrays
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphic Processing Unit
HPC	High Performance Computing
ILP	Instruction Level Parallelism
LZSS	Lempel–Ziv–Storer–Szymanski
MIMD	Multi Instruction Multi Data
NFA	Non deterministic Finite Automaton
PCIe	Peripheral Component Interconnect Express
SIMD	Single Instruction and Multiple Data
SMs	Streaming Multiprocessors
SOA	Structure of Array
SQL	Structure Query Processing

CHAPTER 1

INTRODUCTION

In recent years, the field of computational accelerators has witnessed significant advancements and growing interest due to its potential high performance applications in Big Data, Geographic Information System, Internet of Things Sensor Data Analysis, and others. However, despite the progress made, there remain critical challenges and gaps in this thesis understanding of optimizing the synergy of heterogeneous computation. It involves the multi-core (CPU) and many-core (GPU) acceleration techniques with its unique architecture, while providing insights into the key parallel processing design trade-offs. This thesis aims to address these challenges by harnessing the power of CPU and GPU heterogeneous computing. Through a comprehensive investigation, this research seeks to contribute original insights and extend the boundaries of knowledge in GPU-accelerated structure query processing. By employing a GPU parallel processing to innovate in structure query processing, this thesis explores efficiency in comparison to conventional CPU-based and multi-node Hadoop cluster approaches. The findings of this research hold significant implications for GPU-accelerated structure query processing and heterogeneous computing architectures. These have the potential to dramatically improve the performance and efficiency.

1.1 Background

The exponential growth of data generated and collected annually has led to a data deluge, posing significant challenges for organizations across industries. The IDC Digital Universe Study 2013 annual report predicted that data would rise by a factor of 10, from 4.4 zettabytes to 44 zettabytes through 2022 (Adshead, 2014). According to latest projections from Statista's Technology and Telecommunication team, data is expected to reach 181 zettabytes by 2025 (Taylor, 2022). Data discovery capabilities are becoming increasingly important for making adequate decisions in increasingly complex business rules and environments. It raises huge challenges to the traditional database system today in dealing with massively stored data and rapidly performing query operations on it. Hence, there is a pressing need for diverse and specialized computational accelerators.

1.1.1 The Rise of Heterogeneous Computing in Structured Query Processing

Heterogeneous computing has emerged as a compelling pathway to accelerate structured query processing as data volumes grow beyond what monolithic CPU-only systems comfortably handle. By orchestrating diverse compute elements, CPUs, FPGAs (Field Programmable Gate Arrays), ASICs (Application-Specific Integrated Circuit) and GPUs, these can match the strengths of each processor type to the varied demands of modern database workloads, overcoming uniform-hardware bottlenecks and unlocking significant gains in efficiency and speed (Brodtkorb *et al.*, 2010; Arora, 2012; Sukhwani *et al.*, 2012; Luk *et al.*, 2009). This shift is particularly valuable for analytics on large-scale datasets and complex computations, where traditional

CPU-centric engines may struggle to deliver timely results (Hung *et al.*, 2014; Breß *et al.*, 2014a; Paul *et al.*, 2021). At the same time, although CPUs have advanced through multi-core designs, they remain constrained in data-parallel throughput for database operators, and the slowing of Moore's Law has reduced the pace of single-thread and general-purpose performance scaling, approaching practical limits set by physical and economic factors (Shalf, 2020). These limits have catalysed the adoption of general-purpose GPU computing (GPGPU) and a broader move to heterogeneous computing, tracing a trajectory from specialized graphics processors toward general-purpose accelerators and fused CPU–GPU chips as well as large-scale heterogeneous systems. In common usage, "GPGPU" is used interchangeably with "GPU," and modern heterogeneous programming frameworks now make it feasible to integrate these processors more seamlessly into database execution (Brodtkorb *et al.*, 2010).

FPGA and ASIC are types of integrated circuits with its rigid architecture and restricted flexibility impose notable constraints when compared to more versatile processors like CPUs or GPUs. FPGA provides the lower performance with consuming more power, large form factor with complex design principles, and resulting in a steeper learning curve comparing to software programming (Meza *et al.*, 2013; Das and Wilton, 2011). ASICs cannot be altered after manufacture, as any design changes necessitate a costly and time-consuming process of redesign and fabrication (Kuhn and Alessi, 1989). Thus, it is not a common accessibility for the programmability for the developers (Stok and Cohn, 2003). In contrast, GPGPU is a more flexible and accessible alternative for accelerating query processing, providing high performance, programmability, and cost-effectiveness compared to FPGA and ASIC.

In practice, two main integration strategies have taken shape. The first extends a conventional DBMS with GPU offloading via API-level hooks: established systems like SQLite, PostgreSQL (via PG-Storm/HeteroDB), and Brytlyt can detect GPU-friendly operators at query time, dispatch those parallelizable fragments to the GPU, and then merge the results back into the DBMS execution pipeline (Hordemann *et al.*, 2014; Kohei, 2015; Heyns, 2020). This approach delivers targeted speedups without wholesale architectural upheaval, though it must manage the complexity of identifying CPU-versus-GPU work within queries and the overheads of data transfer and layering between CPU and GPU (Zhang *et al.*, 2013). The second strategy exports data from the host DBMS into a dedicated GPU-accelerated store that maintains its own optimized columnar/parallel format. This approach adopts in GPU-accelerated query processing systems such as SQream, OmniSci, and Kinetica (SQream, 2017; Heavy.AI, 2020; Kinetica, 2021). While this can yield broader, sometimes larger, performance gains across many database functions, it introduces duplicated storage and operational complexity because data must be maintained in both the original and the GPU-native systems (Morgan, 2020).

As CPU performance scaling plateaus and multi-core designs hit limits for data-parallel database work, the need for higher-throughput execution of query operators has intensified (Shalf, 2020). This has propelled the rise of GPGPU and heterogeneous architectures that pair CPUs with accelerators and supportive programming frameworks, enabling efficient mapping of operators to the most suitable hardware (Luk *et al.*, 2009; Brodtkorb *et al.*, 2010; Arora, 2012). In databases, this materializes through either GPU offloading within existing DBMSs or migration to GPU-native analytic stores, each with distinct benefits

and operational trade-offs (Hordemann *et al.*, 2014; Morgan, 2020). Together, these developments establish the technical motivation and architectural options for accelerating structured query processing in the era of heterogeneous computing (Hung *et al.*, 2014; Breß *et al.*, 2014a; Paul *et al.*, 2021).

1.1.2 Identifying and Addressing Bottlenecks in Multi-core CPU

Modern multi-core CPUs have expanded the parallel resources available to database engines, yet they still strain under data-parallel workloads that dominate contemporary analytics. As data sizes grow and operators compete for parallel execution slots, CPUs encounter limits in extracting enough parallelism to keep pace. This strain is reflected in two coupled realities. First, the curve of data growth and transistor density has reached a point where CPU parallelism alone is not enough to serve highly parallel query operators efficiently (Hamilton, 2003; Keckler *et al.*, 2011; Lastovetsky and Manumachu, 2023). Second, even when heterogeneous acceleration is introduced, end-to-end speedups can be blunted by classic systems concerns such as concurrency control, cache coherence, and load balancing, each of which can inject stalls, contention, or idle cores into the pipeline (Boncz *et al.*, 2008; He *et al.*, 2014; Wang *et al.*, 2021). These factors together motivate a design stance that treats the CPU as one part of a broader compute fabric rather than the sole engine of progress.

At the hardware level, the historical gains promised by Moore's Law and Dennard scaling have weakened. Moore observed a doubling of on-chip transistors roughly every 18 months, and Dennard scaling explained how smaller transistors could switch faster without raising power density (Moore, 1965; Dennard *et al.*, 1974). As feature sizes shrank further, leakage currents rose,

static power climbed, and systems ran into a power wall that limited safe frequency and voltage scaling (Kim *et al.*, 2003). Multi-core designs responded by adding cores, but rising transistor counts also increased power and thermal pressure. The Intel Itanium Tukwila generation illustrated this trend, packing roughly two billion transistors, with a large share devoted to caches and instruction-level parallelism support, yet still constrained by power limits (Stackhouse *et al.*, 2008). Subsequent analysis argued that simply pushing for more parallelism on ever smaller processes would bring diminishing returns relative to the costs of further scaling, reinforcing the need to offload massive data-parallel work to accelerators such as FPGAs or GPUs (Esmaeilzadeh *et al.*, 2011; Erdem, 2016).

On the database system side, the friction points that matter most to the performance show up in the critical path of query execution. Concurrency control must protect integrity when many transactions run in parallel, but its locks, latches, or validation steps can heighten contention on hot data and reduce effective parallelism on multi-core machines (Ross, 2014). Cache coherence protocols maintain a consistent view of shared state across per-core caches, which is essential for correctness yet can throttle throughput as cores repeatedly invalidate and fetch shared lines during join, aggregation, and index operations (Pankratius and Heneka, 2011). When working sets exceed cache capacity, miss rates rise and data locality suffers, stretching memory latency and suppressing the gains from additional threads (Chitters *et al.*, 2013). Finally, load balancing determines whether parallel operators actually keep all cores busy. Imbalance can leave some cores overloaded while others idle, particularly in skewed joins or group-by operations, turning the scheduler itself into a bottleneck that erodes

parallel scalability (Ye *et al.*, 2011). Against this backdrop, heterogeneous designs that offload the most data-parallel kernels to accelerators complement the CPU's strengths in control-heavy or latency-sensitive tasks, enabling the system to sidestep power and coherence limits while raising overall query throughput (Hamilton, 2003; Keckler *et al.*, 2011; Lastovetsky and Manumachu, 2023; Boncz *et al.*, 2008; He *et al.*, 2014; Wang *et al.*, 2021).

1.1.3 Operator-Level Acceleration and Programming Models

GPUs have proven valuable for database operators that benefit from high degrees of data parallelism. Prior work shows that scans can be chunked and processed concurrently for fast filtering over large tables, aggregations can combine many inputs into compact summaries with high throughput, and joins can be executed in parallel to manage the heavy costs that typically dominate query processing at scale (Tarditi *et al.*, 2006; Shams and Kennedy, 2007; Kaldewey *et al.*, 2012; Sitaridi and Ross, 2013; Xiangwu and Jinxin, 2016). In practice, these operator-level accelerations translate into better end-to-end responsiveness when datasets grow faster than CPU-only engines can keep up.

Heterogeneous programming models matter because they shape how a system splits work between multi-core CPUs and many-core GPUs, and how efficiently that work moves through the pipeline. Effective models encourage structured data partitioning so that large inputs are divided into manageable pieces that can be processed at the same time, while paying attention to locality and communication costs (Cremonesi and Sorrenti, 1995; Drozd *et al.*, 2012). Naive partitioning can create hot spots and idle resources, which is why strategies must account for imbalance, skew across processing stages, and the

choice of partition count itself (Huo *et al.*, 2011; Ke *et al.*, 2011; Zhong *et al.*, 2012; Yao *et al.*, 2013; Wang *et al.*, 2017; Anupama and Lakshmi, 2022). These choices can be the difference between linear scaling and disappointing returns.

Certain workloads highlight the benefits of these models especially well. When compression and decompression run directly on GPUs, systems move less data through storage and interconnects, which trims bandwidth costs and raises throughput for both distributed settings and high-rate streams (Cloud *et al.*, 2011; Raichand and Aggarwal, 2013). In geospatial analytics, heterogeneous approaches accelerate spatial joins, overlays, and proximity queries across very large spatial datasets, bringing runtimes down to practical levels for interactive or near-real-time use (Stojanovic and Stojanovic, 2014; Gao *et al.*, 2018). String matching, which is central to indexing and text-heavy query processing, can also benefit. Although text size variability and irregular control flow complicate parallelization, applying classic algorithms such as Aho-Corasick (Aho and Corasick, 1975), Boyer-Moore (Boyer, 1977), and Knuth-Morris-Pratt (Knuth *et al.*, 1977) within GPU-friendly formulations has been shown to speed up these operations in database contexts (Kouzinopoulos and Margaritis, 2009; Zha and Sahni, 2013a; Mitani *et al.*, 2017).

At a system level, developer-facing frameworks influence how easily engineers can orchestrate CPU–GPU cooperation. CUDA has become a widely used ecosystem for writing kernels, managing execution, and coordinating data movement with host code. Practical deployments often favour push-based query processing to better feed compute units with shared input streams, which reduces I/O duplication and shifts the dominant cost from movement to computation.

With careful orchestration of concurrent work and data transfers, these models help heterogeneous databases sustain high utilization across both processors while minimizing overheads that would otherwise erode the gains from parallel hardware.

1.2 Embarking On The Research

Databases keep getting bigger, and CPU-only methods often stall on memory bandwidth and cache limits. This thesis asks a simple question with practical stakes: when do GPUs actually help with SQL workloads, and by how much? This thesis studies core operators such as joins, aggregations, and sorting, mapping them to GPU kernels and tuning thread blocks, shared memory use, and warp control. To work within device memory, this thesis use batching and lightweight compression. To avoid transfer bottlenecks, this thesis overlap I/O and compute and reduce PCIe traffic with careful pipeline design. A compact cost model guides when to offload work to the GPU and when to keep it on the CPU. Experiments on realistic datasets, including cloud GPU instances, test these choices across data scales. The outcome is a practical design for heterogeneous execution that improves throughput and energy per query while remaining portable and maintainable. The goal is clear: faster queries without trading away correctness or system simplicity.

1.2.1 Problem Statement

Enterprises now run decision-support and data-engineering workloads that scan, filter, join, and aggregate billions of rows. Traditional CPU execution hits practical limits in memory bandwidth, vector width, and thread-level parallelism. Modern GPUs offer much higher on-board bandwidth, thousands of lightweight

threads, and hardware scheduling that hides memory latency (Govindaraju *et al.*, 2004; Breß and Saake, 2013; Sen *et al.*, 2023). These traits make GPUs a natural fit for column scans, hash aggregation, and join-heavy SQL. Yet in full systems the expected gains do not consistently appear.

This thesis addresses three concrete GPU-side problems that block those gains. Firstly, pre-processing is often CPU-centric. Column layout, dictionary coding, compression, and chunking are chosen without regard to GPU memory coalescing or warp behaviour. That creates unaligned loads, extra host–device transfers, and low occupancy (Breß *et al.*, 2014b; Sen *et al.*, 2023). This thesis targets GPU-aware pre-processing that keeps data resident on device, favours aligned access, and overlaps I/O and compute using CUDA streams.

Secondly, several relational operators, especially hash joins, have not been reworked for current devices and runtimes. Older kernels ignore features like larger shared memory, refined cache behaviour, and concurrent kernel execution. They suffer from skew, branch divergence, and random access that defeats bandwidth (Sioulas *et al.*, 2019). This thesis redesigns and evaluates join variants that manage skew, exploit shared memory for build–probe phases, and maintain coalesced access.

Thirdly, coordination across the pre-processor, the SQL parser with its scheduler, the query engine, and the GPU Columnar File System is ad hoc. Operator placement between CPU and GPU is decided case by case, which inflates PCIe traffic and creates I/O stalls (Bakkum and Chakradhar, 2012; Yogatama *et al.*, 2022). This thesis develops a simple decision model that places

each operator based on data shape, transfer cost, and contention, so the pipeline sustains throughput across one or more GPUs.

These issues are most visible when SQL mixes with text-heavy operators such as LIKE, regex, and dictionary lookups used in cleansing logs and geospatial joins. Many of these paths remain tuned for scalar CPUs (Wu *et al*., 2014; Sitaridi and Ross, 2016). This thesis therefore includes GPU methods for short-pattern matching and hashing so text predicates do not pull execution back to the host. The problem of this thesis solves is to connect GPU-aware pre-processing, GPU-native operators, and a practical scheduler into a single path that delivers repeatable speedups on TPC-H style queries and real data-engineering tasks.

1.2.2 Research Questions

Today's analytic engines mix CPUs and GPUs, but full SQL runs still slow down because pre-processing is shaped for the CPU, many join kernels do not match how new GPUs really work, and operator placement across devices is often a guess. This thesis asks whether GPU-aware pre-processing improves residency and cuts transfers, whether redesigned joins give repeatable gains, and whether a simple placement rule reduces PCIe traffic without harm to accuracy. Below are the list of research questions.

1. How much does a compression-first, GPU-aware pre-processing path using PFOR-Delta with parallel chunking improve device residency, reduce host-device transfers, and cut end-to-end query time compared with CPU-centric layouts or uncompressed columns?

2. What chunk size and partitioning strategy maximise GPU occupancy without spilling, given your formula for chunk sizing and use of streams and pinned memory?
3. Do redesigned hash-join variants that use shared memory for build-probe, coalesced access, and skew handling outperform a tuned multi-core CPU baseline across TPC-H selectivity and scale factors?
4. How do coalesced access and cache-friendly layouts change global load efficiency and join throughput in your kernels?
5. Can a simple decision model for placing operators across CPU and GPU, driven by data shape and transfer cost, reduce PCIe traffic and I/O stalls compared with ad hoc offloading, while preserving accuracy?
6. Under what measurable conditions should your system prefer the GPU over the CPU, and vice-versa, and do those conditions match the decision rule you conclude with?
7. Do GPU methods for short-pattern approximate string matching, paired with an asynchronous multi-stream scheduler, outperform a synchronous baseline as match rate increases, without regressing overall SQL query time?
8. Do the gains you report hold across both TPC-H scale factors and the 32 GB healthcare dataset with realistic schemas, and do results remain consistent on your Kepler GPU workstation versus cloud GPU instances?

9. What are the effects on CPU utilisation and energy per query when the pipeline keeps columns resident and overlaps copies with kernels, compared to CPU-only execution?

1.2.3 Research Objectives

The primary goal of this study is to determine when and how GPU acceleration delivers measurable gains for big data processing. This thesis focuses on heterogeneous execution that combines NVIDIA GPUs with Intel CPUs, and this thesis examines the practical limits that arise in massively parallel workloads. The work assesses bottlenecks such as data movement, memory hierarchy use, and kernel scheduling, then proposes methods to reduce them. The intended outcome is a clear, evidence-based blueprint for an efficient GPU-accelerated structured query system. Thus, this aims to develop a highly efficient GPU-accelerated structure query system. The objectives are below:

Objective 1: Investigation the design of GPU-aware query pipeline for a single-GPU, CPU–GPU system.

- To design a compression-first pre-processing path, size chunks to fit device memory, and implement GPU-centric join operators with coalesced access and skew handling.
- To evaluate gains in device residency, host-device transfers, occupancy, join throughput, and end-to-end query time on NVIDIA Tesla class GPUs.

Objective 2: Enhancement of the efficiency and end-to-end speedups for structured and geospatial queries on real datasets.

- To develop the full pipeline on TPC-H scale factors and a healthcare workload, and include geospatial operators where used.
- To accelerate the speedup, variability, and resource use across GPU to show that improvements hold beyond a single machine or dataset.

Objective 3: Optimisation text predicates with GPU string matching without pulling execution back to the CPU.

- To optimise short-pattern matching, including approximate methods such as Wu–Manber, so LIKE and regex filters run on the device.
- To measure kernel speed, match-rate sensitivity, and the effect on overall SQL query latency when these predicates stay on the GPU.

1.3 Thesis Contribution

Chapter 2 shows that GPUs can accelerate individual operators and specialised workloads, yet it also highlights three gaps that limit their impact on full SQL query processing. First, many GPU database prototypes focus on isolated kernels or partial SQL support rather than an integrated accelerator that combines compression aware columnar storage, joins, and text predicates in one pipeline. Second, lightweight compression schemes such as PFOR-Delta have been evaluated mostly in isolation, so their combined effect with GPU joins and string matching on end-to-end performance is still not well understood. Third, there is limited system level evidence that compares a single GPU accelerator against mainstream CPU and big data platforms on realistic analytical workloads.

This thesis contributes a focused path to faster SQL on a single GPU that works alongside a multi-core CPU. It starts by locating where time is actually

lost in current systems, including CPU centric pre-processing that starves the device, join kernels that underuse bandwidth and parallelism, and ad hoc decisions about when to offload. In response, the work develops a GPU aware pipeline that keeps data resident whenever possible, reduces unnecessary data transfers, and matches operators to the hardware execution model. Design choices are driven by measurements and are validated using complete query runs rather than microbenchmarks alone.

The first contribution is a practical GPU aware query pipeline that integrates a compression first pre-processor, an SQL parser with a simple scheduler, and a GPU query engine. Columns are chunked to fit device memory, pinned buffers and CUDA streams overlap copy and compute, and join operators are rewritten to favour coalesced access while handling skew. This combination improves device residency, reduces bytes moved per query, and raises effective occupancy. The outcome is a design that a database engineer can implement on a single NVIDIA Tesla class device without re-architecting the entire system.

The second contribution is an end-to-end evaluation that demonstrates consistent speedups for structured queries on both standard decision support workloads and a real healthcare dataset. The same pipeline is exercised across TPC H scale factors and a 32 GB healthcare workload, with geospatial operators included when they are used. Results are reported in terms of query level speedup, variability across runs, and GPU resource use. Together, these experiments show how a single GPU accelerator compares with a tuned multi-core CPU engine and representative Hadoop based systems.

The third contribution targets text predicates that usually pull execution back to the host. The thesis designs and evaluates GPU string matching paths for short patterns, including approximate matching based on established algorithms. By keeping LIKE and related filters on the device and scheduling them asynchronously with the rest of the pipeline, the system reduces stalls and preserves the benefit of earlier offloading. Measurements cover kernel speed, sensitivity to match rate and pattern length, and the impact on full query latency when string predicates remain on the GPU.

Taken together, these contributions address the gaps identified in Chapter 2. They move beyond the general observation that GPUs can be fast for individual operators and instead show how an integrated, compression aware query accelerator behaves as a practical component in end-to-end analytical processing.

1.4 Outline of Thesis

This section presents a structured outline of the thesis as follows.

Chapter 1 introduces the motivation and background for heterogeneous processing in structured query workloads. It sets out the problem statement, research questions, and objectives, and summarises the main contributions of the thesis.

Chapter 2 reviews prior work on heterogeneous architectures for relational query processing, GPU aware columnar storage and compression, GPU-accelerated join algorithms, GPU acceleration in big data and geospatial workloads, and GPU based string processing. The literature is organised into

these themes and concludes with a synthesis of research gaps that motivate the architecture and experiments in this thesis.

Chapter 3 presents the design and implementation of the GPU structured query accelerator. It describes the overall architecture, the compression first pre-processing path, and the GPU join operators. The chapter also explains the experimental setup and reports evaluation results for the core pipeline.

Chapter 4 examines the pathways of GPU acceleration in big data applications. It compares the GPU-accelerated query engine with Hadoop based systems on a healthcare dataset and explores GPU based geospatial processing as another demanding workload, followed by a discussion of the implications for real deployments.

Chapter 5 investigates GPU-accelerated string matching for query predicates. It studies hash based matching, histogram optimisation with CUDA, and accelerated bit parallel approximate matching for short patterns, and evaluates their impact on end-to-end query performance.

Chapter 6 summarises the main findings of the thesis, reflects on the limitations of the current work, and outlines directions for future research in GPU-accelerated structured query processing.

CHAPTER 2

LITERATURE REVIEW

2.1 GPU-accelerated Structured Query Processing

The modern database is no longer just a passive storage component. It has become a central computational asset that supports transaction processing, interactive analytics, and increasingly complex data-driven decision making. This shift traces back to early work that framed databases as integrated systems for managing corporate data, in which structured records and navigational access paths formed the backbone of information systems (Bachman, 1973). As data volumes and query complexity increased, conventional CPU-centric engines faced pressure from two directions. On one side, applications demanded fast, interactive analytics over ever larger datasets. On the other, hardware trends moved toward deep memory hierarchies and increasing parallelism rather than simple single-core frequency scaling. These developments created a natural space for heterogeneous acceleration, where GPUs, with their high memory bandwidth and many-core parallelism, became attractive candidates for structured query processing.

2.1.1 From early GPU primitives to full database engines

The first wave of work on GPU acceleration in data management focused on individual relational operators rather than complete systems. Researchers demonstrated that GPUs, originally designed for graphics pipelines, could be repurposed to speed up selection, aggregation, and join operators by mapping

data-intensive kernels onto programmable graphics hardware (Govindaraju *et al.*, 2004; Breß *etl al.*, 2013; Sen *et al.*, 2023). These studies exploited the Single Instruction and Multiple Data (SIMD) style execution model and high throughput of GPUs, showing that even using relatively constrained programming interfaces, it was possible to outperform CPU-only baselines for core relational tasks (He *et al.*, 2009; Huang and Chen; 2015).

Later work took advantage of CUDA and similar general-purpose GPU frameworks to implement more flexible operator pipelines. For example, early GPU database prototypes explored offloading selections and projections to the device, while the CPU remained responsible for orchestration and some control-heavy tasks (Bakkum and Skadron, 2010; Meister and Saake, 2016). Hybrid designs carefully split the query plan between CPU and GPU components in order to balance compute, memory traffic, and control overhead (Heimel and Markl, 2012; Behrens *et al.*, 2018; Li *et al.*, 2025). These proof-of-concept engines established that relational operators could be mapped efficiently to GPUs, and they laid the ground for more ambitious, system-level designs.

2.1.2 GPU-accelerated database architectures

As GPU programming environments matured, research shifted toward complete database management systems that integrate GPU acceleration at the engine level. A recurring theme in this literature is the question of how to structure data and operators so that the GPU can be used effectively, while still supporting a full SQL interface.

Several studies highlight that GPU acceleration is most effective when combined with in-memory, column-oriented storage. One-operator-at-a-time

processing and columnar layouts improve cache locality and reduce unnecessary data movement, which is especially important when data must cross the PCIe bus between CPU and GPU memories (Przymus and Kaczmarski, 2014; Varakin and Gal, 2014). Systems such as MapD (Mostak, 2013; Root and Mostak, 2016) exemplify this approach by maintaining columnar in-memory representations and using a parallel engine that can execute queries either on the GPU or the CPU, depending on resource availability and data placement.

Similarly, GPUDB keeps the primary database in CPU main memory yet uses a column-based, block-oriented processing model to overlap data transfers with computation, carefully using pinned host memory to reduce transfer overhead (Yuan *et al.*, 2013). OmniDB adopts a hardware-oblivious kernel that is connected to devices through adapters, so that the same logical operators can be executed on CPUs or GPUs with minimal changes to the core engine (Zhang *et al.*, 2013). These designs place the GPU as a first-class execution resource, but they differ in how much of the query pipeline is offloaded and how aggressively they assume that data will reside in device memory.

Across these systems, a common insight is that transferring large intermediate results back and forth between CPU and GPU can easily negate the computational gains from massive parallelism (He *et al.*, 2009; Bordawekar and Sadoghi, 2016). As a result, a central design objective is to keep data in memory, minimize data movement between CPU and GPU, and exploit columnar layouts that enable coalesced memory accesses and high throughput (Fang *et al.*, 2010; Breß *et al.*, 2014b).

2.1.3 Cost models and heterogeneous query planning

Heterogeneous engines must decide whether a particular query or sub-plan should execute on CPU or GPU. This decision is rarely trivial. While GPUs offer higher peak throughput and memory bandwidth, they also incur kernel launch overheads and, in discrete configurations, additional data transfer costs. To address this, several works propose cost models that estimate whether executing a given operator or pipeline on the GPU will lead to net speedup (Ranger *et al.*, 2007; Przymus and Kaczmarski, 2014; Wang *et al.*, 2021). These models incorporate factors such as operator selectivity, input cardinality, expected data reuse, and the relative performance of CPU versus GPU for different operator types.

Empirical studies consistently report that GPUs excel at joins and complex aggregations, particularly when large volumes of data must be processed with relatively simple arithmetic on each tuple (He *et al.*, 2009; Breß *et al.*, 2014b). However, when queries are dominated by selective point lookups or involve small result sets, CPU-based execution often remains competitive or superior, especially once data transfer overhead is considered (Varakin and Gal, 2014). This tension motivates hybrid designs in which a cost-based optimizer can choose device assignments on a per-operator basis, and where columnar, in-memory data layouts reduce the penalties of switching between devices (Przymus and Kaczmarski, 2014).

2.1.4 Limitations and implications for this thesis

Despite encouraging results, the literature also reveals several limitations. Many early GPU database prototypes assume relatively small datasets that can

fit in GPU memory, or they rely on streaming mechanisms that treat the GPU as a high-throughput coprocessor without fully addressing the complexity of large-scale, multi-query workloads (Yuan *et al.*, 2013; Xi *et al.*, 2025). Several studies also focus on individual operators in isolation, which simplifies evaluation but does not capture the interactions that arise in realistic query mixes (Heimel and Markl, 2012; Paul *et al.*, 2016).

Furthermore, while cost models have been proposed, they are often tailored to a specific hardware generation and do not fully capture the evolving balance of CPU and GPU capabilities or the emergence of new memory hierarchies. This thesis builds on these insights by adopting in-memory, columnar designs and by explicitly targeting heterogeneous CPU–GPU execution for complete query pipelines, rather than isolated operators.

2.2 GPU Compression for Column Based Data Store

Scaling column-based databases to big-data workloads quickly encounters storage and bandwidth bottlenecks. Compression techniques, particularly when deployed on GPUs, offer a way to alleviate both concerns by reducing the volume of stored data and the amount that must be moved across memory hierarchies (Mensmann *et al.*, 2010; Boeschen *et al.*, 2024). In a columnar setting, where each column stores values of a single attribute, data exhibits stronger regularities and homogeneity, which improves compressibility and makes it easier to design schemes that align with GPU memory access patterns (Fang *et al.*, 2010; Li *et al.*, 2016).

2.2.1 Role of compression in columnar and GPU settings

Columnar databases commonly apply compression at the page or segment level so that compressed blocks can be loaded and decompressed on demand. In a GPU context, compression becomes even more important because device memory is typically smaller than host memory and data transfers are expensive. When applied carefully, compression reduces both storage footprint and I/O, and it can even speed up query processing if decompression is fast enough relative to the saved transfer time (Waidyasooriya *et al.*, 2014; Nelson *et al.*, 2020).

A broad body of work distinguishes between lightweight and heavyweight compression schemes. Disk-based column stores often employ heavyweight algorithms such as Huffman coding to maximize compression ratios (Huffman, 1952). In contrast, in-memory column stores favor lightweight schemes like variants of Lempel–Ziv that trade some compression ratio for higher decompression speed and more predictable performance (Ziv and Lempel, 1977). For GPU-based systems, this trade-off becomes sharper because decompression must keep up with massive parallel execution and must avoid control divergence and irregular memory access patterns (Rahmani *et al.*, 2014; Xiang *et al.*, 2014; Yang *et al.*, 2024).

2.2.2 Heavyweight schemes on GPUs and their limitations

Several studies attempt to port heavyweight algorithms such as Huffman coding to GPUs, seeking to exploit their high memory bandwidth for both compression and decompression (Cloud *et al.*, 2011). However, constructing Huffman trees is inherently sequential and involves complex data dependencies, which makes parallelization challenging (Rahmani *et al.*, 2014). Even when

using advanced GPU architectures with very high memory bandwidth, such as NVIDIA Volta, implementations often suffer from warp divergence and inefficient memory usage when symbol frequencies are skewed (Xiang *et al.*, 2014; Tian *et al.*, 2021).

As a result, although GPU-based Huffman coding can achieve good compression ratios, it tends to incur non-trivial decompression overhead and resource under-utilisation. These limitations motivate a growing focus on schemes that better match GPU execution models, relying on more regular control flow, fixed-length encodings, and fine-grained parallelism (Patel *et al.*, 2012; Vijaykumar *et al.*, 2015).

2.2.3 Lightweight schemes, vectorisation, and GPU-friendly designs

Lightweight compression techniques exploit the fact that many analytical workloads use integer or fixed-width numeric data, and that differences between consecutive values are often small. For CPU-based column stores, vectorised schemes such as PFOR, PFOR-Delta, and patched dictionaries have proved effective in achieving high throughput by aligning data structures with SIMD instructions (Abadi *et al.*, 2006; Guthe and Goesele, 2016; Zhang *et al.*, 2023). These approaches encode frames of data relative to a reference value, then apply bit-packing to store the deltas compactly, while preserving the ability to decompress quickly.

On GPUs, researchers adapt these ideas by combining delta coding, bit-packing, and dictionary compression with parallel chunking and careful memory layout. Work on planning and selecting compression schemes demonstrates that it is beneficial to choose different lightweight schemes for different segments,

based on data characteristics, in order to balance compression ratio and decompression speed (Fang *et al.*, 2010; Przymus and Kaczmarski, 2014). Parallel Lempel–Ziv–Storer–Szymanski (LZSS) variants, such as CULZSS, use fixed-size buffers and asynchronous execution to saturate GPU pipelines while resolving shared-memory conflicts in string compression (Ozsoy and Swany, 2011). PMLZSS extends this by using matrix-based matching to parallelise window-based search, achieving notable speedups over CPU-based LZSS and BZIP2 while maintaining similar compression ratios (Zhou *et al.*, 2014).

Time-series workloads inspire further specialisation. Dynamic selection of lightweight algorithms can accommodate diverse value distributions and temporal patterns while keeping compression overhead small enough for high-volume data ingestion (Przymus and Kaczmarski, 2014). For volumetric and regularly sampled scientific data, random access and on-the-fly decompression are crucial, and GPU-friendly schemes have been proposed to cope with the curse of dimensionality while still offering interactive performance (Guthe and Goesele, 2016). More recently, tile-based integer compression specifically tailored to GPU column stores has been introduced, with single-pass decompression models that align bit-packing and data tiling to global memory access patterns (Shanbhag *et al.*, 2022).

Complementary work revisits classic page and index compression. Two-level relation compression strategies reduce leaf counts in B-trees and R-trees, improve fan-out, and combine page-level and file-level compression to enhance storage and query performance (Goldstein *et al.*, 1998). Delta coding integrated with vectorised encodings further exploits clustered integer distributions, and

when implemented with SIMD instructions, yields substantial improvements in both compression and decoding speed (Lemire and Boytsov, 2015). Empirical evaluations of vectorised compression confirm that these approaches deliver strong energy efficiency and performance benefits for memory-bound applications (Hasib *et al.*, 2015). Recent work such as BTRBLOCKS combines frame-of-reference, bit-packing, and run-length encoding for floating point and string data, offering robust performance across diverse datasets (Kuschewski *et al.*, 2023). In this landscape, PFOR-Delta stands out as a balanced candidate that combines reasonable compression ratios, fast decompression, and limited dependence on specific data properties, making it attractive for GPU acceleration (Zukowski *et al.*, 2006).

2.2.4 Column-wise partitioning and data movement

GPU-based compression pipelines are especially sensitive to data layout and movement. Column-wise partitioning separates data into homogeneous segments, which better match the coalesced access patterns required by GPUs and reduce latency by focusing on contiguous ranges of a single attribute (Raichand and Aggarwal, 2013). This approach allows multiple GPU threads to process different parts of the same column concurrently, increasing throughput and utilising parallel hardware more effectively (Begoli *et al.*, 2016; Ohara and Yamaguchi, 2012).

Selective transfer of only the required columns from CPU to GPU memory further reduces data movement and helps to avoid bandwidth bottlenecks, an effect that becomes more pronounced in large analytical workloads (Shen *et al.*, 2023). By organising data into GPU-friendly column segments and applying

carefully chosen lightweight compression schemes, systems can reduce both storage and communication overhead, while still supporting fast decompression during query processing.

2.2.5 Summary and implications

The literature on GPU compression for column-based data stores suggests a clear trajectory. Heavyweight algorithms remain useful for archival storage, but for query processing in heterogeneous systems, lightweight, vectorised schemes, combined with column-wise partitioning and cost-aware selection, offer better performance and flexibility (Mensmann *et al.*, 2010; Zukowski *et al.*, 2006). This thesis builds upon these insights by adopting GPU-friendly lightweight schemes, such as PFOR-Delta, in conjunction with columnar chunking and compression planning, to reduce the memory and bandwidth footprint of analytical workloads while preserving fast query response times.

2.3 Accelerated Execution for Join Operation

Join processing is often the most expensive component of analytical queries. Joins combine tuples from different relations based on key attributes, and their performance heavily influences end-to-end response time. Traditional database engines invest significant effort in optimising join order and choosing among join algorithms such as nested-loop, sort-merge, and hash join, guided by cost models that account for selectivity, cardinality, and available indexes (Meister and Saake, 2016). As data grows, both CPU and GPU-based solutions must address not only algorithmic complexity but also memory hierarchy behaviour and parallel execution constraints.

2.3.1 CPU-based join optimisation

Early work on join processing focused on improving CPU implementations by rethinking data locality, cache usage, and memory access patterns. Reconfiguring query processing algorithms to better exploit cache hierarchies can produce speedups ranging from modest to several-fold, even without changing asymptotic complexity (Shatdal *et al.*, 1994). Building on this, the radix partitioned hash join reorganises data into cache-friendly partitions, and analytical models have been developed to quantify how partitioning affects memory access cost (Boncz *et al.*, 1999).

Later refinements introduce sophisticated partitioning schemes such as radix-cluster, designed to reduce cache misses and improve main memory bandwidth utilisation (Manegold *et al.*, 2002). Other studies propose compact hash table designs, such as condensed hash tables that achieve very high fill factors using sparse bitmaps and integrated population counts, effectively acting as combined hash structures and Bloom filters for multi-table joins (Barber *et al.*, 2014). These CPU-focused innovations highlight the importance of aligning join algorithms with microarchitectural features and pave the way for analogous optimisations on GPUs.

2.3.2 Vector processors and early GPU-based joins

Before GPUs became the dominant accelerator platform, vector processors were used to speed up common relational operators including selection, projection, and joins (Meki and Kambayashi, 2000; Huang and Chen, 2015). These efforts confirmed that SIMD-style hardware is naturally well suited to the regular, data-parallel operations involved in relational processing.

As programmable GPUs became widely available, researchers revisited these ideas by mapping join operations onto graphics hardware. (Sun *et al.*, 2003) exploited the rendering and search capabilities of GPUs to accelerate spatial joins, demonstrating speedups in the range of roughly five times compared to CPU-based implementations. Subsequent work explored framebuffer-based selectivity estimation for join processing, using GPU rendering primitives to estimate counts of qualifying tuples very quickly (Govindaraju *et al.*, 2004). These early studies, although limited by then-current GPU interfaces, clearly indicated the potential of GPUs to act as powerful co-processors for join-heavy workloads.

2.3.3 Modern GPU join algorithms and comparative studies

With the advent of CUDA and similar programming frameworks, more flexible GPU join algorithms were proposed. A significant body of work focuses on reimplementing hash join and sort-merge join on GPUs, with attention to exploiting shared memory, warp-level parallelism, and coalesced global memory access (He *et al.*, 2009; He *et al.*, 2013; Furst *et al.*, 2017). These algorithms often allocate the smaller relation (inner table) to GPU memory, construct a hash table or sorted representation, and then stream the larger relation (outer table) through parallel probe or merge phases.

However, the relative advantage of GPUs over CPUs for joins is not uncontested. Some studies demonstrate that highly optimised CPU code can outperform GPU-based joins, especially for certain data sizes and skew patterns, achieving speedups of up to between two and eight times over earlier GPU implementations (Kim *et al.*, 2009). Other experiments report that GPUs remain

approximately 2.5 times as efficient as CPUs across a range of join workloads (Lee *et al.*, 2010). On yet another set of benchmarks, GPUs achieve speedups as high as 19 times for sort-merge joins and 14 times for hash joins relative to CPU-only baselines (He *et al.*, 2009).

These mixed findings underscore that join performance depends heavily on data distribution, hardware characteristics, and implementation details. Nevertheless, contemporary GPU join algorithms consistently demonstrate strong absolute performance as datasets grow, provided that memory layout and kernel design are tuned to the specific GPU architecture (He *et al.*, 2013; Barber *et al.*, 2014).

2.3.4 Summary and research gap

The literature on accelerated join processing reveals a rich landscape of CPU and GPU techniques. CPU-based approaches achieve impressive performance by exploiting cache hierarchies and compact data structures (Shatdal *et al.*, 1994; Boncz *et al.*, 1999; Manegold *et al.*, 2002). GPU-based approaches, in contrast, harness massive parallelism and high bandwidth but must deal with kernel divergence, limited device memory, and data transfer overhead (He *et al.*, 2009; He *et al.*, 2013; Nuriev *et al.*, 2024).

A notable gap lies in frameworks that integrate compressed, columnar storage with GPU-accelerated joins in a way that systematically accounts for both computation and data movement. Many studies focus either on join kernels or on data layout and compression in isolation. This thesis aims to bridge this gap by combining compression-aware data partitioning with GPU-friendly join

algorithms within a unified query accelerator, and by evaluating their performance in realistic analytical workloads.

2.4 Accelerated Big Data Processing with Geospatial Computation

Geospatial data plays a central role in domains ranging from environmental monitoring and urban planning to transportation, telecommunications, and public health (Jern *et al.*, 2008). Contemporary applications routinely handle large volumes of spatial and temporal information, such as high-resolution satellite imagery, GPS trajectories, and sensor observations. These datasets are not only large but also computationally intensive, since tasks like spatial joins, distance calculations, interpolation, and visualization often involve complex numeric operations.

2.4.1 Geospatial analytics as a GPU workload

The inherent data parallelism in many geospatial calculations makes them attractive candidates for GPU acceleration. Operations such as computing distances, aggregating points into heatmaps, or performing raster arithmetic can be expressed as per-cell or per-point kernels that map naturally to GPU threads. Prior work demonstrates that GPUs can significantly speed up core geospatial operations, including coordinate transformations, raster-based analyses, and spatial filtering, especially when computations are structured to exploit memory coalescing and avoid irregular control flow (Vetter and Westermann, 2011; Lai *et al.*, 2013; Calarasanu *et al.*, 2015).

Heatmap generation is a particularly illustrative example. By binning large numbers of points into grid cells and applying kernel density or other smoothing

functions, GPUs can produce interactive visualisations that would be prohibitively slow on CPU-only systems for very large datasets (Jern *et al.*, 2008). Similar acceleration has been demonstrated for spatial indexing, path planning on grids, and other compute-intensive spatial tasks (Fechner, 2010; Wal *et al.*, 2015).

2.4.2 Hybrid CPU–GPU architectures and geospatial engines

Several studies investigate hybrid architectures that combine CPU and GPU resources for geospatial workloads. These systems typically maintain geospatial data in an in-memory or distributed database, and invoke GPU kernels for computationally expensive parts of the workflow. Evaluations of NVIDIA K20 (Kepler) GPUs and Intel Many Integrated Core architectures illustrate that GPU-based approaches can outperform both CPU and alternative accelerator designs in many geospatial tasks, especially when data transfers are carefully managed (Vidal *et al.*, 2015; Ji *et al.*, 2024).

At the system level, integrating GPUs with established geospatial software stacks requires thoughtful design. For example, combining GPU kernels with high-level toolsets and in-memory big data systems such as distributed SQL engines or Cloudera Impala allows practitioners to retain familiar query interfaces while benefiting from acceleration in the background (Zhang *et al.*, 2015). Other work explores CUDA-based implementations of common geospatial kernels, including raster operations and spatial filters, highlighting the importance of fine-tuning thread block sizes and memory layouts (Zha and Sahni, 2013; Hung *et al.*, 2014; Zheng *et al.*, 2015).

2.4.3 Challenges and opportunities

Despite promising results, geospatial GPU acceleration faces several challenges. Many spatial datasets are sparse, irregular, or highly skewed, which can lead to load imbalance across GPU threads and reduced utilisation (Calarasanu *et al.*, 2015; Wal *et al.*, 2015). Spatial joins and nearest neighbour queries often require sophisticated indexing structures, such as R-trees or grid-based indexes, which are not straightforward to implement efficiently on GPUs (Goldstein *et al.*, 1998). Furthermore, integrating geospatial acceleration with general-purpose query processing, where spatial predicates are combined with complex SQL filters and joins, adds another layer of complexity.

Nevertheless, the literature shows that with careful data partitioning, representation, and operator design, GPUs can deliver substantial speedups for key geospatial computations, particularly when combined with columnar and in-memory storage models that minimise data movement (Vidal *et al.*, 2015; Zhang *et al.*, 2015). This thesis builds on these insights by treating geospatial workloads as one of several target use cases for a general GPU-accelerated query engine, rather than as a standalone pipeline.

2.5 Accelerated String Matching In GPU Parallel Streaming

String matching is a fundamental operation in many data-intensive domains, including information retrieval, bioinformatics, security monitoring, and log analytics. At its core, string matching aims to find occurrences of one or more patterns within a larger text or sequence. Classical algorithms such as Aho–Corasick (Aho and Corasick, 1975) and Wu–Manber (Wu and Manber, 1992) have been extensively studied and widely deployed. However, as data

volumes and throughput requirements grow, CPU-based implementations struggle to deliver real-time performance, particularly when large pattern sets or approximate matching are required.

2.5.1 String matching fundamentals

Exact string matching requires that patterns match substrings of the text without any differences. Aho–Corasick builds a trie-based automaton that simultaneously scans for many patterns in a single pass, making it attractive for applications such as network intrusion detection and deep packet inspection (Cantone and Faro, 2009). Wu–Manber relies on hash-based heuristics and shift tables to skip ahead in the text, thereby reducing the number of character comparisons (Wu and Manber, 1992).

Approximate pattern matching (APM) relaxes the requirement of exact equality and allows a bounded number of errors, which is essential in domains such as bio sequence analysis or noisy text processing (Cantone and Faro, 2009; Chavan *et al.*, 2016; Tahir *et al.*, 2017). APM is computationally more demanding than exact matching, since it must explore a larger state space or maintain additional information about edit distances. Bit-parallel algorithms mitigate these costs by encoding automaton states in machine words and updating them using bitwise operations, which can apply in parallel across multiple positions (Kusudo *et al.*, 2015).

2.5.2 Hash match on GPU

Large-scale data analytics applications routinely maintain logs and streams of textual events that must be matched against substantial pattern dictionaries. Such workloads are common in fraud detection, security

monitoring, and business intelligence, where timely detection of patterns in semi-structured logs can influence operational decisions (Fechner, 2010; Alba, 2013). To accelerate this process, several studies investigate hash-based string matching on GPUs, taking advantage of the many-core architecture and high memory bandwidth to process large volumes of text in parallel (Alcantara *et al.*, 2009; Vidal *et al.*, 2015).

In these designs, the text is partitioned into chunks that are distributed among GPU threads. Each thread computes hash values for candidate substrings and probes hash tables that represent the pattern set. The challenge is to structure hash tables and access patterns so that memory accesses are coalesced and collision resolution does not introduce excessive divergence. Some work explores hierarchical hashing and compact signatures to balance space usage and lookup latency, while others tune hash table layouts to better align with warp-level execution (Alcantara *et al.*, 2009; Alba, 2013). Across these studies, GPUs consistently show strong potential for hash-based matching, especially when pattern sets are large and text streams are long.

2.5.3 Histogram optimisation with CUDA

Histogram computation is another core primitive in text and log analytics. Histograms summarise frequency distributions of tokens, characters, or other features, and they appear in tasks such as keyword statistics, anomaly detection, and pre-processing for more advanced matching algorithms. Sequential histogram algorithms are straightforward, yet they become bottlenecks when applied to massive data streams (Podlozhnyuk, 2007; Fechner, 2010; Ikeda *et al.*, 2016).

CUDA-based histogram implementations distribute input elements across threads and use shared memory to accumulate partial counts before merging them into global histograms. The primary difficulties are avoiding contention on counters and ensuring that memory access patterns remain efficient as data volume grows. Prior work proposes strategies such as per-block histograms, hierarchical aggregation, and the use of atomic operations in carefully designed patterns (Milic *et al.*, 2013; Wal *et al.*, 2015). These techniques enable near-linear scaling with input size for many workloads, and they naturally integrate into larger GPU pipelines for log or text analytics.

2.5.4 Bit-parallel approximate matching and GPU streaming

Approximate Pattern Matching (APM) algorithms such as Wu–Manber can be adapted to GPUs by exploiting the bit-parallel representation of automaton states. In this approach, pattern states are encoded as bit vectors, and transitions are implemented using bitwise operations that update many state positions simultaneously (Cantone and Faro, 2009). GPU implementations of Wu–Manber break the text into overlapping chunks and assign them to threads or thread blocks, ensuring that potential matches crossing chunk boundaries are handled correctly (Zha and Sahni, 2013). This works by parsing every single character in text T via a matching Non-deterministic Finite Automaton (NFA) constructed from the pattern P. The algorithm for Wu-Manber APM is shown in Pseudo Algorithm 2.1. Note that the first column of the NFA must always be active, which is why the operation $0_{m-1}1$ is conducted in line 4 and 6 in Pseudo Algorithm 2.1. Based on these notations, this thesis formally defines the APM problem as "Given the desired pattern P, the database of T and allowed error k, find all sub-strings in T with edit distances ≤ k". Since the majority of

computations come from bitwise operation (line 6), this thesis motivates to exploit the GPU architecture to accelerate this process. However, Wu-Manber algorithm also involves a lot of memory operations (read, write and swap), which requires special treatment in GPU implementation to achieve good performance.

Pseudo Algorithm 2.1: Wu-Manber Algorithm

Input:
Text: $\boldsymbol{T} = \{\boldsymbol{t_0} \dots \boldsymbol{t_n}\}, \boldsymbol{k}$ (a text of length n characters)
Pattern: $\boldsymbol{P} = \{\boldsymbol{p_0} \dots \boldsymbol{p_n}\}$ (a pattern of length m characters)
Number of Allowed Errors: $\boldsymbol{k}$ (maximum number of allowed errors)

Output:
$\boldsymbol{Positions\ of\ all\ matches\ of\ P\ in\ T\ with\ ED \leq k}$

1. **Precompute** $\boldsymbol{B}$
2. **Initialization:** $\boldsymbol{R_j^{[0]} \leftarrow 0^{m-j}1^j\ where\ 0 \leq j \leq k}$
3. **for** $\boldsymbol{i}$ **from 1 to** $\boldsymbol{n}$
4. $\boldsymbol{R_0^{[i]} \leftarrow \left(\left(R_0^{[i-1]} \ll 1\right) \mid 0^{m-1}1\right) \,\&\, B[t_i]}$
5. **for** $\boldsymbol{j}$ **from 1 to** $\boldsymbol{k}$
6. $R_j^{[i]} \leftarrow \left(\left(R_j^{[i-1]} \ll 1\right) \& B[t_i]\right) \left| R_{j-1}^{[i-1]} \right| R_{j-1}^{[i-1]} \ll 1 \left| R_{j-1}^{[i]} \ll 1 \right| 0^{m-1}1$
7. **end for**
8. **if** $(\boldsymbol{R_k^{[i]} \,\&\, 10^{m-1}}) != \boldsymbol{0}$
9. Return the positions of all matches
10. **end if**
11. **end for**

Early GPU-based string matching demonstrations reported substantial speedups by mapping automaton transitions and hash table lookups onto CUDA kernels (Schatz, 2007; Torres *et al.*, 2012). Later work refines these ideas for Kepler-class GPUs and other accelerators, carefully tuning block sizes, register

usage, and shared memory allocation to reduce divergence and maximise throughput (Zha and Sahni, 2013; Zheng *et al.*, 2015; Hung *et al.*, 2014).

More recent studies explore fully streaming architectures in which GPU kernels filter, classify, and match text as it arrives, integrating pattern matching with other analytics such as histogramming and feature extraction (Calarasanu *et al.*, 2015; Vidal *et al.*, 2015). In these pipelines, GPUs act as high-throughput filters that operate on large batches of input, handing off aggregated or matched results to downstream components. This style of architecture is particularly appealing in settings such as real-time monitoring and deep packet inspection, where latency and throughput constraints are tight (Vasiliadis *et al.*, 2008; Zha and Sahni, 2013; Hung *et al.*, 2014).

2.5.5 Implications of GPU Compression Techniques for Column-Based Analytical Data Stores

The literature on GPU-accelerated string matching demonstrates that many classical algorithms can be successfully mapped to GPU architectures, achieving impressive speedups compared to CPU baselines when data volumes are high and patterns sets are large (Aho and Corasick, 1975; Wu and Manber, 1992; Alcantara *et al.*, 2009; Zha and Sahni, 2013). However, performance is sensitive to details such as chunking strategy, handling of approximate matches, and the interaction between hashing, automata, and memory layout.

From the perspective of this thesis, string matching serves as an important example of an irregular yet highly parallel workload that can benefit from the same GPU acceleration principles used in structured query processing. By examining how hash tables, histograms, and bit-parallel automata are implemented on GPUs, this thesis can extract design patterns for data

partitioning, kernel design, and memory management that also inform the design of GPU-accelerated relational operators.

2.6 Summary of Literature and Research Gaps in GPU-Accelerated Structured Query Processing

This chapter has reviewed research on heterogeneous computing for structured query processing, with a focus on GPU based techniques. The discussion covered GPU-accelerated database architectures, compression for column based data stores, join acceleration, and GPU oriented solutions for geospatial computation and string matching. Across these areas, prior work has demonstrated that GPUs can deliver substantial speedups for selected operators and specialised workloads when data is held in memory, organised in columnar layouts, and processed using carefully tuned kernels (Zukowski *et al.*, 2006; He *et al.*, 2009; Mensmann *et al.*, 2010; Mostak, 2013).

The survey of GPU-accelerated database engines shows that several systems exploit in memory columnar storage and one operator at a time processing to benefit from GPU parallelism. They often support execution of selected operators, such as scans, aggregations, or joins, while leaving other parts of the query pipeline on the CPU (He *et al.*, 2009; Yuan *et al.*, 2013). At the same time, research on lightweight compression and SIMD oriented encodings demonstrates that schemes such as frame of reference, PFOR-Delta and delta coding can reduce memory and I/O costs while preserving fast decompression, both on CPUs and GPUs (Lemire and Boytsov, 2015; Kuschewski *et al.*, 2023). This works on join processing further shows that both CPU and GPU platforms can achieve high performance when hash and sort

merge joins are adapted to cache and memory hierarchies, although reported speedups vary considerably across hardware and data distributions (Shatdal *et al.*, 1994; Boncz *et al.*, 1999; Manegold *et al.*, 2002; Kim *et al.*, 2009; Lee *et al.*, 2010; He *et al.*, 2013).

Beyond classical relational operators, geospatial and string matching workloads provide evidence that GPUs can accelerate more complex analytic tasks such as spatial joins, raster operations, heatmap generation, and multi pattern matching in streams of text (Jern *et al.*, 2008; Zha and Sahni, 2013; Hung *et al.*, 2014; Zheng *et al.*, 2015). These studies confirm that, under suitable data layouts and kernel designs, GPUs are effective for large scale analytical computation. However, they are mostly developed as specialised kernels or pipelines and are not integrated into a general purpose structured query engine.

Taken together, the literature reveals three main gaps that motivate the works in this thesis. First, most existing GPU database prototypes either target individual operators or provide partial support for SQL, and they do not offer an integrated single GPU accelerator that combines compression aware columnar storage, join processing, and string based predicates within one coherent execution framework. The interaction between compression, data partitioning, and GPU join kernels is still not fully addressed in a system that can serve as a drop in accelerator for common analytical workloads.

Second, while lightweight compression and vectorised encodings have been studied extensively, there is limited empirical evaluation of how specific schemes such as PFOR-Delta behave when tightly coupled with GPU based query processing across complete query plans rather than isolated scan

benchmarks. In particular, the combined effect of columnar compression, GPU joins, and GPU string matching on end-to-end performance is not well quantified on realistic datasets.

Third, many performance studies evaluate GPU kernels in isolation or under synthetic workloads. There is relatively little work that positions a GPU oriented query accelerator against mainstream big data platforms and CPU based database engines on domain specific use cases such as healthcare analytics and geospatial queries. As a result, the practical benefits and limitations of a single GPU accelerator, compared to widely deployed CPU centric stacks, remain insufficiently characterised.

This thesis addresses these gaps by designing and implementing a structured query accelerator that integrates compression aware columnar storage, GPU friendly join algorithms, and GPU based string matching into a single coherent execution pipeline. The accelerator is evaluated on established benchmarks and on application motivated workloads, and is compared against representative CPU based and big data systems. In this way, the thesis moves beyond the known fact that GPUs can accelerate individual operators, and instead investigates how an integrated, compression aware GPU query engine behaves as a practical component in end-to-end analytical processing.

CHAPTER 3

GPU STRUCTURE QUERY ACCELERATOR

Section 2.6 identified three main gaps in current work on GPU acceleration for structured queries. Existing systems either focus on isolated kernels or offer only partial support for SQL, so they stop short of an integrated accelerator that combines compression aware columnar storage, join processing, and string predicates within a single coherent execution framework. Lightweight compression schemes such as PFOR-Delta have been studied in detail, yet their behaviour when tightly coupled with GPU joins and string matching across full query plans remains only partially understood. In addition, many evaluations concentrate on kernel level speedups under synthetic workloads rather than comparing a single GPU accelerator with representative CPU based and big data platforms on realistic analytical datasets.

This chapter responds directly to these gaps by presenting the design and implementation of a GPU structured query accelerator that operates alongside a multi-core CPU. The accelerator is conceived as a practical component that can sit between an existing relational front-end and the underlying hardware. It integrates a compression first pre-processor, an SQL parser with a simple scheduler, and a GPU query engine built on chunked columnar storage and GPU friendly join operators. The design aims to reduce unnecessary data movement, keep frequently accessed columns resident in device memory whenever possible, and align operator execution with the GPU programming model while preserving the semantics of conventional SQL processing.

The chapter also positions the accelerator with respect to the research questions stated in Chapter 1. It shows how design choices such as the use of PFOR-Delta compression, the organisation of data into GPU sized chunks, and the mapping of joins and string predicates to CUDA kernels are intended to answer the central questions about when and how a single GPU can provide consistent benefits over tuned CPU engines and big data systems. The implementation is instrumented so that both internal behaviour and end-to-end query performance can be measured on benchmark workloads and on domain specific datasets.

The remainder of the chapter is organised as follows. Section 3.1 describes the overall system architecture and the interaction between the CPU host and the GPU device. Section 3.2 explains the compression first pre-processing path, including the columnar chunking strategy and the use of PFOR-Delta. Section 3.3 presents the design of the GPU join operators and discusses how they address data skew and memory access patterns, and introduces the handling of string predicates that remain on the device. Section 3.4 summarises the main findings from these experiments and prepares the ground for the broader big data and geospatial analysis in Chapter 4.

3.1 Architecture Design of GPU Structure Query Accelerator

In the realm of database architecture design, the integration of Graphics Processing Units (GPUs) has emerged as a transformative approach to enhance data processing capabilities. Leveraging the parallel computing power and high memory bandwidth of GPUs, database systems are evolving to harness these hardware accelerators for improved performance and efficiency. This technical

perspective delves into the intricacies of GPU-accelerated database architecture design, exploring the key components, optimizations, and challenges involved in leveraging GPUs for data-intensive tasks. By examining the intersection of traditional database management systems with GPU technology, this chapter aims to provide insights into the innovative design approaches shaping the future of database architectures.

3.1.1 Research Questions to Architecture Design

The detailed research questions for this thesis were defined in Section 1.2.2. They ask, in summary, how a compression first, GPU aware pre-processing path using PFOR-Delta and parallel chunking affects device residency, host–device transfers and end-to-end query time; how redesigned hash join variants that exploit shared memory and coalesced access compare with a tuned multi-core CPU across TPC-H scale factors; how a simple decision model for placing operators across CPU and GPU can reduce PCIe traffic and IO stalls; and how GPU based methods for short pattern matching behave in terms of performance and their impact on overall SQL query latency. These questions are not repeated here to introduce a new list. Instead, this section explains how they shape the architecture of the GPU structured query accelerator that is developed in the remainder of this chapter.

For the first group of questions, concerned with compression first pre-processing and chunking, the architectural implication is that the accelerator must treat data layout as a first class design choice. The pre-processor and data partitioning stages therefore convert input from existing RDBMS and flat files into a columnar format that is friendly to GPU access patterns, and they divide

each column into chunks that fit device memory while allowing overlapping copy and compute through pinned buffers and CUDA streams. These decisions are realised in the pre-processor and its chunk sizing strategy, described later in Sections 3.2.1 and 3.2.2, and they are evaluated with respect to device residency, number of transfers and query time on NVIDIA Tesla GPUs.

The second group of questions focuses on join processing. They ask whether hash join variants that use shared memory for build and probe phases, maintain coalesced access and handle skew can outperform a tuned multi-core CPU baseline, and how cache friendly layouts influence global load efficiency and join throughput. In architectural terms, this leads to a query engine that places joins at the centre of the GPU execution path. The join operators are designed to work directly on the compressed, chunked columns produced by the pre-processor, and they are written to exploit shared memory and warp level parallelism on Kepler class devices. These choices are developed in Section 3.3 and examined empirically against a CPU baseline using TPC-H style workloads and increasing data sizes.

The third group of questions addresses operator placement across CPU and GPU, and the broader behaviour of the pipeline across different datasets. They ask whether a simple decision model, driven by data shape and transfer cost, can reduce PCIe traffic and IO stalls compared with ad hoc offloading, under what measurable conditions the GPU should be preferred over the CPU, and how GPU based methods for short pattern matching interact with end-to-end SQL query time. These considerations are reflected in the SQL parser with its scheduler, which is responsible for translating queries into an internal plan

and assigning operators either to the CPU or to GPU kernels. The parser and scheduler form the control layer of the architecture described in Section 3.1.2, and they provide the hooks needed to study placement decisions in the evaluation presented in Sections 3.1.3 and 3.3.2, as well as in the application focused studies in Chapters 4 and 5.

For clarity, Table 3.1 summarises the relationship between the research questions from Section 1.2.2, the architectural components introduced in this chapter and the later evaluation chapters. The first block of questions on compression and chunking maps to the pre-processor and data partitioning design in Section 3.2 and to the TPC-H experiments on GPU aware layouts. The second block of questions on join performance maps to the GPU hash and sort merge joins in Section 3.3 and their comparison with a multi-core CPU baseline. The third block of questions on operator placement, workload behaviour and short pattern matching maps to the SQL parser and scheduler in Section 3.1.2.

Table 3.1: Research Questions and Architectural Implications

Research question group (from Section 1.2.2)	Architectural component(s) in this thesis	Where implemented and evaluated
Compression first pre-processing, chunking and device residency	Pre-processor and GPU aware data partitioning. Input from RDBMS and files is converted to a column layout and split into GPU sized chunks with pinned buffers and CUDA streams to overlap transfer and compute.	Design in Sections 3.2.1 and 3.2.2. Behaviour of chunk size, transfer count and query time evaluated on Tesla GPUs with TPC H style workloads in Section 3.2.2.
GPU hash joins, coalesced access	GPU join operators in the query engine. Hash and sort	Architectural role in Section 3.3.1. Performance

and join throughput	merge joins exploit shared memory, coalesced global access and skew handling, and operate directly on compressed chunks from the Pre Processor.	compared with a tuned multi-core CPU across TPC-H selectivity and scale factors in Section 3.3.2, with further analysis in Section 3.3.3.
Operator placement across CPU and GPU, and behaviour on realistic workloads	SQL parser with scheduler and system level evaluations. The parser builds an internal plan and the scheduler assigns operators to CPU or GPU based on data shape and transfer cost, so that simple placement rules can be studied.	Control layer outlined in Section 3.1.2. Placement and behaviour examined in the query level evaluation in Section 3.1.3, and extended to healthcare and geospatial workloads in Chapter 4 and to string matching workloads in Chapter 5.
Short pattern string matching and impact on SQL latency	GPU based string matching kernels and asynchronous scheduling. Hash based, histogram based and bit parallel approximate matching for short patterns, including a multi stream Wu Manber design that overlaps kernel work and memory copies.	Algorithm design in Chapter 5 (Sections 5.1 to 5.3). Performance and match rate sensitivity evaluated on several GPU platforms in Section 5.3, with implications for SQL style workloads summarised in Section 5.4.

3.1.2 Implementation of GPU Query Accelerator

This thesis illustrates GPU structure query accelerator architecture, including details on Pre-Processor, SQL Parser with Scheduler, and Query Engine as shown in Figure 3.1. In order to provide a solid foundation for GPU parallel processing, it is crucial to give priority to optimising particular workloads. This entails the utilisation of specialised data pre-processing approaches that are specifically designed to meet the distinct demands of managing and manipulating data on GPUs. Consequently, the objective to obtain the optimization queries for heterogeneous computing without knowing

the details of database algorithms and processors. Thus, this thesis develops a query optimizer. It should be able to scale effectively with the growing number of heterogeneous processors found in current and future machines, while also minimizing the maintenance effort required to support such scaling. This was designed as a relational GPU-accelerated OLAP engine, primarily for data warehousing workloads which benefit most from GPU acceleration.

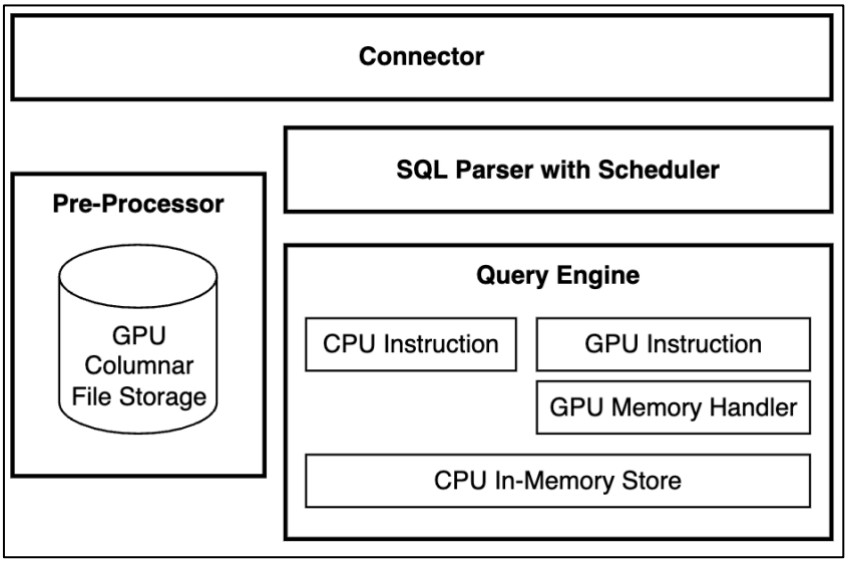


Figure 3.1: Architecture of GPU Query Accelerator

Pre-Processor

The first step involves extracting data from its original source, which could be an existing relational database management system (RDBMS) or flat files. This extraction process might involve filtering or cleaning the data to ensure its suitability for further processing. One of the key strengths of this approach lies in transforming the data into a columnar format. Traditional row-oriented storage, where data is organized by rows with all columns for each record stored contiguously, becomes inefficient when analyzing large datasets. Columnar

storage, on the other hand, groups all values for a specific column together. This format aligns perfectly with how GPUs operate, as they excel at processing data in parallel across multiple cores. Each GPU core can efficiently access and process a specific column simultaneously, significantly accelerating computations. Once the data resides in a GPU-optimized format, queries are translated and executed directly on the GPU. This translation involves converting traditional SQL queries into instructions specifically designed for the GPU's processing model. The GPU then leverages its high memory bandwidth and parallel processing capabilities to execute these queries at significantly faster speeds compared to traditional CPU-bound database systems. Column-oriented processing techniques further enhance performance by allowing the GPU to focus on specific columns relevant to the query, minimizing unnecessary data processing. The current accelerator supports only three data types: 32-Bit integers, 32-Bit floats, and variable-length strings, which allowed for the quick development of a working prototype. Nevertheless, these data types can support the popular Transaction Processing Council Ad-hoc/decision support benchmark, TPC-H (TPC.org, 2016). This method places emphasis on optimising performance for particular workloads through the utilisation of GPUs' capabilities in parallel processing and high memory bandwidth. Nevertheless, it necessitates further procedures for data pre-processing and potentially presents distinct storage management difficulties. On the other hand, source data from database need to be transformed, and output to a parallel accessible storage system. These components are designed to run on an energy efficient commodity of GPU accelerator. In addition, it powers to strive forward for handling big data challenges.

Connector

This is an elemental component that enable communication between the backend of the accelerator and the clients. These frontend application consist of client APIs application and relational database system. It provides way for raw data extraction to pre-process. Also, it interacts the client structure queries and return materialized result set. Pre-Processor stores data in files format. It is stored in column-based orientation which is designed for optimizing parallel processing by GPU. Input data required to be transferred via database connector to the pre-processor module to formulate the file systems. It partitions a column into multiple chunks of files. Also, there are optimized mechanisms to handle different data type's processes. It also compresses the data into smaller size, which reduces the I/O operation and offloads the task to GPU.

SQL Parser with Scheduler

SQL parser has a few modules involved to process user query before executed in GPU. The query needs to break into clause of objects. . It delves deeper than simply validating syntax and semantics. Instead, it analyses the incoming SQL statement to understand the user's intent and identify opportunities for leveraging the GPU's parallel processing power. It composes the clauses in GPU parameters mapping APIs (Application Programming Interface) table. The API tables are a set of functions to perform GPU operations, such as scan, sum, min, max and others. After forming the table, there is an analysis on reshuffling and rearranging the order of mapping APIs in the table. the parser might leverage domain-specific knowledge about the chosen GPU architecture to rewrite queries or identify potential bottlenecks It transforms to

a set of parallel instructions, which are ready to be executed in GPU. However, the system does not support all kind of queries, so it assigns a flag to indicate none GPU executable instruction sets. These are the important steps to strategically execute query in GPU. In order to provide efficient data processing capabilities, it makes use of various fundamental libraries. Flex and Bison (Levine, 2009) uses to handle the input of structure queries.

The data obtained from the parser is of utmost importance in facilitating GPU processing. The system exhibits a close interaction with the scheduler, offering comprehensive analysis of the query structure and the specific regions that have been designated for GPU acceleration. With this analysis at its disposal, the scheduler can subsequently generate an optimised execution plan that effectively leverages the resources of the GPU. It manages the query execution queue either in CPU or GPU. Instruction sets call the GPU executable APIs to perform the parallel operations. During the translation process, the parser may engage in collaboration with the query processing engine, providing supplementary direction for the conversion of parsed query components into instructions that are specifically tailored for the GPU. The SQL parser within a GPU database architecture surpasses a rudimentary comprehension of user intent. The task at hand involves actively harnessing the computational capabilities of the GPU through the identification of appropriate operations and the generation of optimised representations that effectively leverage the GPU's inherent strengths.

Query Engine

This carries out various processes of query investigation by performing the basic parsing and positioning operations. It targets to break down the input user queries to further clause of objects and collects the decomposed SQL objects to determine either CPU or GPU executable instructions to be used. Then, it produces an execution query plan. There is further adjustment of the plan by analysing and tracing parallelizable points and rearranges clause of objects execution. It execution engine performs the accelerated query execution on either CPU or GPU. Those flagged with non GPU executable instruction sets, it then diverted to a standard conventional query engine via the database connector to the CPU execution. Thrust (George *et al.*, 2017) and ModernGPU (Baxter, 2017) are the essential parallel processing libraries for GPU accelerator. This thesis enhances the heterogeneous harmonisation of CPU and GPU parallel processing with adopting Boost C++ (Dawes and Abrahams, 2012) with CUDA (NVIDIA, 2017) development toolkits. This expects the heterogeneous used of libraries to deliver the best performance.

CPU Instruction

This module is likely responsible for receiving user queries, potentially in the form of SQL statements. While the initial parsing and validation of the query might occur on the CPU, this module can play a role in identifying opportunities for GPU acceleration. It could analyse the query structure and flag specific operations (e.g., aggregations, joins) that are well-suited for parallel processing on the GPU. The CPU instruction module might leverage cost-based analysis techniques to compare the efficiency of executing certain parts of the query on

the CPU versus the GPU. Additionally, it interacts with the SQL parser (discussed earlier) to gain insights into the query structure and potential GPU suitability.

GPU Instruction

This module likely deals with translating the parsed query or the identified operations for GPU execution. This is where the core GPU acceleration happens. The GPU instruction module takes the information from the CPU instruction module and translates it into instructions specifically designed for the GPU's processing units. This translation process involve converting the parsed query elements into a format understandable by the GPU's cores (e.g., kernels); Leveraging specialized libraries or APIs optimized for GPU database operations; Offloading appropriate query components to the GPU for parallel processing.

CPU In-Memory Store

This module likely manages data storage and retrieval on the CPU's main memory. The CPU in-memory store acts as a staging area for data before it's transferred to the GPU for processing. The task involves the management of data buffers and the optimisation of data flow between CPU and GPU memory. In order to enhance processing efficiency, it may be essential to conduct pre-processing or filtering on the data prior to its transmission to the GPU.

GPU Memory Handler

This module manages data storage and retrieval on the GPU's dedicated memory. The integration of GPU acceleration is a critical component in the optimisation of data access patterns for the GPU. The GPU memory handler is responsible for the allocation and management of memory on the GPU,

including the storage of data that has to be processed. Utilising methodologies in data chunking with specialised data structures can enhance the efficiency of data access for the cores of the GPU. Optimising the transfer of data between various tiers of the GPU memory hierarchy in global memory and shared memory reduces latency. By cohesively processing, these modules establish a unified system that effectively utilises CPU and GPU resources. The initial query processing and data preparation are handled by the CPU instruction and in-memory store modules, whilst the GPU instruction and memory handler modules are responsible for translating queries and managing data on the GPU to facilitate parallel processing. The utilisation of a collaborative approach enables the utilisation of GPUs for database workloads, resulting in notable enhancements in performance.

3.1.3 Evaluation of GPU Query Accelerator on TPC-H Workloads

The efficiency and effectiveness of GPU Structure Query Accelerator is evaluated through the experimentation of TPC-H queries. This experiment has been conducted on a traditional relational database management system, which is PostgreSQL; and GPU Structure Query Accelerator, a system designed to leverage GPUs for database workloads. This focuses on queries involving computationally intensive operations, including aggregations, table joins, and string comparisons against the performance of multi-core CPU and many-core GPU.

Experimental Hardware and Software Setup

This thesis experimental setup involves the use of two distinct GPU models from the NVIDIA Tesla Kepler series: the Kepler K20c and the Kepler

K40. The Kepler K20c is equipped with 2496 CUDA cores and 6GB of GDDR5 RAM, while the Kepler K40 boasts 2880 CUDA cores and 12GB of GDDR5 RAM. This thesis uses these GPUs for their computational power and memory capacity, which are crucial for the data partitioning strategy.

To facilitate this experiments, each GPU was installed in a separate HP Z800 workstation, ensuring that each workstation had the same hardware configuration. The workstations were equipped with Dual Sockets motherboard in 2 x Intel Xeon X5680 processors, which are with total of 12 CPU cores, run at 3.33GHz, 32GB of DDR3 RAM with 1333MHz, 1TB hard disk with 5400 RPM rate, an ATI FireMV 2260 for display card, and 850W power supply. Table 3.2 shows the comparison of the processing units of CPU and GPU models. This configuration provides a stable and consistent environment for conducting the experiments.

Table 3.2: Specification of Xeon and Kepler

Features	Intel Xeon X5680	NVIDIA K20	NVIDIA K40
Architecture	Westmere	Kepler	Kepler
Manufacturing Process	32nm	28nm	28nm
Thermal Design Power	130W	225W	235W
Memory	12MB L2 Cache	5GB GDDR5	12GB GDDR5
CPU Cores	6 Cores, 12 Threads	2496 CUDA Cores	2880 CUDA Cores
Speed	3.33GHz	706MHz Base, 823MHz Boost	745MHz Base, 810MHz Boost

For software, both workstations were configured with Microsoft Windows 7 (64-bit) as the operating system and PostgreSQL 9.2 as the database system.

The choice of Windows 7 was based on its widespread use in research environments and its compatibility with the CUDA toolkit. Additionally, the K20c workstation was set up with CUDA 5.5 (Production Release), while the K40 workstation was set up with CUDA 6 (Release Candidate). The selected CUDA versions exploit recent features and optimisations to improve GPU-accelerated computations.

Dataset

This has been tested on TPC-H's data sources. TPC-H implements an ad-hoc decision support benchmark. The ad-hoc nature of the benchmark is intended to mimic a real-life use case for businesses. The database administrators (DBAs) do not know which queries are executed against the database system. It poses a more challenging workload that is more customer-relevant and has garnered the support of the industry for database engines. This thesis uses the TPC-H data generator application to create three different sizes of datasets namely 1 GB, 10 GB, and 50 GB.

Query workload employs three queries from TPC-H to evaluate various aspects. These three queries focus on different aspects of business analysis using SQL. The first query examines shipping and return trends up to a specific date, providing insights into operational efficiency and customer satisfaction. The second query delves into customer purchasing behaviour, segmenting data by market segment, order date, and ship priority to understand customer preferences and trends over time. Lastly, the third query analyses revenue, quantity, and order count for a specific market segment, offering valuable

insights into the performance and characteristics of customers within that segment.

Query 1 provides a detailed analysis of shipping and return trends in the dataset up to 2 September 1998, focusing on quantities, prices and order counts. It groups records by return flag and line item status to summarise quantities, extended prices, discounts and revenues. These aggregates reveal operational efficiency, return behaviour and revenue impact, and also form a representative workload for comparing the evaluated query processing platforms. It calculates various aggregate values from the 'lineitem' table to analyse shipping and return trends within the dataset. The sum function uses to calculate the total quantity 'sum_qty' and total extended price 'sum_base_price' of items. Additionally, the 'avg' function computes the average quantity 'avg_qty' and average extended price 'avg_price' per item. The min and max functions determine the minimum 'min_qty', 'min_price' and maximum 'max_qty', 'max_price' quantities and prices, respectively. Finally, the count function calculates the total number of orders 'count_order'. The WHERE clause filters the data to include only records with a 'l_shipdate' on or before September 2, 1998. The results group by 'l_returnflag' and 'l_linestatus' to provide insights into different return and line item statuses. The ORDER BY clause organizes the results by 'l_returnflag' and 'l_linestatus' for better readability. In short, Figure 3.2 summarises quantities, prices and order counts by return flag and line item status. This breakdown clarifies shipping performance and return patterns across the dataset, highlights their impact on overall order behaviour, and provides a concrete reference for interpreting subsequent performance results across the evaluated platforms.

```
SELECT      l_returnflag, l_linestatus,
            sum(l_quantity) AS sum_qty,
            sum(l_extendedprice) AS sum_base_price,
            avg(l_quantity) AS avg_qty,
            avg(l_extendedprice) as avg_price,
            min(l_quantity) AS min_qty,
            max(l_quantity) AS maxqty,
            min(l_extendedprice) AS min_price,
            max(l_extendedprice) AS max_price,
            count(l_quantity) AS count_order
FROM        lineitem
WHERE       l_shipdate <= 19980902
GROUP BY    l_returnflag, l_linestatus
ORDER BY    l_returnflag, l_linestatus
```

Figure 3.2: Query 1 - Shipping and Return Trends Analysis

Query 2 is designed to analyse the revenue, quantity, and order statistics of customers segmented by market segment, order date, and ship priority. By aggregating data from the 'customer', 'orders', and 'lineitem' tables, valuable insights can be gained into the purchasing behaviour and trends of different market segments up to March 15, 1995. It selects the 'c_mktsegment' field from the customer table, the 'o_orderdate' and 'o_shippriority' fields from the 'orders' table, and calculates aggregate values from the 'lineitem' table, including the total revenue, total quantity, average quantity per order 'avg_qty', average price per order 'avg_price', and total number of orders 'total_order'. In addition, it joins the 'customer', 'orders', and 'lineitem' tables using the 'c_custkey' and 'o_custkey' fields to link customers to their orders, and the 'l_orderkey' and 'o_orderkey' fields to link orders to line items. The WHERE clause filters the data to include only orders placed before March 15, 1995. The results are then grouped by 'c_mktsegment', 'o_orderdate', and 'o_shippriority' to provide insights into the purchasing behaviour of different market segments over time. This analysis can help identify trends, such as which market segments are more active. In short, Figure 3.3 provides a detailed analysis of revenue, quantity, and

order statistics grouped by market segment, order date, and ship priority. This analysis offers valuable insights into customer behaviour and purchasing patterns.

```
SELECT      customer.c_mktsegment,
            orders.o_orderdate, orders.o_shippriority,
            sum(lineitem.l_extendedprice) AS revenue,
            sum(lineitem.l_quantity) AS quantity,
            avg(lineitem.l_quantity) AS avg_qty,
            avg(lineitem.l_extendedprice) AS avg_price,
            count(orders.o_orderkey) AS total_order
FROM        customer, orders, lineitem
WHERE       customer.c_custkey = orders.o_custkey  AND
            lineitem.l_orderkey = orders.o_orderkey  AND
            orders.o_orderdate < 19950315
GROUP BY    customer.c_mktsegment,
            orders.o_orderdate,
            orders.o_shippriority
```

Figure 3.3: Query 2 - Customer Purchasing Behaviour by Market Segment

Query 3 calculates the total revenue, total quantity, and number of orders for customers in the 'BUILDING' market segment. By joining data from the 'customer', 'orders', and 'lineitem' tables, this analysis provides insights into the purchasing behaviour of customers in this specific market segment. It selects the 'l_extendedprice' and 'l_quantity' fields from the 'lineitem' table and the 'o_orderkey' field from the orders table, then calculates the sum of 'l_extendedprice' as revenue, the sum of 'l_quantity' as quantity, and the count of 'o_orderkey' as 'cnt'. Also, it joins the 'customer', 'orders', and 'lineitem' tables using the 'c_custkey' and 'o_custkey' fields to link customers to their 'orders', and the 'l_orderkey' and 'o_orderkey' fields to link orders to line items. The WHERE clause filters the data to include only customers in the 'BUILDING' market segment. In short, Figure 3.4 provides a comprehensive overview of the revenue, quantity, and order count for customers in the 'BUILDING' market segment. By

aggregating data from multiple tables, this analysis provides valuable insights into the purchasing patterns and trends of customers within this segment.

```
SELECT      sum(lineitem.l_extendedprice) AS revenue,
            sum(lineitem.l_quantity) AS quantity,
            count(orders.o_orderkey) AS cnt
FROM        customer, orders, lineitem
WHERE       customer.c_custkey = orders.o_custkey AND
            lineitem.l_orderkey = orders.o_orderkey AND
            customer.c_mktsegment = 'BUILDING'
```

Figure 3.4: Analysis of 'BUILDING' Market Segment

Results

The results include the overall timing from issuing queries, executing queries in the GPUs, processing output results, and displaying final results on the screen. Figure 3.5 shows the results from three sets of experiments. A logarithmic scale is used for the x-axis to represent execution times of the CPU and GPUs results as the ratio between is the results are extremely large. All running time value is expressed in seconds.

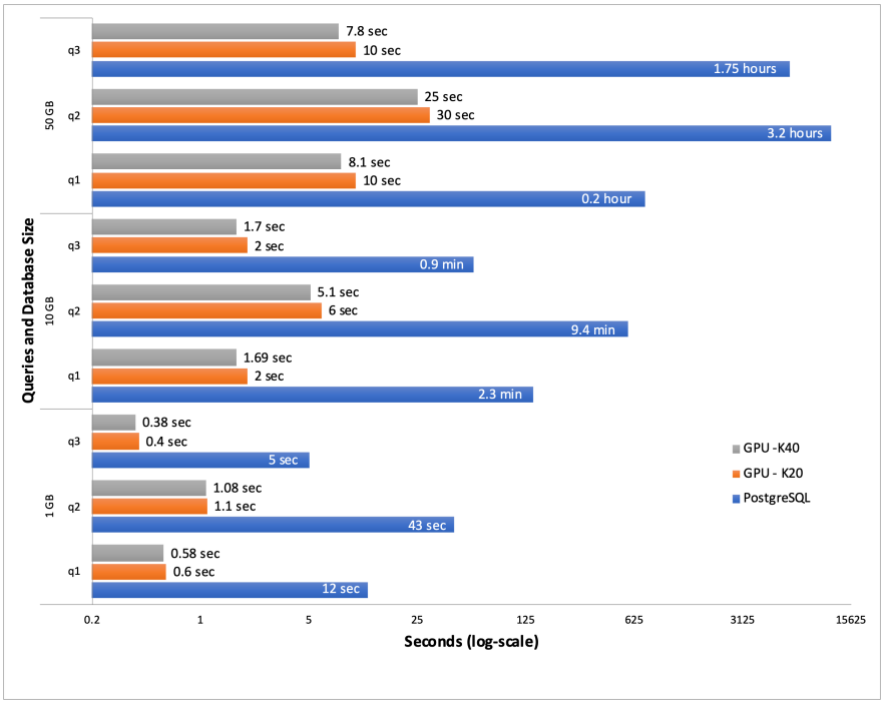


Figure 3.5: Three Queries Execution

For PostgreSQL, as the data size increases from 1 GB to 50 GB, there's a substantial increase in execution time for all queries, indicating higher processing times due to larger volumes of data. For instance, query 1 takes 12 seconds for a 1 GB dataset but increases to 734 seconds for a 50 GB dataset. Furthermore, the execution time for Query 2 is 3.2 hours, while Query 3 takes 1.75 hours. This shows that PostgreSQL's performance is significantly affected by increases in data size.

The GPU K20 and K40 run all queries and data sizes much faster than PostgreSQL, confirming the speed advantage of GPUs for query processing. The K40 usually outperforms the K20, but only by a modest margin, which suggests diminishing performance gains with newer GPUs. For Query 1 on the 50 GB dataset, the K20 completes in 10 seconds and the K40 in 8.1 seconds. Across all systems, execution time rises with data size as expected, but PostgreSQL shows a much steeper increase than the GPU based configurations. This pattern indicates that the GPU accelerators scale more effectively than PostgreSQL as workload size grows.

Table 3.3: Query Complexity Comparison

Query	Total of Aggregation	Group By	Sort	Search	Remarks
1	8	2	2	1	Complexity, suitable for parallelisation
2	5	3	0	1	Moderate complexity, suitable for parallelisation
3	3	0	0	1	Low complexity, suitable for parallelisation with minimal overhead

The implications for both PostgreSQL and GPU query accelerator can be further derived by combining the comparison of query complexity in Table 3.3 and the performance of query execution in Figure 3.5. Query 1 is the most complex query consisting of highest aggregated functions call with 2 join of sub dataset via "Group By", also two sorting operation. The speed-up is particularly pronounced for larger size of data, with K40 consistently up to 91x in 50GB, and outperforming K20 from 21x in 10GB. GPU is increasingly being performed for data parallelization, even for queries with moderate and low complexity. The choice between GPU models (K20 vs. K40) depends on the bigger size of data workload requirements, with GPU-K40 generally providing higher performance gains but at a potentially higher cost.

This thesis seeks to comprehend the progression during the execution of these three queries on a GPU. Profiling results are gathered through the utilisation of the NVIDIA NSight profiling tool. It enables users to gather data pertaining to CUDA-related operations, encompassing events and metrics associated with CUDA kernels. This thesis generally utilises the event/metric summary mode and default summary mode, while the profiler also supports other modes, as shown in Table 3.4. "Sum of Occupancy Values" is added up all the individual occupancy values (percentages) obtained at different points in time during the profiling session; "Number of Samples" is the total number of occupancy measurements taken during the profiling session. Conversely, the average CPU usage during execution time session is shown in Table 3.5. "Total of CPU Time" is the total amount of time that the CPU spends executing instructions for a particular task or process.

Table 3.4: Averaging of GPU Occupancy in %

```
Average GPU Occupancy (%) = (Sum of Occupancy Values) /
                            (Number of Samples) * 100
```

Table 3.5: Averaging of CPU Usage in %

```
Average CPU Usage (%) = (Total CPU Time / (Total Elapsed Time
                        * Number of Cores)) * 100
```

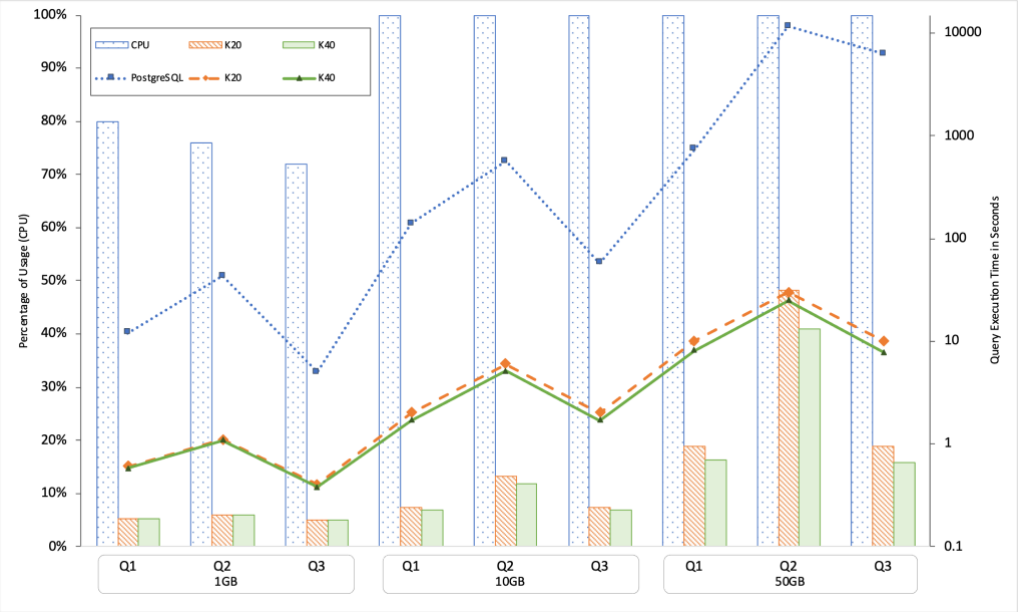


Figure 3.6: Resources Usage for CPU & GPU

As in Figure 3.6, it contrasts CPU and GPU resource use with query runtimes for TPC-H Q1 to Q3 on data sizes from 1 GB to 50 GB. On PostgreSQL, CPU utilisation is already high at 1 GB, around 72% to 80% across the three queries, and reaches 100% for all queries at 10 GB and 50 GB. This shows that the CPU only engine is quickly compute bound and that growing the data size mainly stretches execution time. At 50 GB, Q2 requires about 11 529 seconds on PostgreSQL, while Q1 and Q3 also run into the range of many minutes to more than an hour. In contrast, when using the GPU based engine, the host CPU

remains lightly loaded because the heavy operators run on the device. GPU utilisation on K20 and K40 is relatively modest, typically below 15% for 1 GB and 10 GB, and rising only to around 48% for K20 and about 41% for K40 at 50 GB. Even with this partial utilisation, runtimes drop sharply. For example, at 50 GB Q2 falls from 11529 seconds on PostgreSQL to 30 seconds on K20 and 25 seconds on K40, which corresponds to speedups of roughly 384 times and 461 times respectively. Across all queries and sizes, K40 is consistently faster than K20, but usually by margins in the range of 15% to 25% rather than orders of magnitude. This pattern indicates that both GPUs have ample headroom and that the limiting factors lie in PCIe transfers and host side processing, not in raw GPU throughput.

Table 3.6: Impact of PCIe communication on GPU for Query 2 (50 GB)

System	Base query time (s)	PCIe rounds (H→D & D→H)	PCIe transfer time (s)	Total time incl. PCIe (s)	Effective speedup vs PostgreSQL in (x)
PostgreSQL CPU	11529	0	0	11529	1
K20 GPU	10	86	537.5	547.5	21
K40 GPU	8.1	72	450	458.1	25

The extended results for TPC-H Query 2 at 50 GB include a simple PCIe transfer model and give a more realistic picture of end-to-end performance from the CPU perspective, as shown in Table 3.6. Query 2 at 50 GB is used as a representative case because it is the most demanding among Q1–Q3 in the earlier experiments, with the longest PostgreSQL runtime and the largest data volume flowing through the pipeline. It therefore stresses both computation and data movement. It is a suitable scenario for examining how PCIe communication

affects the apparent GPU speedup. PostgreSQL still forms the baseline with an execution time of 11 529 seconds. The GPU only kernel times are 10 seconds on K20 and 8.1 seconds on K40, which by themselves suggest very large acceleration. Once PCIe-2 transfers are included, using the bandwidth assumptions introduced earlier, the total times rise to 547.5 seconds for K20 and 458.1 seconds for K40. The corresponding effective speedups are 21 times and 25 times relative to PostgreSQL.

Table 3.6 shows that repeated data movement between host and device dominates the GPU execution time. For K20, 537.5 seconds out of 547.5 seconds, about 98 percent, is spent in 86 PCIe transfer rounds for host to device and device to host exchanges of intermediate result sets. The actual GPU computation accounts for less than 2% of the total. K40 exhibits a similar pattern. Out of 458.1 seconds, 450 seconds, again about 98%, is due to 72 PCIe transfer rounds, while only 8.1 seconds is spent in kernel execution. These numbers are consistent with the earlier observation that GPU utilisation in the chart remains moderate even for large data sizes. The device finishes its kernels quickly, then waits for data to arrive or leave over a relatively slow interconnect.

The modest gap between K20 and K40 also becomes easier to interpret. On the GPU alone, K40 is about 19 percent faster than K20 for Query 2 at 50 GB. When PCIe transfers are included, the speedup improves from 21 times to 25 times relative to PostgreSQL, an increase of roughly 19 percent as well. Because both devices share the same PCIe 2 interface and follow similar transfer patterns, almost all of the 450 to 537.5 seconds of communication cost is identical. Only the small compute fraction benefits from the stronger K40

hardware. These results support the earlier conclusion that the pipeline is limited by data movement and host side processing rather than raw GPU throughput, and they reinforce the motivation for later chapters that focus on compression, GPU resident processing and reduced host–device exchanges.

3.1.4 Discussion of Performance Gains and GPU Underutilisation

This thesis investigates the utilization of GPU to accelerate structural query execution within the TPC-H, and further evaluates the performance implications of deploying a single GPU in conjunction with a CPU within a heterogeneous computing environment. The authors investigate if this design can effectively use parallel processing for performance improvements. There discusses designing a GPU query accelerator architecture. It details components like a pre-processor, SQL parser with scheduler, and query engine. The pre-processor transforms data into a columnar format suitable for GPUs. The SQL parser analyses queries to identify opportunities for GPU acceleration and translates them into instructions understandable by the GPU. The query engine manages query execution on either CPU or GPU. The effectiveness of the GPU query accelerator is evaluated using TPC-H queries on a traditional database system and the accelerator system. The experiments use two NVIDIA Tesla Kepler GPU models, K20c and K40. The results show that the GPU-accelerated systems outperform the traditional database system for all queries and data sizes. The performance gain is more significant for complex queries and larger datasets. There is extensive analysis of GPU and CPU utilization during query execution. The results show that PostgreSQL heavily utilizes the CPU, while the GPU-accelerated systems exhibit lower GPU utilization. This indicates that there is

still potential for improvement by offloading more tasks to the GPU. Overall, the chapter demonstrates that using GPUs in a heterogeneous architecture can significantly improve database query performance.

3.2 Data Partitioning for GPU Compressor

In this context, pre-processing involves several key steps, including data formatting, compression, encoding, partitioning, and saving to file. By organizing data into a columnar structure and applying compression techniques such as delta-based encoding with bit packing for numeric columns and dictionary encoding for string columns, the system stores repeated or slowly changing values more compactly. This reduces the amount of data that must be moved between CPU and GPU, cuts storage requirements, and shortens scan time for analytical queries. Encoding categorical variables and partitioning data into manageable chunks further enhance the efficiency of data loading and processing on GPUs. Optimising the loading process from columnar storage, including the use of metadata for efficient data retrieval, is also crucial. Taken together, these pre-processing steps prepare the data in a format that maximises the performance of GPU-based computations and connect directly to the PFOR-Delta and dictionary schemes described later in Section 3.2.1.

Efficient pre-processing for GPUs also entails choosing algorithms that minimise memory overhead and make effective use of the available bandwidth. In this prototype, integer and floating point attributes are compressed using PFOR-Delta, while string attributes use dictionary compression. These lightweight schemes reduce the size of column partitions so that more data can reside in device memory and keep decompression simple enough for massively

parallel execution. The compression experiments later in this chapter compare PFOR-Delta with GPU LZSS, block-sorting and Huffman coding, and show that block-sorting and Huffman achieve the highest compression ratios, whereas PFOR-Delta attains competitive ratios at the cost of longer compression and decompression times on the evaluated text datasets. This means that the choice of compression scheme influences both how much data can be stored and transferred efficiently on the GPU and how much pre-processing time is added before query execution. To keep these costs manageable, the decompression kernels are structured to match the GPU's execution model rather than using complex branch-heavy logic. During partitioning, it is also important to align data segments with the GPU warp size so that all threads in a warp progress together, and to apply a lightweight data shuffling step when needed to avoid non-coalesced memory accesses that would otherwise limit effective throughput.

Metadata storage should be optimized for the fast arithmetic capabilities of the GPU. Storing metadata in structures that can be directly interpreted by the GPU can substantially reduce the need for intermediate data interpretation and avoid unnecessary computation. When saving data to the file for GPU consumption, the use of file formats in CUDA is adopted, which can map memory directly for the GPU, thereby allowing data to be pre-loaded and directly accessed by GPU kernels, cutting down on loading times.

3.2.1 Implementation Pre-processor

An obstacle in the implementation of data partitioning for a GPU compressor is the challenge of transferring data between the CPU and GPU. This can involve chunking the data by co-processing schema structure for given

database architecture differently. A series of essential procedures are undertaken to pre-process data partitions for GPU compression as shown below:

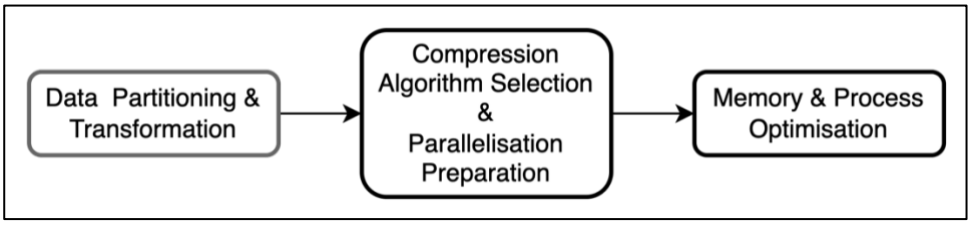


Figure 3.7: Essential Procedure of GPU Data Partition

Data Partitioning & Transformation

Once partitioned, the data undergoes preparation tailored to the data type and the chosen compression method. For integers (int), this may involve techniques like byte alignment or utilizing bit packing to compact the data if the integers do not require the full byte range. Floating-point numbers (float) may be standardized to a fixed precision or converted to a more compact representation if the specific algorithm can handle it without significant loss of information. When preparing textual data (string), common approaches include dictionary encoding, where repeated strings are replaced with shorter codes, and transforming strings into their binary ASCII or Unicode equivalents. The preparation phase can also involve other steps such as data normalization, encoding conversion, or the application of specific formatting rules to ensure compatibility with the GPU compression algorithm and data structure alignment.

In this combined stage, the focus is on organizing the data to minimize dependency and synchronization between GPU threads, facilitating compression algorithms to work on each partition efficiently. The transformation of data into a uniform type, where possible, can significantly speed up the compression process on a GPU by reducing the complexity of operations performed during

the compression. Through careful partitioning and thoughtful preparation, including consideration of three major data types for int, float, and string, the data is rendered into an optimal state for fast and effective GPU-assisted compression.

The initial pre-process stage is to optimize data access patterns that maximizing memory locality. This fixes the chunk size into total rows for each chunk files. Below illustrate the implementation of the GPU approachable chunking process:

Pseudo Algorithm 3.1: Pre-Processing of Data Partition

```
The size of the chunk of data, C_size
The size of the entire data set, D_size
The amount of memory available on the GPU, GPU_mem
The size of a thread block, G_tb_size
The number of thread blocks, G_tb_total

// Get the optimum chunk size
C_size = min(D_size, GPU_mem / (G_tb_size * G_tb_total))

// Split the Data into Chunks
X_chunk_i = {x_i, x_i+1, ... , x_i+m-1}(chunk i as C_size)

// Preprocess and Transform the Data:
Y_chunk_i = Output(X_chunk_i)
```

This formula ensures that the chunk size is always less than or equal to the amount of memory available on the GPU. This ensures that the data can be loaded into the GPU's memory and processed without any errors. The formula also ensures that the chunk size is a multiple of the thread block size. This is important because it allows the data to be divided evenly among the thread blocks. This can improve the performance of the parallel processing. **X_chunk_i** represents the **i-th** chunk of the raw data, which contains data points

starting from index **i** to **i+m-1**. **Y_chunk_i** represents the preprocessed and transformed version of the chunk **X_chunk_i** using the function **Output**.

Compression Algorithm Selection & Parallelisation Preparation

The effectiveness of a GPU compressor is largely contingent on the compression algorithm chosen. As discussed in the previous section on data partitioning for GPU compression and in the earlier background on compression techniques, this choice must reflect the data type, the desired compression ratio, and the balance between compression speed and decompression efficiency. Lossless compression algorithms are preferable in this context because they preserve data fidelity. On the GPU, two compression schemes are employed, one for integer and float attributes and one for string attributes. The first scheme, used for integer and float data, is based on the PFOR-Delta method from Zukowski's work (Zukowski *et al.*, 2006). It stores differences instead of the actual values, so only the change between subsequent values is recorded. A bit packing mechanism can then further optimise storage by using only the number of bits required to represent each element. This scheme is efficient for sequential access patterns but does not provide efficient random access, because decompressing a particular element usually requires decompressing the preceding elements. The effectiveness of PFOR-Delta is also influenced by data distribution and features, and it may compress irregular data poorly, which can limit compression improvements. The corresponding dictionary-based scheme for string data is discussed later in this section.

To reduce the sensitivity of PFOR-Delta to data distribution and to prepare it for GPU parallelisation, a pre-process for parallel chunking has been designed,

as described in the preceding section. Column partitions are divided into multiple chunks so that different regions of the data can be compressed independently. These chunks are then distributed across GPU processors in a way that aims to balance the workload and utilise all available GPU resources more evenly. This approach helps to mitigate the impact of skewed or irregular data on compression performance and makes it easier to exploit the parallel processing capabilities of the GPU during compression and decompression.

The GPU-based implementation of the PFOR-Delta technique can leverage the parallel processing capabilities of GPUs to achieve high-speed compression. Pseudo algorithm is express the compression process. The GPU execution assigns a unique identifier, tid, to each thread that is used to access the input and output data arrays. data, the array of input data, contains the integers to be compressed, while compressed_data, the array of output data, holds the compressed data. The num_elements variable stores the number of elements in the array of input data. The algorithm encodes the difference, delta, between the current element, data[i], and the previous element, data[i-1], using variable bit width. The encoded delta value is then combined with the current element using a bitwise OR operation, (delta << 16) | data[i], and stored in the array of compressed data at the same index, compressed_data[i], as shown below:

Pseudo Algorithm 3.2: PFOR-Delta Compression

```
compress_pfor_delta_gpu(data, compressed_data, num_elements):
    tid = thread_index
    for i = tid to num_elements - 2 step (blockDim * gridDim):
        delta = data[i + 1] - data[i]
        compressed_data[i] = (delta << 16) | data[i]
    if tid = num_elements - 1:
        compressed_data[tid] = data[tid]
```

Shifting the delta value by 8 bits instead of 16 bits is a valid choice depending on the dataset's requirements and characteristics. However, the decision depends on factors like the range of delta values, compression ratio, and available storage. Shifting by 8 bits provides fewer bits for representing the delta value, making it suitable for small ranges and accurate representation within 8 bits. However, shifting by 8 bits may limit the range and precision of accurately represented delta values. As the TPC-H dataset spans a larger range or requires more precision, shifting by 16 bits or more may be more appropriate.

Pseudo Algorithm 3.3: PFOR-Delta Decompression

```
decompress_pfor_delta_gpu(compressed_data, decompressed_data,
num_elements, max_string_length):

    tid = threadIdx.x + blockIdx.x * blockDim.x
    for i = tid to num_elements - 2 step (blockDim.x *
        gridDim.x):

        delta = (compressed_data[i] >> 16)
        copy(decompressed_data + i * max_string_length,
          decompressed_data + (i - 1) * max_string_length,
          max_string_length)
        for j = 0 to max_string_length - 1:
            decompressed_data[i * max_string_length + j] +=
              delta

    if tid == 0:
        copy(decompressed_data + (num_elements - 1) *
          max_string_length, compressed_data + (num_elements -
          1) * max_string_length, max_string_length)
```

The above function is designed to efficiently decompress data that has been compressed using a variant of the PFOR-Delta method for GPU processing. It uses CUDA thread indexing to process the compressed data in parallel. For each element in the compressed data array, the function extracts the delta value, which represents the difference between consecutive elements in the original data. It then copies the corresponding string from the decompressed data array

to the current position and adds the delta value to each character of the string at that position. Finally, to handle the last element of the decompressed data array, the function checks if the thread index is 0 and copies the last string from the compressed data array directly to the decompressed data array. Overall, the function efficiently reconstructs the original data from the compressed format, making it suitable for large-scale data processing tasks on GPUs.

Dictionary compression scheme was applied on string data type for much better compression mechanism. It has a dictionary of the unique values in the chunk file for decompression. As such, ultra-fast retrieval on the dictionary is the key of the efficiency. It offloads the lookup process to the GPU. However, this string compression mechanism is only benefited with a lot of duplicated words in the input column of data.

Memory & Process Optimisation

With the goal to enhance memory access efficiency during the compression and decompression procedure, it is possible to employ a modified version of the PFOR-Delta compression technique that aims to minimise memory transfers while simultaneously maximising data localization. A optimisation process has formulated as below:

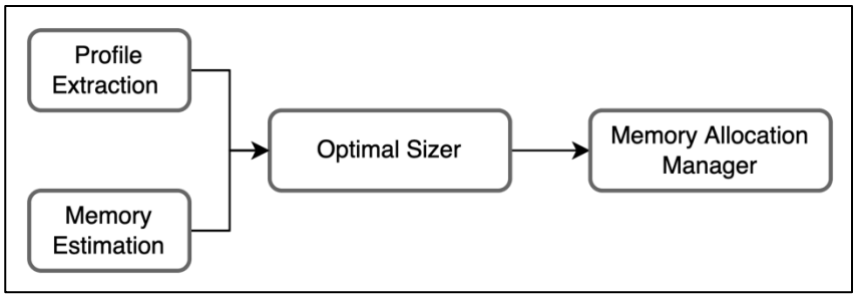


Figure 3.8: Step of Memory & Process Optimisation

The steps have briefly illustrated as below:

Profile Extraction	To obtain data pertaining to the GPU, such as the quantity of CUDA cores, memory bandwidth, and accessible memory, one can call CUDA device functions library.
Memory Estimation	Estimate the memory requirements for the PFOR-Delta algorithms based on the input data size, compression ratio, and algorithm complexity. Factor of block size, thread count, and shared memory usage.
Optimal Sizer	Determine the optimal memory size by considering the GPU's memory constraints and performance characteristics. Use the estimated memory requirements and GPU specifications to calculate the optimal memory size for both compression and decompression.
Memory Allocation Manager	Allocate memory on the GPU based on the calculated optimal size. Implement efficient memory management strategies, such as reusing memory buffers and minimizing memory transfers, to improve performance.

Through the implementation of the subsequent procedures, one can effectively analyse the GPU specifications, make estimations regarding the most suitable memory size for the operations, and optimise algorithms to achieve efficient memory utilisation and enhanced performance on the GPU.

3.2.2 Comparative Study of GPU Compression Schemes

Three more GPU data compression techniques adopts alongside the PFOR-Delta algorithm to evaluate their performance. The algorithms encompass GPU LZSS, which utilises Lempel-Ziv sliding window compression, GPU block-sorting compression, which applies block-level sorting techniques, and GPU Huffman coding, which facilitates lossless data compression by Huffman encoding. The implementation and optimisation of each method were conducted with meticulous attention to GPU parallel processing, leveraging the highly parallel architecture of contemporary GPUs to expedite the compression procedure. An assessment was conducted to compare the performance of these

algorithms with the GPU PFOR-Delta algorithm in terms of compression ratio, compression speed, and memory utilisation. This analysis yielded useful insights into the efficiency and effectiveness of various GPU data compression strategies.

Experimental Hardware and Software Setup

The test platform used in this work is an Intel® Xeon® CPU with 12 GB RAM and an NVIDIA TESLA K40c GPU card. The operating system is Ubuntu Linux 14.04 with CUDA 7.5, and all codes are compiled using the NVIDIA nvcc toolchain. For benchmarking, 6 datasets are used in the experiments, covering data sizes and selectivity levels to stress both computation and PCIe data transfer behaviour.

Dataset

This experiments uses *enwik8 and enwik9* (Wikipedia, n.d.b) which is in text dump format; *rctail96* (Manzini, 2015) which is in XML format, *app1* (CompressionRatings.com, n.d.) which is type of application file; TPC-H (TPC.org, 2016) which is in CSV format. These datasets are carefully picked to provide us a good span of data size from 100MB to 1GB.

Results

This evaluation uses the CUDA lossless data compression algorithms in various datasets in terms of compression ratio, compression time and decompression time. Compression ratio is given by the ratio between the compressed file and the original file, Compression Ratio = $\frac{Compressed\ file\ size}{Original\ file\ size}$, while the time to compress/decompress the file is measured in seconds.

Table 3.7: Compression Ratio

Dataset	MB	GPULZSS	GPUHuffman Coding	GPUBlock-Sorting	GPUPFOR-Delta
enwik8	100	43	25	21.5	42
rctail96	114	62.3	25	10	61
app1	503	69.2	27.3	22.8	65
enwik9	1024	84.2	25.1	18.6	83
TPC-H	1024	48	26	19	55

A comparative of the compression ratio between the four GPU compression methods is presented in Table 3.8. Huffman coding (Guo *et al.*, 2013) and block-sorting compression algorithms (Patel *et al.*, 2012) outperform the LZSS (Ozsoy and Swany, 2011) and PFOR-Delta algorithm in terms of compression ratio. Among the four, the CUDA block-sorting compression technique achieves the highest compression ratio, ranging from 10% to 22.8%. The block-sorting compression technique achieves a high compression ratio performance mostly because of the additional transformation step employed in the algorithm, which enhances the efficiency of the encoding process. The transformation employed in this context is the sort transformation, which has resemblance to the Burrows-Wheeler Transform (Burrows and Wheeler, 1994) algorithm. Additionally, the block-sorting compression algorithm demonstrates effective performance on GPUs due to the minimal data dependency that occurs between the blocks of chunked data.

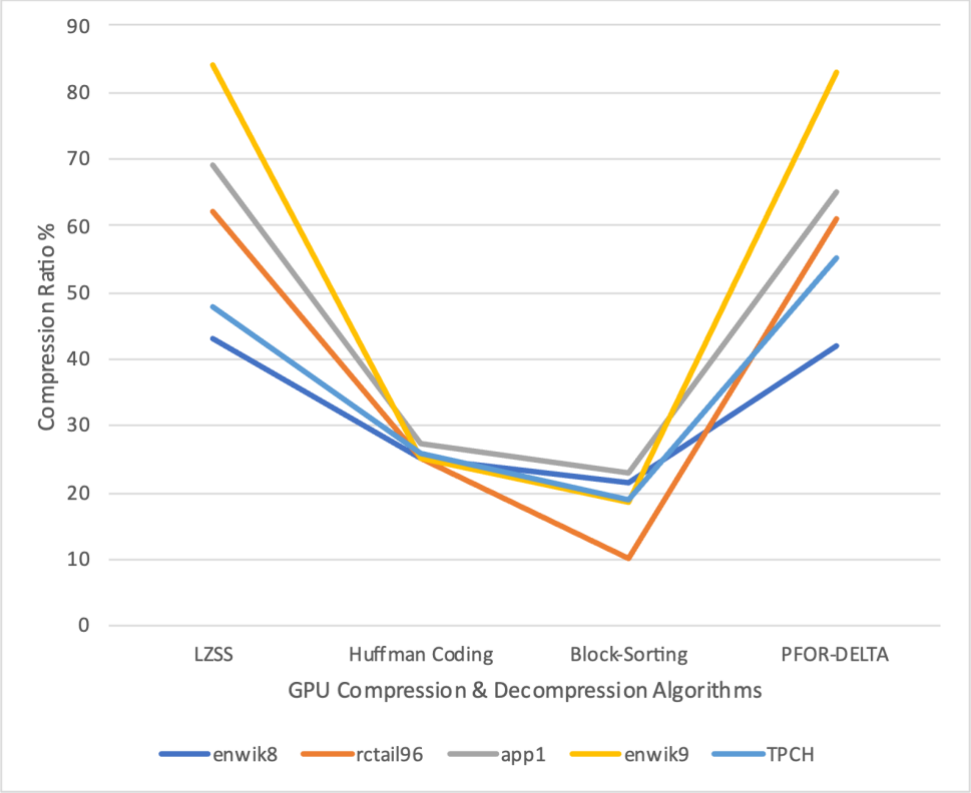


Figure 3.9: Compression Ratio of GPU Compression Algorithms

Conversely, Huffman coding employs variable bit coding to encode data, resulting in a greater reduction in file size following compression compared to the fixed-length byte coding utilised in LZSS encoding. The compression ratio provided by the LZSS and PFOR-Delta method, as depicted in Figure 3.9, has a comparatively high value, ranging from 43% to 84%, as the size of the test file grows. The aforementioned findings suggest that the PFOR-Delta algorithm demonstrates efficacy in data compression and can be regarded as a feasible substitute for alternative compression techniques. It is noteworthy that the DELTA method exhibits compression ratios that are competitive in comparison to alternative techniques, with values ranging from 42% to 83%. The effectiveness of the PFOR-Delta algorithm can be ascribed to its capacity to use the attributes of the data, including the utilisation of differential coding and a

modified version of the DELTA algorithm for optimal integer compression, rendering it very suitable for GPU acceleration.

Table 3.8: Compression & Decompression Time (Seconds)

Dataset	GPULZSS		GPUHuffman Coding		GPUBlock-Sorting		GPUPFOR-Delta	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
enwik8	0.92	0.51	1.67	0.49	5.08	10.42	1.01	0.56
rctail96	1.09	0.69	1.76	0.58	3.24	8.04	1.23	0.78
app1	4.63	4.33	7.41	6.92	26.19	17.1	4.17	3.90
enwik9	9.17	8.4	13.22	12.85	42.21	42.32	8.44	7.73
TPCH	3.67	3.53	5.29	5.40	16.88	17.77	10.40	10.80

***c** = compression; **d**= decompression*

The primary incentive for developing parallel compression algorithms on is the significant speed improvement it provides. Previous studies have documented that the version of lossless compression algorithms exhibits a certain degree of speed improvement compared to their serial counterparts (Patel *et al.*, 2012; Zhou *et al.*, 2014). Rather than examining the disparity in compression speed between GPU and versions of methods, this thesis centers on comparing the compression and decompression time across various categories of lossless compression algorithms implemented on GPU. In Table 3.8, it records the duration of compression and decompression for the three compression algorithms. The results indicate that the Huffman coding technique exhibits a 60% average compression speed advantage over the block-sorting compression technique across all datasets. In contrast, the LZSS method demonstrates a somewhat superior compression time compared to the Huffman

coding algorithm. In terms of decompression time, LZSS demonstrates superior speed compared to the block-sorting compression and Huffman coding methods for most datasets, with the exception of the enwik8 and rctail96 datasets, as depicted in Figure 3.10.

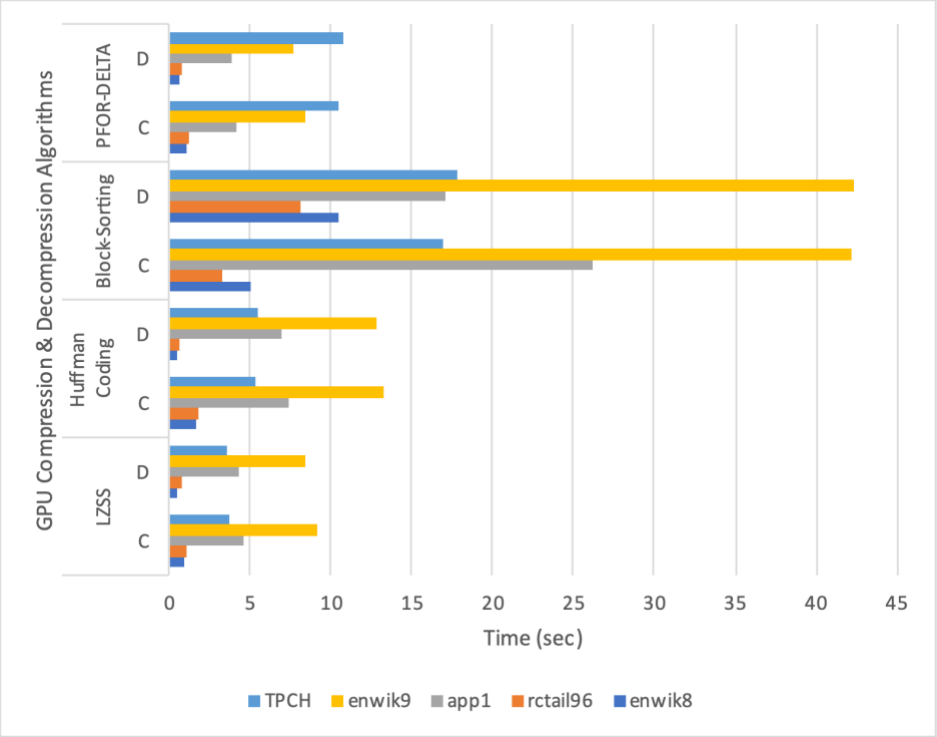


Figure 3.10: GPU Compression and Decompression Time

Due to the lack of parallelization for GPU in certain components of the block-sorting method, it is expected that the compression and decompression processes of this algorithm takes slightly longer compared to the other two CUDA compression techniques. The LZSS algorithm exhibits efficient compression and decompression performance due to two key factors. Firstly, the algorithm continuously updates its dictionary during the input data scan. Secondly, this operation is executed concurrently in each GPU block for every input data chunk. Regarding the Huffman coding technique, the sequential operation during the generation of the Huffman tree has resulted in a decrease

in the overall speed of the algorithm's compression and decompression processes. On the whole, it has been observed that various compression algorithms exhibit distinct performance characteristics in relation to both compression ratio and compression speed. However, the compression time and decompression time for all compression algorithms remain quite quick in comparison to the majority of lossless compression methods designed for execution.

Upon comparing the compression time (C) of the PFOR-Delta method with that of other algorithms, it becomes evident that, in the majority of instances, the PFOR-Delta algorithm exhibits satisfactory performance. The algorithm often exhibits compression times that are comparable to those of other algorithms, although there are certain cases where it may be either faster or slower. The compression time of the PFOR-Delta algorithm in the 'enwik8' dataset is 1.01 seconds, which is similar to the compression times of LZSS (0.92 seconds) and Huffman Coding (1.67 seconds). However, it is slower than the compression time of Block-Sorting (5.08 seconds). In the 'app1' dataset, the compression time of the PFOR-Delta algorithm (4.17 seconds) is comparatively slower than that of LZSS (4.63 seconds), although it is faster than that of Huffman Coding (7.41 seconds) and Block-Sorting (26.19 seconds).

The PFOR-Delta algorithm demonstrates competitive performance when evaluating decompression time (D). The algorithm often exhibits decompression speeds that are comparable to those of other algorithms, although there are certain cases where it may be either faster or slower. In the 'enwik8' dataset, the decompression time of the PFOR-Delta algorithm is 0.56 seconds, which is similar to the decompression times of LZSS (0.51 seconds) and Huffman Coding

(0.49 seconds). However, it is slower than the decompression time of Block-Sorting (10.42 seconds). In the 'app1' dataset, the decompression time of the PFOR-Delta algorithm (3.90 seconds) is somewhat slower than that of LZSS (4.33 seconds), although it surpasses the decompression times of Huffman Coding (6.92 seconds) and Block-Sorting (17.1 time). In terms of compression and decompression time, the PFOR-Delta algorithm has competitive performance compared to other algorithms, indicating its efficacy as a GPU data compression technique.

3.2.3 Interpretation of Compression Trade-offs and PFOR-Delta Performance

The present thesis aims to assess the efficacy of four GPU data compression approaches, including GPU LZSS, GPU block-sorting compression, GPU Huffman coding, and GPU PFOR-Delta. The methods were engineered to enhance GPU parallel processing capabilities, and their efficacy was evaluated based on metrics such as compression ratio, compression speed, and memory utilisation. In terms of compression ratio, the findings indicate that GPU block-sorting compression and GPU Huffman coding exhibit superior performance compared to GPU LZSS and GPU PFOR-Delta. Notably, GPU block-sorting compression demonstrates the highest compression ratio. The GPU PFOR-DELTA method has compression ratios that are competitive, ranging from 42% to 83%. This suggests that the algorithm is effective as a strategy for compressing data on a GPU.

The investigation also examined the duration of compression and decompression for the methods, demonstrating that GPU Huffman coding demonstrates a 60% average advantage in compression performance compared

to GPU block-sorting compression. GPU LZSS has enhanced compression efficiency in certain scenarios; however, it is surpassed by GPU Huffman coding in terms of decompression time. GPU PFOR-DELTA has competitive performance in GPU data compression applications, despite significantly longer compression and decompression durations compared to other techniques. In general, the research offers significant contributions to the understanding of the efficacy and proficiency of several GPU data compression methodologies.

3.3 GPU-accelerated Execution for Join Operation

This section presents a revolutionary integration of join algorithms that exhibit exceptional performance within the contemporary NVIDIA GPU environment. This thesis enhances the widely used radix hash join and redesigns sort merge join algorithms on GPUs. This is achieved through the implementation of innovative techniques that leverage the hardware capabilities of the current GPU architecture and the advancements introduced by the CUDA programming framework. Consequently, although the implementation uses code for common data primitives such as sorting, searching and prefix scanning from widely used CUDA libraries, the algorithms retain essential characteristics that distinguish them from prior research. In the last few years, there has been a notable advancement in the hardware architecture and software environment of GPUs, coinciding with the exponential development in processing capability. Nevertheless, current join algorithms are predominantly tailored for previous GPU architectures, which creates doubt about their capacity to effectively utilise the most recent devices available, particularly in relation to the "join" function. To achieve optimal performance, it is necessary to optimise GPU code

specifically for the new GPU components and capabilities in the runtime system software, even if GPU code may scale successfully with the increasing computing resources in newer GPU devices.

3.3.1 Implementation accelerated GPU Query Processing

This thesis introduces new GPU hash and sort-merge join algorithms that take advantage of such features to effectively utilise GPU resources. The purpose of the hash join is to first store the smaller table, which is referred to as the inner table, in memory. After that, it will traverse the bigger table, which is referred to as the outer table, in order to locate the tuple of the outer table within the hash table. The process of sort-merge involves sorting two tables as first, and then traversing them in order to determine whether or not to connect them. These hash and sort-merge join algorithms on GPUs efficiently handle inner and outer tables and optimize performance through strategic sorting and traversal techniques.

Pseudo Algorithm 3.4: GPU Join

```
s be the output table name.
j1 be the join table name.
grp be a grouping indicator.
start_seg be the starting segment.
end_seg be the ending segment.

gpu_join(s,j1,grp,start_seg,end_seg):
   process_error(2,"Join : couldn't find variable " + j1),
   process_error(2,"Join : couldn't find variable " +
   op_join.front()),
   mystat[s]=statement_count
   mystat[j1]=statement_count
   check_used_vars()
   while op_join is not empty:
      mystat[op_join.front()]=statement_count
      op_join.pop()
```

In Pseudo Algorithm 3.4, the 'gpu_join' function is a complex algorithm that handles the joining of tables in a database-like system. It takes several parameters, including the output table name 'j1', the join table name 'j1'; a grouping indicator 'grp', and segment boundaries 'start_seg' and 'end_seg'. The function first checks if the join table 'j1' and the table specified by the front of the operation queue ' op_join.front()' exist in the system. If either of these tables is not found, it triggers an error message using the 'process_error' function.

Next, the function updates the 'mystat' hash table to mark the output table 's' and the join table 'j1' as being processed in the current statement. It then checks for any other variables used in the join operation and updates their status in 'mystat' as well. After processing all variables, it iterates through the 'op_join' queue to mark each variable as processed in the current statement.

In the context of the 'gpu_join' function, the sort-merge join operation is not explicitly represented in the symbolic form. However, the concept of joining two tables based on a common attribute can be inferred from the function's logic. The 'op_join' queue likely contains the attributes used for joining, and the function performs operations related to joining these attributes. Similarly, the integration of hash tables is not explicitly represented in the symbolic form. However, the use of 'mystat' interprets as performing lookups on a hash table structure. These structures likely store information about variables and their states, which is crucial for the join operation.

In order to improve the management of GPU memory, this thesis present the idea of partitioned hash tables, which expand the integration of hash tables by integrating partitioning functionality. A partitioning key is used to divide the

data into smaller partitions, which is what is meant by the term "partitioning". After then, each division is processed independently, which can boost parallelism and lower the size of each individual hash table, ultimately leading to improved memory utilisation and speed. Other benefits include increased efficiency.

The process begins by defining a partitioning key, an attribute from the input data that facilitates even distribution across partitions. Each record is then assigned a partition ID based on this key using a partitioning function. Subsequently, both input datasets are iterated through, and each record is added to an in-memory structure dedicated to its corresponding partition. Next, for each partition, a separate hash table is constructed. A hash function is defined to map record keys to unique positions within the hash table. Each record in the partition is then processed, its key hashed, and the record inserted into the corresponding partition's hash table using the calculated hash value as the key. The join operation leverages these partitioned hash tables. This thesis iterate through the original input data sets once more. For each record, the partitioning function determines the partition ID, and the hash function calculates a hash value based on the record's key. This hash value is then used to probe the hash table corresponding to the partition ID, retrieving any matching records (records that share the same hash value). Finally, the desired join operation is performed between the current record and the retrieved matches, with the results stored in a temporary data structure. The beauty lies in parallelism. Since each partition has its own hash table, the join processing for each partition can be executed concurrently using multiple threads or processes. This significantly improves performance, especially when dealing with large datasets. The final step

involves combining the partial join results collected from each partition's temporary data structure into the final output. Pseudo Algorithm 3.5 demonstrates how partitioning empowers hash joins to handle massive datasets efficiently, making it a valuable tool for large-scale data processing.

Pseudo Algorithm 3.5: Hash Table Partition

```
'R' be the input relation 'R' that is partitioned into 'n'
   partitions: R1,R2,...,Rn
'S' be the input relation 'S' that is partitioned into 'm'
   partitions: S1,S2,...,Sm
'H' be the hash table constructed for each partition

Partitioning Step:
R={R1,R2,...,Rn}(Partitioned 'R' into 'n' partitions)
S={S1,S2,...,Sm}(Partitioned 'S' into 'm' partitions)

Hash Table Construction:
   For each partition Ri :
      Construct hash table HRi for Ri
   For each partition Sj
      Construct hash table HSj for Sj

Join Processing:
   For each record r in Ri:
      Hash r and determine the partition Sj it belongs to
      Insert r into HSj

   For each record s in Sj :
      Hash s and determine the partition Ri it belongs to
      Insert s into HRi

Parallel Processing:
   Perform join operation in parallel for each pair of
   partitions (Ri,Sj)

Merge Sort:
   After the join operation, the results from each pair of
   partitions (Ri,Sj) are merged using a parallel merge sort
   algorithm.
   Merge sort is used to efficiently merge the results from
   different partitions, ensuring that the final result is
   correctly ordered.

Result Aggregation:
   Aggregate the results from all hash tables HRi and HSj to
   produce the final join result.
```

'R', 'S', and 'H' are used in the context of partitioning, hash table construction, and join processing. Each partition of 'R' and 'S' has its corresponding hash table 'H' for efficient join operations. Integrating merge sort in this way ensures that the results of the join operation are correctly ordered and achieved an efficiently merged, even when processing large datasets in parallel.

The typical methodology utilises a configurable number of pass partitions. Conversely, this thesis observed that the number of divisions increases in a non-linear manner as the table size increases when employing this approach. Consequently, there is a non-linear increase in the duration of execution. This implementation uses a duapro partition technique (Xiangwu and Jinxin, 2016). The partition size maintains to ensure optimal fit inside the shared memory. Consequently, the cooperation stage only requires to read the data once from the global memory. Therefore, this leads to a substantial decrease in the transfer of data from global memory to shared memory. In order to attain a compact partition size for a substantial input, it is necessary to generate a considerable quantity of partitions. Therefore, the appropriate data is stored in the shared memory, necessitating the storage of a substantial amount of hash table data. Through the utilisation of this approach, this thesis handles a solitary table including 250 million pairs of integers, comprising of a key and a value. The K40c with 12 GB RAM provides enough capacity to hold two arrays of 64 million tuples, together with intermediate data, so it suits the experiments.

In order to rearrange the tuples, it is necessary for each thread block to possess knowledge of its initial positions within the partitions. The hash tables that are shared are transferred to the global memory. Following this, a prefix

scan is conducted in order to ascertain the initial position of all partitions within each block. After computing the target positions, all threads reorder the tuples in parallel by atomically incrementing the write pointer associated with each partition. A shared hash table together with its prefix sum confines write positions to threads inside the same block. This layout improves memory coalescing, allows the cache to retain most written data and reduces traffic to device memory. The approach also balances write contention across blocks and shortens the critical section for each atomic update. As a result, the join kernel sustains higher effective memory bandwidth and achieves better throughput when the number of tuples and partitions increases.

3.3.2 Evaluation of GPU Hash and Sort-Merge Joins

The evaluation compares the GPU based join algorithms with both existing GPU implementations and recent multicore CPU join code. It also studies how relation size, join selectivity, hash table parameters and device occupancy influence performance, and documents the hardware and software configurations used in the experiments to support reproducibility and later cross chapter comparisons.

Experimental Hardware and Software Setup

This thesis evaluates the performance of the merge sort algorithm on three hardware platforms, namely the Intel i7 6700K CPU, NVIDIA K40c GPU and NVIDIA GTX1080 GPU, and compares their execution time and efficiency for this classic sorting algorithm. The hardware specifications shows in Table 3.9. This comparison highlights how differences in core counts, memory bandwidth

and compute resources influence sorting throughput, which is important when selecting platforms for analytics workloads.

Table 3.9: Specification of CPU and GPU

	Intel i7-6700K	NVIDIA K40c	NVIDIA GTX1080
Clock Rate	4GHz	0.84GHz	1.73GHz
Core	4 / 8 Threads	2880	2560
L1 Cache	128KB	64KB x 14	48KB
L2 Cache	2MB	1.5MB	2MB
L3 Cache	8MB	-	-
Memory	32GB DDR3	12GB GDDR5	8GB GDDR5
Memory Bandwidth	34.1GB/s	288GB/s	320GB/s
Max GFLOPS	207.23	4494	8873

The evaluation environment comprises an Intel i7-6700K CPU, NVIDIA K40c GPU, and NVIDIA GTX1080 GPU. The i7-6700K features a clock rate of 4GHz, 4 cores with 8 threads, 128KB of L1 cache, 2MB of L2 cache, 8MB of L3 cache, and 32GB of DDR3 RAM, offering a memory bandwidth of 34.1GB/s and a max GFLOPS of 207.23. In contrast, the K40c GPU operates at 0.84GHz, boasts 2880 CUDA cores, 64KB x 14 of L1 cache, 1.5MB of L2 cache, 12GB of GDDR5 VRAM, and a memory bandwidth of 288GB/s, delivering a max GFLOPS of 4494. On the other hand, the GTX1080 GPU runs at 1.73GHz, has 2560 CUDA cores, 48KB of L1 cache, 2MB of L2 cache, 8GB of GDDR5 VRAM, and a memory bandwidth of 320GB/s, providing a max GFLOPS of 8873. These specifications are essential for understanding the computational power and memory capabilities of each component in the evaluation

environment, crucial for performance evaluation and benchmarking in research. A PCIe 3.0 16X interface is used to connect the NVIDIA GPU to the workstation that contains Intel i7. Ubuntu 18.04 LTS serves as the operating system, and NVCC 7.5 compiles the GPU code. NVIDIA Nsight analyses the runtime characteristics of the GPU kernels. The CPU uses up to 8 threads in the tests, which gives the best performance for this setup.

This work evaluates the performance of join algorithms against baseline methods using the TPC-H benchmark. The experiments focus on the LINEITEM and ORDERS tables, where tuples are joined on the common ORDERKEY attribute and executed over datasets generated at multiple TPC-H scale factors (SF). This configuration reflects a typical decision support workload and stresses both the build and probe phases of the join, allowing a fair comparison of throughput and scalability across implementations.

Results

This thesis begins by comparing the performance of hash join algorithms with that of baseline methods on large relations whose tuples are generated using the approach described in (Kim *et al.*, 2009; Paul *et al.*, 2019). In the subsequent evaluations, the parameters of the proposed join algorithms are systematically varied and analysed to understand their impact on performance and scalability. Each relation contains up to 512 million tuples, which corresponds to the maximum size that fits into the main memory of the available NVIDIA GTX 1080 Ti host, and this configuration is chosen to emulate realistic large scale analytical workloads typically seen in contemporary database systems.

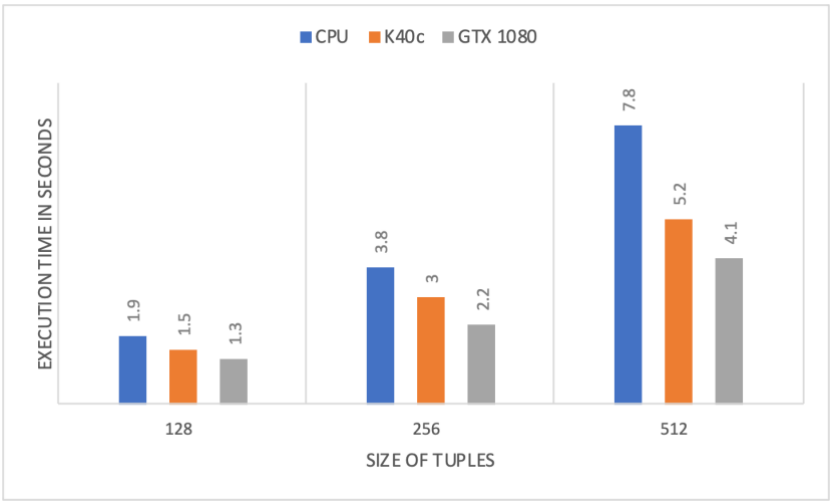


Figure 3.11: Evaluation of Hash Join on Generated Relations

Figure 3.11 shows that the proposed hash join algorithm on both GPUs consistently outperforms the hash join implementation on the multicore CPU as the relation size increases. The numbers in the table represent the execution times in seconds for different dataset sizes (128, 256, and 512). Lower values indicate better performance, as they represent shorter execution times. For the CPU configuration, the execution times range from 1.9 seconds for a dataset size of 128 to 7.8 seconds for a dataset size of 512. This indicates that the CPU's performance decreases as the dataset size increases, which is expected due to the sequential nature of CPU processing. K40c GPU shows better performance compared to the CPU, with execution times ranging from 1.5 seconds for a dataset size of 128 to 5.2 seconds for a dataset size of 512. GTX 1080 GPU exhibits even better performance than the K40c GPU, with execution times ranging from 1.3 seconds for a dataset size of 128 to 4.1 seconds for a dataset size of 512.

The thesis next compares the performance of the proposed hash join algorithms with baseline methods on the TPC H benchmark. The evaluation reads tuples from the LINEITEM and ORDERS tables and performs a join on

the common ORDERKEY attribute. The TPC H scaling factor (SF) controls the total number of tuples in each dataset. Experiments use datasets at multiple SF values to examine how join performance and scalability change as data volume increase.

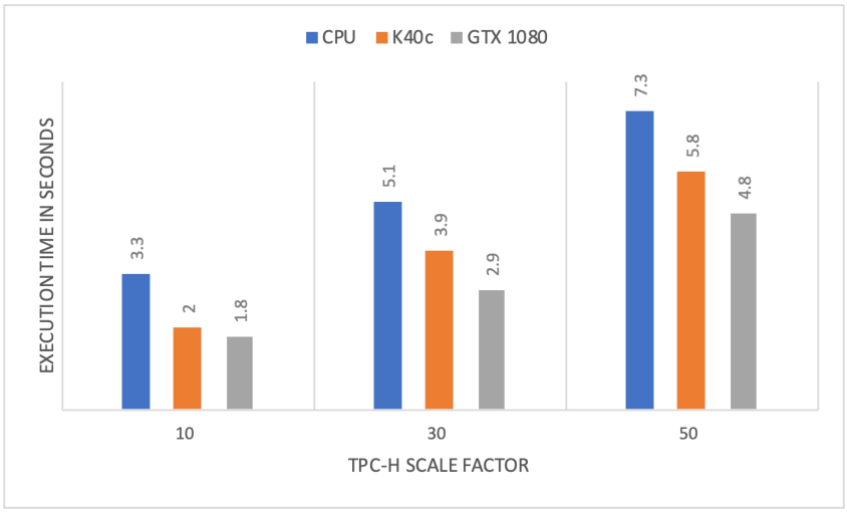


Figure 3.12: Evaluation of Hash Join on TPC-H

The results presented in Figure 3.12 demonstrate that the proposed join algorithms operating on both GPUs outperform CPU baselines as the size of the relations increases. As the dataset size grows from 10GB to 30GB and 50GB, the speedup of GPU acceleration compared to CPU processing decreases slightly but remains significant. For the NVIDIA Tesla K40c GPU, the speedup over the CPU decreases from 3.3x to 2x for 10GB and 30GB datasets, respectively, and from 5.1x to 3.9x for 30GB and 50GB datasets, respectively. Similarly, for the NVIDIA GeForce GTX 1080 GPU, the speedup decreases from 3.3x to 1.8x for 10GB and 30GB datasets, respectively, and from 5.1x to 2.9x for 30GB and 50GB datasets, respectively. Despite the decrease in speedup with larger dataset sizes, both GPUs still provide significant performance

improvements over the CPU, with the GTX 1080 consistently outperforming the K40c across all dataset sizes.

The evaluation results of hash join performance on generated relations highlight the significant performance benefits of GPU acceleration over traditional CPU-based processing. Across all dataset sizes, both the NVIDIA Tesla K40c and GeForce GTX 1080 GPUs outperform the CPU, with the GTX 1080 showing the best performance. This demonstrates the potential of GPUs to significantly improve the efficiency of hash join operations in database systems. The scalability of GPU-accelerated hash join operations is also evident, as the GPUs maintain a performance advantage even with larger datasets. This indicates that GPUs can effectively handle increasingly large datasets without a significant drop in performance, making them suitable for processing big data workloads. The performance difference between the K40c and GTX 1080 GPUs can be attributed to their architectural differences, with the GTX 1080's newer and more powerful architecture leading to better performance. Overall, these results have practical implications for database systems and applications that rely on hash join operations, as GPU acceleration can lead to faster query processing times and improved overall performance for end-users.

3.3.3 Analysis of Join Scalability and GPU–CPU Comparison

This study introduces new GPU hash and sort-merge join algorithms designed to efficiently utilize GPU resources. The hash join algorithm focuses on storing the smaller inner table in memory and then traversing the larger outer table to locate matching tuples. Conversely, the sort-merge process involves sorting two tables and then traversing them to determine connectivity. These

algorithms optimize performance through strategic sorting and traversal techniques. The pseudo code provided for the GPU join function outlines the process of joining tables in a database-like system, involving error checking, updating variable statuses, and iterative operations on the operation queue. The integration of hash tables in the join operation is implied through operations on variables and the use of a hash function for mapping keys. Additionally, the study introduces the concept of partitioned hash tables to improve GPU memory management, dividing data into smaller partitions and processing them independently for enhanced parallelism and memory utilization. The partitioning technique involves defining a partitioning key, assigning partition IDs, constructing hash tables for each partition, and performing join operations using these hash tables. The study also discusses the implementation of compact partition sizes to optimize memory usage for large datasets, achieving efficient data storage and processing. The evaluation results compare the performance of GPU-based join algorithms with existing GPU and CPU join code on the TPC-H benchmark, demonstrating superior performance of the GPU algorithms as dataset sizes increase. Overall, the study highlights the potential of GPU acceleration for improving hash join operations in database systems, offering faster query processing and enhanced scalability for handling large datasets.

3.4 Summary of GPU Query Accelerator Architecture and Performance

In Chapter 3 builds and evaluates GPU based query accelerator for analytical SQL workloads. A multi-core CPU parses, plans and schedules queries, while a many-core GPU executes data parallel operators on columnar chunks. Experiments with selected TPC-H queries at 1 GB, 10 GB and 50 GB

show large gains over PostgreSQL on the same workstation. For the heaviest case, TPC-H Query 2 on 50 GB, PostgreSQL requires 11 529 seconds, whereas the GPU kernels finish the core computation in 10 seconds on the K20 and 8.1 seconds on the K40. When this thesis includes PCIe transfer time, the total GPU runtimes increase but still deliver effective speedups of 21 times and 25 times. These results show that the architecture turns hour long queries into workloads that complete within minutes and shift the main bottleneck from computation to data movement.

This thesis designs a pre-processing and partitioning path that feeds the GPU with compressed, chunked columns. The system converts data from relational tables and flat files into a columnar layout, then divides each column into chunks that fit device memory and allow overlap of transfer and computation. Chapter 3 compares four GPU compression schemes along this path: GPULZSS, GPU Huffman coding, GPU block sorting and a GPU PFOR Delta variant. GPUPFOR Delta compresses the benchmark datasets to between 42% and 83% of their original size and runs much faster than GPU block sorting on large inputs, while decompression stays competitive with GPULZSS and GPU Huffman. These properties enable a compression first design that reduces PCIe traffic, keeps more data on the device and turns compression into an enabler rather than a new bottleneck for GPU query processing.

This thesis implements GPU hash join and sort merge join operators that work directly on the compressed, chunked columns that the pre-processor produces. The join evaluation on synthetic relations shows that the GPU hash join scales better than the multicore CPU join and keeps response time in the

sub second range as relation size grows, while the CPU time rises to several seconds. TPC-H based experiments confirm that the GPU joins keep this advantage on more complex workloads and that newer GPUs such as the GTX1080 improve throughput further compared with the K40c. Resource profiles show that the GPU often waits for data over PCIe while the CPU stays lightly loaded once it hands work to the device. Chapter 3 therefore establishes that a single, carefully programmed GPU can act as a practical accelerator for multi gigabyte analytical SQL workloads when combined with columnar chunking and lightweight compression. It identifies PCIe transfers and device under-utilisation as key targets for the later optimisation work in this thesis.

CHAPTER 4

PATHWAYS OF GPU IN BIG DATA APPLICATIONS

This chapter examines how the GPU query accelerator behaves when it is used in realistic big data settings rather than in controlled benchmark queries. It focuses on two studies that represent common analytics workloads. The first study compares a GPU-accelerated query engine with Hadoop based platforms and a PostgreSQL database for a healthcare dataset of about thirty two gigabytes. The dataset contains twenty five tables with mixed data types and includes one fact table with around one hundred and twenty million rows, which allows the systems to be tested on queries that involve large scans, grouping and joins. The second study investigates GPU acceleration for geospatial processing, using a heat map application as a case study to see how the GPU pipeline performs when it is integrated with an existing GIS toolchain.

In the healthcare study, the GPU engine is evaluated alongside an eight node Hadoop cluster that runs CSV based Hadoop processing, Hive and Impala, and a workstation that runs PostgreSQL as a traditional relational baseline. Each platform executes six SQL style queries that cover simple aggregation, grouped counts and multi table joins over the largest table. By comparing execution times across these systems, the chapter provides concrete evidence of where a single GPU workstation can outperform distributed CPU based solutions, and where factors such as query complexity and data layout still limit performance.

The geospatial study turns to heat map computation driven by ArcGIS. Here the GPU-accelerated query engine uses a column oriented file structure

and the CUDASET representation to store geospatial attributes in a structure of array layout. This organisation improves memory coalescing and makes it easier to offload filtering and aggregation to the GPU. The performance of the GPU based pipeline is contrasted with Apache Spark, using the same input data and SQL statements generated by the GIS tool. The analysis focuses on both the speedup achieved for map generation and the way performance changes as the volume of spatial points grows.

Together, these two strands show how the GPU query accelerator operates when it is placed next to widely used big data engines and GIS tools rather than being measured in isolation. The remainder of the chapter is organised as follows. Section 4.1 describes the experimental environments, the healthcare dataset and the six queries, and reports the comparative results for PostgreSQL, Hadoop and the GPU workstation. Section 4.2 introduces the GPU-accelerated geospatial processing pipeline, explains the role of the columnar file system and CUDASET, and presents the evaluation of heat map generation against Apache Spark. Section 4.3 summarises the main observations and highlights the implications and limitations of using a single GPU accelerator in big data and geospatial applications.

4.1 GPU and Big Data Application

In the present study, this thesis assumes that the use of GPU-accelerated query processing engines in comparison to Hadoop on healthcare datasets yield expedited query processing durations and enhanced efficiency. The underlying basis of this idea is in the parallel processing capabilities exhibited by GPUs, which are very suitable for effectively managing the computing requirements

associated with big data analytics. The GPU-accelerated technique is expected to exhibit superior speed and performance compared to Hadoop, particularly in the context of healthcare datasets where prompt analysis is essential for decision-making and patient care.

4.1.1 Implementation

Building upon the setup outlined in Chapter 4, which forms the core framework of the proposed GPU-accelerated query processing system, this thesis integrates techniques for data partitioning and join operations to further evaluate the performance and efficiency of the proposed system. As established in Section 4.1, serves as the starting point for integrating the advancements in data partitioning discussed in Section 4.2 and the optimized join operations detailed in Section 4.3.

With the Chapter 4 methodology in mind, the optimization of the GPU query accelerator engine is a critical aspect of enhancing its performance and efficiency in handling complex data analytics tasks. In this context, the focus is on leveraging existing optimised multi-threaded libraries and APIs to improve the system's functionality, rather than inventing new ones. By utilizing these resources, the GPU query accelerator engine can achieve higher levels of parallelism and concurrency, leading to faster query processing times and improved overall performance. This approach not only streamlines the development process but also ensures compatibility with established standards and practices in the field of data analytics.

The current query engine, while employing a combination of Bison and Flex for parsing, experiences a bottleneck. During the CPU phase, there is lack

of multi-threaded tasks parallelism (He *et al.*, 2009; Wang *et al.*, 2014; Breß *et al.*, 2016; Liu *et al.*, 2022). The CPU phases are in charge of parsing clause into objects. It identifies the required data source, translates the operation into low level instruction sets. Then, it arranges execution sequence and dispatch for execution. It uses the combination of Bison and Flex implementing a SQL query parser by using traditional single thread mechanism. There is lack of handling the CPU task parallelism with GPU data parallelism related workloads. However, CPU starts initializing GPU contexts, preparing input data, launching GPU kernel functions, and materializing query results in controlling the steps of query progress in CPU. GPU phases are in executing specific optimized kernel functions. These are mostly aggregate and compute intensive functions. The data are used thousands of core to process at once. This limits overall performance, especially for complex queries involving operations like joins and aggregations. This limits overall performance, especially for complex queries involving operations like joins and aggregations.

To address this limitation, this thesis integrates the Boost libraries into the query engine. Boost provides optimised parallel processing functionalities that complement the existing use of CUDPP and NVIDIA Thrust libraries. This integration is expected to improve the processing time of complex queries. In particular, the Boost API is used to extend the scheduler so that incoming queries are placed into a shared work queue and dispatched across a pool of CPU worker threads. The scheduler controls the number of active threads and, in turn, the level of resource contention on the CPU. A resource monitor collects the current status of GPU device usage, and the scheduler uses this information to assign queued tasks to available GPUs. In this design, the CPU plays an important role

in managing concurrent queueing. The queue implementation allows data to be safely added by one thread and removed or joined by other threads without corrupting the data. This helps maintain an appropriate concurrency level and workload on the GPUs. A data swapping mechanism is employed to maximise the effective utilisation of GPU device memory. Through these processes, overall resource utilisation is improved, using a mixture of Application Program Interface (API) calls from the Boost libraries.

This optimization maximizes concurrency by implementing a pipelining mechanism to overlap data transfer via the PCIe bus and arithmetic computation. Utilizing CUDA streams enables asynchronous execution, queuing work and promptly returning control to the CPU. Additionally, a pinned memory mechanism is often employed in specific query implementations, leveraging the Direct Memory Access (DMA) engine to achieve a higher percentage of peak bandwidth. The Hype-Q feature in Kepler architecture further enhances performance by allowing multiple CPU threads to be launched on the GPU simultaneously, significantly increasing GPU utilization and reducing CPU idle times.

4.1.2 Performance Assessment of GPU and Hadoop Platforms for Healthcare Queries

This experiment comprises the execution of six distinct queries, with each query being conducted five times on a Hadoop cluster consisting of multiple nodes. The response times of each query vary significantly depending on the approach employed. Furthermore, this thesis conducts a comparison between a solitary workstation and a distributed system consisting of multiple nodes.

Experimental Hardware and Software Setup

The experiment is conducted using two distinct setups. The first setup involved a Hadoop 2.7 based distributed system comprising eight virtual machines. These virtual machines were distributed across four HP DL380p G8 servers, each equipped with a 12-core processor and 32GB of RAM. The setup included seven worker nodes and one master node, facilitating the distributed processing of data across the virtual machines.

The second setup utilises a GPU workstation consisting of a DELL T5500 with an Intel® Xeon® E5630 @ 2.53GHz Quad-Core Processor, NVIDIA Tesla K20c GPU, 8GB of RAM, and a 1TB WD HDD. The workstation was running Windows Server 2008 R2 Enterprise 64-bit operating system. This setup provided the computational power required for GPU-accelerated processing, enabling efficient execution of complex queries and computations. By comparing the performance of these two setups, the experiment aimed to evaluate the effectiveness of GPU-accelerated query processing compared to traditional Hadoop-based distributed systems in handling big data analytics tasks. In short, the two experimental environments are depicted below:

Table 4.1: Specification of Hadoop Vs GPU Workstation

8 Nodes Hadoop Cluster	GPU Workstation
4 x HP DL380pG8 Servers: Duo Intel Xeon E5-2630 @ 2.3GHz with 12 Cores, 32GB RAM DDR3 and SAS HardDrive 20TB (10K RPM), Centos 7 with Cloudera Hadoop 2.7	GPU Workstation: DELL T5500, Intel Xeon E5630 @ 2.53GHz with Quad Core Processor, NVIDIA Tesla K20c, 8GB RAM, WD HDD 1 TB, Windows Server 2008 R2 Enterprise 64-bit

There is process of exporting and processing dataset from PostgreSQL to the Hadoop file system for Hadoop Hive and Impala query processing involved the following steps. Initially, the data extracted from PostgreSQL was transformed into a compatible format for Hadoop in CSV. The export technique assured data compatibility with the Hadoop file system and facilitated seamless processing by Hive and Impala. After exporting the data, it was moved to the Hadoop file system using tools like Apache Sqoop or custom scripts. These tools facilitated the seamless movement of data from PostgreSQL to Hadoop, guaranteeing the availability of the data for query processing in Hive and Impala. The data was imported into tables in Hive using the suitable HiveQL statements. The statements established the schema of the data and determined the mapping of the data to the underlying Hadoop files. After importing the data, it became possible to query it using HiveQL, which enabled advanced analytics and data processing functionalities. The data in Impala was imported into tables using analogous statements to HiveQL. Impala demonstrated superior query performance in comparison to Hive due to the utilisation of a distinct query engine that is specifically optimised for interactive queries. Impala's capability for real-time data analysis and processing makes it a vital tool for rapid and efficient data processing in Hadoop. The process of exporting and processing data from PostgreSQL to the Hadoop file system for Hive and Impala query processing required a series of procedures to guarantee the correct formatting, transfer, and import of the data into the Hadoop ecosystem.

Dataset and Queries

The dataset utilised in this investigation was supplied by a member of the local healthcare sector and has a total data size of 32 gigabytes. The dataset was initially stored in a PostgreSQL database for the management of the substantial amount of data. The database consisted of 25 tables, each holding diverse forms of information pertinent to healthcare analytics. One of the tables in the database was significantly extensive, encompassing around 120 million records. This table was helpful in assessing the efficiency of the query processing engines, since it simulated a practical situation where datasets can be very vast and intricate. The tables in the database had a variety of fields, which included data types such as integers, floats, texts, and dates. The variety of data kinds enabled a thorough examination of the query processing engines' capacity to effectively manage diverse data types.

Six queries were designed to represent typical data processing tasks in the healthcare dataset used in this study, as summarised in Table 4.2. These queries were carefully crafted to assess various aspects of query processing, such as data retrieval, aggregation, and join operations, which are commonly performed in healthcare analytics scenarios. By executing these queries on each platform and measuring the query execution times, this thesis aims to gain insights into the relative strengths and weaknesses of each system in handling complex data analytics tasks. This comprehensive evaluation approach allowed us to make informed recommendations for selecting the most suitable platform for healthcare analytics applications, considering factors such as query performance, scalability, and ease of use.

Table 4.2: Queries Description for Healthcare Dataset

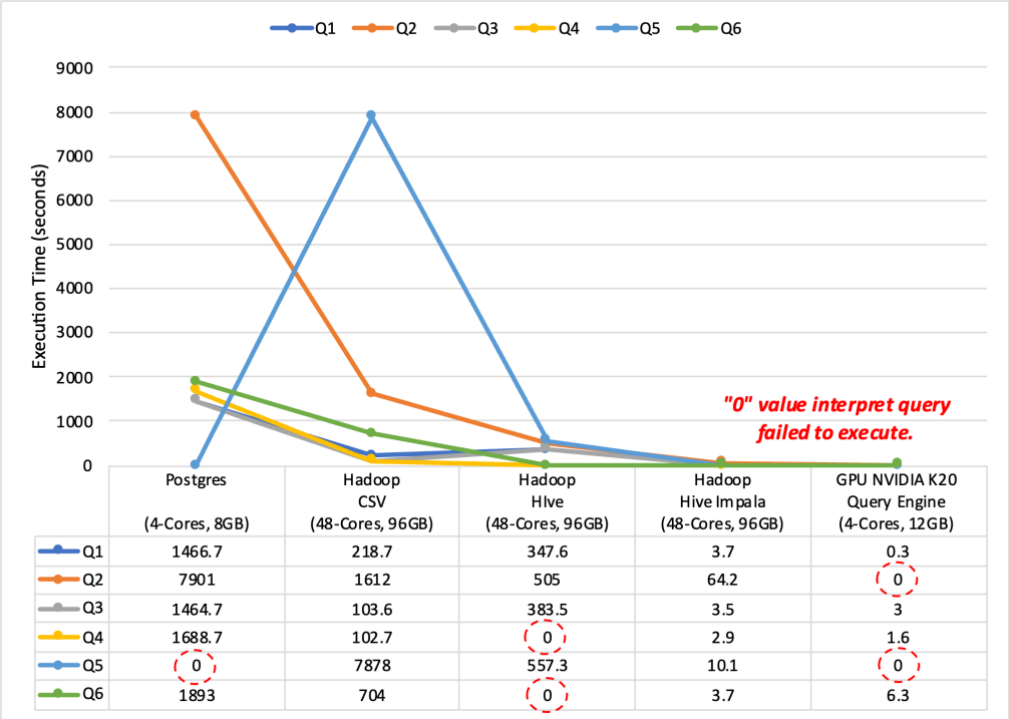
Query	Description
Q1	Selecting sum from one column of 120 million records
Q2	Selecting a name column, counting the name and ordering by top 10 names
Q3	Selecting state code, years from hospital patient records with one disease code selected group by years and state code, order by years and state code
Q4	Selecting state, years, disease name from hospital patient records wither one disease code with disease name and state code with state names, grouping by years, states, and disease names, ordering years and state code
Q5	Inserting the results of selecting state code, years from hospital patient records with ALL diseases type group by years and state code, order by years and state code
Q6	Selecting state, years, disease name from hospital patient records where three disease name types are selected and joining disease type with disease name state code with state names, grouping by years, states and disease names, ordering by years, states and disease names.

4.1.3 Cost and Performance Implications for Healthcare Analytics

The execution times for the six queries across the five systems varied significantly, highlighting the performance differences between them. The execution time for each query type was analysed for PostgreSQL database and GPU query accelerated engine on the DELL T5500 workstation; Hadoop based distributed system on the eight virtual machines. There are Apache Hadoop CSV, Hadoop with Hive, Impala with Hive, and a GPU workstation in order to compare the processing time (in seconds). Figure 4.1 displays the performance outcomes in terms of the execution time for the queries conducted on 120 million

rows. Since certain portions of the execution time have been recorded as "0", this is considered a failure to complete the execution.

PostgreSQL on the 4 core, 8 GB RAM workstation shows high execution times for all queries, with Q2 reaching 7 901 seconds, which indicates limited suitability for the large scale workloads in this experiment. In contrast, Hadoop CSV, Hadoop Hive and Hadoop Impala on the 48 core, 96 GB RAM cluster execute most queries substantially faster, confirming the benefit of distributed processing for large datasets. The GPU accelerated query engine on the NVIDIA K20 with 12 GB RAM delivers the strongest improvements, particularly for Q4 and Q5, with runtimes of 2.9 and 10.1 seconds. These results highlight GPU acceleration as a competitive option for analytical query processing alongside cluster based Hadoop systems.



	Postgres (4-Cores, 8GB)	Hadoop CSV (48-Cores, 96GB)	Hadoop HIve (48-Cores, 96GB)	Hadoop Hive Impala (48-Cores, 96GB)	GPU NVIDIA K20 Query Engine (4-Cores, 12GB)
Q1	1466.7	218.7	347.6	3.7	0.3
Q2	7901	1612	505	64.2	0
Q3	1464.7	103.6	383.5	3.5	3
Q4	1688.7	102.7	0	2.9	1.6
Q5	0	7878	557.3	10.1	0
Q6	1893	704	0	3.7	6.3

Figure 4.1: Comparison of Query Execution

GPU query accelerated engine takes the shortest time to process ~120 million records and the cost is also cheaper than implementing eight nodes of Hadoop. SQL was not able to compute on big data especially for real time analytics. It even failed in the insertion of output data in the same database. Hadoop with Hive is not suitable for real time processing, and would only be useful for batch processing of big data. Impala-Hive is as beneficial as GPU for general queries and could be used to complement the hybrid parallel processing approach especially led by the orchestration engine with predefined rules within the scripts, developed for the middleware layer. Impala Hive is in-memory accelerated query engine, whereby is a faster as there are multiple tables to be joined and with huge strings operations to be performed.

In this experiment, the query failures represented by 0 values in Figure 4.1, which are interpreted as cases where a system could not complete the healthcare workload under the chosen configuration, rather than as precisely diagnosed software or hardware faults. For PostgreSQL, a value of 0 indicates that the query did not finish successfully or could not materialise the result, which is consistent with the earlier observation that SQL was unable to handle the largest query and failed when inserting the derived output back into the same database. For the GPU workstation, an entry of 0 for the NVIDIA K20 means that the GPU based query engine returned a generic failure status instead of a valid execution time. In practice this reflects limitations of the prototype engine running on the K20, where the full query pipeline could not be completed for some healthcare queries. This could be due to resource constraints, insufficient memory or processing power and limitations in the query processing algorithms implemented in these systems (Doraiswamy *et al.*, 2016; Chrysogelos *et al.*,

2019). For the Hadoop based systems (CSV, Hive and Impala), failed queries point to problems at the level of data preparation or query configuration in the Hadoop stack, such as mismatches between the exported CSV data and the Hive or Impala table definitions, or issues in the way particular queries were expressed for these engines. This interpretation is consistent with benchmarking studies, which often report Big Data and DBMS results with limited methodological and diagnostic detail, making it difficult to explain anomalous outcomes in depth (Carey 2013; Raasveldt *et al.*, 2018). Similarly, guidance on GPU performance analysis stresses the importance of architecture aware metrics such as occupancy, memory bandwidth and kernel level counters when diagnosing performance limiters on Fermi and Kepler GPUs (Micikevicius 2012; Konstantinidis and Cotronis 2017). In the absence of such detailed measurements, the safest interpretation is to use the failures only as markers of where each system was unable to cope with the workload, and not as evidence for specific internal bottlenecks.

Under these conditions, it is still possible to make a careful and defensible claim about better performance of the GPU based query engine. For the subset of healthcare queries that completed successfully, the NVIDIA K20 based engine consistently achieved shorter execution times than the PostgreSQL workstation and the Hadoop based systems, for tables containing approximately twelve million rows. These comparative results are drawn directly from the measured timings and do not depend on speculative interpretations of the failure cases. They are also aligned with architectural expectations for Kepler class GPUs, where high memory bandwidth and large numbers of parallel cores are designed to benefit data parallel analytical workloads, provided that the kernels

expose sufficient parallelism and use memory efficiently (Micikevicius, 2012; Konstantinidis and Cotronis, 2017). At the same time, the presence of failures and the lack of configuration sweeps mean that this advantage cannot yet be generalised to all query types, data sizes or deployment configurations. A balanced conclusion is therefore that, in the configurations and workloads studied, the K20 based GPU engine demonstrates clear performance benefits for the queries that succeed, while the failure cases highlight areas where further engineering, tuning and diagnostic instrumentation are needed to make such GPU based engines more robust and to broaden the range of situations in which those benefits can be confidently claimed.

Despite the query failures, the results remain reliable for the subset of queries that completed, because the reported execution times are direct measurements on clearly specified hardware, software and data configurations. The study is also valid within this defined scope, since the main limitations, in particular the absence of detailed diagnostic information for failed queries and the lack of systematic configuration exploration, are explicitly acknowledged and tied to those specific cases rather than to all observations. Within this framework, a cautious performance claim is still justified: for the queries that did finish, the K20 based GPU query engine consistently achieved shorter execution times than PostgreSQL and the Hadoop based systems on the same healthcare workload. Taken together, these findings support the view that GPU acceleration can provide tangible benefits for structured query processing in the tested configurations, while the failure cases underline the need for further engineering, tuning and instrumentation to improve robustness across a wider range of analytical queries.

4.2 GPU-accelerated Geospatial Processing

In the context of geospatial computation, this thesis assumes that GPU acceleration leads to notable performance enhancements in handling complex geospatial calculations, showcasing superior speed and efficiency compared to CPU-based approaches. This work delves into two pivotal GPU acceleration techniques for optimizing Geo-Spatial processing. Firstly, the CUDASET methodology for Structure of Array optimization and the strategic offloading of CPU parallel processing to the GPU. The CUDASET mechanism is employed to streamline data organization and expedite data pre-processing, a critical step in Geo-Spatial applications. By restructuring data layout to align with GPU memory access patterns, CUDASET enables more efficient data transfer and manipulation, ultimately enhancing overall processing performance. Secondly, another implementation is the offloading of computationally intensive tasks from the CPU to the GPU, leveraging the GPU's unparalleled capabilities in massive data parallelism. This approach is facilitated by GPU streams, which allow for concurrent execution of multiple kernels and data transfers, effectively addressing the dual challenges of data parallelism and task parallel processing. By strategically distributing data parallelism workload across GPU, this thesis aims to maximize resource utilization and minimize latency, thereby significantly boosting processing throughput.

This geo spatial processing on GPU-accelerated functions involves in integrating ESRI (Environmental Systems Research Institute, Inc) software, named ArcGIS Desktop for performing accelerating geo-spatial data computation via the generated SQL statement. These works have shown the

specific implementation details, challenges encountered, and performance metrics achieved through the application of CUDASET and GPU-accelerated structure query for geo spatial processing. The output of result set is to generate a heat map diagram. By critically examining these GPU acceleration techniques, this thesis contributes to the growing body of knowledge on high-performance computing for geospatial applications and shed light on the potential for further advancements in this field.

4.2.1 Implementation

Source data in the database requires a pre-processing stage. It converts the data into a parallelizable files structure, GPU FS (File System), then it stores into a column-based orientation, as shown in Figure 4.2. Thus, data can access independently, compute parallelly and maximize CPU multithreaded processing. Each column segments into multi files and the size of each segment is customizable. Therefore, CPU and GPU have sufficient amount of memory to compute larger data set. Furthermore, it allows GPU to independently process each column in the segment.

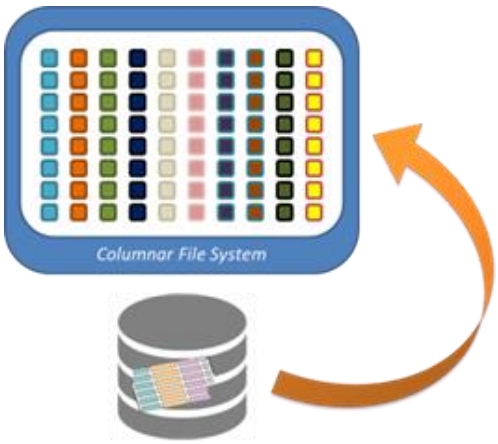


Figure 4.2: GPU Columnar File System

The change of the data in the database does not automatically trigger an update on the pre-processed data. Thus, it needs to be re-created or complemented (when only new data is added). The CUDASET is a parallel file structure, which improves the parallel geo-spatial processing jobs in GPU. It is not a legacy array of structure (AOS) design that losing of bandwidth and wasting of L2 cache memory. GPU query accelerated engine uses CUDASET representation to arrange data in Structure of Array (SOA) access pattern. It certainly gains high throughput by coalescing the memory access in GPU. Also, it is critical to memory-bound kernel functions.

CUDASET uses Thrust library device_vector and host_vector containers to store each column and supports multiple data types, including int, float and char. It maintains metadata for every column, such as name, type, size and record count, so the engine can manage schema level operations efficiently. The class allocates GPU device memory for all columns based on the current record count and updates the metadata accordingly. It also provides methods to copy data for individual columns or for the entire table between host and device in a single operation. This design simplifies memory management, reduces boilerplate CUDA code and offers a consistent abstraction for higher level query operators that consume columnar data. The required elements of structure can load individually and no data interleaving as shown in Figure 4.3. Thus, it achieves high global memory performance. The SOA CUDASET data structure used in this thesis originates from earlier work on GPU accelerated SQL query processing (Yong *et al.*, 2016) which introduces a CUDA based class for storing columnar data in a structure of arrays layout.

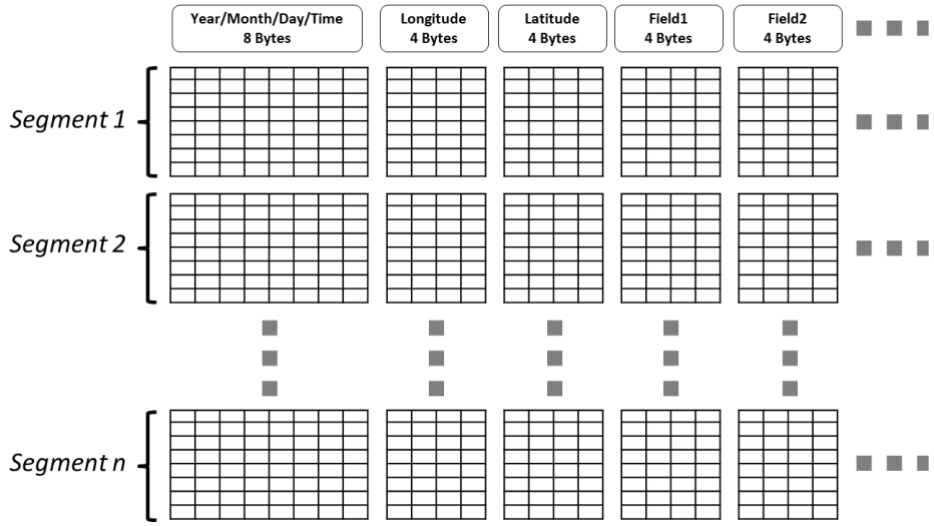


Figure 4.3: SOA CUDASET Structure

Thus, the CUDASET class provides a high-level abstraction for managing and processing large datasets on NVIDIA GPUs using CUDA. It handles data storage, memory management, data transfer, compression, loading, operations, indexing, sorting, and retrieval, making it easier to work with big data efficiently on GPU devices. The compression method is applied as in Section 3.2.1 in PFOR-DELTA compression techniques.

There are a series of processes to produce a heat map. It represents the geographic density of features on a map. The coloured areas represent points that making for layers with a large number of features. It requires certain toolboxes in ArcGIS to complete entire process and visualize the map, such as Density toolset, Spatial Analyst & Statistics toolbox. These set of toolboxes generate the SQL statement and pass to database system to execute. Figure 4.4: shows a sample of SQL statement for Heat map.

GPU-accelerated processing carries out processes of query investigation by performing the basic parsing string into query object and positioning operations. It targets to break down the input user queries to further clause of

objects and collects the decomposed SQL (Figure 4.4) objects to determine either CPU or GPU executable instructions to be used. Then, it produces an execution query plan. There is further adjustment of the plan by analysing and tracing parallelizable points and rearranges clause of objects execution. GPU-accelerated execution engine performs the query execution on either CPU or GPU.

The query execution in GPU starts with filtering process. The filtering process aims to filter out unrelated geospatial data in order to accelerate the processing time needed. It identify the records to be filtered from the "WHERE" statement from the query executed. After that, it creates CUDASET structure clone for output. For each column of the input CUDASET, it allocates host memory for the particular column in output CUDASET. The CUDASET that is separated into multiple segment due the large volume of geospatial data. For each segment, it checks if the partial data are in range. If it is not in range, and proceed to the next segment. If it is in range, then it load fields data from pre-processed binary file for current segment into host memory. Then, the fields data is being copied from host memory to GPU memory.

In GPU memory, the process of filtering is divided into three phases. Firstly, the system calculates the number of blocks and threads based on number of data in that particular segment. In phase two, each threads evaluates condition expression into Boolean result. Then, the Boolean result for each data is being assigned into corresponding position in a Boolean list. A filter count variable is created to store to amount of each 'true' result from the Boolean list. In phase three, GPU allocates memory according to the number filter count. Then, each

threads is responsible to copy index with ‘true’ value in to original Boolean list into filtered index list. Finally, the filtered index list is copied into host memory and set into output CUDASET. The filtering process repeats for each column and segment.

In the geospatial dataset, longitude and latitude values transform into planar x and y coordinates using the Mercator projection (Battersby *et al.*, 2014). Mercator represents the Earth on a flattened cylinder, with meridians and parallels intersecting at right angles and preserving local angles, which originally supported global marine navigation. This projection also simplifies spatial indexing and window based queries in modern GIS systems. The conversion step ensures that all points align correctly in ArcGIS and that subsequent analyses operate in a consistent projected coordinate space. In this study, the projected coordinates support aggregation of rainfall measurements, in particular the computation and visualisation of yearly rainfall totals over the relevant spatial regions.

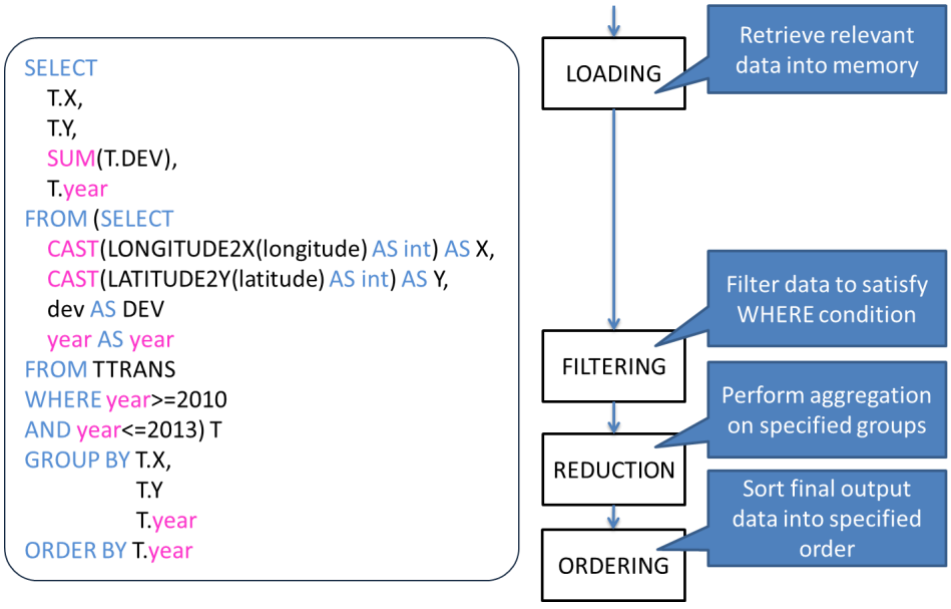


Figure 4.4: SQL for Heat Map Computation

4.2.2 Performance Study of GPU-Accelerated Geospatial Heat Map Generation

This section describes the experimental setup and analysis of the result. There are two experiments conducted to evaluate the proposed implementation techniques; data pre-processing and computation result set for heatmap. It utilizes GPU accelerator and comparing with a CPU based system, Apache Spark for a fast engines for big data processing system. This thesis measures the execution period by adding fixed number of data records for each test run.

Experimental Hardware and Software Setup

This thesis performs the experiments into a single NVIDIA Tesla K20c GPU computational device. It has 2496 CUDA cores and 5GB GDDR5 RAM. The workstation is HP Z800. It contains dual sockets Intel Xeon X5680 CPU with the total of 12 cores, clock rate is 3.33GHz, 32GB DDR3 RAM, 1TB Hard Disk, and ATI FireMV 2260 as a display device. For software, it has Microsoft Windows 7 (64bits), Spark Version 1.3.0 and CUDA 7 (Release Candidate).

Dataset

This is a weather simulated dataset for a precipitation data from 2010 to 2099. The location data is stored in a 3x3 km grid throughout the whole area of the map. It consists of about 127 million records with each record includes seven columns which are year, month, day, longitude, latitude, rainfall value, and runoff value. The memory size of test data is 4.1GB.

Results

Pre-processing for Geo-Spatial Dataset

The computation time of data pre-processing is tested with various size of data. The raw input data stores in CSV format. It need to process and import into corresponding data warehouse systems. Spark is converted into Parquet format, which is in columnar storage layout. GPU-accelerated structure query system is stored in a GPU accessible columnar format. Figure 4.5 shows the execution time on data pre-processing. The timing includes the data transfer between hard disk, CPU and GPU memory. For the smallest dataset of 1 million records, the GPU-accelerated query system demonstrates a significant performance advantage, completing the pre-processing in just 1.10 seconds compared to Apache Spark's 4.91 seconds. This represents a 77.6% reduction in pre-processing time, highlighting the GPU system's efficiency in handling smaller datasets. As the dataset size increases to 5 million records, the GPU-accelerated query system maintains its lead, processing the data in 5.34 seconds compared to Apache Spark's 7.02 seconds. However, the performance gap begins to narrow, with the GPU system offering a 23.9% reduction in pre-processing time. For the 10 million record dataset, the GPU-accelerated query system completes the pre-processing in 10.47 seconds, while Apache Spark takes 13.54 seconds, resulting in a 22.7% improvement in favour of the GPU system.

Interestingly, as the dataset size reaches 15 million records, the performance difference between the two systems becomes less pronounced. The GPU-accelerated query system processes the data in 15.42 seconds, while Apache Spark takes 17.42 seconds, offering an 11.5% reduction in pre-

processing time. Finally, for the largest dataset of 20 million records, Apache Spark slightly outperforms the GPU-accelerated query system, completing the pre-processing in 19.56 seconds compared to the GPU system's 20.85 seconds, a 6.6% difference in favour of Apache Spark.

The data pre-processing is not a compute intensive operation and requiring high I/O data transfer between the CPU and GPU memory. In addition, Spark has a minimum start-up overhead without enabling the data compression for processing the raw CSV files. As observed in Figure 4.5, the processing speed of Spark eventually overtakes the GPU system. Spark applies the optimization of utilizing the CPU multi-threading features to preparing these CSV files.

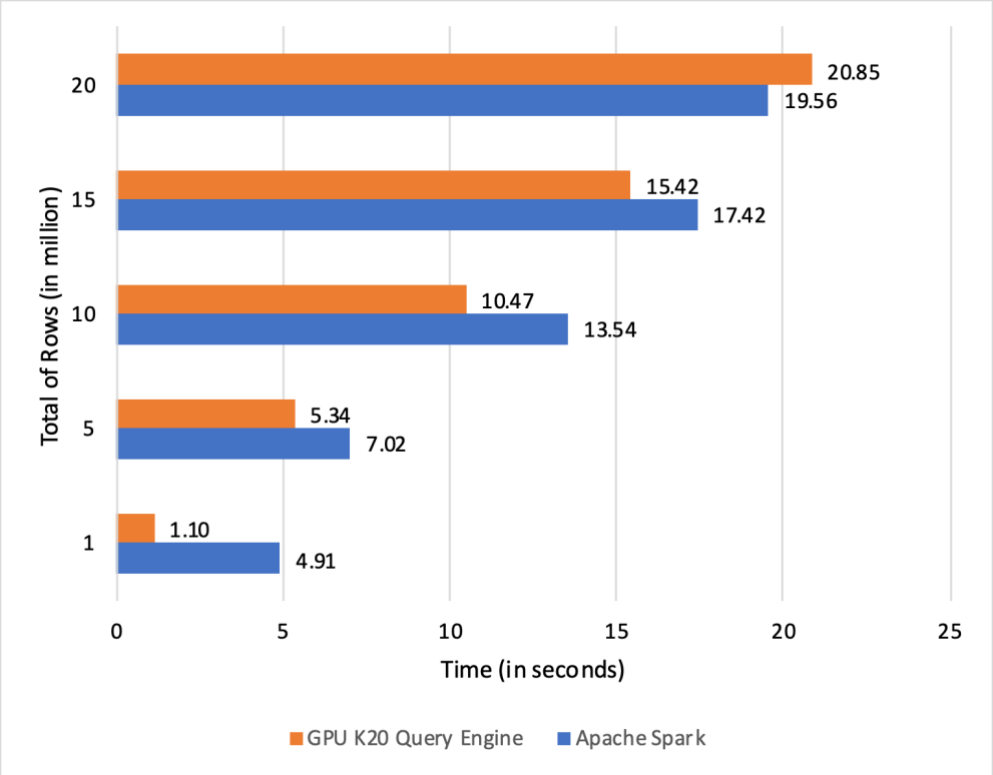


Figure 4.5: Result of Data Pre-processing

The findings suggest that the GPU-accelerated query system excels in pre-processing performance for smaller datasets, making it an attractive choice for

applications dealing with data sizes up to 10 million records. The significant reduction in pre-processing time can lead to faster query response times and improved overall system performance. However, as the dataset size increases beyond 15 million records, the performance advantage of the GPU-accelerated query system diminishes, and Apache Spark becomes more competitive. This implies that the choice between the two systems should be based on the expected dataset sizes and the specific requirements of the application. For larger datasets, factors such as scalability, distributed processing capabilities, and ease of integration with existing big data ecosystems should be considered when deciding between Apache Spark and the GPU-accelerated query system.

GPU-accelerated Mapping Density

There are a series of processes to produce a heat map. It represents the geographic density of features on a map. The coloured areas represent points that making for layers with a large number of features. It requires certain toolboxes in ArcGIS to complete entire process and visualize the map, such as Density toolset, Spatial Analyst & Statistics toolbox. These set of toolboxes generate the SQL statement and pass to database system to execute. Figure 4.4 shows a sample of SQL statement for Heat map.

The speedup measurement calculates in (Spark's Execution Time) / (Mi-Galactica Execution Time). The results turn out that GPU-accelerated query engine had out performed Spark. The speedup is between 4x to 18x speedup in executing the SQL statement shown in Figure 4.6. A snapshot of final visualization output of the heat map is shown below, Figure 4.7.

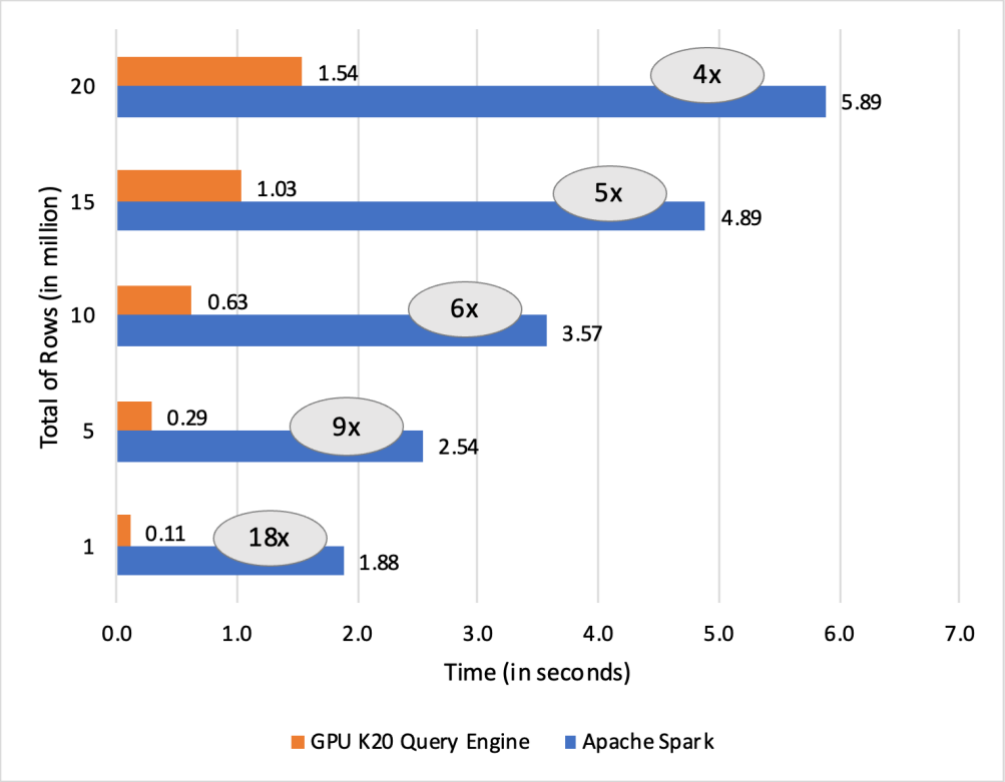


Figure 4.6: Heat Map query execution

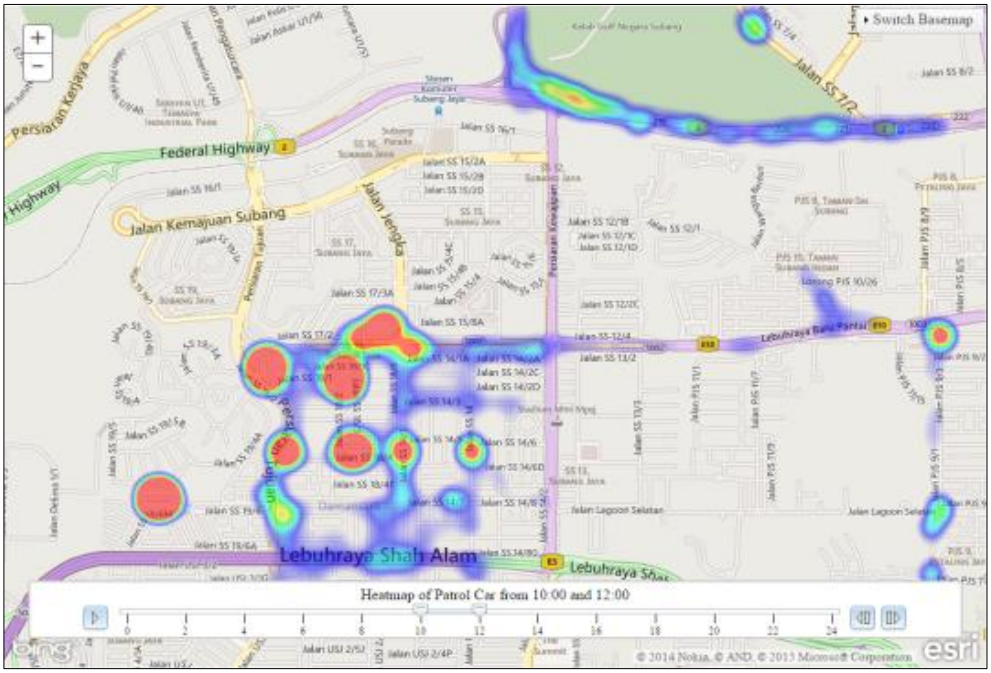


Figure 4.7: Visualization of Heat Map

4.2.3 Discussion of GPU Geospatial Pipeline and Spark Comparison

As observed, GPU-accelerated query engine GPU-accelerated query engine reduces the speedup while the rows of data are increasing. It is due to the handling of the I/O (Input/Output) movement between CPU and GPU memory without applying an efficient streaming mechanism during this preprocessing stage. In fact, there does not have complex computation to maximize the GPU resource utilization as well. Thus, it lost the optimization effort at transferring data between CPU and GPU. However, GPU-accelerated query engine performance is still good enough to provide timely visualization on geo-location data

4.3 Summary and Implications for Single-GPU Big Data and Geospatial Workloads

This chapter presents a GPU-accelerated query engine as an alternative to traditional databases for large scale data processing. The engine supports SQL based analytic operations, including data cleansing, as illustrated in the example application. Experimental results show that the GPU based solution offers a practical alternative to Hadoop and MapReduce style systems for big data analytics. The GPU-accelerated database design also integrates with front-end applications through a standard database connector. It allows users to harness the power of the GPU by providing efficient work distribution across GPU cores, particularly in terms of I/O access and compression. Furthermore, this thesis has successfully applied the GPU query engine to Geographic Information Systems, specifically in applications that employ SQL-like operations, striving to support the SQL92 standard and enable parallel accelerated query processing.

However, the current thesis still has clear limitations, especially for query processing in multi-GPUs environments. This thesis demonstrates the potential of GPU acceleration for big data and geospatial workloads, but does not examine how to execute queries across multi-GPUs or manage the associated coordination and data distribution challenges. Future work therefore needs to extend the architecture and algorithms to multi-GPUs settings and explore the additional opportunities and bottlenecks that arise in that context. Multi-GPUs processing introduces additional complexities, such as data partitioning, load balancing, and communication overhead between GPUs. These factors can significantly impact the performance and scalability of query processing in a multi-GPUs setting.

To fully realize the potential of GPU acceleration in big data and geospatial processing, future research should investigate the specific limitations and challenges of multi-GPUs query processing. This may involve developing novel algorithms and techniques for efficient data partitioning and distribution across multiple GPUs, optimizing inter-GPU communication, and addressing load balancing issues. Additionally, further studies could explore the integration of multi-GPUs query processing with distributed computing frameworks, such as Apache Spark or Hadoop, to enable large-scale data processing across multiple nodes and GPUs.

Extending the architecture to multi-GPUs query processing and addressing these limitations can unlock new opportunities for accelerating big data and geospatial analytics. It enables organizations to process and analyse even larger datasets, extract valuable insights more quickly, and make informed

decisions in various domains, such as business intelligence, scientific research, and geospatial applications.

To wrap up, this thesis has demonstrated the significant potential of GPU acceleration for big data and geospatial processing, there are still critical areas that require further exploration, particularly in the realm of multi-GPUs query processing. By acknowledging these limitations and focusing future research efforts on addressing them, this thesis continues pushing the boundaries of GPU-accelerated computing and develop more powerful and efficient solutions for handling the ever-growing demands of big data and geospatial analytics.

CHAPTER 5

STRING MATCHING QUERY

String matching is a core building block in many data intensive applications. It appears in data cleansing, log analysis, security monitoring and bioinformatics, and it also underpins a wide range of text based predicates in database systems. As data volumes grow, the cost of scanning large text collections and evaluating patterns becomes a bottleneck, particularly when existing engines still execute many of these predicates on the CPU. This chapter investigates how different classes of string matching algorithms behave on modern NVIDIA GPUs and what kinds of speedup can be achieved in practice.

The work focuses on three complementary strands. First, it studies an exact string matching mechanism for name lists using a GPU friendly Hash Match design (Alcantara *et al.*, 2009). The hash based method compresses each name into a fixed width 64 bit value before comparison, and is evaluated on a dataset of up to ten million names on Fermi and Kepler GPUs. The experiments show that Hash Match on Kepler achieves speedups of about six to eight times over Boyer–Moore–Horspool and about two to three times over a brute force Column Search, with the gains arising from both reduced data movement and better use of GPU parallelism.

Second, the chapter examines histogram based matching as a representative of algorithms that aggregate character frequencies rather than comparing strings directly. A series of implementations are developed for multi

core CPUs with OpenMP and for GPUs using CUDA, including a streaming version that overlaps data transfer and computation. On large inputs, the GPU optimised streaming histogram runs around seven times faster than the multithreaded CPU version when transfer time is included, which highlights how careful use of the GPU memory system can turn a simple but heavy weight primitive into an efficient building block.

Third, the chapter returns to approximate pattern matching, focusing on the Wu–Manber algorithm for short patterns. Building on earlier work that mapped Wu–Manber to GPUs and Xeon Phi, a new asynchronous implementation is proposed that divides the workload into multiple CUDA streams to overlap kernel execution with memory copies. The technique is evaluated on three GPU generations and improves the speed of pattern matching for short patterns by roughly three to seventeen percent compared with the previous state of the art, particularly when many matches are found and overlapping becomes effective.

These three investigations do not yet integrate directly into the GPU query accelerator of Chapter 3, but they provide concrete evidence that common string operations required by SQL style workloads can benefit from GPU acceleration. The results also expose the conditions under which GPU based string matching is most effective, such as high degrees of parallelism, sufficient pattern density and careful management of data transfers. The remainder of the chapter is organised as follows. Section 5.1 presents the Hash Match mechanism and its evaluation on large name datasets. Section 5.2 introduces the histogram based implementations on CPU and GPU and reports their performance. Section 5.3

details the asynchronous Wu–Manber design for short patterns and its evaluation on several GPU platforms. Section 5.4 closes the chapter with a summary of the main observations and their implications for future integration of string matching into GPU-accelerated query processing.

5.1 Hash Match on GPU

The Hash Match on GPU mechanism consists of two components; Hash function and Matching. The following section briefly illustrates these components and the functionalities to support the GPU hash mechanism through the implementation on GPU. The diagram in Figure 5.1 presents an overview of this proposed Hash Match mechanism.

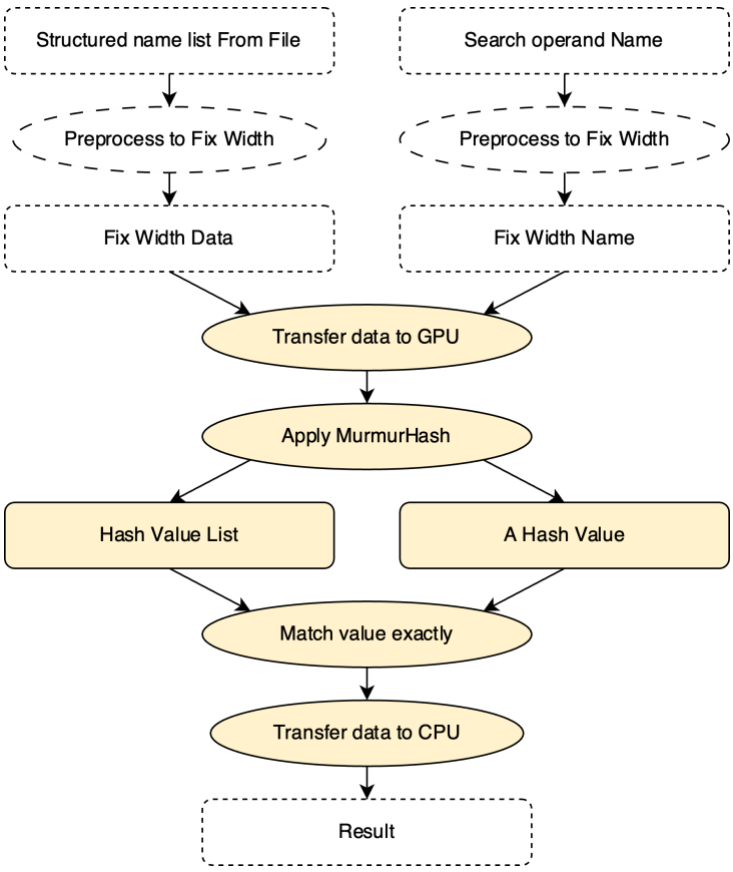


Figure 5.1: Hash Match Workflow

5.1.1 Implementation

The initial step in the process involves fragmenting the data into a size that has a constant width. This fragmentation process is carried out on the CPU, and the data that has been fragmented is saved in the RAM that is located on the CPU. Before moving on to the next stage, which is the hashing process, the fragmented data is first transferred into the Random Access Memory (RAM) of the GPU, more particularly the Global Memory. In the event that the search term or name is being used, a copy of it will also be placed in the shared memory of the GPU. Moving on to the third step, each fixed width block of names is hashed by the MurmurHash2A function. This completes the third stage. This function will produce a list of 64-bit integers, one for each name in the list, and those numbers will represent the list. After the data have been hashed, they are saved in the Global Memory in preparation for the next stage. When the matching engine reaches the fourth stage, it begins the process of completing a precise matching of the name list and continues until it reaches the goal. The fifth and final stage of the matching process consists of transferring the results of the matching process back to the RAM of the CPU for further processing in the user interface.

Hash Function

It is a formula primarily to generate a hash value from a given input data. It acts as a shortened reference to the original data. The principle of accelerating the string matching is using one or more good hash functions to generate a unique hash value and reduce collisions. The criteria of choosing a hash function is based on the consideration of efficient hash algorithm, non-cryptographic

string hashing, and able to hash in 32/64-bit formats. There are a few popular non-cryptographic hash functions which are listed in (Wikipedia, n.d.a). MurmurHash serves as the hash function for the proposed hash based lookup.

GPU Matching

The GPU assigns parallel comparison tasks for matching a list of hash value against a targeted searchable hash value. Each of the streaming multiprocessors process the kernel functions for matching the data. If it detects the match, it flags "1" to the output buffer, which is stored in the GPU's global memory. This is an embarrassingly parallel comparison, by separating each chunk for parallel execution of the value comparison and there is no dependency between these parallel tasks. The result is stored into each pair's index when it detects a collision.

After hash value comparison, it verifies the match result. As the hash function cannot guarantee that there is no collision, it compares byte-by-byte for those matched names against the user input name. Furthermore, there it is necessary to consider the assigned thread block size and eliminating branch divergence to ensure that they fully occupy the SMPs.

Column Search (Brute Force)

There are three major steps to perform Column Search. Firstly, the input data need to be pre-processed into a fixed column width (45 characters). Thus, each row stores a person's name. Secondly, it performs parallel row by row comparison of a name using character matching from left to right. This starts from the leftmost pattern of the character. If a character matches with the character in the pattern, it flags "1" to the index of positions. Thirdly, once the

above step has completed, the next kernel function performs a parallel atomic sum operation for each row. If the total is matched with the length of the pattern, then it indicates this row (name) is matched. This thesis implements this naïve mechanism to perform brute-force matching the name. It is used to compare with the proposed Hash Match implementation.

Boyer-Moore-Horspool

It uses a window of size m (length of the pattern) to search the text from right to left. There is a pre-processing to compute an array of the rightmost character of the window to skip the position of characters when a mismatch has detected. Building on earlier work by (Tay, 2010), this thesis focuses on extending the Boyer–Moore–Horspool implementation to GPU based query processing. First version of this matching is without the usage of shared memory, and the second version applies the shared memory concept.

5.1.2 Experimental Results for Hash Match on Fermi and Kepler GPUs

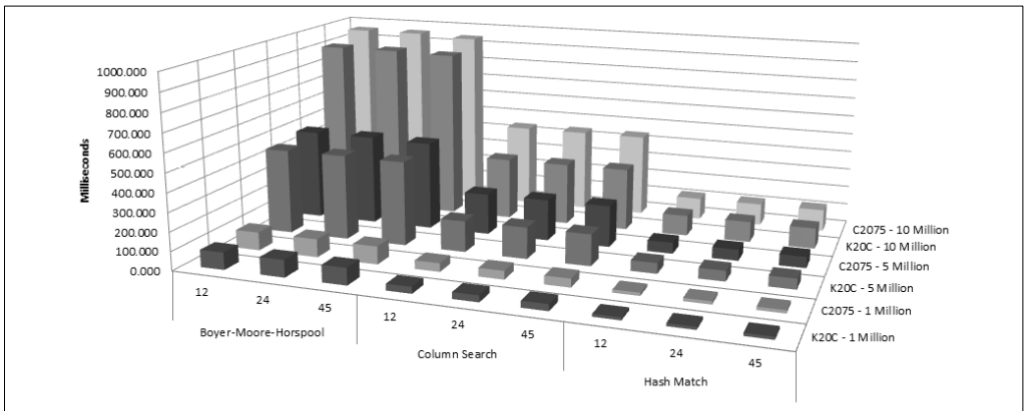
This thesis implements the GPU Hash Match on a Tesla C2075 Fermi GPU and a Kepler K20c GPU based on an 8-core Intel Xeon Server. Fermi consists of 448 CUDA cores with 1.15 GHz and 6GB DDR5 memory. Kepler consists of 2496 CUDA cores with 706 MHz and 5GB DDR5 memory. Both boards have 48KB shared memory and 16KB L1 cache per thread block and 384-bit wide memory bus. Three string matching mechanisms and algorithms were implemented on the GPUs using Microsoft Visual Studio 2010 VC++ along with NVIDIA CUDA 5 (nvcc compiler). These string-matching mechanisms are the Boyer-Moore-Horspool, Column Search (Brute force) and Hash Match mechanisms.

The dataset comes from a data cleansing application project. The database contains 25 columns, but the experiments use only the name column. This consists of 10 million names, which are saved in a text file. The maximum length of the name is 45 bytes. Those names are the combination of first name and last names. The query pattern was constructed of randomly chosen names from the dataset. The total size of the dataset is approximately 429.15 Megabytes. The experiments use three dataset sizes: about 42 MB (1 million names), 214 MB (5 million names) and 429 MB (10 million names).

This thesis concentrates on comparing the speedup performance of various GPU string matching algorithms against the GPU Hash Match. The total time is measured in milliseconds for the time needed for an algorithm to find the match of a keyword in the name dataset and is composed of the data transfer and kernel execution times. The dataset was transferred in one batch to the global memory of the GPU. The keyword and relevant lookup tables were using for the pre-processing, which were copied to the GPU shared memory area of each streaming multiprocessor.

In this experiment, this thesis evaluates the performance of different exact string-matching implementations with the proposed GPU Hash Match mechanism on the Fermi and Kepler NVIDIA computation cards. Figure 5.2 shows the speedup of implementation of the proposed Hash Match mechanism on GPU. The result has shown the Hash Match on Kepler outperforms the speedup to 6.25x/8.24x/8.34 (1/5/10 million) against Boyer-Moore-Horspool, and 2.36x/2.95x/3.04x (1/5/10 million) against Column Search. Column search does not need to build a lookup table, since it is based on byte-by-byte

comparisons. It gets a 2.64x/2.75x/2.74x (1/5/10 million) speedup when compared to Boyer-Moore-Horspool, and shows that it is well suited to be applied on parallel systems for matching. The major boost of the Hash Match performance is the shrinking of data to a 64-bit integer, and optimizing the execution steps, after applying hashing and matching value. In addition, there is performance gain on the shortened transfer time because the data is compressed.



	Boyer-Moore-Horspool			Column Search			Hash Match		
	12	24	45	12	24	45	12	24	45
K20C - 1 Million	90.775	90.772	89.945	34.341	34.342	34.362	14.522	14.651	14.521
C2075 - 1 Million	96.094	95.981	94.454	45.188	45.220	45.292	16.229	16.289	16.232
K20C - 5 Million	453.875	453.860	449.725	164.863	166.848	166.852	55.065	55.021	55.129
C2075 - 5 Million	480.470	479.905	472.270	219.285	220.849	221.335	58.809	58.912	58.951
K20C - 10 Million	907.750	907.720	899.450	331.211	332.020	333.050	108.861	108.723	108.916
C2075 - 10 Million	960.940	959.810	944.540	440.104	440.930	441.090	115.168	115.371	115.418

Figure 5.2: Comparison Performance of String Matching

Figure 5.3 shows the execution time for comparing a hash value by assigning various block and thread size. Results show that computation speed is highly dependent on the block and thread sizes and can affect performance by up to 102 times. Kernel execution can be less than 1 millisecond for a properly configured system. Either insufficient threads assignment for a large dataset or too many threads in the small datasets has caused the slowdown performance. Thus, it is important to examine the complication of computation and assignment of sufficient thread sizes and the occupancy for each thread. A warp comprises 32 threads, and the scheduler issues instructions at the granularity of

a warp. By carefully coalescing global memory loads and stores within each warp, the design reduces DRAM traffic and improves effective bandwidth utilisation. Experiments show that blocks with 32 threads deliver the best performance on the evaluated kernels. This configuration matches the warp size, keeps all CUDA cores within each streaming multiprocessor fully occupied, and simplifies tuning of occupancy, memory access patterns and latency hiding for subsequent kernels in the query pipeline.

Block & Thread Size	1 Million	5 Million	10 Million
(65535,1024)	3.01	3.24	3.53
(65535,128)	0.361	0.536	0.76
(65535,64)	0.423	0.783	1.238
(31250, 32)	0.0293	7.825	15.597
(31250, 160)	0.679	3.003	5.699

Figure 5.3: Hash Value Kernel Matching on K20c

Figure 5.4 shows the analysis on the kernel computation time only for the string matching mechanisms. Experiments measure the speedup of GPU kernel functions on two generations of NVIDIA cards across several datasets. On the K20c, without the NVIDIA Kepler architecture specific optimisation or porting, the kernels achieve about 1.4 times speedup relative to the earlier card in this setup. The Boyer–Moore–Horspool kernel runs faster than the Column Search kernel, but Column Search still delivers better end to end performance. The Boyer–Moore–Horspool implementation generates large additional shift and lookup tables, which increase data transfer volume and overhead to the GPU and offset its kernel time advantage. Hash match kernels provide the fastest computation for fixed width parallel hashing and independent matching on each

name, and they form an efficient building block for the text processing pipeline evaluated in this chapter.

String Matching Algorithms	1 Million		5 Million		10 Million	
	K20c	C2075	K20c	C2075	K20c	C2075
Boyer-Moore-Horspool	7.631	11.218	38.185	56.152	76.327	112.188
Column Search	23.900	34.740	119.160	173.580	238.090	347.070
Hash Match	4.080	5.718	9.340	13.081	15.740	22.039

Figure 5.4: Kernel Execution Time

Based on the experiment for Boyer-Moore-Horspool, it shows that shared memory can optimize the repetitive usage of a small chunk of data to communicate and collaborate on computations. This provides 23x speedup by using the shared memory. It takes the advantage of high speed memory access within the same thread block. This provides approximately 150x faster computation than the un-cached global memory. Boyer-Moore-Horspool utilizes this in the pattern of the lookup tables.

5.1.3 Discussion of Hash Match Behaviour and Thread Configuration

This thesis has proposed and implemented the Hash Match mechanism for parallel exact string matching on GPUs. This thesis also evaluates it against Boyer-Moore-Horspool and Column Search mechanisms performed on GPU as well. The experimental results show that the Hash Match mechanism achieves significant speedup by shrinking and reducing operations to parallel process the data. This has shown Hash Match mechanism for exact string matching tends to be an embarrassingly parallel task. Future research in the area of string matching for parallel processing needs to be focused on the latest GPU compute capability. This computation engine incorporates the new Kepler dynamic parallelism methodology. By splitting the internal loop, it can design a child

kernel function to recursively compute it. This method can optimistically maximize the usage of all the CUDA cores. Moreover, it is important to have a study on continues streaming with parallel matching mechanism with the latest NVIDIA HyperQ technology. Finally, it would be interesting to examine the performance of these matching mechanisms executing on a MPI distribution system with GPU Direct.

5.2 Histogram Optimization with CUDA

A histogram counts how many input elements fall into each of a set of predefined value ranges or buckets. The resulting values describe the data distribution and reveal skew, sparsity or concentration patterns. Both the domain of the input values and the number of buckets can vary widely across applications, from simple frequency counts in text analytics to fine grained measurements in scientific and geospatial workloads.

5.2.1 Implementation

Pseudo Algorithm 5.1. shows a sequential histogram design. The array input of size n stores the values to analyse. The code samples each element of input and maps it to a bucket index that represents the observed value. The array histo of size buckets stores the histogram counts, where each element corresponds to one bucket. Lines 1 to 4 initialise all entries of histo to zero. Lines 6 to 10 iterate over the input array and increment the appropriate element of histo for each value. This procedure produces a simple baseline for later parallel and GPU based histogram implementations.

Pseudo Algorithm 5.1: Simple Computation for Sequential Histogram

```
1   for (int i = 0; i < buckets; i++)
2   {
3       histo[i] = 0;
4   }
5
6   for (int i = 0; i < n; i++)
7   {
8       /* 0 <= input[i] < buckets */
9       histo[input[i]]++;
10  }
```

Parallel programming techniques can be used to accelerate histogram computation which can be divided into micro and macro parallelization levels. Micro level exploits data parallelism. It uses the same computational model for every thread, but feeds it with different parts of the data. The data to be examined is decomposed into smaller segments, which are then assigned to different threads. Each thread will then examine the assigned segment and update the histogram. It is also known as “Single Instruction Multiple Data” (SIMD). In general, it is well-suited for many-core accelerator. On the other hand, macro level works on simultaneous task execution, which is suitable for processing highly complex computations.

This thesis implements the optimization of histogram computation by using both multi-core CPU and many-core GPU hardware. OpenMP multi-threaded directive method is adopted for the multi-core CPU implementation. To further exploit the many- core GPU compute capabilities and accelerator features, there are three different optimization mechanisms explored. It includes the use of global memory, shared memory and streaming. In this investigation, the input data is a random stream of 1 octet-wide, which may represent a stream

of ASCII characters. As the 8-bits input can be anything ranging from 0x00 and 0xFF, a minimum of 2^8 number of buckets is required to keep count of the number of occurrences.

Multi-core CPU with OpenMP

The first attempt of histogram optimization is to use OpenMP to maximize the usage of multi-core CPU resources. It is a conventional optimization technique for multi-thread processing on a standalone machine. The parallel execution of the tasks is performed simultaneously across a defined number of CPU cores. However, in this implementation, due to the high possibility of multiple threads attempting to increase the same bin array element simultaneously, a local bin array per thread is necessary to avoid read-write conflicts. The final bin array is calculated by combining these local bin arrays when all threads have finished processing.

Many-core GPU Programming Model

This section provides a quick overview of the programming model for CUDA on NVIDIA GPU architecture. For the purpose of computation in GPU, data retrieved from a hard disk which is stored in the CPU memory needs to be transferred to GPU memory device via the PCIe bus. After GPU processing has been completed, the output will be transferred back to the CPU memory. These processes are initiated in the CPU host code.

CUDA uses the concept of threads, blocks, and grids. These kernel codes are executed in parallel by a series of threads. Blocks gather into thread-blocks to enable synchronization and the usage of shared memory, whereas grids group all the blocks. These worker threads are executed on streaming multiprocessors

(SMXs). Thus, different blocks can be dynamically scheduled into different configurations such as concurrent block execution on different SMXs, concurrent execution on the same SMX by using CPU multi- threads; concurrent execution on the same or different multiprocessors at different times. All of the threads have dedicated registers but share the GPU global memory.

A warp is a group of multiple threads within a single block. It is a basic unit of thread scheduling in the SMXs. Once a block is assigned to a SMX, it is divided into units of warps. Each warp can support up to 32 threads in the Kepler architecture. Each thread in a warp executes in parallel. For optimized warp execution, branch divergence should be avoided as much as possible. Branch divergence occurs when threads inside a warp branch into different execution paths. All threads in the same grid execute the same kernel code but handle different data (SIMD). With the Kepler/Maxwell architecture, a block can consist of 1024 threads in each of the x, y, and z dimensions. The maximum of x dimension in blocks of grid can go up to 2^{32-1}.

There are three different GPU implementations: Global Memory, Shared Memory and Streaming. There are essential factors to be considered on the design of the kernel code in order to obtain the best performance such as data transfer latency, fine tuning of kernel parameters for data parallel execution and resource utilization. Both global and shared memory accesses should be performed in a coalesced manner. In such cases, GPU combines multiple load and store instructions from each thread within a warp into fewer numbers of transactions necessary to service all of the threads. As all memory accesses are cached, the number of transactions are dependent on the number of cache lines,

128 byte lines. Consequently, it keeps the workload efficiency high. Memory tiling and stride access techniques are employed to enhance the per thread workload. In the memory tiling implementation, each thread computes two adjacent input elements. In contrast, the stride access pattern implementation deliberately computes multiple non-adjacent data elements while striding across the input data.

Global Memory

Both the input data and bucket array are stored in the GPU global memory area. Each thread evaluates different elements of the input data buffer while increasing the bucket array elements accordingly. The same race condition similar to the CPU OpenMP implementation is highly probable, thus it is necessary to increment the bucket array atomically. An atomic operation is guaranteed to be performed with no interference from any other threads in the GPU.

Pseudo Algorithm 5.2: Global Memory Stride Access Kernel Code

```
1   __global__ void
2   gpu_histogram_stride_access(
3       unsigned char * data,
4       unsigned int * bin,
5       unsigned long long size)
6   {
7       unsigned long long int i;
8       unsigned int offset;
9       i = blockIdx.x * blockDim.x + threadIdx.x;
10      offset = blockDim.x * gridDim.x;
11      while (i < size) {
12          atomicAdd(&bin[data[i]], 1);
13          i += offset;
14      }
15  }
```

Consequently, this atomic operation might cause some of the write operations to be serialized instead, which in turn may dampen performance. Pseudo Algorithm 5.2 shows the globally allocated input and bucket memory locations are supplied via the kernel's function parameters. The "cudaMalloc()" CUDA API calls to allocate the GPU memory in the main caller section code before launching the kernel.

Shared Memory

This optimization uses the GPU shared memory where its latency is roughly 100x lower than global memory latency with appropriate coalesced access. Each thread has its own private copy of the bucket array. Input data is stored in the global memory, while the private bucket array is located in the shared memory, as shown in Pseudo Algorithm 5.3, where the private copy is declared using the __shared__ variable specifier. Each thread utilizes stride access across the assigned input data, and then increases these local buckets. Therefore, it does not directly access the final bucket array in the global memory. After all threads have finished processing, ensured by the synchronization barrier, the final bucket array is updated with the sum of all of these local buckets.

Pseudo Algorithm 5.3: Shared Memory with Stride Access Kernel Code

```
1   __global__ void
2   gpu_histogram_shared_mem_stride_access(
3         unsigned char * data,
4         unsigned int * bin,
5         unsigned long long size)
6   {
7         unsigned long long i;
8         unsigned int stride;
9         __shared__ unsigned int shared_bin[BIN_SIZE];
10        shared_bin[threadIdx.x] = 0;
11        __syncthreads();
```

```
12      i = blockIdx.x * blockDim.x + threadIdx.x;
13      stride = blockDim.x * gridDim.x;
14      while (i < size) {
15          atomicAdd(&shared_bin[data[i]], 1);
16          i += stride;
17      }
18      __syncthreads();
19      atomicAdd(&bin[threadIdx.x],
20      shared_bin[threadIdx.x]);
21  }
```

Streaming

In the final optimization phase, the kernel implementation with the best performance is modified to perform computation in multiple streams. In essence, the compute cycle of one kernel overlaps with the host to device data transfer of another kernel, thus improving utilization efficiency. In order to stream, one of the requirements is to use host pinned memory for the input data using the API "cudaMallocHost()". This pinned memory is generally used as a staging area when transferring data from host pageable memory to the device. This allow sthe CUDA driver to directly copy these data into the device. Pseudo Algorithm 5.4 launches the same kernel on all streams that handle data processing, except on the default stream. The design avoids launching kernels on the default stream in order to prevent unintended serialisation of work. The "cudaMemcpyAsync()" API performs asynchronous memory transfers from pinned host memory to device memory on the non-default streams. These asynchronous copies overlap with kernel execution in other streams, which hides transfer latency and improves overall throughput for the histogram workload.

Pseudo Algorithm 5.4: CUDA Stream Implementation

```
1   for (int j = 0; j < no_streams; j++)
2   {
3       checkCuda(cudaMemcpyAsync(
4           d_data + (j * stream_size * sizeof(unsigned char)),
5           h_data + (j * stream_size * sizeof(unsigned char)),
6           stream_size * sizeof(unsigned char),
7           cudaMemcpyHostToDevice, streams[j]));
8
9       gpu_histogram_shared_mem_stride_access <<<
10      dimGrid, dimBlock, 0, streams[j]>>> (
11          d_data + (j * stream_size * sizeof(unsigned char)),
12          d_bin, stream_size);
13  }
14
15  cudaDeviceSynchronize();
```

5.2.2 Experimental Results for Histogram Optimisation on CPU and GPU

This section describes the setup of the test environment, shows the experiment results and discusses various implemented optimization mechanisms for the parallelization of histogram.

Table 5.1: Specification of GPU K20c & GTX980Ti

	K20c	GTX980Ti
CUDA Cores	2496	2816
Memory BUS	320-bit	384-bit
Memory Size	5GB GDDR5	6GB GDDR5
Memory Bandwidth	208 GB/s	336 GBs

The experiments are executed on a workstation, which contains Intel Xeon Processor E5-1607@3GHz, 8GB DDR3 RAM and 400GB SATA hard disk storage. The operating system is Ubuntu 14.04 TLS (x64) with NVIDIA Tesla driver 352.93. There are two different types of NVIDIA GPU cards used for

testing; NVIDIA Kepler K20c and GeForce Maxwell GTX980Ti. The specification is shown in Table 5.1. The source codes are compiled using NVIDIA NVCC. It is based on CUDA 7.5 toolkit with compute capabilities version 3.5 for NVIDIA Kepler architecture. The Nsight Eclipse IDE tool is used for code editing and performance profiling.

The input data size ranges from 512 MB to 3584 MB. For each kernel, the speedup is computed using the formula in Figure 5.5, where it is defined as the ratio of the execution time of the sequential CPU implementation to that of the corresponding GPU implementation. This definition follows established GPU performance studies that quantify speedup in the same way (Bauke and Keitel, 2011; Bard and Dorelli, 2014), and it suits the CPU and GPU comparison in this thesis. The final average speedup is obtained by averaging the individual speedups of all kernel codes over the different input sizes. Each configuration is executed multiple times, and the mean value is reported to reduce variability and improve the accuracy of the measurements.

$$\textbf{Speedup} = \frac{\textit{CPU Computation Time}}{\textit{GPU Computation Time}}$$

Figure 5.5: The Speedup Calculation

The average speedup of the GPU implementation in the first optimization phase using global memory with maximum threads per block shows almost 8x speedup comparing to the CPU OpenMP implementation, as shown in Figure 5.6. Although this implementation has utilized all available CPU cores from the workstation, the sheer number of threads executed on the GPU is simply enormous. The GPU architecture is designed to be massively parallel with its thousands of much smaller CUDA cores. As an example, the NVIDIA Tesla

K20 consists of 2496 cores allowing the GPU to efficiently handle multiple tasks simultaneously.

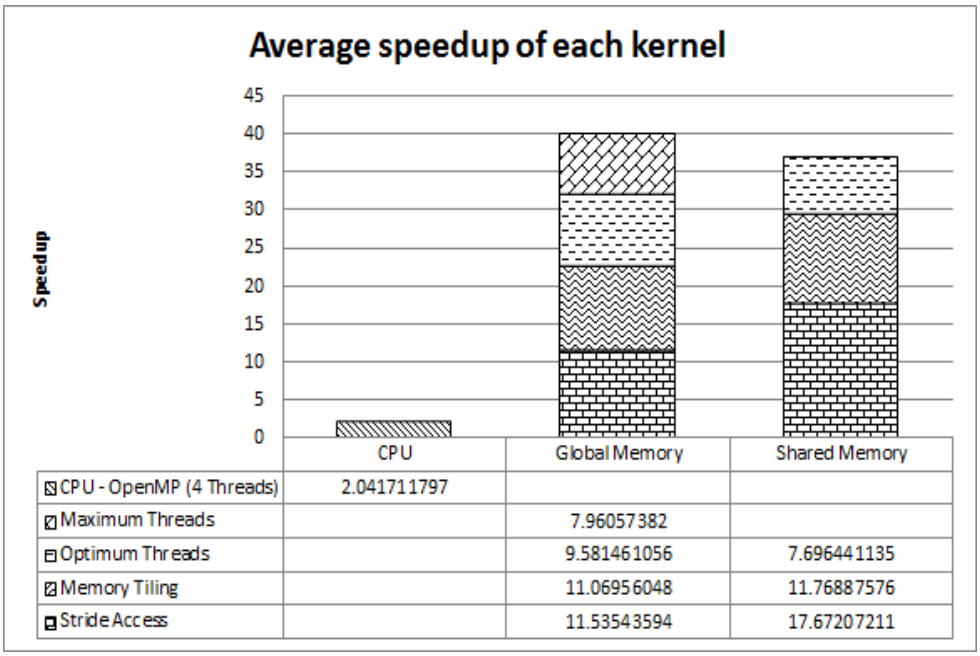


Figure 5.6: Average Speedup for Both Global and Shared Memory

The average speedup of the GPU implementation in the first optimization phase using global memory with maximum threads per block shows almost 8x speedup comparing to the CPU OpenMP implementation. Although this implementation has utilized all available CPU cores from the workstation, the sheer number of threads executed on the GPU is simply enormous. The GPU architecture is designed to be massively parallel with its thousands of much smaller CUDA cores. As an example, the NVIDIA Tesla K20 consists of 2496 cores allowing the GPU to efficiently handle multiple tasks simultaneously.

In the same global memory implementation, one of the kernel parameters is then tuned to allot 256 threads per block. The performance of this particular kernel gains 3x more speedup. Thus, this is deemed as an optimum configured block-thread parameter. Although this is a successful optimization, the Nsight

profiler has shown it has a lower occupancy rate compared to the previous kernel as shown in Figure 5.7. This leads to the realization that optimum parameters may not necessarily mean maximum usage of the GPU resources.

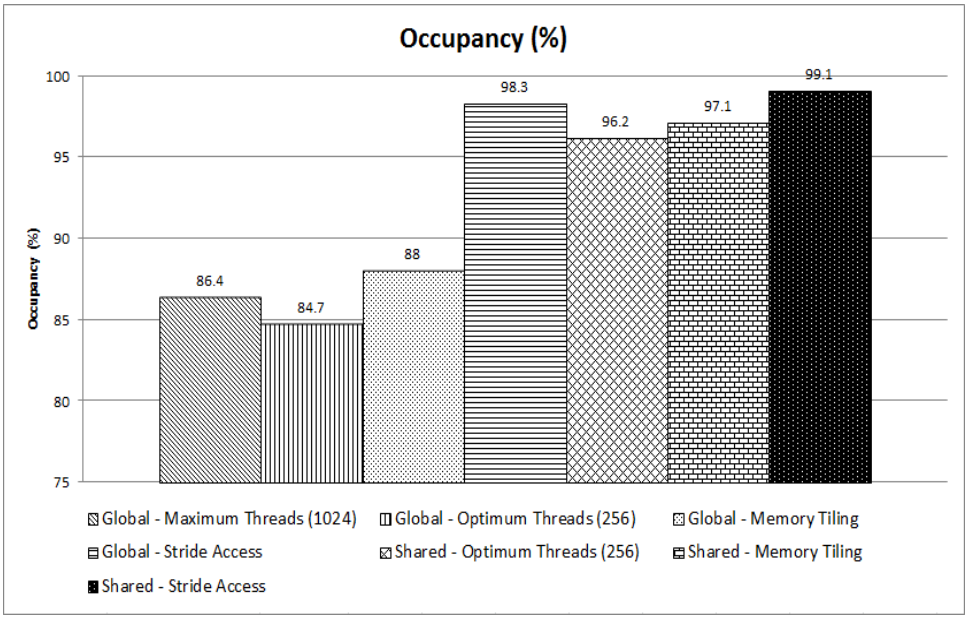


Figure 5.7: GPU Occupancy from NVIDIA Nsight

Although the occupancy has slightly reduced, the number of atomic request transactions presented in Figure 5.8 has equally reduced by at least 76 million. This reduction is a result of aligning the number of threads per block to the number of bucket array where the possibility of read-write access conflicts on the same element has declined. This seems to suggest that the reduced number of atomic request transactions has contributed to the slight increase in performance.

While the global memory tiling implementation managed to further boost the speedup up to 11 times by increasing the occupancy by slightly less than 5% compared to the global optimum threads implementation, the atomic request transactions has surged up by 3 million. Moreover, the memory tiling implementation has caused the memory access to be performed in a non-

coalescing manner as shown in Figure 5.8. Atomic requests transactions for both L1/Shared Memory and L2 Cache while computing an input data size of 1024MB. This appears the instruction level parallelism introduced within this kernel is the contributing factor in the performance enhancement.

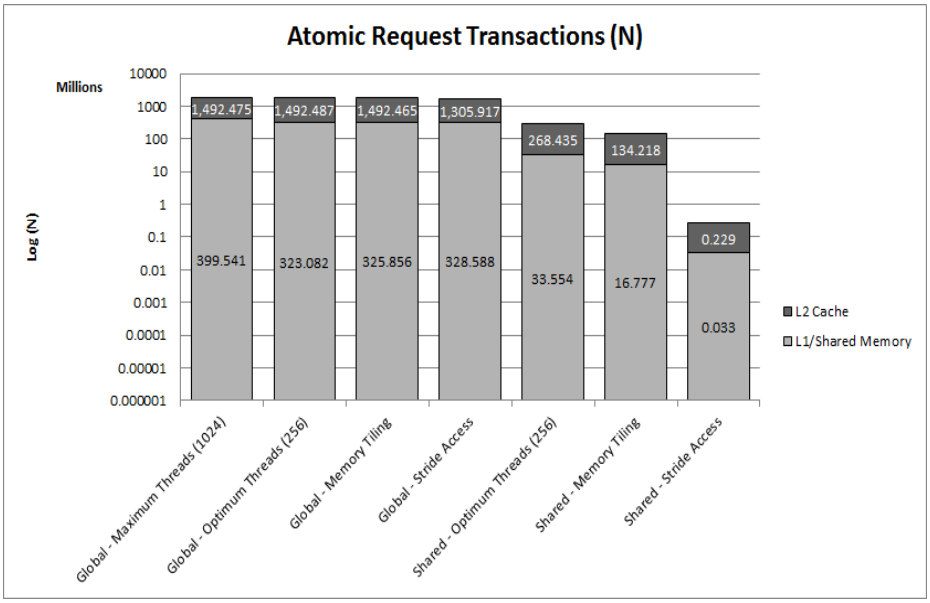


Figure 5.8: Atomic Requests Transactions

With the global stride kernel implementation where the instruction level parallelism increases by adding more computation load per thread, the occupancy has risen up to a little bit more than 98% while keeping the global load to be 100% efficient. Yet, the speedup growth is only minimal despite the fact that the atomic transactions were able to be reduced by almost 200 million. In the second GPU optimization phase, the usage of the shared memory has shown that it does not guarantee better performance. The speedup is comparable or slightly lower than the first GPU implementation (Global Memory) by maximizing the threads usage. Other than that, this kernel implementation has tremendously reduced the less favourable usage of atomic transactions to a mere total of 300 million transactions, a reduction of almost 1.5 billion.

However, in the shared memory stride access kernel implementation, the speedup has shot up to more than 17x compared to the sequential implementation. This kernel has achieved the highest occupancy of 99% and the lowest number of atomic transactions at less than 300 thousand transactions. This seems to suggest how vital is the atomic request transactions to the performance of the kernel.

Table 5.2: GPU Kernel Implementations Global Load Efficiency

Functions		**Global Load Efficiency (%)**
Global Memory	Maximum Threads	100
	Optimum Threads	100
	Memory Tiling	50
	Stride Access	100
Shared Memory	Optimum Threads	100
	Memory Tiling	50
	Stride Access	100

Figure 5.9 shows the comparison of performing the kernels on different GPU architectures. Due to the fact that these kernels are optimized on the K20c GPU card, these kernels are performing poorly on the GTX980Ti GPU card. However, a massive growth of an order in speedup is displayed by the GPU shared memory stride access implementation in comparison. The Maxwell architecture has a faster shared memory atomic operation which may have contributed to this increment. The atomic operation is implemented using a native shared memory 32/64 bit compare-and-swap (CAS) instead of the lock-

update-unlock implementation which can be found on both Fermi and Kepler architecture.

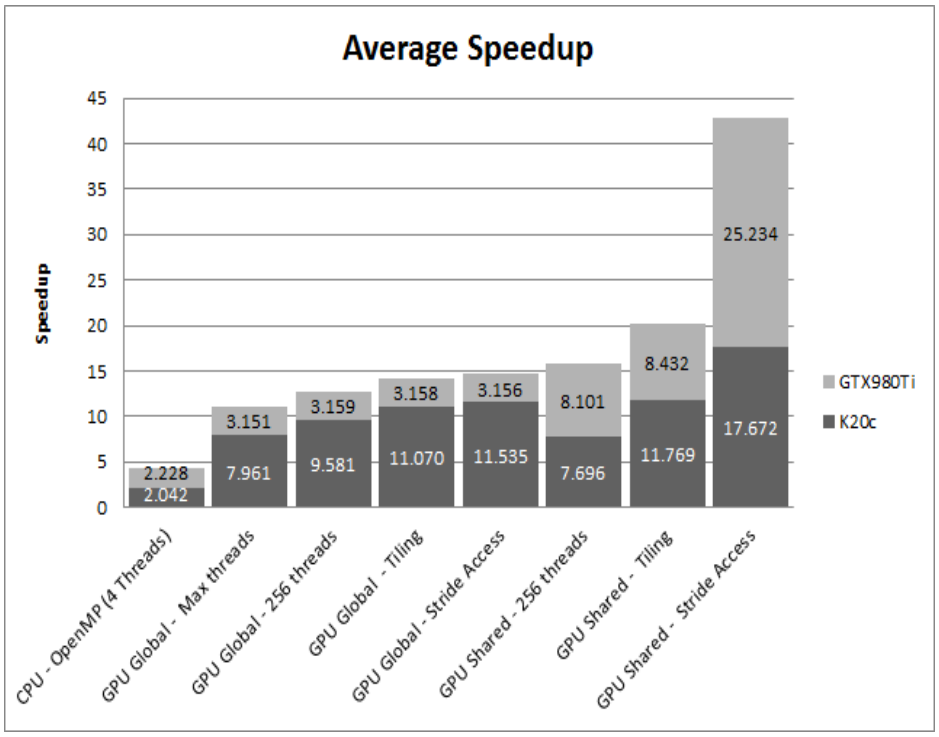


Figure 5.9: Average Speedup Of All Kernels On Both GPU Cards

Figure 5.10 shows the comparison of the overall processing time inclusive of memory transfer for the shared memory stride access kernel with and without streaming implementation. As shown here, the overall speedup of the non-streaming implementation inclusive of the memory transfer is dramatically reduced to merely 6x faster, even when using the most optimized kernel available that has almost 18x speedup. This is due to limitations on the memory transfer speed which is the key contributing factor to this reduction. Hence, it is important to implement optimization techniques such as streaming which could possibly help in diminishing this memory transfer latency and improves the overall performance.

By configuring the kernel into 4 streams, it squeezes out 1.5x speedup. It shows in the Figure 5.11. The data transfers from host to device and the kernel code executions are overlapping with each other. Therefore, the memory latency during data transfer is effectively hidden and the performance immediately has improved.

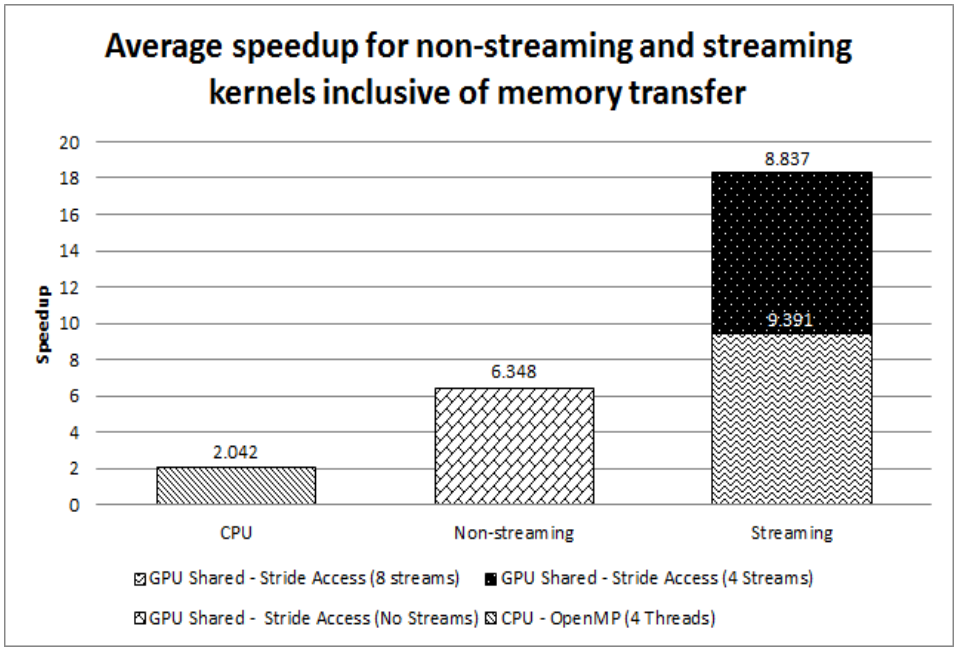


Figure 5.10: Average Speedup for Non-Streaming And Streaming

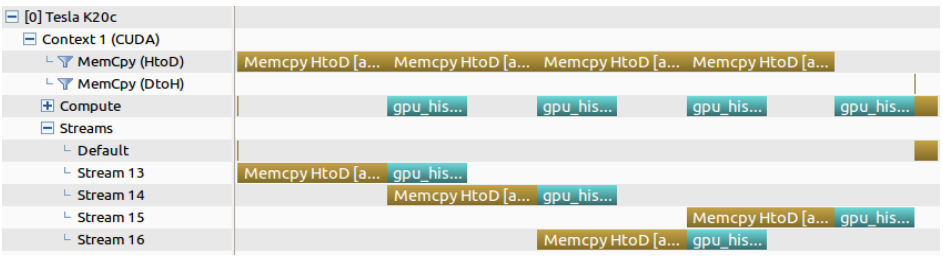


Figure 5.11: Time Profile of The Streaming Implementation (4 streams)

Nevertheless, Figure 5.12 shows that the increments of streams are the significant growth. It can be achieved more by using the GTX980Ti GPU card. In brief, streaming on the Maxwell GPU card can additional boost up 2x to 5x more speedup without streaming. In contrast, adaptation of streaming, there is

a maximum of 7x and 12x speedup, which comparing to CPU OpenMP implementation with both K20c and GTX980Ti GPU card respectively.

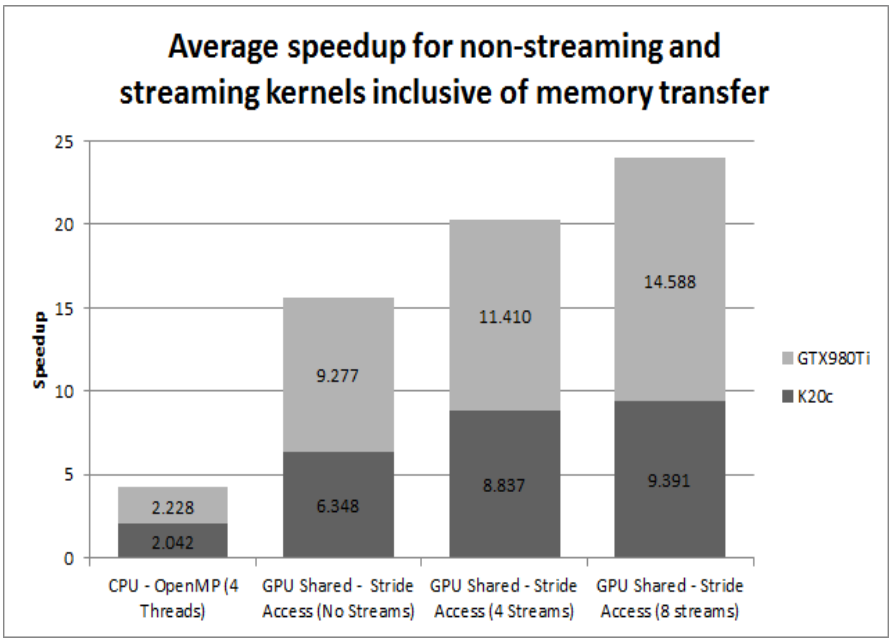


Figure 5.12: Average Speedup of K20C and GTX 980Ti

5.2.3 Analysis of Streaming Histogram Design and GPU Occupancy

This thesis evaluates various optimizations on multi-core CPU and many-core GPU for histogram algorithm. The optimization of an algorithm on the GPU architecture has different challenges compared to the CPU. It observed that the best GPU optimization uses CUDA streams. In addition, a higher amount of shared memory available can certainly boost up performance. Furthermore, there is still performance bottleneck of PCIe bus, which transferring data between host and device. One future work can be developed a tuning mechanism that automates configuring the optimum parameters for various GPU devices. It is able to approach the GPU optimization from a different angle, such as CUDA stream scheduling may need special attention for good overlapping; parallelism execution plan avoids stalls on data dependencies

in large cache usage, branch prediction and divergence. Thus, these can further squeeze out some more performance from the application.

5.3 Accelerated Bit Parallel Approximate Matching

In a recent study, Tran and colleagues (Tran *et al.*, 2016) demonstrated that graphics processing units (GPUs) may be used to speed up the Wu-Manber method. The goal of this thesis is to make their implementation approach more efficient so that it can do rapid pattern matching for short patterns (P). After providing a concise overview of the implementation strategies suggested by (Tran *et al.*, 2016), then, this move on to discussing the method that this thesis proposed method.

After a concise overview of the implementation strategies that (Tran *et al*., 2016) propose, the chapter then discusses the method developed in this thesis.

5.3.1 Implementation Techniques in GPU

T is first divided into multiple sub-strings, whereby each warp in GPU can perform APM for pattern P against one substring. Since many warps are performing APM in parallel, this his essentially an embarrassingly parallel problem, which is very suitable for GPU acceleration. (Tran *et al.*, 2016) also proposed to utilized one warp as 1024-bit vector to search for one pattern (P), wherein each thread within a warp holds a 32-bit variable (i.e. 32 threads Å~ 32-bit = 1024-bit vector). Within each warp, most of the bitwise operations can be executed independently within a thread, except the left shift operation that involve memory operation. The left shift operation is implemented using

warp_shuffle instruction provided by GPU architecture (i.e. Kepler, Maxwell and Pascal).

The NFA in Wu-Manber APM is represented in a matrix form, which is transferred from CPU to GPU global memory for processing. Instead of updating the NFA matrix directly on global memory, the (Tran *et al.*, 2016) proposed tiling method to buffer the frequently accessed data in registers. This can effectively reduce the slow access to global memory, which eventually improved the overall performance for pattern matching. For detail descriptions on the implementation, this refers the readers to the original article (Tran *et al.*, 2016).

Figure 5.13: Sync - Pattern Matching (GPU)

Referring to Figure 5.13, the database text (T), pattern (P), error (k) and all relevant is first copied to global memory in GPU for processing, followed by pattern matching in GPU, then results are copied back to CPU. Note that this process is executed in strictly serial manner; it is used by (Tran *et al.*, 2016) and it names this as sync.

In this proposed implementation, it is to first divide the workload in GPU pattern matching into multiple parts and assign them into different CUDA streams. Once the first stream completes the pattern matching in GPU, it immediately copies back the data to CPU for post-processing; at the same time the GPU continues to perform next pattern matching. This overlapping of workload is possible because GPU has independent execution and memory

engine; kernel execution and memory copy can take place at the same time. This thesis refers this proposed technique as async.

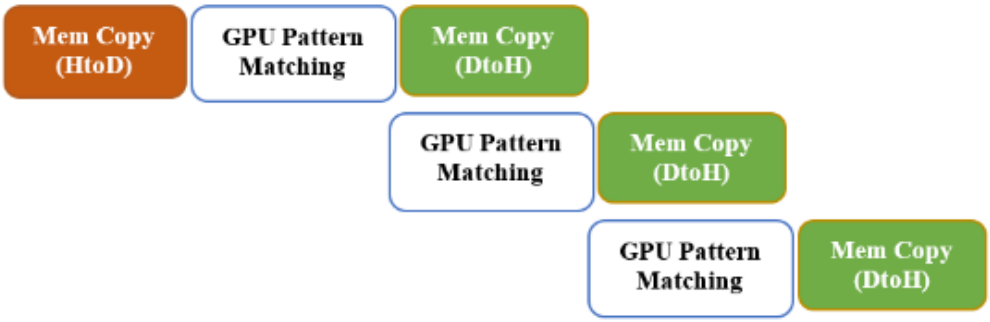


Figure 5.14: Async - Overlapping Pattern Matching (4 Streams)

When the pattern matching phase produces many matches, the synchronous (sync) method retains all results on the device until kernel execution finishes, then performs a single bulk transfer of the result buffers to the CPU. In contrast, the asynchronous (async) method initiates result transfers earlier and overlaps them with ongoing GPU execution, which yields clear benefits when the match density is high and buffers fill quickly. Under sparse match conditions, however, the overlapping effect becomes weak, and the additional stream management overhead may offset any gains, so overall time does not improve. For this reason, the technique targets short patterns, where the probability of finding many matches is higher and the async pipeline can keep multiple streams busy, hide transfer latency more effectively and deliver more stable speedups across different GPU architectures.

5.3.2 Experimental Evaluation of Asynchronous Wu–Manber for Short Patterns

The performance of the proposed implementation is evaluated on three GPU platforms: K40c, GTX980 and GTX1080. The host system uses an Intel i7-4790K quad core CPU (eight hardware threads) with 32 GB RAM and runs

Windows 8.1. This configuration represents a typical high end desktop environment and provides a consistent baseline for comparing architectural differences across the evaluated GPUs. This thesis has used the part of the human chromosome 21 which has 33554432 characters (i.e., combination of C, T, G and A). The short patterns (p) are generated randomly with sizes varying from 8-128 characters. Patterns longer than 128 characters are not evaluated as the aim of this paper is to accelerate the speed performance of short patterns. Different allowed errors (k), range from 2-8, are also evaluated in the experiments. However, cases for k>8 are not evaluated as it is larger than the shortest pattern (p=8) in our experiments. This section reports results for four streams, which give the best performance in the experiments. Increasing the number of independent streams does not yield additional speedup and can introduce extra scheduling and memory management overhead, so the analysis focuses on this configuration.

Figure 5.15 to Figure 5.20 report the timing performance of the APM kernels on the K40c, GTX980 and GTX1080, including both kernel execution and copying results back to the CPU. The implementation by (Tran *et al.*, 2016) appears as the sync baseline, while the proposed solution appears as async. Across all three GPUs, async generally reduces end to end time, and the newer devices amplify this benefit by providing higher memory bandwidth and more cores. The figures also highlight that architectural improvements across generations and the use of overlapping transfers jointly determine the overall throughput of the APM pipeline.

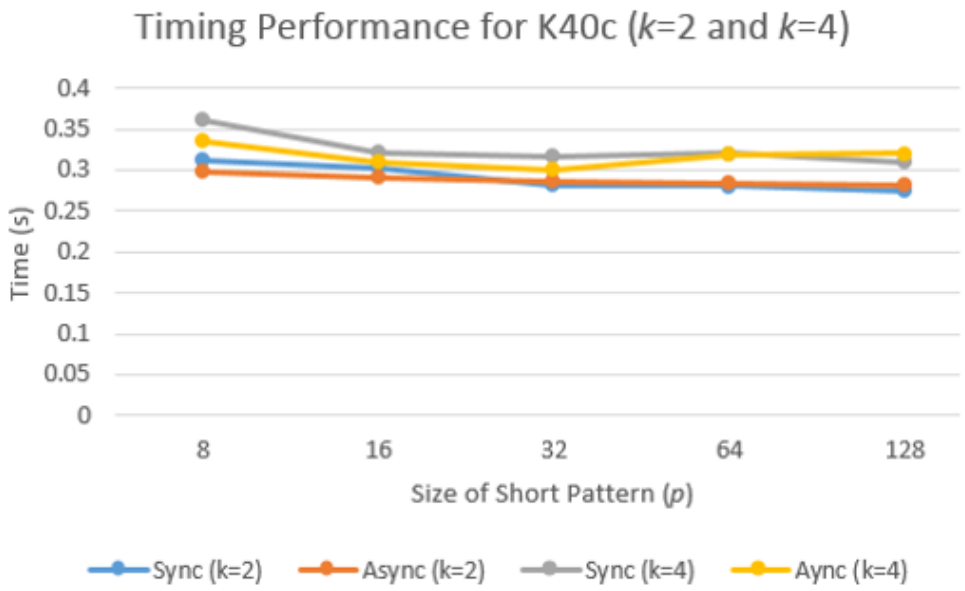


Figure 5.15: Timing performance for APM in K40c with k=2 and 4.

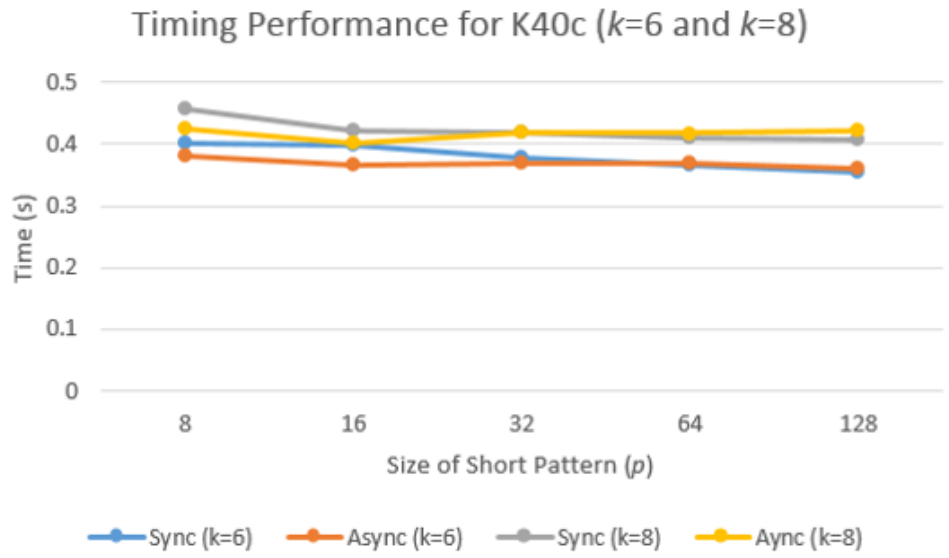


Figure 5.16: Timing performance for APM in K40c with k=6 and 8.

Figure 5.15 to Figure 5.16, it is observed that the pattern matching with stream programming model (*async*) is always faster than the implementation by (Tran *et al.*, 2016) (*sync*) for small patterns ($p \leq 16$). For larger patterns ($p > 16$), the performance of *async* drops because it is more difficult to find the desired patterns, so the size of results also reduced, which in turn jeopardised the benefit

of using multiple streams. In other words, *async* is only good to be used for pattern matching in short patterns as it is easy to find many matching patterns.

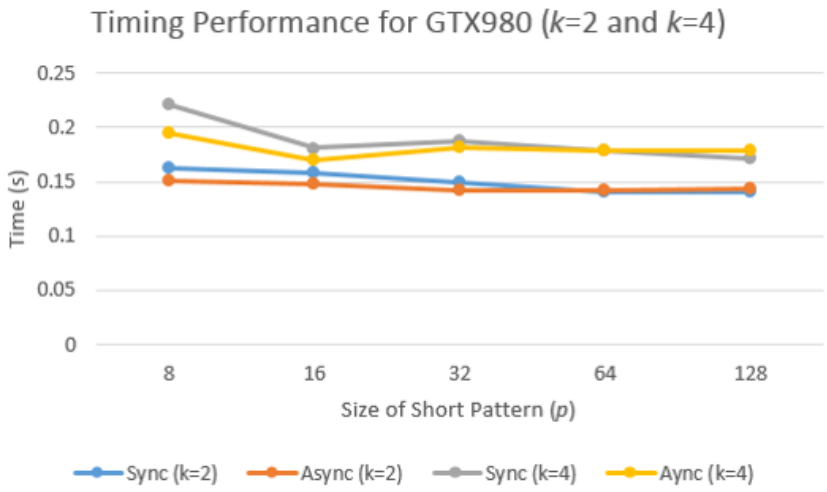


Figure 5.17: Timing performance for APM in GTX980 with k=2 and 4.

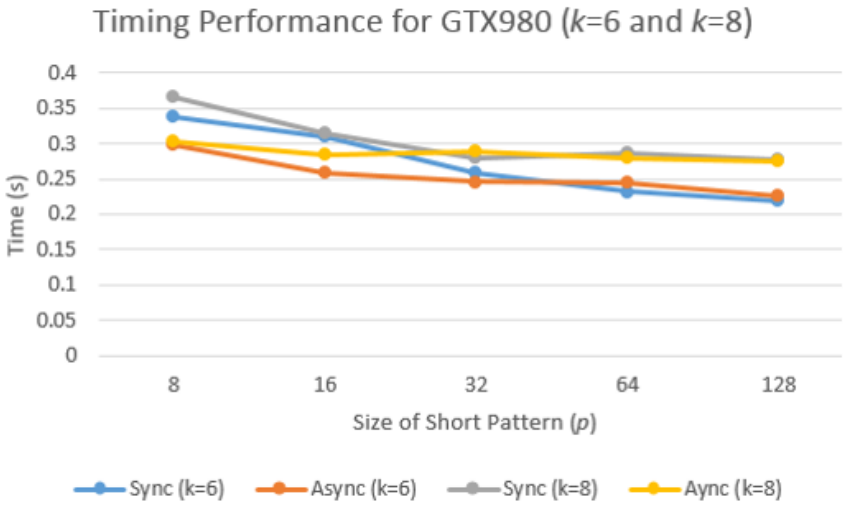


Figure 5.18: Timing performance for APM in GTX980 with k=6 and 8.

Figures 5.17 and 5.18, as well as Figures 5.18 and 5.19, exhibit the same qualitative trend: the proposed asynchronous implementation consistently outperforms the implementation by (Tran *et al.*, 2016) for small patterns ($p \leq 16$) across all reported pattern lengths. In each figure, the async configuration achieves lower execution times than both the synchronous variant and the baseline, while preserving the relative ranking of the three GPU platforms. The

GTX1080 with Pascal architecture delivers the best timing results, reflecting the benefits of its newer architecture. The GTX980 (2048 cores) is faster than the K40c (2880 cores) even though it has fewer cores, because the Maxwell architecture provides higher instruction throughput and larger shared memory. Table 5.3 summarises the performance gains of async over sync, quantifying the improvements that are visually evident in Figure 5.17 to Figure 5.19.

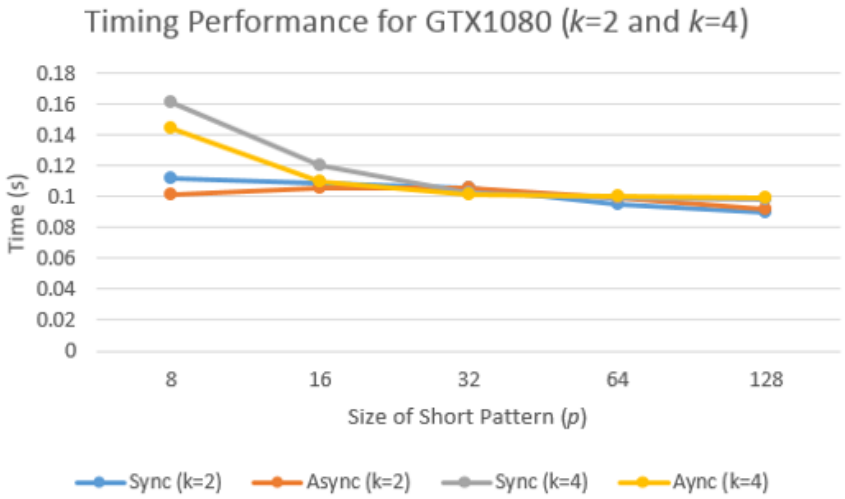


Figure 5.19: Timing performance for APM in GTX1080 with k=2 and 4.

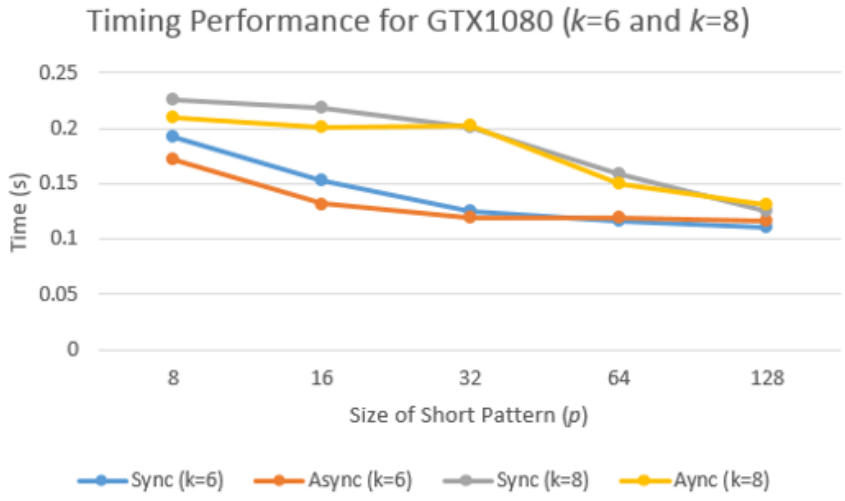


Figure 5.20: Timing performance for APM in GTX1080 with k=6 and 8.

In Table 5.3, this thesis also observes that as *k* increases the benefit of using this proposed technique *async* is more obvious. For example, the

performance gain for pattern size p=8 is increased from 7.3% (k=2) to 17.15% (k=8). This is due to the fact that there are more results obtained when the permitted error (k) increases, so the benefit of utilizing multiple streams is also increases.

Table 5.3: Summary Of Performance Gain For Async Against Sync

NVIDIA TESLA GPU	Pattern Size (p)	Performance Gain (%)			
		k=2	k=4	k=6	k=8
K40c	8	**7.3**	**11.98**	**11.6**	**17.15**
	16	**6.3**	**5.97**	**16.44**	**9.99**
	32	5.07	3.36	5.06	-3.44
	64	-0.78	-0.17	-5.83	2.31
	128	-2.86	-4.68	-3.2	0.29
GTX980	8	**9.26**	**10.24**	**11.26**	**16.39**
	16	**2.3**	**9.18**	**13.6**	**8.05**
	32	-0.09	1.17	4.81	-0.3
	64	-4.83	-0.3	-2.94	5.42
	128	-2.78	-1.11	-4.71	-5.29
GTX1080	8	**4.26**	**7.34**	**10.68**	**10.47**
	16	**3.32**	**3.7**	**8.26**	**4.84**
	32	-1.35	4.53	2.25	-0.29
	64	-1.54	0.03	-0.99	-1.83
	128	-2.69	-3.73	-1.24	-3.69

5.3.3 Implications of Approximate Matching for GPU Query Pipelines

Most of the previous attempts to implement APM in GPU focuses on accelerating the speed performance for large patterns (Xu *et al.*, 2013; Tran *et al.*, 2016; Ho *et al.*, 2017). However, the pattern matching for short patterns is equally important. This thesis presents a new technique to improve the speed performance of APM (Wu-Manber algorithm) implementation in GPU. This thesis implementation has shown the improvement of the speed of pattern

matching for small patterns (p≤16) by 3.32% - 17.15% on K40c, GTX980 and GTX1080 respectively, compared to the previous state-of-the-art implementation by (Tran *et al.*, 2016).

5.4 Summary of GPU-Accelerated String Matching for Query Predicates

Most of the previous attempts to implement approximate pattern matching (APM) on GPUs, such as the design by (Tran *et al.*, 2016), have focused on maximising throughput for long patterns and large pattern sets. In contrast, this thesis targets APM performance for short patterns in GPU based query pipelines, where predicates typically use relatively short string values. The experimental results in Figures 5.15 to 5.20 show that, for patterns with length $p \leq 16$, the asynchronous implementation (async) consistently outperforms the synchronous baseline (sync) on all three evaluated GPUs (K40c, GTX980 and GTX1080). The corresponding percentage improvements are summarised in Table 5.3, where the async version achieves between 3.32% and 17.15% lower execution time for small patterns compared with the implementation by (Tran *et al.*, 2016), with the largest gain observed for $p = 8$ and $k = 8$. These results indicate that overlapping kernel execution and data transfer via multiple CUDA streams is an effective strategy for approximate matching on short patterns, and suggest that this technique is well suited for integration into GPU-accelerated query pipelines that require APM on short string predicates.

CHAPTER 6

CONCLUSION

This final chapter brings the study back to its starting point. Chapter 1 motivated the use of GPU assisted structured query processing and posed three research questions about when a single GPU can deliver reliable benefits over a tuned CPU database and a small Hadoop cluster, how compression and data layout interact with GPU execution, and whether text oriented predicates can remain on the device without eroding those gains. Chapter 2 mapped the landscape of heterogeneous query processing and concluded, in Section 2.6, that existing work leaves open the integration of compression, joins and string predicates in one accelerator, the understanding of lightweight schemes such as PFOR-DELTA in complete query plans, and the system level comparison of a single GPU accelerator with mainstream CPU and big data platforms.

Chapters 3, 4 and 5 turned these open points into concrete designs and measurements. Chapter 3 introduced a GPU structured query accelerator that couples a compression first pre-processing path with chunked columnar storage and GPU friendly hash and sort merge joins, and defined the control points that make its behaviour observable. Chapter 4 moved from design to deployment by running a subset of TPC H queries at several scale factors and six healthcare queries on both a GPU workstation and a Hadoop based cluster with PostgreSQL as the CPU baseline, showing how the accelerator behaves under realistic data engineering conditions. Chapter 5 extended the same pipeline to short string predicates by adapting hash based matching, histogram based filtering and bit

parallel approximate matching so that LIKE style filters can remain on the GPU rather than forcing a return to host execution. Together, these chapters provide an integrated view of how a single GPU, combined with lightweight compression, can act as a practical accelerator for structured queries.

The role of this chapter is not to introduce new mechanisms, but to interpret what these results mean for the original problem statement and the research questions. It summarises the main findings, revisits the questions from Chapter 1 in the light of the evidence obtained, and clarifies the scope and limits of the conclusions. The chapter also reflects on the assumptions built into the accelerator, identifies the types of workloads and deployment settings where the approach is most suitable, and discusses situations where alternative designs such as multi-GPUs configurations or purely CPU based engines may be preferable. It concludes by outlining several directions for future work that follow naturally from the strengths and limitations observed in this study.

6.1 Summary of Contributions

This thesis makes several contributions to GPU based structured query processing. First, it develops and evaluates a GPU centric query processing architecture and engine. The thesis shows how to organize query operators and data movement so that GPUs act as primary processors rather than simple coprocessors, and demonstrates that a carefully tuned GPU-accelerated engine can serve as an alternative to traditional CPU centric databases for selected analytical workloads. Second, it investigates data layout and compression for column-based stores on GPUs by evaluating GPU friendly encodings and compression schemes. The results show how these schemes reduce storage cost

while maintaining or improving query throughput, and they characterise the tradeoff between PCIe transfer overhead and the benefits of faster scans once data resides in GPU memory. Third, the thesis redesigns relational join operators for GPUs, presenting join variants that exploit fine grained parallelism while remaining compatible with conventional SQL processing so they can be integrated into a practical query engine without changing the relational abstraction.

The thesis also contributes experimental evidence for GPU based geospatial and big data analytics, and for histogram based analytical patterns. In the geospatial study, the GPU-accelerated query engine uses a column-oriented file structure with the CUDASET representation to store spatial attributes in a structure of arrays layout, which improves memory coalescing and supports efficient offloading of filtering and aggregation to the GPU. On top of this representation, the thesis implements GPU kernels for aggregation, filtering and spatial operations and integrates them into an end-to-end analytical pipeline. The evaluation with large and complex data sets shows that this CUDASET based pipeline can process geospatial and big data workloads effectively when the full stack is tuned for GPU execution. In addition, the thesis studies histogram-based analytics on multi core CPUs and many core GPUs. It compares optimisation techniques across these platforms and shows that GPUs can deliver substantial gains for histogram computation even when PCIe transfer costs are taken into account, highlighting histograms as an important analytical pattern that benefits from heterogeneous hardware.

A further set of contributions lies in GPU based string processing for query predicates. The thesis proposes and evaluates a hash based exact string-matching mechanism for GPUs, and shows in the reported experiments that it outperforms baseline algorithms such as Boyer Moore Horspool and column search by aligning control flow and memory access with massively parallel execution. It also presents a GPU based implementation of the Wu–Manber approximate pattern matching algorithm that targets short patterns and uses multiple CUDA streams to overlap kernel execution and data transfer. Experiments on several GPU models show measurable speed improvements for small patterns compared with baseline implementations.

6.2 Recommendations For Future Work

To further improve the performance of GPU-accelerated query processing, there are three major areas can be scrutinized.

6.2.1 Multi-GPUs Acceleration for Query Processing Techniques

In light of the identified intricacies and performance constraints in multi-GPUs environments, the creation of advanced algorithms capable of effectively overseeing data partitioning, load balancing, and inter-GPU communication is of the utmost importance. Further investigation is warranted into the development of scalable and resilient frameworks capable of seamlessly integrating with established distributed computing systems, such as Apache Spark or Hadoop. This integration would enable the processing of massive amounts of data across numerous GPU nodes.

6.2.2 Enhanced Parallelism With Leveraging GPU Advancement

The potential of GPU acceleration in database query processing has been clearly demonstrated. However, the underutilization of GPU resources suggests ample room for optimization. Future work should aim to develop recursive child kernel functions and continuous streaming mechanisms that maximize the use of CUDA cores for parallel processing tasks.

6.2.3 Integration of GPU-Accelerated String Matching With MPI Systems

The Hash Match mechanism has shown promise in accelerating string matching tasks on GPUs. To build upon this success, future research should investigate the integration of GPU-accelerated string matching algorithms with Message Passing Interface (MPI) distribution systems, leveraging GPU Direct for efficient data transfer and communication. This integration could lead to significant improvements in the performance of distributed applications that require high-throughput string matching capabilities.

BIBLIOGRAPHY

Abadi, D., Madden, S. and Ferreira, M. (2006) 'Integrating compression and execution in column-oriented database systems', Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM, pp. 671–682. Available at: https://dl.acm.org/doi/10.1145/1142473.1142548 (Accessed: 3 November 2025).

Adshead, A. (2014) Data set to grow 10-fold by 2020 as internet of things takes off. Available at: http://www.computerweekly.com/news/2240217788/Data-set-to-grow-10-fold-by-2020-as-internet-of-things-takes-off (Accessed: 11 November 2025).

Aho, A.V. and Corasick, M.J. (1975) 'Efficient string matching', Communications of the ACM, 18(6), pp. 333–340. Available at: https://dl.acm.org/doi/10.1145/1142473.1142548 (Accessed: 3 November 2025).

Alba, D. (2013) 'Diamond blankets will keep future chips cool', IEEE Spectrum. Available at: https://spectrum.ieee.org/diamond-thermal-conductivity (Accessed: 11 November 2025).

Alcantara, D.A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owen, J.D. and Amenta, N. (2009) 'Real-time parallel hashing on the GPU', ACM Transactions on Graphics, 28(5), pp. 154:1–154:9. Available at: https://dl.acm.org/doi/10.1145/1618452.1618500.

Anupama, C.G. and Lakshmi, C. (2022) 'A comprehensive review on data partitioning and sampling techniques for processing big data', Proceedings of the International Conference on Power, Energy, Control and Transmission Systems (ICPECTS). Chennai, India: IEEE, pp. 1–6. Available at: https://doi.org/10.1109/ICPECTS56089.2022.10047766 (Accessed: 5 November 2025).

Arora, M. (2012) The architecture and evolution of CPU-GPU systems for

general purpose computing. University of California, San Diego. Available at: https://cseweb.ucsd.edu/~marora/files/papers/REReport_ManishArora.pdf (Accessed: 16 November 2025).

Bachman, C.W. (1973) 'The programmer as navigator', Communications of the ACM, 16(11), pp. 653–658.

Bakkum, P. and Chakradhar, S. (2012) Efficient data management for GPU databases. Available at: https://www.eecis.udel.edu/~cavazos/cisc879/papers/datamanagement.pdf (Accessed: 16 November 2025).

Bakkum, P. and Skadron, K. (2010) 'Accelerating SQL database operations on a GPU with CUDA', Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. Pittsburgh, USA: ACM, pp. 94–103. Available at: http://doi.acm.org/10.1145/1735688.1735706 (Accessed: 6 November 2025).

Barber, R., Lohman, G., Pandis, I., Raman, V., Sidle, R., Attaluri, G., Chainani, N., Lightstone, S. and Sharpe, D. (2014) 'Memory-efficient hash joins', Proceedings of the VLDB Endowment, 8(4), pp. 353–364. Available at: https://dl.acm.org/doi/10.14778/2735496.2735499 (Accessed: 8 November 2025).

Bard, C.M. and Dorelli, J.C. (2014) 'A simple GPU-accelerated two dimensional MUSCL Hancock solver for ideal magnetohydrodynamics', Journal of Computational Physics, 259, pp. 444–460.

Battersby, S.E., Finn, M.P., Usery, E.L. and Yamamoto, K.H. (2014) 'Implications of Web Mercator and its use in online mapping', Cartographica, 49(2), pp. 85–101.

Bauke, H. and Keitel, C.H. (2011) 'Accelerating the Fourier split operator method via graphics processing units', Computer Physics Communications,

182(12), pp. 2454–2463.

Baxter, S. (2017) ModernGPU - patterns and behaviors for GPU computing. Available at: https://github.com/moderngpu/moderngpu (Accessed: 13 November 2025).

Begoli, E., Patel, P. and Christian, J.B. (2016) 'Storage and read-optimized data placement structures for high-performance analysis', in Optimization Challenges in Complex, Networked and Risky Systems. Catonsville: INFORMS, pp. 171–184. Available at: https://doi.org/10.1287/educ.2016.0143 (Accessed: 9 November 2025).

Behrens, T., Rosenfeld, V., Traub, J., Breß, S. and Markl, V. (2018) 'Efficient SIMD vectorization for hashing in OpenCL', Proceedings of the 21st International Conference on Extending Database Technology (EDBT). Vienna: OpenProceedings.org, pp. 489–492. Available at: https://doi.org/10.5441/002/edbt.2018.54 (Accessed: 5 November 2025).

Boeschen, N., Ziegler, T. and Binnig, C. (2024) 'GOLAP: a GPU-in-data-path architecture for high-speed OLAP', Proceedings of the ACM on Management of Data, 2(6), pp. 1–26. Available at: https://doi.org/10.1145/3698812 (Accessed: 9 November 2025).

Boncz, P.A., Manegold, S. and Kersten, M.L. (1999) 'Database architecture optimized for the new bottleneck: memory access', Proceedings of the 25th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., pp. 54–65.

Boncz, P.A., Kersten, M.L. and Manegold, S. (2008) 'Breaking the memory wall in MonetDB', Communications of the ACM, 51(12), pp. 77–85. Available at: https://dl.acm.org/doi/10.1145/1409360.1409380 (Accessed: 7 November 2025).

Bordawekar, R.R. and Sadoghi, M. (2016) 'Accelerating database workloads by software-hardware-system co-design', Proceedings of the 32nd International Conference on Data Engineering (ICDE). Piscataway: IEEE, pp. 1428–1431.

Available at: https://doi.org/10.1109/ICDE.2016.7498362 (Accessed: 6 November 2025).

Boyer, R.S. and Moore, J.S. (1977) 'A fast string searching algorithm', Communications of the ACM, 20(10), pp. 762–772.

Breß, S. and Saake, G. (2013) 'Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS', Proceedings of the 40th International Conference on Very Large Data Bases. New York: ACM, pp. 1398–1403. Available at: https://doi.org/10.14778/2536274.2536325 (Accessed: 3 November 2025).

Breß, S., Funke, H. and Teubner, J. (2016) 'Robust query processing in co-processor-accelerated databases', Proceedings of the International Conference on Management of Data (SIGMOD'16). New York: ACM, pp. 1891–1906. Available at: https://doi.org/10.1145/2882903.2882936 (Accessed: 12 November 2025).

Breß, S., Heimel, M., Siegmund, N., Bellatreche, L. and Saake, G. (2014a) 'Exploring the design space of a GPU-aware database architecture', in Catania, B. et al. (eds) New Trends in Databases and Information Systems. New York: Springer, pp. 225–234. Available at: http://link.springer.com/10.1007/978-3-319-01863-8_25 (Accessed: 8 November 2025).

Breß, S., Heimel, M., Siegmund, N., Bellatreche, L. and Saake, G. (2014b) 'GPU-accelerated database systems: survey and open challenges', in Transactions on Large-Scale Data- and Knowledge-Centered Systems XV. Berlin Heidelberg: Springer, pp. 1–35. Available at: http://link.springer.com/10.1007/978-3-662-45761-0_1 (Accessed: 3 November 2025).

Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M. and Storaasli, O.O. (2010) 'State-of-the-art in heterogeneous computing', Journal of Scientific Programming, 18(1), pp. 1–33. Available at:

https://doi.org/10.1155/2010/540159 (Accessed: 13 November 2025).

Burrows, M. and Wheeler, D.J. (1994) A block-sorting lossless data compression algorithm. Maynard: Digital Equipment Corporation.

Calarasanu, S., Fabrizio, J. and Dubuisson, S. (2015) 'Using histogram representation and Earth Mover's Distance as an evaluation tool for text detection', Proceedings of the 13th International Conference on Document Analysis and Recognition (ICDAR). New York: ACM, pp. 221–225.

Cantone, D. and Faro, S. (2009) 'Pattern matching with swaps for short patterns in linear time', in Nielsen, M. et al. (eds) Lecture Notes in Computer Science. Berlin: Springer, pp. 255–266. Available at: http://link.springer.com/10.1007/978-3-540-95891-8_25 (Accessed: 2 November 2025).

Carey, M.J. (2013) 'BDMS performance evaluation: practices, pitfalls, and possibilities', in Nambiar, R. and Poess, M. (eds) Selected Topics in Performance Evaluation and Benchmarking (TPCTC 2012). Berlin: Springer, pp. 108–123.

Chavan, H., Alghamdi, R. and Mokbel, M.F. (2016) 'Towards a GPU-accelerated spatial computing framework', Proceedings of the IEEE 32nd International Conference on Data Engineering Workshops (ICDEW). Piscataway: IEEE, pp. 135–142. Available at: http://ieeexplore.ieee.org/document/7495634 (Accessed: 6 November 2025).

Chitters, V., Midvidy, A. and Tsui, J. (2013) Reducing synchronization overhead using hardware transactional memory. Berkeley: University of California. Available at: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project6_report_ver2.pdf (Accessed: 19 November 2025).

Chrysogelos, P., Sioulas, P. and Ailamaki, A. (2019) 'Hardware-conscious query processing in GPU-accelerated analytical engines', Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR). Available

at: http://infoscience.epfl.ch/record/262529 (Accessed: 2 November 2025).

Cloud, R.L., Curry, M.L., Ward, H.L., Skjellum, A. and Bangalore, P. (2011) 'Accelerating lossless data compression with GPUs', Journal of the University of Alabama, 3, pp. 26–29.

CompressionRatings.com (no date) Download. Compression Ratings. Available at: http://www.compressionratings.com/download.html (Accessed: 20 November 2025).

Cremonesi, P. and Sorrenti, D. (1995) 'A control architecture for managing instructions among partitions of a data parallel structure', Proceedings Euromicro Workshop on Parallel and Distributed Processing. Piscataway: IEEE, pp. 262–269. Available at: https://doi.org/10.1109/EMPDP.1995.389135 (Accessed: 10 November 2025).

Das, J. and Wilton, S.J.E. (2011) 'Accelerated FPGA architecture design: capabilities and limitations of analytical models', Proceedings of the International Conference on Field-Programmable Technology. Piscataway: IEEE, pp. 1–8. Available at: https://doi.org/10.1109/FPT.2011.6132684 (Accessed: 11 November 2025).

Dawes, B. and Abrahams, D. (2012) Boost C++ Libraries. Available at: https://www.boost.org (Accessed: 12 November 2025).

Dennard, R.H., Gaensslen, F.H., Yu, H.N., Rideout, V.L., Bassous, E. and LeBlanc, A.R. (1974) 'Design of ion-implanted MOSFET's with very small physical dimensions', Journal of Solid-State Circuits, 9(5), pp. 256–268. Available at: https://doi.org/10.1109/JSSC.1974.1050511 (Accessed: 3 November 2025).

Doraiswamy, H., Vo, H.T., Silva, C.T. and Freire, J. (2016) 'A GPU-based index to support interactive spatio-temporal queries over historical data', Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE). Piscataway: IEEE, pp. 1086–1097. Available at:

https://ieeexplore.ieee.org/document/7498315 (Accessed: 21 November 2025).

Drozd, A., Maruyama, N. and Matsuoka, S. (2012) 'Sequence alignment on massively parallel heterogeneous systems', Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. Piscataway: IEEE, pp. 2498–2501. Available at: https://doi.org/10.1109/IPDPSW.2012.311 (Accessed: 12 November 2025).

Erdem, O. (2016) 'Tree-based string pattern matching on FPGAs', Computers & Electrical Engineering, 49, pp. 117–133. Available at: https://doi.org/10.1016/j.compeleceng.2015.11.025 (Accessed: 19 November 2025).

Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K. and Burger, D. (2011) 'Dark silicon and the end of multi-core scaling', Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA). Piscataway: IEEE, pp. 365–376.

Fang, W., He, B. and Luo, Q. (2010) 'Database compression on graphics processors', Proceedings of the VLDB Endowment, 3(1–2), pp. 670–680. Available at: https://dl.acm.org/doi/10.14778/1920841.1920927 (Accessed: 5 November 2025).

Fechner, B. (2010) 'GPU-based parallel signature scanning and hash generation', Proceedings of the 23rd Architecture of Computing Systems (ARCS), International Conference on. Piscataway: IEEE, pp. 1–6.

Furst, E., Oskin, M. and Howe, B. (2017) 'Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries', Proceedings of the 13th International Workshop on Data Management on New Hardware (DaMoN'17). New York: ACM, pp. 3:1–3:6. Available at: https://doi.org/10.1145/3076113.3076119 (Accessed: 12 November 2025).

Gao, C., Baig, F., Vo, H., Zhu, Y. and Wang, F. (2018) 'Accelerating cross-matching operation of geospatial datasets using a CPU-GPU hybrid platform', Proceedings of the IEEE International Conference on Big Data (Big Data).

Piscataway: IEEE, pp. 3402–3411. Available at: https://doi.org/10.1109/BigData.2018.862260 (Accessed: 3 November 2025).

George, A.V., Manoj, S., Gupte, S.R., Mitra, S. and Sarkar, S. (2017) 'Thrust++: extending Thrust framework for better abstraction and performance', Proceedings of the 24th International Conference on High Performance Computing (HiPC). Piscataway: IEEE, pp. 368–377. Available at: https://doi.org/10.1109/HiPC.2017.00049 (Accessed: 16 November 2025).

Goldstein, J., Ramakrishnan, R. and Shaft, U. (1998) 'Compressing relations and indexes', Proceedings of the 4th International Conference on Data Engineering. Piscataway: IEEE, pp. 370–379. Available at: https://doi.org/10.1109/ICDE.1998.655800 (Accessed: 3 November 2025).

Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M. and Manocha, D. (2004) 'Fast computation of database operations using graphics processors', Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (DaMoN). New York: ACM, pp. 215–226. Available at: https://doi.org/10.1145/1007568.1007594 (Accessed: 3 November 2025).

Guo, G.X., Qiu, S., Ye, Z.Q., Wang, B.Q., Fang, L. and Lu, M. (2013) 'GPU-accelerated adaptive compression framework for genomics data', Proceedings of the IEEE International Conference on Big Data. Piscataway: IEEE, pp. 181–186. Available at: https://doi.org/10.1109/BigData.2013.6691572 (Accessed: 13 November 2025).

Guthe, S. and Goesele, M. (2016) 'GPU-based lossless volume data compression', Proceedings of The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON). Piscataway: IEEE, pp. 1–4. Available at: https://doi.org/10.1109/3DTV.2016.7548892 (Accessed: 12 November 2025).

Hamilton, S. (2003) 'Intel research expands Moore's Law', Computer, 36(1), pp. 31–40. Available at: https://doi.org/10.1109/MC.2003.1160054 (Accessed: 3 November 2025).

Hasib, A.A., Cebrian, J.M. and Natvig, L. (2015) 'V-PFORDelta: data

compression for energy efficient computation of time series', Proceedings of the 22nd International Conference on High Performance Computing (HiPC). Piscataway: IEEE, pp. 416–425. Available at: https://doi.org/10.1109/HiPC.2015.11 (Accessed: 17 November 2025).

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q. and Sander, P.V. (2009) 'Relational query coprocessing on graphics processors', ACM Transactions on Database Systems, 34(4), pp. 1–39. Available at: https://doi.org/10.1145/1620585.1620588 (Accessed: 17 November 2025).

He, J., Lu, M. and He, B. (2013) 'Revisiting co-processing for hash joins on the coupled CPU-GPU architecture', Proceedings of the VLDB Endowment, 6(10), pp. 889–900. Available at: https://doi.org/10.1145/1620585.1620588 (Accessed: 12 November 2025).

He, J., Zhang, S. and He, B. (2014) 'In-cache query co-processing on coupled CPU-GPU architectures', Proceedings of the 40th International Conference on Very Large Data Bases. New York: ACM, pp. 329–340. Available at: http://dx.doi.org/10.14778/2735496.2735497 (Accessed: 3 November 2025).

Heavy.AI (2020) Using IBM with OmniSci GPU-accelerated analytics. San Francisco: Heavy.AI.

Heimel, M. and Markl, V. (2012) 'A first step towards GPU-assisted query optimization', Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures. Istanbul, Turkey.

Heyns, M. (2020) Brytlyt 4.0 now 4x faster with strong PostgreSQL integration. Available at: https://brytlyt.io/news/brytlyt-4-0-now-4x-faster-with-strong-postgresql-integration/ (Accessed: 13 December 2020).

Ho, T., Oh, S.-R. and Kim, H. (2017) 'A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle

operations', PLOS ONE, 12(10).

Hordemann, G., Lee, J.K. and Smith, A.H. (2014) 'Accelerated SQLite database using GPUs', Proceedings of the 22nd Computer Graphics, Visualization and Computer Vision (CGVCVIP). Plzen, Prague.

Huang, Y.-F. and Chen, W.-C. (2015) 'Parallel query on the in-memory database in a CUDA platform', Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC). Piscataway: IEEE, pp. 236–243. Available at: https://ieeexplore.ieee.org/document/7424569 (Accessed: 18 November 2025).

Huffman, D. (1952) 'A method for the construction of minimum-redundancy codes', Proceedings of the IRE, 40(9), pp. 1098–1101. Available at: https://doi.org/10.1109/JRPROC.1952.273898 (Accessed: 3 November 2025).

Hung, C.-L., Lin, C.-Y. and Wang, H.-H. (2014) 'An efficient parallel-network packet pattern-matching approach using GPUs', Journal of Systems Architecture, 60(5), pp. 431–439. Available at: https://doi.org/10.1016/j.sysarc.2014.01.007 (Accessed: 3 November 2025).

Huo, X., Ravi, V.T. and Agrawal, G. (2011) 'Porting irregular reductions on heterogeneous CPU-GPU configurations', Proceedings of the 18th International Conference on High Performance Computing. Piscataway: IEEE, pp. 1–10. Available at: https://doi.org/10.1109/HiPC.2011.6152715 (Accessed: 2 November 2025).

Ikeda, K., Ino, F. and Hagihara, K. (2016) 'An OpenACC optimizer for accelerating histogram computation on a GPU', Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). Piscataway: IEEE, pp. 468–477. Available at: https://doi.org/10.1109/PDP.2016.14 (Accessed: 3 November 2025).

Jern, M., Åström, T. and Johansson, S. (2008) 'GeoAnalytics tools applied to large geospatial datasets', Proceedings of the 12th International Conference Information Visualisation. Piscataway: IEEE, pp. 362–372. Available at:

https://doi.org/10.1109/IV.2008.27 (Accessed: 3 November 2025).

Ji, Y., Qian, J., Yang, R., Zhang, L. and Huang, Z. (2024) 'GPU-based parallel generation algorithm of Geohash index keys', Proceedings of the 2024 10th International Conference on Humanities and Social Science Research (ICHSSR 2024). Piscataway: IEEE, pp. 158–164. Available at: https://doi.org/10.1109/ichci63580.2024.10808128 (Accessed: 21 November 2025).

Kaldewey, T., Lohman, G., Mueller, R. and Volk, P. (2012) 'GPU join processing revisited', Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN '12). New York: ACM, pp. 55–62. Available at: https://dl.acm.org/doi/10.1145/2236584.2236592 (Accessed: 21 November 2025).

Ke, Q., Prabhakaran, V., Xie, Y., Yu, Y., Wu, J. and Yang, J. (2011) 'Optimizing data partitioning for data-parallel computing', Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems. New York: ACM.

Keckler, S.W., Dally, W.J., Khailany, B., Garland, M. and Glasco, D. (2011) 'GPUs and the future of parallel computing', IEEE Micro, 31(5), pp. 7–17. Available at: https://doi.org/10.1109/MM.2011.89 (Accessed: 12 November 2025).

Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Blas, A.D. and Dubey, P. (2009) 'Sort vs. hash revisited: fast join implementation on modern multi-core CPUs', Proceedings of the VLDB Endowment, 2(2), pp. 1378–1389. Available at: https://dl.acm.org/doi/10.14778/1687553.1687564 (Accessed: 3 November 2025).

Kim, N.S., Austin, T., Baauw, D., Mudge, T., Flautner, K. and Hu, J.S. (2003) 'Leakage current: Moore's Law meets static power', Computer, 36(12), pp. 68–75. Available at: https://doi.org/10.1109/MC.2003 (Accessed: 11 November

2025).

Kinetica DB Inc (2021) Kinectica - the new era of big data parallelism. Arlington: Kinetica. Available at: https://www.kinetica.com/wp-content/uploads/2021/07/Vectorization-Whitepaper-1.pdf (Accessed: 13 November 2025).

Knuth, D., Morris, J.J. and Pratt, V. (1977) 'Fast pattern matching in strings', SIAM Journal on Computing, 6(2), pp. 323–350. Available at: https://doi.org/10.1137/0206024 (Accessed: 12 November 2025).

Kohei, K. (2015) 'PG-Strom query acceleration engine of PostgreSQL powered by GPGPU', Proceedings of the GPU Technology Conference. San Jose, USA: NVIDIA.

Konstantinidis, E. and Cotronis, Y. (2017) 'A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling', Journal of Parallel and Distributed Computing, 107, pp. 37–56.

Kouzinopoulos, C.S. and Margaritis, K.G. (2009) 'String matching on a multi-core GPU using CUDA', Proceedings of the 13th Panhellenic Conference on Informatics. Piscataway: IEEE, pp. 14–18. Available at: https://doi.org/10.1109/PCI.2009.47 (Accessed: 12 November 2025).

Kuhn, J.A. and Alessi, R.V. (1989) 'Mixed signal ASIC design issues and methodologies', Proceedings of the 2nd Annual IEEE ASIC Seminar and Exhibit. Piscataway: IEEE, pp. T4-1/1-9. Available at: https://doi.org/10.1109/ASIC.1989.123160 (Accessed: 3 November 2025).

Kuschewski, M., Sauerwein, D., Alhomssi, A. and Leis, V. (2023) 'BtrBlocks: efficient columnar compression for data lakes', Proceedings of the ACM on Management of Data, 1(2), pp. 1–26. Available at: https://dl.acm.org/doi/10.1145/3589263 (Accessed: 3 November 2025).

Kusudo, K., Ino, F. and Hagihara, K. (2015) 'A bit-parallel algorithm for

searching multiple patterns with various lengths', Journal of Parallel and Distributed Computing, 76, pp. 49–57. Available at: https://doi.org/10.1016/j.jpdc.2014.11.003 (Accessed: 11 November 2025).

Lai, C., Huang, M., Shi, X. and You, H. (2013) 'Accelerating geospatial applications on hybrid architectures', Proceedings of the IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. Piscataway: IEEE, pp. 1545–1552. Available at: https://doi.org/10.1109/HPCC.and.EUC.2013.217 (Accessed: 16 November 2025).

Lastovetsky, A. and Manumachu, R.R. (2023) 'Energy-efficient parallel computing: challenges to scaling', Information, 14(4), p. 248. Available at: https://doi.org/10.3390/info14040248 (Accessed: 16 November 2025).

Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P. (2010) 'Debunking the 100X GPU vs. CPU myth', Proceedings of the 37th Annual International Symposium on Computer Architecture. New York: ACM, pp. 451–460. Available at: https://dl.acm.org/doi/10.1145/1815961.1816021 (Accessed: 16 November 2025).

Lemire, D. and Boytsov, L. (2015) 'Decoding billions of integers per second through vectorization', Software: Practice and Experience, 45(1), pp. 1–29. Available at: https://doi.org/10.1002/spe.2203 (Accessed: 16 November 2025).

Levine, J.R. (2009) Flex & Bison. Edited by S. St. Laurent. Sebastopol: O'Reilly Media, Inc.

Li, J., Tseng, H.-W., Lin, C., Papakonstantinou, Y. and Swanson, S. (2016) 'HippogriffDB: balancing I/O and GPU bandwidth in big data analytics', Proceedings of the VLDB Endowment, 9(14), pp. 1647–1658. Available at: https://doi.org/10.14778/3007328.3007331 (Accessed: 16 November 2025).

Li, Y., Ding, B., Wei, Z., Maas, L.M., Al-Ghosien, M., Blanas, S., Bruno, N.,

Curino, C., Interlandi, M., Peeper, C., Rajan, K.S., Chaudhuri, S. and Gehrke, J. (2025) 'Scaling GPU-accelerated databases beyond GPU memory size', Proceedings of the VLDB Endowment, 18(11), pp. 4518–4531. Available at: https://doi.org/10.14778/3749646.3749710 (Accessed: 16 November 2025).

Liu, J., Zhang, F., Li, H., Wang, D., Wan, W., Fang, X., Zhai, J. and Du, X. (2022) 'Exploring query processing on CPU-GPU integrated edge device', IEEE Transactions on Parallel and Distributed Systems, 33(12), pp. 4057–4070. Available at: https://ieeexplore.ieee.org/document/9782546 (Accessed: 16 November 2025).

Luk, C.-K., Hong, S. and Kim, H. (2009) 'Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping', Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Piscataway: IEEE, pp. 45–55.

Manegold, S., Boncz, P. and Kersten, M. (2002) 'Optimizing main-memory join on modern hardware', IEEE Transactions on Knowledge and Data Engineering, 14(4), pp. 709–730. Available at: https://doi.org/10.1109/TKDE.2002.1019210 (Accessed: 16 November 2025).

Manzini, G., 2015, rctail96 [Online]. Available at: https://people.unipmn.it/manzini/lightweight/corpus/ (Accessed: 3 November 2025).

Meister, P. and Saake, G. (2016) 'Challenges for a GPU-accelerated dynamic programming approach for join-order optimization', Proceedings of the 28th GI-Workshop Grundlagen von Datenbanken (GvD 2016). Nörten Hardenberg, Germany: CEUR-WS, pp. 81–86.

Meki, S. and Kambayashi, Y. (2000) 'Acceleration of relational database operations on vector processors', Systems and Computers in Japan, 31(8), pp. 79–88. Available at: https://onlinelibrary.wiley.com/doi/10.1002/1520-684X(200007)31:8%3C79::AID-SCJ9%3E3.0.CO;2-C (Accessed: 12

November 2025).

Mensmann, J., Ropinski, T. and Hinrichs, K. (2010) 'A GPU-supported lossless compression scheme for rendering time-varying volume data', Proceedings of the 8th IEEE/EG International Conference on Volume Graphics. Piscataway: IEEE, pp. 109–116.

Meza Escobar, J.-H., Sachsse, J., Ostendorff, S. and Wuttke, H.-D. (2013) 'ISA configurability of an FPGA test-processor used for board-level interconnection testing', Proceedings of the 14th Latin American Test Workshop - LATW. Piscataway: IEEE, pp. 1–6. Available at: http://ieeexplore.ieee.org/document/6562678/ (Accessed: 16 November 2025).

Micikevicius, P. (2012) 'GPU performance analysis and optimization'. Presentation at GPU Technology Conference 2012, San Jose.

Milic, U., Gelado, I., Puzovic, N., Ramirez, A. and Tomasevic, M. (2013) 'Parallelizing general histogram application for CUDA architectures', Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). Piscataway: IEEE, pp. 11–18. Available at: https://doi.org/10.1109/SAMOS.2013.6621100 (Accessed: 16 November 2025).

Mitani, Y., Ino, F. and Hagihara, K. (2017) 'Parallelizing exact and approximate string matching via inclusive scan on a GPU', IEEE Transactions on Parallel and Distributed Systems, 28(7), pp. 1989–2002. Available at: https://doi.org/10.1109/TPDS.2016.2645222 (Accessed: 16 November 2025).

Morgan, T.P. (2020) Accelerated databases in the fast lane. Available at: https://www.nextplatform.com/2020/06/25/accelerated-databases-in-the-fast-lane/ (Accessed: 13 November 2025).

Mostak, T. (2013) An Overview of MapD (Massively Parallel Database). MSc Thesis. Massachusetts Institute of Technology.

Nelson, M., Sorenson, Z., Myre, J.M., Sawin, J. and Chiu, D. (2020) 'Parallel

acceleration of CPU and GPU range queries over large data sets', Journal of Cloud Computing, 9(1), p. 44. Available at: https://doi.org/10.1186/s13677-020-00191-w (Accessed: 16 November 2025).

Nuriev, M., Zaripova, R., Sinicin, A., Chupaev, A. and Shkinderov, M. (2024) 'Enhancing database performance through SQL optimization, parallel processing and GPU integration', BIO Web of Conferences, 113, p. 04010. Available at: https://doi.org/10.1051/bioconf/202411304010 (Accessed: 16 November 2025).

NVIDIA (2017) CUDA Toolkit Documentation. Available at: https://docs.nvidia.com/cuda/archive/8.0/ (Accessed: 11 November 2025).

Ohara, M. and Yamaguchi, T. (2012) 'Lossless compression of double-precision floating-point data for numerical simulations: highly parallelizable algorithms for GPU computing', IEICE Transactions on Information and Systems, E95.D(12), pp. 2778–2786. Available at: https://doi.org/10.1587/transinf.E95.D.2778 (Accessed: 18 November 2025).

Ozsoy, A. and Swany, M. (2011) 'CULZSS: LZSS lossless data compression on CUDA', Proceedings of the IEEE International Conference on Cluster Computing. Piscataway: IEEE, pp. 403–411. Available at: https://doi.org/10.1109/CLUSTER.2011.52 (Accessed: 18 November 2025).

Pankratius, V. and Heneka, M. (2011) 'Moving database systems to multi-core: an auto-tuning approach', Proceedings of the International Conference on Parallel Processing. Piscataway: IEEE, pp. 582–591. Available at: https://doi.org/10.1109/ICPP.2011.24 (Accessed: 18 November 2025).

Patel, R.A., Zhang, Y., Mak, J., Davidson, A. and Owens, J.D. (2012) 'Parallel lossless data compression on the GPU', Proceedings of the Innovative Parallel Computing (InPar). Piscataway: IEEE, pp. 1–9. Available at: https://doi.org/10.1109/InPar.2012.6339599 (Accessed: 18 November 2025).

Paul, J., He, J. and He, B. (2016) 'GPL: a GPU-based pipelined query processing engine', Proceedings of the International Conference on Management of Data

(SIGMOD'16). New York: ACM, pp. 1935–1950. Available at: https://doi.org/10.1145/2882903.2915224 (Accessed: 18 November 2025).

Paul, J., He, B., Lu, S. and Lau, C.T. (2019) 'Revisiting hash join on graphics processors: a decade later', Proceedings of the IEEE 35th International Conference on Data Engineering Workshops (ICDEW). Piscataway: IEEE, pp. 294–299. Available at: https://doi.org/10.1109/ICDEW.2019.00008 (Accessed: 18 November 2025).

Paul, J., Lu, S. and He, B. (2021) 'Database systems on GPUs', in Foundations and Trends® in Databases. Netherlands: Now Publishers inc., pp. 1–108. Available at: http://dx.doi.org/10.1561/1900000076 (Accessed: 18 November 2025).

Podlozhnyuk, V. (2007) Histogram calculation in CUDA. Santa Clara: NVIDIA. Available at: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf (Accessed: 2 November 2025).

Przymus, P. and Kaczmarski, K. (2014) 'Compression planner for time series database with GPU support', Transactions on Large-Scale Data and Knowledge-Centered Systems, 15, pp. 36–63. Available at: http://dx.doi.org/10.1007/978-3-662-45761-0_2 (Accessed: 18 November 2025).

Raasveldt, M., Holanda, P., Gubner, T. and Mühleisen, H. (2018) 'Fair benchmarking considered difficult: common pitfalls in database performance testing', Proceedings of the 7th ACM SIGMOD Workshop on Testing Database Systems (DBTest 2018). New York: ACM.

Rahmani, H., Topal, C. and Akinlar, C. (2014) 'A parallel Huffman coder on the CUDA architecture', Proceedings of the IEEE Visual Communications and Image Processing Conference. Piscataway: IEEE, pp. 311–314. Available at: https://doi.org/10.1109/VCIP.2014.7051566 (Accessed: 19 November 2025).

Raichand, P. and Aggarwal, R.R. (2013) 'A short survey of data compression techiques for column oriented databases', Journal of Global Research in

Computer Science, 4(7).

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C. (2007) 'Evaluating MapReduce for multi-core and multiprocessor systems', Proceedings of the 13th International Symposium on High Performance Computer Architecture. Piscataway: IEEE, pp. 13–24. Available at: https://doi.org/10.1109/HPCA.2007.346181 (Accessed: 19 November 2025).

Root, C. and Mostak, T. (2016) 'MapD: a GPU-powered big data analytics and visualization platform', Proceedings of the International Conference on Special Interest Group on Computer Graphics and Interactive Techniques Conference (SIGGRAPH'16). New York: ACM, pp. 73:1–73:2. Available at: https://doi.org/10.1145/2897839.2927468 (Accessed: 19 November 2025).

Ross, K.A. (2014) 'Multi-core processors and database systems', Ubiquity, 2014(August), pp. 1–7. Available at: https://dl.acm.org/doi/10.1145/2618403 (Accessed: 19 November 2025).

Schatz, M.C. (2007) Fast exact string matching on the GPU. College Park: Center for Bioinformatics and Computational Biology.

Sen, R., Interlandi, M., Arulraj, J. and Kim, H. (2023) 'GPU database systems characterization and optimization', Proceedings of the VLDB Endowment.

Shalf, J. (2020) 'The future of computing beyond Moore's Law', Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 378(2166), p. 20190061. Available at: https://doi.org/10.1098/rsta.2019.006 (Accessed: 19 November 2025).

Shams, R. and Kennedy, R.A. (2007) 'Efficient histogram algorithms for NVIDIA CUDA compatible devices', Proceedings of the International Conference on Signal Processing and Communication Systems (ICSPCS'07).

Shanbhag, A., Yogatama, B.W., Yu, X. and Madden, S. (2022) 'Tile-based lightweight integer compression in GPU', Proceedings of the International Conference on Management of Data. New York: ACM, pp. 1390–1403.

Available at: https://dl.acm.org/doi/10.1145/3514221.3526132 (Accessed: 19 November 2025).

Shatdal, A., Kant, C. and Naughton, J.F. (1994) 'Cache conscious algorithms for relational query processing', Proceedings of the 20th International Conference on Very Large Data Bases. New York: ACM, pp. 510–521.

Shen, J., Long, L., Deng, X., Okita, M. and Ino, F. (2023) 'A compression-based memory-efficient optimization for out-of-core GPU stencil computation', The Journal of Supercomputing, 79(10), pp. 11055–11077. Available at: https://link.springer.com/10.1007/s11227-023-05103-8 (Accessed: 19 November 2025).

Sioulas, P., Chrysogelos, P., Karpathiotakis, M., Appuswamy, R. and Ailamaki, A. (2019) 'Hardware-conscious hash-joins on GPUs', Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE). Piscataway: IEEE, pp. 698–709. Available at: https://doi.org/10.1109/ICDE.2019.00068 (Accessed: 19 November 2025).

Sitaridi, E.A. and Ross, K.A. (2013) 'Optimizing select conditions on GPUs', Proceedings of the 9th International Workshop on Data Management on New Hardware (DaMoN'13). New York: ACM, p. 1. Available at: https://doi.org/10.1145/2485278.2485282 (Accessed: 19 November 2025).

Sitaridi, E. and Ross, K.A. (2016) 'GPU-accelerated string matching for database applications', The VLDB Journal, 25(5), pp. 719–740. Available at: https://link.springer.com/article/10.1007/s00778-015-0409-y (Accessed: 19 November 2025).

SQream Technologies (2017) SQream DB technical whitepaper. Available at: https://www.temperfield.com/wp-content/uploads/2018/06/SQream-DB-Technical-Whitepaper-Tf.pdf (Accessed: 16 November 2025).

Stackhouse, B., Bhimji, S., Bostak, C., Bradley, D., Cherkauer, B. and Desai, J. (2008) 'A 65nm 2-billion-transistor quad-core Itanium® processor', Proceedings of the IEEE International Solid-State Circuits Conference - Digest

of Technical Papers. Piscataway: IEEE, pp. 92–598. Available at: https://doi.org/10.1109/ISSCC.2008.4523072 (Accessed: 19 November 2025).

Stojanovic, N. and Stojanovic, D. (2014) 'High performance processing and analysis of geospatial data using CUDA on GPU', Advances in Electrical and Computer Engineering, 14(4), pp. 109–114.

Stok, L. and Cohn, J. (2003) 'There is life left in ASICs', Proceedings of the 2003 International Symposium on Physical Design. New York: ACM, pp. 48–50. Available at: https://doi.org/10.1145/640000.640013 (Accessed: 20 November 2025).

Sukhwani, B., Min, H., Thoennes, M., Dube, P., Iyer, B. and Brezzo, B. (2012) 'Database analytics acceleration using FPGAs', Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). Piscataway: IEEE, pp. 411–420.

Sun, C., Agrawal, D. and Abbadi, A.E. (2003) 'Hardware acceleration for spatial selections and joins', Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM, pp. 455–466. Available at: https://dl.acm.org/doi/10.1145/872757.872813 (Accessed: 20 November 2025).

Tahir, M., Sardaraz, M. and Ikram, A.A. (2017) 'EPMA: efficient pattern matching algorithm for DNA sequences', Expert Systems with Applications, pp. 162–170. Available at: https://doi.org/10.1016/j.eswa.2017.03.026 (Accessed: 20 November 2025).

Tarditi, D., Puri, S. and Oglesby, J. (2006) 'Accelerator: using data parallelism to program GPUs for general-purpose uses', ACM SIGARCH Computer Architecture News, 34(5), pp. 325–335. Available at: https://dl.acm.org/doi/10.1145/1168919.1168898 (Accessed: 20 November 2025).

Tay, R. (2010) Sample implementations of exact string matching algorithms in CUDA. Available at: https://code.google.com/archive/p/exactstrmatchgpu/

(Accessed: 20 November 2025).

Taylor, P. (2022) Digital & Trends: database software. New York: Statista Inc. Available at: https://www.statista.com/statistics/871513/worldwide-data-created/ (Accessed: 11 November 2025).

Tian, J., Rivera, C., Di, S., Chen, J., Liang, X. and Tao, D. (2021) 'Revisiting Huffman coding: toward extreme performance on modern GPU architectures', Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS). Piscataway: IEEE, pp. 881–891. Available at: https://doi.org/10.1109/IPDPS49936.2021.00097 (Accessed: 20 November 2025).

Torres, J.S., Espert, I.B., Dominguez, A.T., Hernendez, V., Medina, I., Terraga, J. and Dopazo, J. (2012) 'Using GPUs for the exact alignment of short-read genetic sequences by means of the Burrows-Wheeler transform', IEEE/ACM Transactions on Computational Biology and Bioinformatics, 9(4), pp. 1245–1256. Available at: https://doi.org/10.1109/TCBB.2012.49 (Accessed: 20 November 2025).

TPC.org (2016) TPC-H Vesion 2. Available at: https://www.tpc.org/tpch/ (Accessed: 13 November 2025).

Tran, T.T., Liu, Y. and Schmidt, B. (2016) 'Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi', Journal of Parallel Computing, 54, pp. 128–138. Available at: https://doi.org/10.1016/j.parco.2015.11.001 (Accessed: 20 November 2025).

Varakin, K. and Gal, A. (2014) A method for pre-processing and processing query operation on multiple data chunk on vector enabled architecture. WIPO Pat. WO2014020605A1.

Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P. and Ioannidis, S. (2008) 'Gnort: high performance network intrusion detection using graphics processors', in Lippmann, R., Kirda, E. and Trachtenberg, A. (eds) Recent

Advances in Intrusion Detection. Berlin Heidelberg: Springer, pp. 116–134.

Vetter, C. and Westermann, R. (2011) 'Optimized GPU histograms for multi-modal registration', Proceedings of the IEEE International Symposium on Biomedical Imaging: From Nano to Macro. Piscataway: IEEE, pp. 1227–1230. Available at: https://doi.org/10.1109/ISBI.2011.5872623 (Accessed: 20 November 2025).

Vidal, R., Casas, M., Moreto, M., Chasapis, D., Ferrer, R., Martorell, X., Ayguade, E., Labarta, J. and Valero, M. (2015) 'Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads', in Terboven, C. et al. (eds) OpenMP: Heterogeneous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015. Cham: Springer International Publishing, pp. 60–72.

Vijaykumar, N., Pekhimenko, G., Jog, A., Bhowmick, A., Ausavarungnirum, R. and Das, C. (2015) 'A case for core-assisted bottleneck acceleration in GPUs', Proceedings of the 42nd Annual International Symposium on Computer Architecture. New York: ACM, pp. 41–53. Available at: https://dl.acm.org/doi/10.1145/2749469.2750399 (Accessed: 20 November 2025).

Waidyasooriya, H.M., Ono, D., Hariyama, M. and Kameyama, M. (2014) 'Efficient data transfer scheme using word-pair-encoding-based compression for large-scale text-data processing', Proceedings of the Asia Pacific Conference on Circuits and Systems (APCCAS). Piscataway: IEEE, pp. 639–642. Available at: https://doi.org/10.1109/APCCAS.2014.7032862 (Accessed: 20 November 2025).

Wal, G., Zhang, D., Kandaswamy, I., Marakowitz, J., Kaighn, K. and Zhang, J. (2015) 'FPGA acceleration for feature based processing applications', Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). Piscataway: IEEE, pp. 42–47. Available at:

https://doi.org/10.1109/CVPRW.2015.7301365 (Accessed: 20 November 2025).

Wang, D., Zhang, F., Wan, W., Li, H. and Du, X. (2021) 'FineQuery: fine-grained query processing on CPU-GPU integrated architectures', Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER). Piscataway: IEEE, pp. 355–365. Available at: https://doi.org/10.1109/Cluster48925.2021.000200 (Accessed: 20 November 2025).

Wang, J., Papenhausen, E., Wang, B., Ha, S., Zelenyuk, A. and Mueller, K. (2017) 'Progressive clustering of big data with GPU acceleration and visualization', Proceedings of the New York Scientific Data Summit (NYSDS). Piscataway: IEEE, pp. 1–9. Available at: https://doi.org/10.1109/NYSDS.2017.8085036 (Accessed: 20 November 2025).

Wang, K., Zhang, K., Yuan, Y., Ma, S. and Lee, R. (2014) 'Concurrent analytical query processing with GPUs', Proceedings of the VLDB Endowment, 7(12), pp. 1011–1022. Available at: https://doi.org/10.14778/2732967.2732976 (Accessed: 20 November 2025).

Wikipedia (no date a) Non-cryptographic hash function. Available at: https://en.wikipedia.org/wiki/Non-cryptographic_hash_function. (Accessed: 2 November 2025).

Wikipedia (no date b) enwik8 and enwik9. Available at: http://mattmahoney.net/dc/textdata.html (Accessed: 4 November 2025).

Wu, H., Diamos, G., Cadambi, S. and Yalamanchili, S. (2014) 'Red Fox: an execution environment for relational query processing on GPUs', Proceedings of the International Symposium on Code Generation and Optimization (CGO '14). New York: ACM, pp. 44–54. Available at: http://gpuocelot.gatech.edu/wp-content/uploads/cgo14_redfox.pdf (Accessed: 12 November 2025).

Wu, S. and Manber, U. (1992) 'Fast text searching', Communications of the ACM, 35(10), pp. 83–91. Available at:

https://dl.acm.org/doi/10.1145/359842.359859 (Accessed: 20 November 2025).

Xi, J., Mo, C., Karsin, B., Chirkin, A., Li, M. and Zhang, M. (2025) 'VecFlow: a high-performance vector data management system for filtered-search on GPUs', Proceedings of the ACM on Management of Data, 3(4), pp. 1–27. Available at: https://doi.org/10.1145/3749189 (Accessed: 20 November 2025).

Xiang, P., Yang, Y. and Zhou, H. (2014) 'Warp-level divergence in GPUs: characterization, impact, and mitigation', Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). Piscataway: IEEE, pp. 284–295. Available at: https://doi.org/10.1109/HPCA.2014.6835939 (Accessed: 20 November 2025).

Xiangwu, D. and Jinxin, C. (2016) 'Optimizing parallel join of column-stores on heterogeneous computing platforms', Proceedings of the 1st Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). Piscataway: IEEE, pp. 621–625. Available at: https://doi.org/10.1109/ITNEC.2016.7560435 (Accessed: 20 November 2025).

Xu, K., Cui, W., Hu, Y. and Guo, L. (2013) 'Bit-parallel multiple approximate string matching based on GPU', Procedia Computer Science, 17, pp. 523–529.

Yang, Y., Fu, Y., Liu, S., Zhixiang, Z. and Zhang, G. (2024) 'Optimizing Top-K query processing on GPU using data compression and pre-filtering', Applied and Computational Engineering, 105(1), pp. 38–43. Available at: https://doi.org/10.54254/2755-2721/105/2024tj0059 (Accessed: 20 November 2025).

Yao, D., Zeng, G. and Ding, C. (2013) 'Partition strategies for C source programs to support CPU+GPU coordination computing', Proceedings of the International Conference on Information Science and Cloud Computing. Piscataway: IEEE, pp. 39–48. Available at: https://doi.org/10.1109/ISCC.2013.23 (Accessed: 20 November 2025).

Ye, Y., Ross, K.A. and Vesdapunt, N. (2011) 'Scalable aggregation on multi-core processors', Proceedings of the 7th International Workshop on Data

Management on New Hardware. New York: ACM, pp. 1–9. Available at: https://dl.acm.org/doi/10.1145/1995441.1995442 (Accessed: 20 November 2025).

Yogatama, B.W., Gong, W. and Yu, X. (2022) 'Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS', Proceedings of the VLDB Endowment, 15(11), pp. 2491–2503. Available at: https://doi.org/10.14778/3551793.3551809 (Accessed: 20 November 2025).

Yong, K.K., Yap, V.V. and Ong, H.H. (2016) 'GPU SQL query accelerator', International Journal of Information Technology, 22(1), p. 18.

Yuan, Y., Lee, R. and Zhang, X. (2013) 'The Yin and Yang of processing data warehousing queries on GPU devices', Proceedings of the VLDB Endowment, 6(10), pp. 817–828. Available at: https://dl.acm.org/doi/10.14778/2536274.2536281 (Accessed: 20 November 2025).

Zha, X. and Sahni, S. (2013) 'GPU-to-GPU and host-to-host multipattern string matching on a GPU', IEEE Transactions on Computers, 62(6), pp. 1156–1169. Available at: https://doi.org/10.1109/TC.2012.61 (Accessed: 20 November 2025).

Zhang, B., Tian, J., Di, S., Yu, X., Swany, M., Tao, D. and Cappello, F. (2023) 'GPULZ: optimizing LZSS lossless compression for multi-byte data on modern GPUs', Proceedings of the 37th International Conference on Supercomputing. New York: ACM, pp. 348–359. Available at: https://doi.org/10.1145/3577193.3593706 (Accessed: 20 November 2025).

Zhang, J., You, S. and Gruenwald, L. (2015) 'Large-scale spatial data processing on GPUs and GPU-accelerated clusters', SIGSPATIAL Special, 6(3), pp. 27–34. Available at: https://doi.org/10.1145/2766196.2766201 (Accessed: 20 November 2025).

Zhang, S., He, J., He, B. and Lu, M. (2013) 'OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures', Proceedings of

the VLDB Endowment, 6(12), pp. 1374–1377. Available at: https://doi.org/10.14778/2536274.2536319 (Accessed: 21 November 2025).

Zheng, K., Cai, Z., Yang, B. and Wang, Z. (2015) 'Algorithms to speedup pattern matching for network intrusion detection systems', Computer Communications, 62, pp. 47–58. Available at: https://doi.org/10.1016/j.comcom.2015.02.004 (Accessed: 21 November 2025).

Zhong, Z., Rychkov, V. and Lastovetsky, A. (2012) 'Data partitioning on heterogeneous multi-core and multi-GPU systems using functional performance models of data-parallel applications', Proceedings of the IEEE International Conference on Cluster Computing. Piscataway: IEEE, pp. 191–199. Available at: https://doi.org/10.1109/CLUSTER.2012.34 (Accessed: 21 November 2025).

Zhou, B., Jin, H. and Zheng, R. (2014) 'A high speed lossless compression algorithm based on CPU and GPU hybrid platform', Proceedings of the IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. Piscataway: IEEE, pp. 693–698. Available at: https://doi.org/10.1109/TrustCom.2014.90 (Accessed: 21 November 2025).

Ziv, J. and Lempel, A. (1977) 'A universal algorithm for sequential data compression', IEEE Transactions on Information Theory, 23(3), pp. 337–343. Available at: https://doi.org/10.1109/TIT.1977.1055714 (Accessed: 21 November 2025).

Zukowski, M., Heman, S., Nes, N. and Boncz, P. (2006) 'Super-scalar RAM-CPU cache compression', Proceedings of the 22nd International Conference on Data Engineering (ICDE'06). Piscataway: IEEE, pp. 59–59. Available at: https://doi.org/10.1109/ICDE.2006.150 (Accessed: 21 November 2025).

APPENDIX

Compute Capability (CUDA SDK support vs. Microarchitecture)