

**OPTIMIZING GPU-ACCELERATED DATA PROCESSING
FRAMEWORKS FOR SCALABLE DEEP LEARNING**

BY

Hamed Aidaros Al-Mehdhar

A Thesis Presented to the
**COLLEGE OF GRADUATE AND INTERDISCIPLINARY STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS**

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

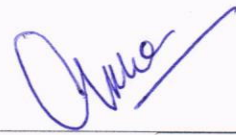
December, 2024

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

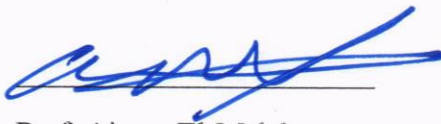
DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Hamed Aidaros Al-Mehdhar** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING.**



Dr. Ayaz Ul Hassan Khan
(Advisor)



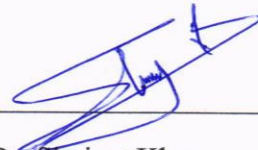
Prof. Aiman El-Maleh
Department Chairman



Dr. Suliman Saleh Al-Homidan
Dean of Graduate Studies



Prof. Mohamed Elrabaa
(Member)



Dr. Shujaat Khan
(Member)

30-12-2024

Date



© Hamed Aidaros Al-Mehdhar

2024

To my family, friends, and loved ones.

To the deserving ones who wanted but couldn't.

ACKNOWLEDGMENTS

Firstly, I begin with the praise of Allah the Almighty, the most Gracious, and the most Merciful, though our praise is insufficient and meagre in relation to His glory. Secondly, I ask for His blessings on all the prophets and specially on His last and greatest Prophet Muhammad (peace be on them all), his family, and companions.

I am forever grateful and thankful for being able to go through this experience, and for those who supported me and guided me throughout this journey. To my advisor and mentor Dr. Ayaz Khan, your guidance and patience gave me the courage to explore ideas allowing me to achieve goals I never thought were possible. I am also grateful to the committee members, Prof. Mohamed Elrabaa and Dr. Shujaat Khan for their valuable and constructive input that allowed me to complete my thesis to the best of my abilities. My gratitude for my parents for their endless prayers and support. To my brothers Mohammed and Hamza and to my sisters Khadijah, Fatimah and Amena, for all their help emotionally, mentally and financially. To my nephews and nieces for being a part of my life. This accomplishment wouldn't have been possible without the unwavering support and guidance. To my friends Ahmed and Salem Basamad, Ibrahim Albulushi, Adnan Munir, Ahmed Saeed, Mohammed Makarem, Muneef, Akram, Baber, Abdussamad, Abdulmumin, Ali, and Aliyu for their support even beyond the academic realm. Lastly, I am thankful to King Fahd University of Petroleum and Minerals (KFUPM) for giving me the opportunity to go through this experience, and to SDAIA-KFUPM Joint Research Center for Artificial Intelligence for providing us with environment that made this work possible.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	III
TABLE OF CONTENTS	IV
LIST OF TABLES	VIII
LIST OF FIGURES.....	IX
LIST OF ABBREVIATIONS.....	XII
ABSTRACT	XIV
ملخص الرسالة.....	XVI
CHAPTER 1 INTRODUCTION	1
1.1 Background and Context	1
1.2 Research Problem and Motivation.....	3
1.3 Thesis Organization.....	7
CHAPTER 2 BACKGROUND	8
2.1 DataFrames	8
2.2 GPU Execution Model	10

2.3	Challenges	13
2.3.1	GPU Execution Challenges	13
2.3.2	DataFrame Challenges	15
CHAPTER 3 LITERATURE REVIEW		17
3.1	GPU-Accelerated Data Processing.....	17
3.2	GPU Optimization Approaches	21
3.3	GPU Acceleration Challenges.....	23
CHAPTER 4 WORK OVERVIEW		27
4.1	Contributions.....	27
4.2	Experimental Setup.....	28
4.2.1	Development Environment Setup	28
4.2.2	Dataset	29
4.2.3	Experimental Procedure	31
4.2.4	Deep Learning Model	32
CHAPTER 5 OPTIMIZATION TECHNIQUES.....		34
5.1	Optimization Approach	34
5.1.1	Multi-GPU Support	34
5.1.2	Customized Memory Allocation Strategy	38
5.1.3	Copy-Compute Overlap	39
5.1.4	Optimized Utilization of I/O Bandwidth	40

5.2	Optimization Implementation	41
5.2.1	Multi-GPU cuDF.....	41
5.2.2	Dynamic Memory Management Strategy	49
5.2.3	Utilizing Multiple CUDA Streams.....	51
5.2.4	Processing Large Chunks of Data	54
CHAPTER 6	RESULTS AND DISCUSSION	56
6.1	Results.....	56
6.1.1	Baseline Evaluation	56
6.1.2	RMM Evaluation.....	57
6.1.3	SMs Evaluation	58
6.1.4	Optimal Chunk Size Evaluation.....	59
6.1.5	RMM & SMs Evaluation.....	60
6.1.6	RMM & Optimal Chunk Size Evaluation	60
6.1.7	SMs & Optimal Chunk Size Evaluation	61
6.1.8	RMM & SMs & Optimal Chunk Size Evaluation	62
6.1.9	RMM & SMs & Optimal Chunk Size Vs. Baseline Evaluation	62
6.1.10	Deep Learning Model Data Preprocessing	66
6.2	Discussion.....	68
6.2.1	Execution Time Analysis	68
6.2.2	Memory Capacity Analysis	73
6.2.3	Deep Learning Model Data Preprocessing Analysis.....	75
6.2.4	Profiling Analysis	76
CHAPTER 7	CONCLUSION AND FUTURE WORK.....	79

7.1	Conclusion	79
7.2	Future Work	80
7.2.1	Multi—GPU Data Distribution	80
7.2.2	Dynamic Chunk Size.....	82
7.2.3	cuDF Memory Consumption	82
7.2.4	Application Adaptation	82
	REFERENCES	83
	VITAE.....	95

LIST OF TABLES

Table 1. Summary of GPU Acceleration Challenges and Their Effect on GPU-Accelerated Data Processing Frameworks.....	26
Table 2. Operating Systems and Hardware Specification.....	29
Table 3. Optimized cuDF Data Preprocessing Operations Speedup (tested on 3.39GB file).....	66
Table 4. Speedup in Execution time against Original cuDF (Single GPU)	69
Table 5. Speedup in Execution time against Pandas (Single-GPU).....	70
Table 6. Speedup in Execution time against Original cuDF (Multi-GPU)	71
Table 7. Speedup in Execution time against Pandas (Multi-GPU)	72

LIST OF FIGURES

Figure 1. CPU vs GPU Deep Learning Model Workflow	4
Figure 2. Data Structure Transition in Conventional Deep Learning Pipeline	6
Figure 3. Envisioned Data Structure Transition in Deep Learning Pipeline	6
Figure 4. General DataFrame Data Model.....	9
Figure 5. Visual Representation of CUDA Thread Hierarchy and associated GPU Processing Unit “Modified from [33]”	11
Figure 6. Copy-Computer-Overlap Concept.....	12
Figure 7. Generic CPU-GPU Control Flow Sequence.....	13
Figure 8. Thread Divergence	15
Figure 9. RAPIDS Ecosystem Layers “Adapted from [41]”	18
Figure 10. Hardware-to-Software Stack for the Data Science Pipeline “Adapted from [46]”	19
Figure 11. Flowchart of the Python Script Used in the Experimental Procedure	32
Figure 12. Data Distribution Challenges in Multi-GPU Environment	35
Figure 13. Memory Access Challenges in Multi-GPU Environment	36
Figure 14. Conceptual Representation of Single-GPU Execution Vs. Multi-GPU Execution	37
Figure 15. cudf.read_csv Operation workflow Diagram w/&w/o RMM.	39
Figure 16. Parallel Multi-Stream Processing Concept.....	40
Figure 17. Visualization of the I/O Bandwidth Utilization Effect on Data Transfer Time	41
Figure 18. Flowchart illustrating the logical flow of the cudf.read_csv_mgpu function.	43

Figure 19. Flowchart illustrating the logical flow of the cudf.max_mgpu function	47
Figure 20. NVIDIA Nsight Systems Profiling showing Multi-Stream Execution of GPU Kernels	53
Figure 21. Execution Time of cudf.read_csv with Multiple Chunk Sizes (Tested on ~6GB file)	55
Figure 22. Execution Time of Read CSV & Max (Baseline)	57
Figure 23. Execution Time of Read CSV & Max (RMM)	57
Figure 24. cudf_RMM GPU memory consumption overhead (manual setting of pool size) against memory consumed by cuDF without RMM.	58
Figure 25. Execution Time of Read CSV & Max (SM)	59
Figure 26. Execution Time of Read CSV & Max (1GB_Chunk)	59
Figure 27. Execution Time of Read CSV & Max (RMM & SMs)	60
Figure 28. Execution Time of Read CSV & Max (RMM & 1GB_Chunk)	61
Figure 29. Execution Time of Read CSV & Max (SM & 1GB_Chunk)	61
Figure 30. Execution Time of Read CSV & Max (RMM & SM & 1GB_Chunk)	62
Figure 31. Execution Time of Read CSV & Max - Baseline Vs. Optimized cuDF (RMM & SM & 1GB_Chunk)	63
Figure 32. Speedup of Read CSV & Max Against Pandas	63
Figure 33. Execution Time Speedup Visualization of the different optimizations Against Plain cuDF	64
Figure 34. Execution Time Speedup Visualization of the different optimizations Against Pandas	65
Figure 35. Load Capacity Gain in Readable File Sizes over Original cuDF	66

Figure 36. Data Preprocessing Speedup Against Pandas DataFrame.....	67
Figure 37. Dataset Size Capacity for Deployed Deep Learning Model Data Preprocessing.....	68
Figure 38. CUDA API Function Calls plain cuDF vs Optimized multi-GPU cuDF	77
Figure 39. Execution Time of Optimized Multi-GPU cuDF w/o Manually Distributed Data (tested on 18.3GB file)	81

LIST OF ABBREVIATIONS

GPU	:	Graphics Processing Unit
AI	:	Artificial Intelligence
CUDA	:	Compute Unified Device Architecture
NLP	:	Natural Language Processing
SIMT	:	Single Instruction Multiple Thread
SM	:	Streaming Multiprocessor
SMs/SMX	:	Streaming Multiprocessors
HtoD	:	Host to Device
DtoH	:	Device to Host
HPC	:	High-Performance Computing
DNNs	:	Deep Neural Networks
CLX	:	Cyber Log Accelerators
SIEM	:	Security Information and Event Management
CCO	:	Copy-Compute-Overlap
RMM	:	RAPIDS Memory Manager
NVCC	:	NVIDIA CUDA Compiler
CNN	:	Convolutional Neural Network

LSTM	:	Long Short-Term Memory
E.T.	:	Execution Time
PCA	:	Principal Component Analysis (PCA)
KNN	:	K-Nearest Neighbor

ABSTRACT

Full Name : Hamed Aidaros Al-Mehdhar
Thesis Title : Optimizing GPU-Accelerated Data Processing Frameworks for Scalable
Deep Learning
Major Field : Computer Engineering
Date of Degree : December, 2024

In recent years, deep learning has proven effective in tackling a wide range of applications surpassing conventional deterministic, handcrafted alternatives. Given the demand for immense amounts of data to be used by deep learning models during training to produce accurate results, preprocessing such data can greatly affect the deep learning model lifecycle. Data preprocessing is a primary stage in any deep learning model, especially during training, as it can impact a model's accuracy and deployment time. The need for streamlined data preprocessing led NVIDIA to come up with RAPIDS, which is a suite of libraries specialized in data analytics tasks leveraging GPU Acceleration. cuDF is a library within RAPIDS that is used primarily for data preprocessing. It also acts as a backbone for other libraries to achieve GPU-accelerated performance. This makes cuDF a great candidate for optimization as it can streamline the development and deployment of deep-learning models. This thesis presents several novel contributions and optimizations to cuDF, focusing on improving its performance and scalability. The focus of those optimizations is on data loading as preprocessing routines may differ from one model to another, yet all models must start with data loading. The implemented optimizations include developing a novel version of cuDF that supports multi-GPU execution, improving the workload that can be handled by cuDF, improving its performance in executing data

manipulation operations, and improving the overall utilization of resources in systems used for machine learning tasks in general. The optimizations include multi-GPU support, a dynamic memory management strategy, streamed data loading with kernel execution, and the optimal utilization of I/O bandwidth. On the system-wide optimization that utilizes multi-GPU support, the best-case scenarios yield a 3.09X increase in the data loading capacity and a 1.26X execution time speedup against the original implementation of cuDF. Meanwhile, the core optimizations to cuDF running on a single GPU offer a 1.65X increase in data loading capacity and faster execution time by 13%. The version supporting multi-GPU execution stands to gain a substantial 2.59X faster execution over the optimized version of cuDF, with a 2X increase in the data loading capacity it can handle. The optimized versions produced improved cuDF performance and scalability, allowing democratizing GPU acceleration for deep learning model development and deployment on commonly available hardware.

ملخص الرسالة

الاسم الكامل: حامد عيروس المحضار
عنوان الرسالة: تحسين أطر معالجة البيانات المسرّعة بوحدة معالجة الرسومات لتعلم عميق قابل للتوسع
التخصص: هندسة الحاسب الآلي
تاريخ الدرجة العلمية: رجب 1446 هـ

في السنوات الأخيرة، أثبت التعلم العميق فعاليته في معالجة مجموعة واسعة من التطبيقات، متفوقاً على البدائل التقليدية ذات الطابع الحتمي والمصممة يدوياً. نظراً للاعتماد على كميات هائلة من البيانات لاستخدامها من قبل نماذج التعلم العميق أثناء التدريب لتحقيق نتائج دقيقة، يمكن أن تؤثر المعالجة المسبقة لهذه البيانات بشكل كبير على دورة حياة نموذج التعلم العميق. تُعد معالجة البيانات المسبقة مرحلة أساسية في أي نموذج من نماذج التعلم العميق، خاصة أثناء مرحلة التدريب، حيث يمكن أن تؤثر على دقة النموذج ووقت جاهزيته للاستخدام. أدى الاحتياج إلى تحسين عمليات معالجة البيانات المسبقة إلى قيام شركة إنفيديا بتطوير "رايبرز"، وهي مجموعة من المكتبات البرمجية المتخصصة في مهام تحليل ومعالجة البيانات، مستفيدةً وحدة معالجة الرسومات لتسريع هذه المهام. "كو دي إف" هي مكتبة برمجية ضمن مجموعة "رايبرز" تُستخدم بشكل أساسي في معالجة البيانات المجدولة. وتعمل أيضاً كوحدة بناء أساسية لمجموعة من المكتبات البرمجية لتحقيق أداء مُسرّع باستخدام وحدة معالجة الرسومات. هذا يجعل من "كو دي إف" مرشحاً ممتازاً للعمل على تحسين أداء هذه المكتبة البرمجية، حيث يمكنها تبسيط عملية تطوير وتسريع جاهزية نماذج التعلم العميق. تقدم هذه الرسالة العلمية عدة إسهامات وعمليات تحسين جديدة لمكتبة "كو دي إف"، مع التركيز على تحسين أدائها وقابليتها للتوسع. تركز هذه التحسينات على عملية تحميل البيانات، حيث قد تختلف إجراءات المعالجة المسبقة من نموذج إلى آخر، ولكن جميع النماذج يجب أن تبدأ بتحميل البيانات. تشمل التحسينات المُنفذة تطوير إصدار جديد من مكتبة "كو دي إف" يدعم استخدام وحدات معالجة رسومات متعددة، وتحسين عبء العمل الذي يمكن لـ "كو دي إف" معالجته، وتعزيز الأداء في تنفيذ عمليات معالجة البيانات، بالإضافة إلى تحسين الاستفادة الشاملة من الموارد في الأنظمة المستخدمة لمهام الذكاء الاصطناعي بشكل عام. تشمل التحسينات دعم المعالجة باستخدام وحدات معالجة الرسومات متعددة، استراتيجية ديناميكية لإدارة الذاكرة، تحميل البيانات بطريقة تزامنية عبر أكثر من نواة معالجة، والاستفادة المثلى من معدل نقل البيانات المتاح لوحدة الإدخال/الإخراج. في التحسينات على مستوى النظم التي من الممكن أن تستفيد من دعم عملية المعالجة باستخدام وحدات معالجة الرسومات المتعددة، تُظهر أفضل الحالات زيادة قدرها 3.09 في كمية البيانات التي يمكن معالجتها وتقليص وقت التنفيذ بنسبة

26 بالمئة مقارنة بالنسخة الاصلية لمكتبة "كو دي إف". أما في حالة النظم التي تعمل باستخدام وحدة معالجة رسومات مفردة، فإن التحسينات المطبقة أدت إلى زيادة في كمية البيانات التي يمكن معالجتها بنسبة 65 بالمئة وتقليص وقت التنفيذ بنسبة 13 بالمئة مقارنة بالنسخة الاصلية لمكتبة "كو دي إف". النسخة المطورة التي تعمل تستفيد من دعم عملية المعالجة باستخدام وحدات معالجة الرسومات المتعددة فإن أفضل الحالات التي تم تسجيلها في هذا العمل أوضحت سرعة عالية في التنفيذ حيث أن المعالجة في هذه الحالة أسرع بمقدار 2.59 مرة مقارنةً بالنسخة التي تعمل باستخدام وحدة معالجة رسومات مفردة، مع قدرة هذه النسخة على معالجة ضعف كمية المعلومات. أسفرت النسخ المحسنة عن تحسين أداء "كو دي إف" وتحسين قابلية هذه المكتبة البرمجة للتوسع، مما مكن من توسيع نطاق الاستفادة من تسريع وحدة معالجة الرسومات لتطوير وتسريع جاهزية نماذج التعلم العميق في الأجهزة المتاحة بشكل شائع.

CHAPTER 1

INTRODUCTION

1.1 Background and Context

Given the multitude of directions in which researchers are working towards enhancing the computing power of devices, some of those approaches became an industry of their own. Some of the proposed approaches are still in their infancy, while others are considered to be mainstream enhancement methods adopted in a wide variety of devices, from highly specialized ones to devices used by the public in their day-to-day lives. Graphics Processing Unit (GPU) acceleration is at the center of well-established and widely adopted technologies for the purpose of enhancing computation power. The Artificial Intelligence (AI) revolution has created a demand never before been witnessed for GPU acceleration, which made NVIDIA, as a leading manufacturer of GPUs, “the world's most valuable company” [1].

GPUs were developed solely to accelerate graphics-related tasks. Given their high performance and unmatched computational throughput compared to CPUs, developers started to look for ways to use GPUs for tasks other than graphics related using APIs and other techniques, yet this showed to be extremely difficult [2]. Noticing the efforts put by the research community into generalizing the use of GPUs, in 2006, NVIDIA released its first GPU equipped with CUDA (Compute Unified Device Architecture). The GeForce 8800 GTX was the first GPU to allow users to program for general-purpose processing [3].

CUDA has enabled researchers and developers to accelerate a wide range of applications by executing those applications or components of their applications on GPUs. Its benefits can be witnessed in fields such as medicine [4], signal processing [5], and engineering [6] to name a few. All of those examples witnessed speedups that can reach 100s of times when compared to with other works in their respective fields. CUDA has also enabled researchers to tackle problems that were unfeasible to solve due to their high runtime [7] by utilizing the high computational throughput of GPUs.

Data processing is yet another field that benefits greatly from GPU acceleration. Given the increasing popularity of DataFrames, making them the de-facto data structure for structured data processing and manipulation. DataFrame libraries with limited scalability, e.g. single node or single thread execution libraries, reach their limits when handling moderately sized datasets [8]. This raised the need for accelerating DataFrames with GPU execution. CUDA proved to play a pivotal role in achieving streamlined data processing.

In this thesis, we look into how GPU-accelerated data processing libraries, cuDF in particular, can be optimized for streamlined deep-learning model development. Deep learning has proven itself to be more accurate and efficient than conventional systems in various applications; however, this unparalleled performance is achieved at the cost of high computational power demand. This computational demand exceeds the hardware performance growth [9]. The trajectory deep learning is moving along regarding its computational demand can pose a constraint in achieving better performance. This shows the relation between deep learning and GPUs, as they try to harness the high computational capabilities of GPUs to meet the computational demand of deep learning models in achieving their prospected performance.

1.2 Research Problem and Motivation

Deep learning models have achieved tremendous performance in many applications due to their ability to capture features that are next to impossible to account for using conventional handcrafted methods [10]. Medicine and healthcare are among the most impactful areas where deep learning is showing promising results. Deep learning models were able to achieve superior results in cancer detection and diagnosis [11], preventing health deterioration in diabetic patients [12], along with many other areas in medicine. Deep learning models proved to give equivalent accuracy to healthcare professionals [13]. Weather forecasting is another area that has benefited greatly from deep learning. Quite recently, as of Nov 2023, a deep learning model called “GraphCast” by Google proved to be more accurate in med-range weather forecasting than the most accurate conventional methods that use operational deterministic systems [14]. Deep learning has also proved to be critical in other tasks, such as fraud detection [15], autonomous systems [16], Natural Language Processing (NLP) [17], manufacturing [18], and many other applications from a wide range of fields.

Data preprocessing is a critical stage in the lifecycle of deep learning model development, as preprocessing the data before it can be fed to the model can greatly affect the accuracy and performance of a model [19], [20], [21]. *FIGURE 1* shows a simplified representation of the deep learning model development lifecycle and the difference between model development on CPU vs GPU. The workflow execution time in the figure refers to the data preprocessing time and training time of a deep learning recommender system model executing on CPU and GPU, showing the ratio of preprocessing to training time [22].

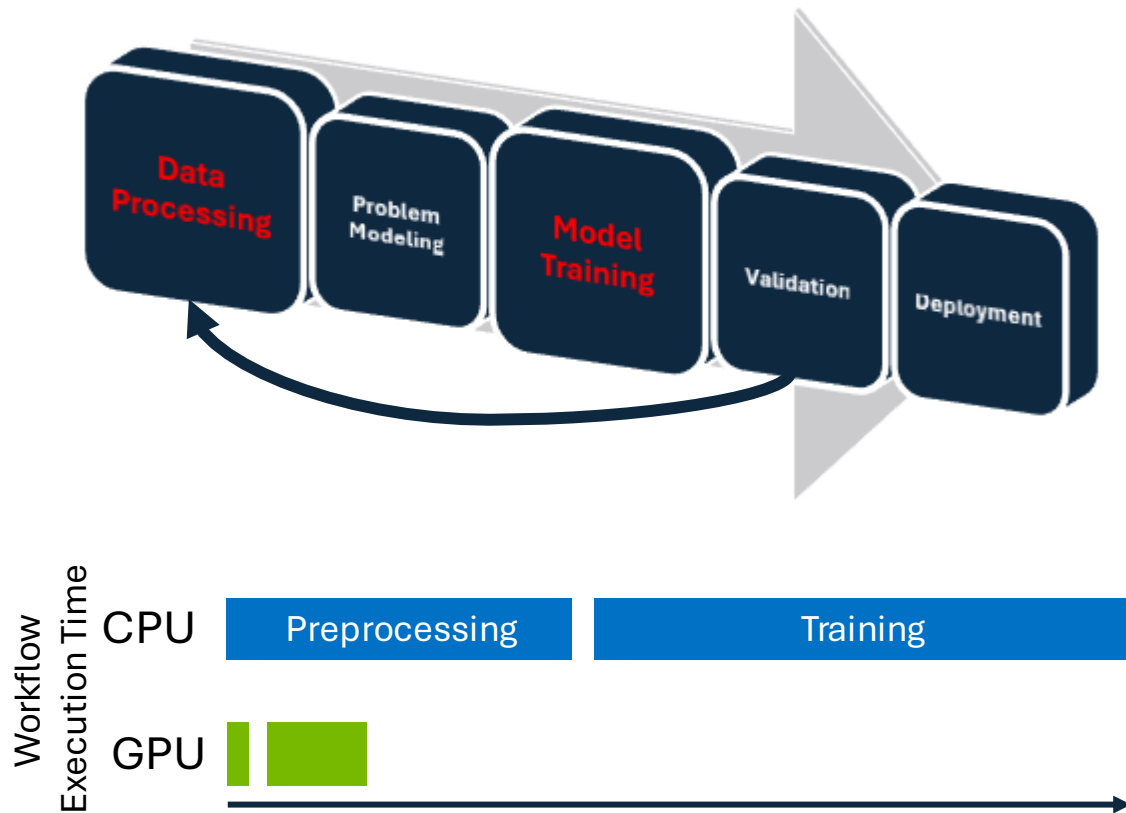


Figure 1. CPU vs GPU Deep Learning Model Workflow

Data processing is an iterative process, meaning it can be repeated many times over until adequate results are obtained [23]. Optimizing this component presents a key opportunity to significantly enhance the overall model’s performance, making it a central focus of our work.

When addressing tabular data processing, DataFrames are among the most powerful and widely used data processing tools in Python [24]. Rule and Tabard et. al in [25] have examined over 1.2 million Jupyter notebooks on GitHub and found that Python notebooks account for over 96% of them. Pandas library is among the most imported libraries and packages into those notebooks, with 42.3% of the notebooks using It. Pandas library has

been downloaded over 7 billion times [26], proving that data science and analysis are of paramount importance nowadays.

Studying the occurrences of each Pandas DataFrame operation on over 1 million Jupyter notebooks [27], it was found that the most common operations of Pandas DataFrame include operations to create a DataFrame, such as `pandas.read_csv()`. Data aggregation operations such as `mean`, `sum`, and `max` are also among the most used operations on a daily basis along with other operations supported by Pandas. They have also identified areas for optimizing Pandas DataFrame, among the most noticeable ones is the support of multi-core capabilities as Pandas is limited by its single-threaded implementation, and improving the overall scalability limitations of Pandas beyond the scalability gains of parallelism.

The need for streamlined data preprocessing led NVIDIA to develop RAPIDS, a suite of open-source libraries specialized in data analytics tasks leveraging GPU Acceleration. cuDF is a library within RAPIDS that is used primarily for data preprocessing. It also acts as a backbone for other libraries to achieve GPU-accelerated performance. cuDF can be blazing fast compared to CPU-based data preprocessing libraries, in certain instances, cuDF can be up to 150x faster than Pandas [28]. However, cuDF suffer from several major constraints that can hinder its adoption. For instance, cuDF does not scale across systems, meaning it only supports single-GPU execution. The cuDF DataFrame is also quite inefficient with GPU memory usage. Testing cuDF, it was found that the largest CSV file size cuDF can read into a cuDF DataFrame is around 31% of the available GPU memory. Inefficient use of limited GPU memory greatly reduces cuDF's capabilities in handling large datasets, as to be expected when used for deep learning data preprocessing tasks.

Mitigating those challenges can produce a version of cuDF that is more suitable for data preprocessing of deep learning models.

Streamlining data preprocessing framework could potentially alter the transition from one data structure to another in a deep learning pipeline. Utilizing a single, and uniform data structure within the pipeline would help in optimizing memory usage, and improve the end-to-end performance of a deep learning model development lifecycle. *FIGURE 2* shows the data structure transitions in a generic deep learning model.

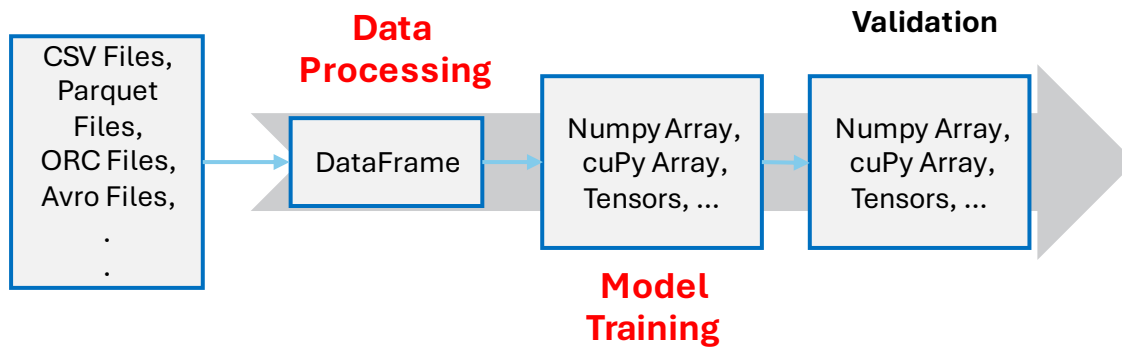


Figure 2. Data Structure Transition in Conventional Deep Learning Pipeline

FIGURE 3 shows the reduction in data structure transitions in the pipeline if a data structure is streamlined well enough to suit the requirements of the different stages in the pipeline.

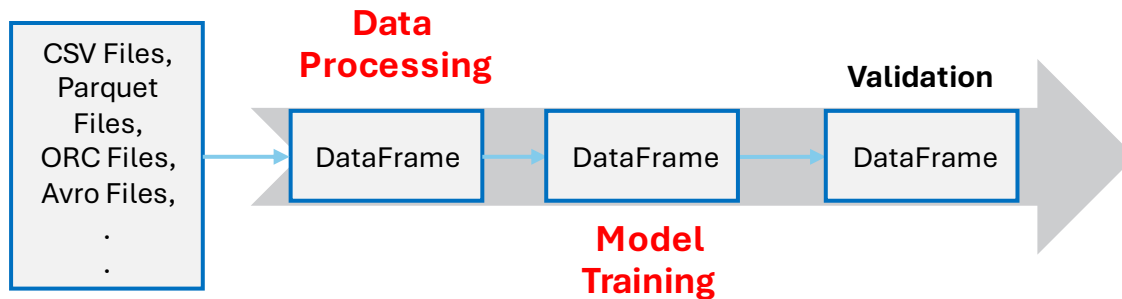


Figure 3. Envisioned Data Structure Transition in Deep Learning Pipeline

In this thesis, we present several novel contributions and optimizations to cuDF, focusing on improving its performance and scalability. The implemented optimizations include

developing a novel version of cuDF that supports multi-GPU execution. As for the optimizations implemented on the core of the cuDF library include implementing a dynamic memory management strategy, concurrent data loading kernel execution, and improving the utilization of I/O bandwidth between the host and the device. Thus, improving the workload that can be handled by cuDF, the overall utilization of resources in systems used for machine learning tasks in general, and the performance of cuDF.

1.3 Thesis Organization

The thesis starts with the *BACKGROUND*, giving the principles of the GPU execution model and the challenges accompanied by this model. The *LITERATURE REVIEW*, covers GPU-accelerated data processing, GPU optimization approaches, and with GPU acceleration challenges. The *WORK OVERVIEW* chapter highlights the main contributions of this work and the experimental setup. Followed by the adopted *OPTIMIZATION TECHNIQUES* explaining the concept behind the optimizations and their implementation details. The *RESULTS AND DISCUSSION*, present the outcomes of this work and explain the key points of the results. Lastly, the work is concluded in the *CONCLUSION AND FUTURE WORK* summarizing the outcomes of this work and identifying areas of optimization that can be targeted in future endeavors.

CHAPTER 2

BACKGROUND

2.1 DataFrames

DataFrames were first introduced by Chambers & Hastie et al. in 1990 as an S programming language object class `data.frames` [29]. In their book, they state that data frames are designed to mimic matrix operations, however, the difference between DataFrames and matrices is in their ability to support elements of different types into the same DataFrame. Pandas, developed in 2008, brought DataFrames capabilities into Python [30], creating flexible, yet versatile data science capabilities.

Data in a DataFrame model belongs to a predefined set of domains, which we can think of as datatypes. Supported domains differ from one DataFrame library to another. For ease of use, each element within a domain within a DataFrame takes the type of its domain. This also applies to missing values, where each domain has its own distinguishable `Null` value. Another aspect adding to the flexibility of DataFrames, domains don't need to be declared beforehand, meaning they can be deduced when needed.

FIGURE 4 shows the general DataFrame data model consists of an array containing the data, labels for the rows and the columns, and the column domain representing the column's metadata. Referencing rows and columns in DataFrames can be done using positional notation, with numerical indexes (e.g. `DF[2, 1]`), or using named notation, using labeled indexes (e.g. `DF["col 0 Label"]`). With DataFrames, we can perform operations

row-wise, column-wise, or on individual elements within the array of data, using the row and column indices to reference a specific element.

	Col 0 Label	Col 1 Label	Col 2 Label	
	Col 0 Domain	Col 1 Domain	Col 2 Domain	
Row Labels				
	2D Array of Data			

Figure 4. General DataFrame Data Model

DataFrames versatility comes from the fact that it combines the capabilities of relational algebra in relational databases (e.g. join, filter, group by,...etc.), linear algebra in matrices (e.g. matrix multiplication, transpose, element-wise operations,...etc.), and spreadsheet computations utilizing formulas (e.g. conditional formatting, aggregation, fill/replace, ...etc.). DataFrames differ from relational databases as relational databases require a rigid declaration of a column's data, the deferred datatype declaration of data in DataFrames allows, for instance, to perform the transpose operation, unlike relational data models. As for matrices, DataFrames differ from them by allowing heterogeneous data to be present within the same DataFrame. Spreadsheets can be modified to represent something similar to DataFrames using extensions, however, the simplicity of DataFrame operations allows for easier implementation and optimization [27].

According to [27], the most common Pandas DataFrame operations [31] used on a daily basis include:

- `read_csv`: used to read CSV data into a Pandas DataFrame.
- `DataFrame`: used to construct a Pandas DataFrame.
- `shape`: used to represent the dimensionality of the DataFrame
- `head`: used to return the top rows in the DataFrame
- `plot`: used to make plots of the DataFrame or a specific series within the DataFrame.
- `mean`: used to obtain the mean value of an axis within the DataFrame.
- `sum`: used to obtain the sum of the values of an axis within the DataFrame.
- `max`: used to obtain the maximum value of an axis within the DataFrame.
- `groupby`: a versatile operation that can be used to group the data within the DataFrame and apply a function over the grouped data and combine results.
- `join`: used to join columns from different DataFrames.
- `loc`: used to access the columns and rows of a DataFrame.
- `iloc`: used to access the data of a DataFrame through select by integer position only.
- `drop`: used to drop specific rows or columns within a DataFrame.
- `apply`: used to apply a function on an axis within a DataFrame.

2.2 GPU Execution Model

GPUs follow a Single Instruction Multiple Thread (SIMT) architecture, which means multiple threads are issued to process specific instructions in parallel. A thread is responsible for executing a specific kernel, i.e. function, on a GPU core. A group of threads is organized into a Thread Block which is executed in parallel on a Streaming

Multiprocessor (SM/SMX), which consists of multiple cores. It is worth mentioning that a group of 32 threads within a thread block is called a Warp, where all threads within the warp execute the same instruction. Combining a group of thread blocks produces what is known as a Grid. All thread blocks in a grid consist of the same number of threads per block. A grid is executed on several streaming multiprocessors (SMs/SMX) [32]. *FIGURE 5* gives a simplified visual representation of how threads are grouped and how they are processed by the processing units within a GPU.

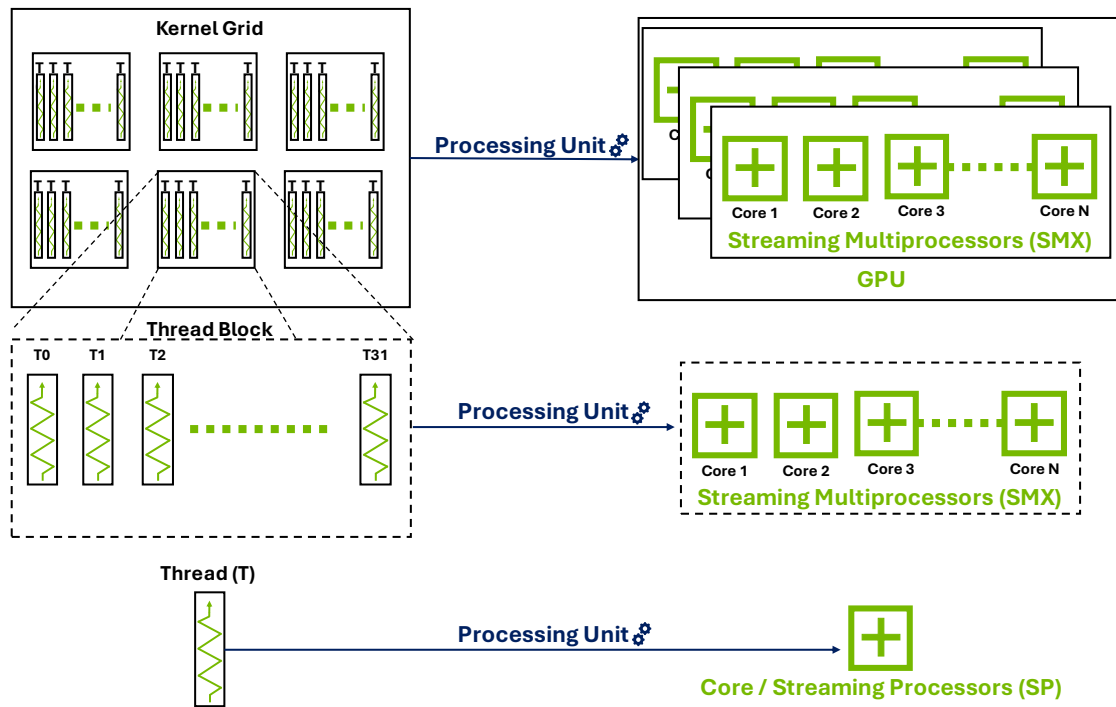


Figure 5. Visual Representation of CUDA Thread Hierarchy and associated GPU Processing Unit “Modified from [33]”

Launching a CUDA kernel to execute GPU code, i.e. device code, requires specifying the dimensions of the block and the grid within the code executed on the CPU, i.e. host code. Those are known as the launch parameters. The syntax for launching a CUDA kernel in C/C++ language is as follows:

```
CUDA_kernel <<< grid, block, unified_memory_bytes, stream >>> ();
```

A typical CUDA program requires 3 main steps, the first is to copy the data from Host memory to Device memory (HtoD), followed by performing the required computations specified in the CUDA kernel, and lastly, copying the data from Device memory to Host memory (DtoH). CUDA kernels can also specify which “stream” to launch a kernel, where a is an abstract concept in CUDA. Streams can be thought of as processes running their code on a processor in a sequential manner. Streams, just like processes, allow concurrent execution of multiple streams.

Dividing the data to be processed into smaller portions/chunks allows for faster copying of the data between device and host memories and allows for concurrently running the intended computation utilizing multiple streams. This strategy helps reduce the time required to complete a task by dividing it into multiple smaller tasks (handling smaller data) that can be executed concurrently, as can be seen in *FIGURE 6* [34].

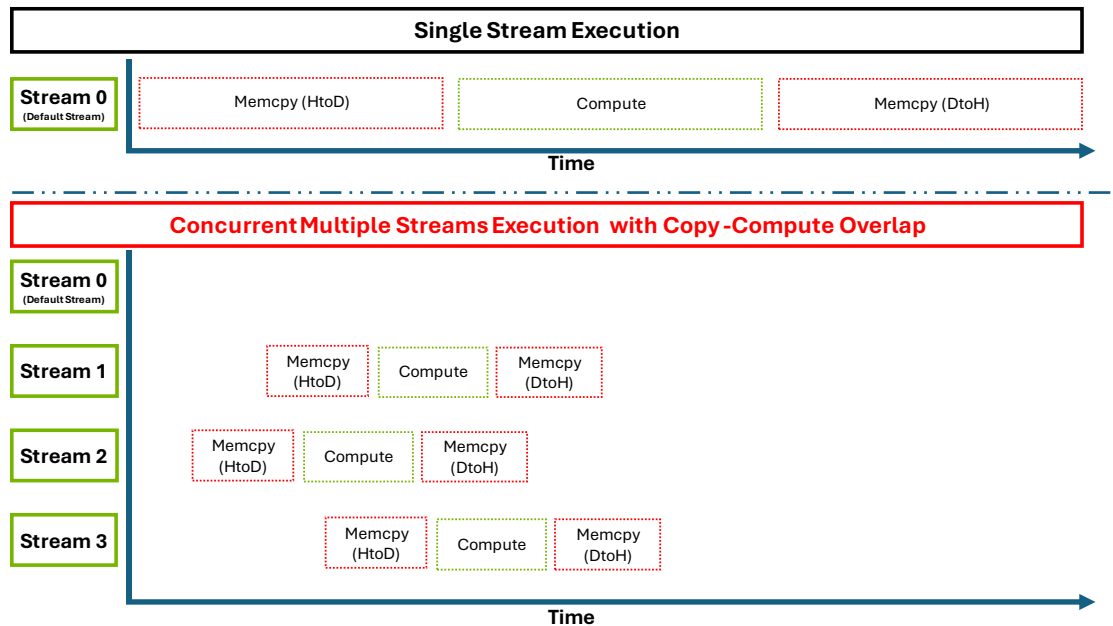


Figure 6. Copy-Computer-Overlap Concept

2.3 Challenges

The challenges faced in carrying out this work can be grouped into two categories. The first are challenges related to the GPU execution model, while the second category is related to DataFrames, and their execution on GPUs.

2.3.1 GPU Execution Challenges

CPU-GPU Communication Overhead

All GPU control flows are initiated by the CPU, as the CPU orchestrates the overall control flow of programs executing in a system. *FIGURE 7* shows the generic control flow sequence required to launch a kernel on the GPU. The figure shows the extent at which GPUs relay on CPUs to perform their tasks.

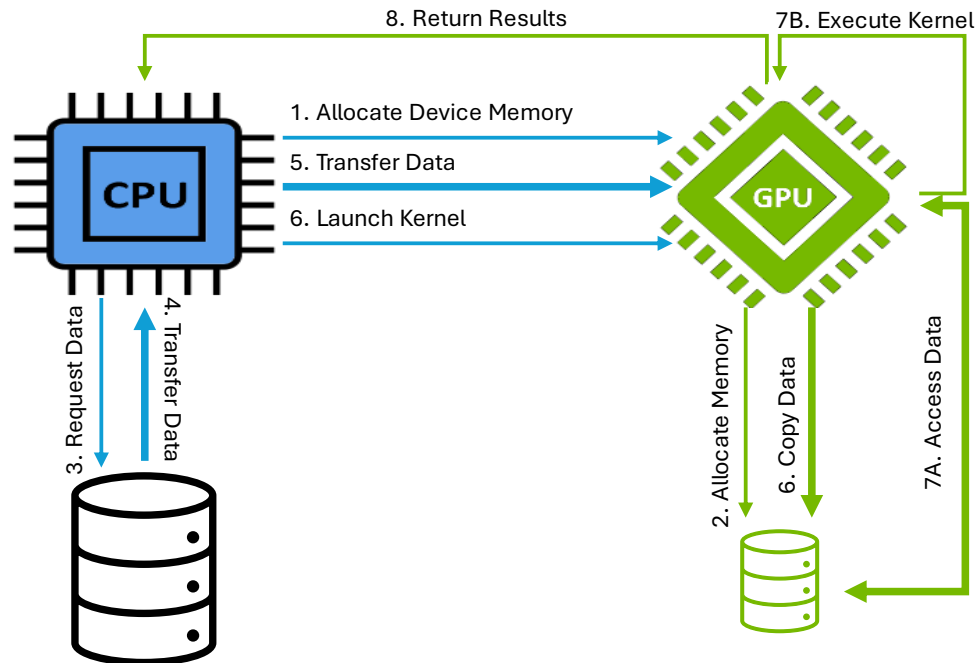


Figure 7. Generic CPU-GPU Control Flow Sequence

The CUDA programming guide released by NVIDIA [32] encourages developers to minimize data transfers between the host and the device. This is because the high overhead of data transfers can limit the benefits of utilizing GPUs' high computational throughput. To reduce the cost of data transfers, developers might need to migrate portions of their host code into device code even if their code does not efficiently execute on GPUs. All of this added effort is to reduce the overall cost of data transfers, as it can completely diminish the gains of accelerating applications on GPUs. Another aspect of this overhead arises when handling small transfers, as each transfer acquires its own overhead. Performing many small transfers will accumulate those overheads, severely impeding GPU acceleration.

Locality

There are two dimensions of locality, temporal and spatial. temporal locality indicates that a program has a high tendency towards using the same memory locations over and over. While spatial locality indicates the adjacency of memory accesses [35]. There are many factors that can impact the degree of locality that can be achieved, such as the memory allocation policy, where the use of memory coalescing for example can improve spatial locality [36]. Another factor is related to how the data is accessed within the device code [37]. This makes handling locality quite a challenge for any GPU-accelerated application.

Thread Divergence

As stated in the GPU execution model, all threads within the warp execute the same instructions. When threads within a warp do not have the same control flow, this creates what is known as thread divergence. Thread divergence can impact the benefits of

parallelism in GPU execution [23]. *FIGURE 8* shows the concept of thread divergence and how it can occur.

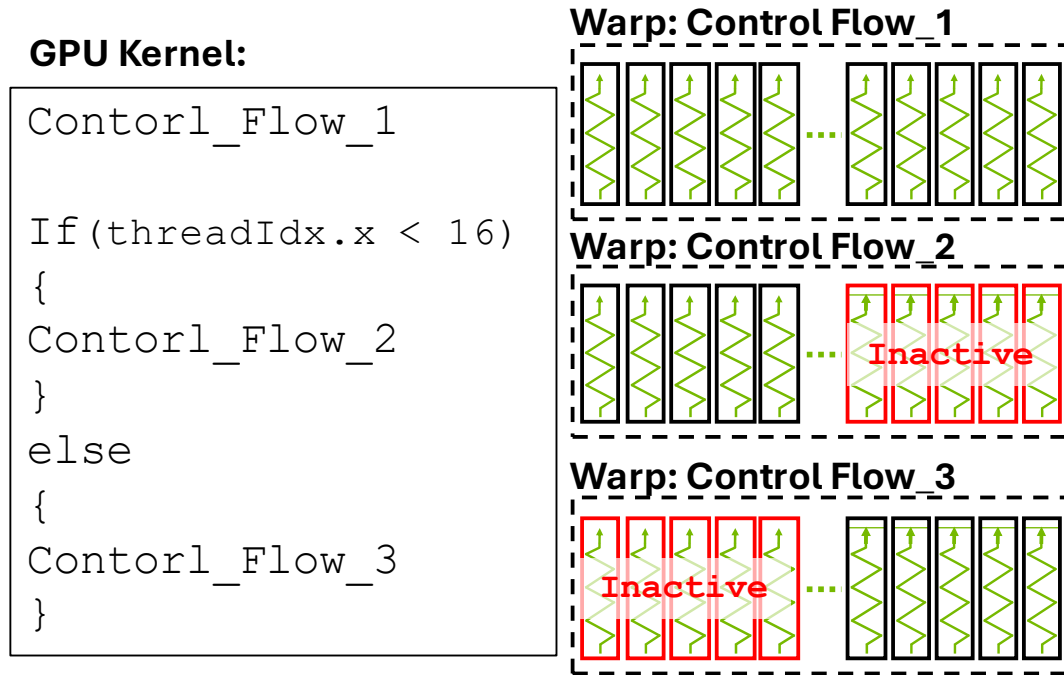


Figure 8. Thread Divergence

2.3.2 DataFrame Challenges

Scalability

How scalable a DataFrame framework is can greatly affect its effectiveness. Limiting execution to a single resource in a system further limits this user's ability to handle larger workloads. Frameworks that require highly sophisticated setups to handle their workloads impose a very high cost on their users.

There are two main approaches to scaling systems, the first is scaling up, where more resources are added to a single machine. The second approach is to scale out, where more nodes are added to a system instead of relying on a single machine. Scaling out approach is preferred and is becoming the norm because of the lower overall system cost, as the

nodes in the system can be moderate instead of relying on highly sophisticated machines. The second reason that makes scaling out a better approach is due to the added resilience it offers, where in the case of failure of certain nodes, the remaining ones can carry out the work [38]. DataFrame libraries that do not offer adequate scalability greatly limit the utilization of scaled-out systems and they make achieving their objectives more difficult and expensive for the user.

Efficient Memory Utilization

Memory demand is more severe when dealing with GPU-accelerated DataFrames, as GPUs generally have more limited memory resources than CPU memory, and there is no feasible option to upgrade GPU memory, as the case is with CPUs. According to McKinney, the developer of Pandas, the rule of thumb when working with Pandas DataFrame is to have a memory that is 5-10x the size of the dataset targeted for processing [39]. As discussed later in *SECTION 6.2.2*, cuDF is even more memory-demanding than Pandas!

In data processing, GPU accelerated DataFrames, in this case cuDF, seems to be an excellent candidate for loading, exploring, transforming, and cleaning data. However, due to memory limitations on top of limited scalability with single GPU execution only, it excludes cuDF from handling large datasets, hence from being a viable option to use in certain scenarios and applications [40].

CHAPTER 3

LITERATURE REVIEW

The literature review conducted for this thesis reflects what the scientific community and industry have achieved in regard to GPU-accelerated data processing and what approaches are explored in the literature to optimize GPU-accelerated workflows. Concluding with the challenges that can hinder and degrade the performance of GPU acceleration.

The investigated areas in the literature review help us in planning a roadmap on how to approach GPU optimizations, what to look for, and what to avoid to gain improved performance on GPUs.

3.1 GPU-Accelerated Data Processing

RAPIDS is an open-source suite of API libraries developed to execute Data Analytics and Data Science on NVIDIA's CUDA-enabled GPUs. RAPIDS suite includes a wide range of projects, including cuDF for DataFrame Processing with a matching API to Pandas, cuML for machine learning with an API similar to scikit-learn, cuGraph for graph processing with an API similar to networkX, RMM that provides configurable GPU memory management, along with other projects that provide an end-to-end data science experience [41]. *FIGURE 9* shows the RAPIDS ecosystem structure layers and how it integrates with NVIDIA's technology stack utilized by libraries within RAPIDS.

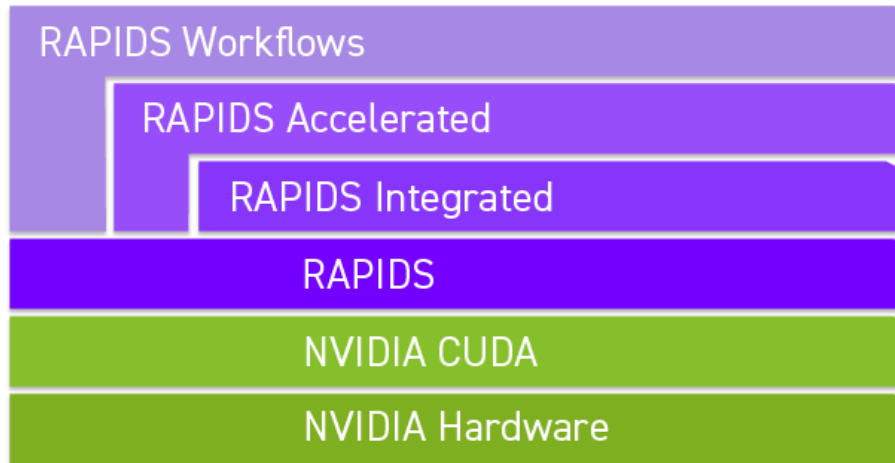


Figure 9. RAPIDS Ecosystem Layers “Adapted from [41]”

cuDF is a GPU DataFrame with an API mimicking that of the Pandas DataFrame. It provides users with all they need for data processing and manipulation on NVIDIA CUDA-enabled GPUs. cuDF comes as a part of RAPIDS. The core of the cuDF Python library uses libcudf, which is a C++ library that is used by cuDF to execute some of its operations. cuDF is developed to be perceived as a GPU-accelerated Pandas DataFrame [42]. cuDF falls under the RAPIDS layer shown in *FIGURE 9* as a key component of the RAPIDS ecosystem. CuDF directly utilizes CUDA for GPU acceleration to operate on tabular data.

Dask-cuDF is another library under RAPIDS for data manipulation [43]. It utilizes the Dask framework for parallel computation. Dask framework is a Python library developed for distributed parallel computing. Dask DataFrames uses Pandas as the backbone, creating multiple Pandas DataFrames that execute in parallel [44]. On the other hand, Dask-cuDF uses cuDF as the backbone, executing its operations on cuDF DataFrames. Dask-cuDF falls under the RAPIDS Integrated layer shown in *FIGURE 9*, as it integrates the Dask parallel computing framework with GPU-accelerated capabilities of cuDF to enable distributed and parallel workflows leveraging GPU acceleration.

RAPIDS also provides a cuDF Pandas accelerator. This accelerator runs Pandas’ operations on cuDF in order to accelerate its performance by running on GPU. This accelerator automatically falls back to normal Pandas running on the CPU in case an operation does not execute properly on cuDF [45]. The cuDF Pandas accelerator falls under the RAPIDS Accelerated layer shown in *FIGURE 9*, as it accelerates Pandas trenchantly by replacing Pandas operations with cuDF operations to gain the benefits of GPU acceleration when possible.

FIGURE 10 demonstrates how RAPIDS enables data science and AI libraries to build on CUDA and CUDA-enabled hardware making GPU acceleration accessible across workflows in various environments.

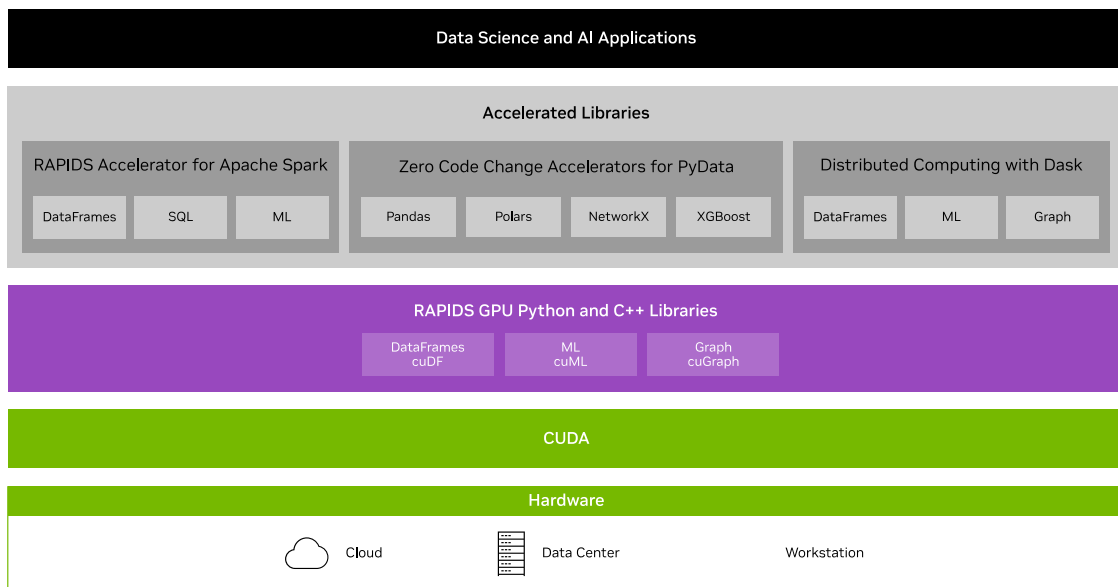


Figure 10. Hardware-to-Software Stack for the Data Science Pipeline “Adapted from [46]”

Hernández et al. in [47] have conducted an extensive evaluation of the RAPIDS suite, including cuDF. The evaluation was conducted on the Summit supercomputer to gain a better understanding of the performance of the RAPIDS libraries. The operations evaluated

include reading CSV files of various sizes, finding the unique elements in the DataFrame, merging the DataFrame with another DataFrame, and grouping entries within the DataFrame. Throughout the testing of the various operations, the GPU accelerated cuDF was at least an order of magnitude faster than Pandas DataFrame running on CPU. It was noticed in this evaluation that the merge, groupby, and unique operations showed poor scalability, unlike the read_csv operation, as the number of GPUs increased, which could be a good direction for our work to look into.

The work of Topcu in [48], is dedicated to evaluating the performance of a meteorological model for forecasting a meteorological phenomenon using a High-Performance Computing (HPC) system. The part that concerns us in this study is running the algorithm on GPU using CUDA. The model was implemented to run serially on the CPU, then to run in a parallel fashion using GPU. They showed that the required calculations, in comparison with CPU serial execution, were 13.1% faster when running on GPU using the local memory of the GPU and 94.1% faster when using global memory. Using the RAPIDS suits, Rabhi et al. [49] were able to accelerate a recommender system by 15.6x using several optimizations, among which was the use of cuDF for data preprocessing. By migrating from Pandas DataFrame to cuDF, they were able to achieve a speedup of 9.7x in the process of feature engineering and data processing with noticeable speedups in 12 different DataFrame operations.

The works in the literature are mostly concerned with measuring the performance of GPU-accelerated DataFrames and data processing libraries showing remarkable speedups when compared with CPU-executed libraries. However, there is a severe lack of research directed toward optimizing and enhancing the inner workings of cuDF. The work of Yogatama et

al. in [50] is among the very scarce works directed toward optimizing cuDF DataFrame, which NVIDIA RAPIDS has already adopted in the current versions of cuDF. Their work focuses on accelerating user-defined aggregation functions, achieving remarkable speedups compared with the baseline version of cuDF.

3.2 GPU Optimization Approaches

General optimization enhancing GPU performance found in the literature involves memory management and exploiting the GPU’s massive parallel computation powers. In this subsection of the literature, we will look into optimization concepts that could be employed to improve the performance of cuDF.

Yang et al. [51], investigated GPU memory management techniques for Deep Neural Networks (DNNs) focusing on nonlinear networks to address the constraints imposed by limited GPU memory. The work utilized unified memory pooling to address the problems of fragmentation with a novel placement policy for allocations. The proposed system reduced memory consumption by 31% with a throughput improvement of 42% when compared with the state-of-the-art.

The survey conducted by Winter et al. in [52] evaluated the proposed dynamic GPU memory managers’ approach. The work evaluated ten different GPU memory managers based on their performance concerning allocations, scalability, fragmentation, and out-of-memory. The survey concludes that there is no such thing as a perfect memory manager, it depends on the focus and primary concern of the user to decide the manager that suits their

application best. One major outcome is regarding future proofing, as the majority of the tested memory managers are not supported on newer GPU architectures.

Becker et al. in [53], proposed a framework called Cyber Log Accelerators (CLX) that utilizes RAPIDS libraries to streamline loading Security Information and Event Management (SIEM) data and preprocessing. The work emphasized the benefits of using memory pooling utilizing RMM and developing nine cyberpandas functions under the RAPIDS suite yielding an average speed up of 295x when compared to the cyberpandas version running on CPU.

The work of Kawakami et al. in [54] analyzed the performance of atomistic models on different computed models, and they concluded that GPU-accelerated versions of the models leveraging CUDA were 54x faster than the baseline on a single GPU and 215x faster on six GPUs compared to the baseline. The experiment also leveraged RMM, and using unified memory, they were able to accommodate datasets exceeding the GPU memory capacity. However, the performance did not always improve as some memory allocation strategies led to a degradation in performance compared to the baseline.

We have exhausted the literature for works that utilize RMM. However, some works utilize and benefit from the use of RMM for memory allocation [55], [56], [57], yet those works did not give specific details or analysis on the effects of using RMM. There are other works [58], [59] that investigated the use of RMM memory allocators; however, their use cases required large allocations, and they noticed that large allocations led to fragmentation issues, rendering the RMM memory allocation strategy used unbeneficial in their test cases.

Another common approach to optimize CUDA kernels is the use of multiple streams to work on a kernel concurrently. A Thread Block in CUDA is executed in parallel on a SM, consisting of multiple cores. CUDA allows overlapping copy operations with computation by splitting the data into smaller chunks and allowing the copy engine to start copying the data, while the kernel engine, which performs the computation, executes the kernel independently of the copy engine. Decoupling the copy engine from the kernel engine allows what is called “Copy-Compute-Overlap (CCO)”. This technique improves the performance by allowing the computation to start before the data is fully copied to memory.

This optimization technique made possible by CUDA has proven its efficiency, as seen in [60], [61] along with many other sources in the literature that apply the same principle within the scope of their application. For brevity, other references are not mentioned as none of them employ this technique to optimize cuDF or any other CUDA-enable GPU-accelerated data manipulation library.

3.3 GPU Acceleration Challenges

Not all types of tasks benefit from GPU parallel execution. There are a number of challenges that must be addressed in order to reap the benefits of GPU acceleration.

Vanderbauwhede et al. [62] experimentations identified the following possible challenges that might affect the gains that GPU acceleration can achieve. Among the possible concerns would be CPU-GPU communication. The PCI Express system latency can significantly reduce the effectiveness of parallelism. Another factor would be the size of the data available for parallel execution for the GPU/s. GPUs require large data sizes to maximize performance. Lastly, how much of the model can be executed on the GPU can greatly affect

the overall performance and speedup. Without proper analysis of the model, the effort to accelerate some components would be futile.

In the work of Solano-Quinde et al [63], they successfully accelerated a module within a system by 3.5x using multi-GPU execution. However, their multi-GPU implementation was designed carefully to avoid what could possibly impede the benefits of using multi-GPUs, which is host-to-device (CPU to GPU) data transfer. They had to minimize the amount of data transfers, using pinned memory (locked memory). They reduce per-transfer overhead by adopting batched transfers where they combine small transfers into a larger transfer reducing the overall number of transfers required. This indicates that the transition to GPU-accelerated implementation is not straightforward and requires careful planning on how to properly benefit from GPU acceleration, especially in regard to data transfers.

Whetstone et al in [64], touched upon a critical subject when it comes to GPU acceleration. The principle of temporal data locality is how frequently a program uses the same data items again and again during the program execution. If the nature of the program uses many data items (e.g., a large number of array elements) for minimal processing, this will create a mismatch between computation vs memory access. This mismatch will require more data transfers, which in turn will incur more overheads. This overhead will undermine the GPU's strong suit, reducing the effectiveness of GPU acceleration.

The work conducted by Puranik in [65], examines the effect of running different DataFrame operations on CPU vs GPU. The work touches upon concerns related to working with GPUs. Among those concerns, as discussed, is the problem of memory and data transfers. Another concern is what is known as thread divergence, where different

threads have to concurrently perform different operations from each other, which can potentially harm the overall performance in comparison with how gracefully CPUs handle thread divergence. Those concerns can potentially affect GPU performance, making it worse than CPUs. However, for tasks that are embarrassingly parallel, GPUs tend to perform much better than CPUs, as witnessed by the finding of Puranik’s work. The ‘Map’ operation, which is a DataFrame operation used to apply custom functions to elements of a specific column, has gained a 20x speedup running on GPU DataFrame when compared to CPU DataFrame as the array size increases. As for the ‘Reduce’ operation, the speedup can reach around 80x, while the ‘Filter’ operation can have a 2x speedup by running the operation on GPU when compared to CPU. The reason for the lower speedup gains in the ‘Filter’ operation is due to one of the aforementioned concerns, mainly thread divergence, leading to less-than-optimal utilization of the GPU.

In the work of Topcu in [48] discussed earlier, the paper inaccurately attributes this speedup to the memory transfer difference between local and global memory schemes. However, most likely this speedup is attributed to efficient memory access patterns when utilizing global memory instead of local memory. The work also demonstrated the effect data transfer has over potential performance improvements, which is highly exacerbated in the case of relatively low computation when compared with data transfer.

Based on the identified GPU acceleration challenges and optimization approaches, we can look into those challenges from a GPU-accelerated DataFrame perspective, and how those challenges can affect the performance of GPU-accelerated DataFrames. *TABLE 1* summarizes the challenges and optimization approaches and maps them to their possible effects on GPU-accelerated DataFrames, specifically cuDF, performance.

Table 1. Summary of GPU Acceleration Challenges and Their Effect on GPU-Accelerated Data Processing Frameworks

Challenge	Description	Effect	Reflection on This Work
CPU-GPU Communication	The overhead caused by frequent data transfers between CPU and GPU.	Increases latency; impacts real-time processing efficiency.	Optimization of this aspect to minimize transfers.
Operational Intensity	The ratio of computations to memory operations on the GPU.	Since the data always resides on the GPU, the effect of data transfer operational intensity can be somewhat limited.	NA
Memory Access Pattern	Irregular or non-coalesced memory access patterns.	Leads to suboptimal GPU performance.	Utilization of coalesced memory.
Memory Allocation Strategy	How memory is allocated and managed on the GPU.	Inefficient allocation increases fragmentation and slows down execution.	Dynamic selection of custom memory strategies.
Sub-optimal Resource Utilization	Certain operations do not utilize multi-stream processing by dividing data into chunks for concurrent processing.	Missing on the opportunity to overlap data transfer and computation was missed.	Leveraging CUDA's copy-compute-overlap to improve performance for large datasets.
Single GPU Execution	Executing operations on a single GPU, not utilizing available devices.	Missing on the opportunity to benefit from the available computation resources in a system.	Support multi-GPU execution.

CHAPTER 4

WORK OVERVIEW

4.1 Contributions

The thesis contributions can be summarized as follows:

- Adding multi-GPU capabilities to cuDF,
 - Multi-GPU support helps improve the performance of cuDF CSV data loading operation and drastically improves the scalability of cuDF, allowing it to handle larger datasets and addressing single GPU execution limitations in cuDF.
- Integrating RAPIDS Memory Manager (RMM) into cuDF to dynamically utilize memory pooling and managed memory,
 - This contribution aids in streamlining cuDF operations and scalability.
 - Improving performance by optimizing memory access patterns, and memory allocation strategy.
- Utilizing the concept of CUDA streams to streamline the `cudf.read_csv` operation.
 - The utilization of multiple CUDA streams allows concurrent execution of GPU kernels developed to read CSV files into cuDF.
 - Allowing for more efficient processing leveraging CUDA's copy-compute-overlap optimization technique.
- Improving CPU-GPU communication,

- Optimizing how cuDF handles data transfers between the host and the device, effectively improving its performance and reducing the CPU-GPU communication overhead.
- Improving the overall performance and scalability of cuDF on single-GPU and multi-GPU environments.

Those improvements act as an enabler for people working on developing deep-learning models by streamlining the data processing stage of the deep-learning model development lifecycle.

4.2 Experimental Setup

This section discusses the methodology followed in this work, starting with *DEVELOPMENT ENVIRONMENT SETUP*, detailing the environment setup on which the experiment is conducted, including hardware and software components. Then, *DATASET* explores the dataset used to test the effect of the different optimizations. Lastly, the subsection *EXPERIMENTAL PROCEDURE* of the methodology explains the procedure for measuring the performance and the details of the conducted experiments.

4.2.1 Development Environment Setup

TABLE 2 shows the operating system and hardware specification of the systems used in this work.

Table 2. Operating Systems and Hardware Specification

Operating System	CPU Specs	GPU Specs
Ubuntu 22.04.2 LTS	Architecture: x86-64	Architecture: NVIDIA Ampere architecture
	CPU op-mode(s): 32-bit, 64-bit	Multi-node GPU workstation: - 2 NVIDIA RTX A4500 devices.
	Model name: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz	
Kernel: Linux 5.19.0-43-generic	CPU Cores: 10	Memory: 20GB of memory per node, total 40GB of memory.
	CPU Threads: 20	
	RAM: 130GB	Cuda Version: 12.4

As for the software components used in this work, Python 3.11.8, Pandas version 2.2.1, cuDF version 24.04.00, Dask version 24.04.00, RMM version 24.04.00a36, and NVIDIA Nsight Systems version 2023.4.4.54-234433681190v0 to profile the applications. The GPU Driver version 550.90.07 with CUDA version 12.4. with NVIDIA CUDA Compiler (NVCC) version .33961263_0 and GCC version 12.3.0. GCC is used as it is the default compiler ,on Linex, used by NVCC to compile C++ files. Jupyter Notebook, to run and document the code, version 7.1.1.

4.2.2 Dataset

The experimentations carried out in this work use the MeteoNet meteorological dataset provided by the French National Meteorological Service [66]. The Dataset consists of a time series of ground station data represented in CSV format. This work focused on the Southeastern France ground stations data, spanning 2016, 2017, and 2018. The dataset is

sampled at a high resolution of 6 minutes between samples. The overall size of the dataset on disk is ~9.81 GB.

The meteorological parameters are represented in a tabular format which are:

- dd: representing the wind direction (in degrees °)
- ff: representing the wind speed (m/s)
- precip: representing the precipitation rate in (kg/m²)
- hu: humidity (%)
- td: Dew Point in Kelvin (K)
- t: temperature in Kelvin (K)
- psl: pressure at sea level in Pascal (Pa)

The dataset is not clean, and it contains some missing/invalid entries.

The dataset was split into different sizes to measure the performance in reading varying CSV file sizes. The sizes of the files used to perform the tests are 10MB, 100MB, 1GB, 3.39GB, 5.93GB, 9.81GB, 18.3GB, and 27.4 GB. The first four files (from 10MB to 3.39GB) are taken as part or as a whole from the year 2016 of the MeteoNet dataset, the 5.93GB file is produced by combining the data from 2016 and 2018 files. The 9.81 file is produced by combining the full dataset (2016, 2017, and 2018) files. The 18.3GB file duplicated and combined the three files into one file. Lastly, the 27.4GB file combines three duplicates of the full data dataset files.

4.2.3 Experimental Procedure

Three different libraries and accelerators are tested to measure the execution time of loading a CSV file into a GPU DataFrame. The libraries are `cuDF`, `dask_cudf`, and `cudf.pandas`.

The results documented in the *RESULTS* section are obtained by running seven trials of a Python script from a Jupyter notebook for each test case using the magic command “`%%timeit`”. The reported execution time is the average wall time of executing the Python script. The Python scripts include importing the library, reading the CSV file, and then performing the `max` operation on the DataFrame produced after reading the CSV file. The reason for including the `max` operation is because of the use of the `dask_cudf` library. Dask performs a lazy `dask_cudf.read_csv()`, meaning the data after this operation is not read into memory. It only creates a DataFrame populated with the file’s metadata. The actual reading starts when an operation that needs to work on the data is invoked. Hence, the `max` operation is included in the time measurement to make sure all of the tested libraries perform the file reading operation fully, emulating an eager evaluation for Dask. To keep comparable measurements, all of the scripts are tested following the same standard (import, read, max) for each library. The results obtained are based on those Python scripts described unless explicitly stated otherwise. *FIGURE 11* depicts the logical flow followed in the Python scripts that were used obtain the results.

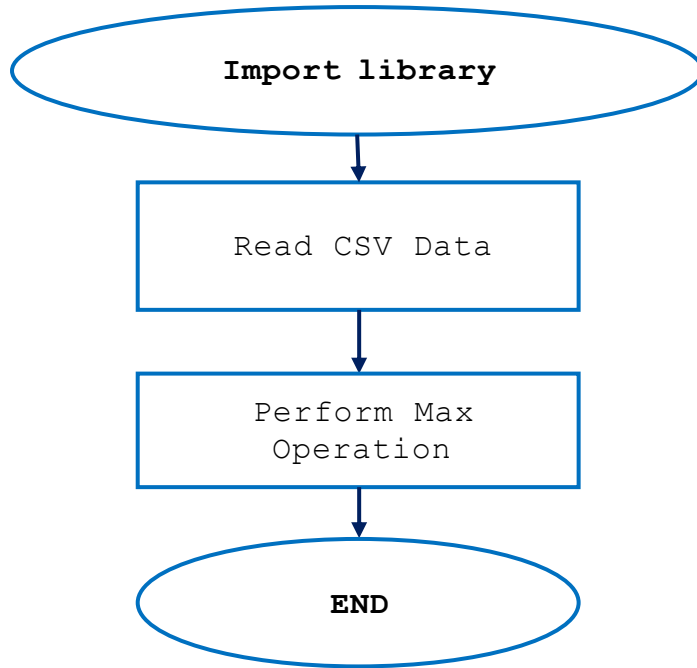


Figure 11. Flowchart of the Python Script Used in the Experimental Procedure

The trials were executed under eight scenarios. The multi-GPU version of cuDF is included in each scenario. Starting with the “plain” scenario, where all the inner workings of the libraries are not altered to document the baseline performance. The following three scenarios test the effect of the three optimizations (i.e. RMM, SM, and optimal chunk size), individually. The next three scenarios test all the combinations of two optimizations together. The last scenario tests the effect of applying all of the optimizations.

4.2.4 Deep Learning Model

Experiments have been conducted to measure the effect of optimizations on other frequently used operations on time series datasets for deep learning model training. The tested data preprocessing operations are CSV data loading, datetime conversion, indexing, and resampling the dataset on a daily mean value with backward filling of missing data. A

deep-learning weather forecasting model was used as a test case and fed the processed dataset for training. The model uses a hybrid architecture combining a one-dimensional Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) layers. The focus regarding the results obtained by training the model is on the data preprocessing side of the overall model development lifecycle.

CHAPTER 5

OPTIMIZATION TECHNIQUES

5.1 Optimization Approach

The optimization approach explores the general ideas and concepts behind applied optimizations to cuDF to improve its overall performance and scalability. Areas of optimization explored in this section include customized memory allocation, copy-compute overlap, optimizing I/O bandwidth utilization, and adding multi-GPU support.

5.1.1 Multi-GPU Support

Dividing the jobs to be accomplished by a single processing unit can stand to be significantly improved by dividing the work over multiple units. In the era of HPC, the concept of utilizing multiple processing units has become the norm, otherwise known as scaling out [38]. In our case, the focus is on utilizing multiple GPUs to improve performance and scalability by utilizing available hardware within a system. Utilizing multi-GPU is quite appealing. However, several considerations can greatly limit the benefit of adding multi-GPU support to an application. If carried out poorly, a multi-GPU design can lead to worse performance than a single-GPU design.

Among those considerations is how the data is distributed among the GPUs available in the system, *FIGURE 12* shows the nuanced cases of distributing a CSV file to be read by two GPUs into cuDF DataFrames. The naïve case does not take into account that

DataFrames required the full row to be read into the DataFrame, if the naïve implementation is followed, this will lead to producing incorrect DataFrames on each GPU, as the first GPU containing the first half of the file might have missing data, while the second half would have misaligned data.

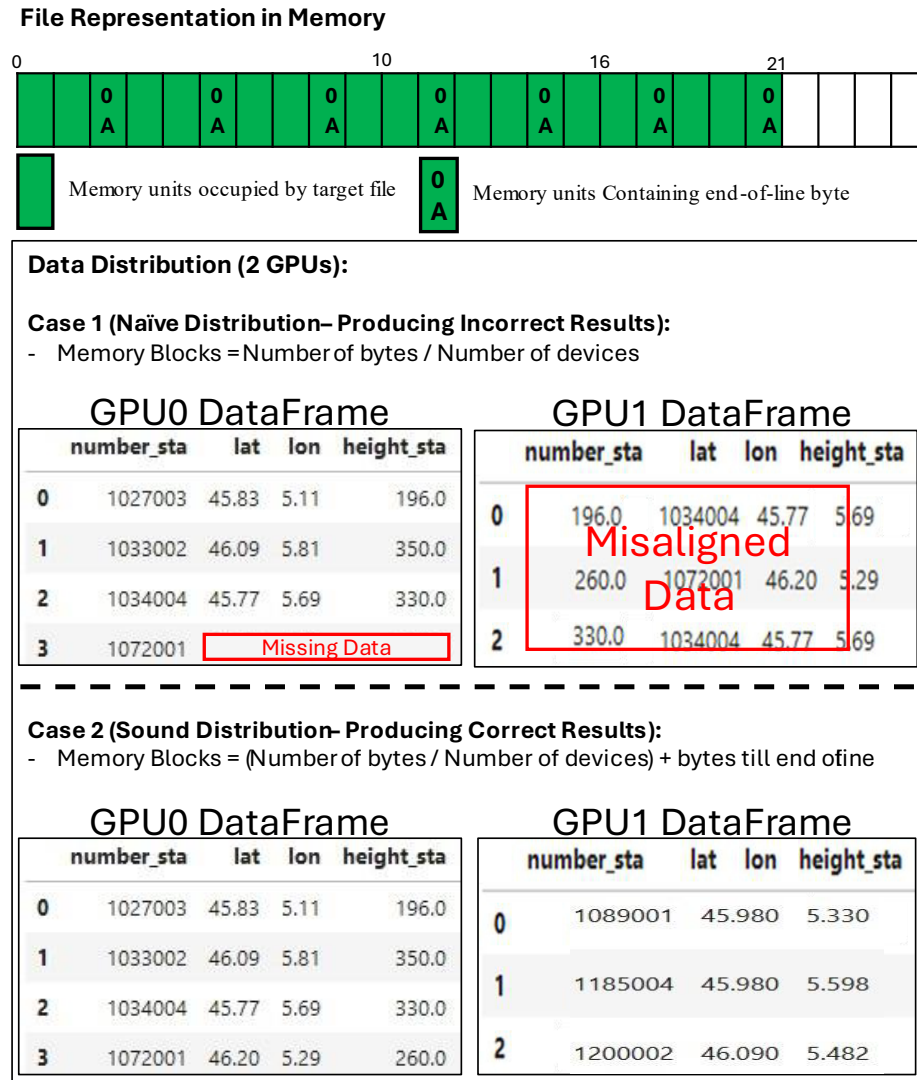


Figure 12. Data Distribution Challenges in Multi-GPU Environment

Another concern with multi-GPU support is how to manage the memory space of multiple GPUs from a single host. *FIGURE 13* illustrates a possible scenario when handling data that is distributed over multiple GPUs. CUDA's design prevents the host from interfacing with

multiple GPUs using the same thread. Multiple threads are required to properly access data distributed over multiple GPUs.

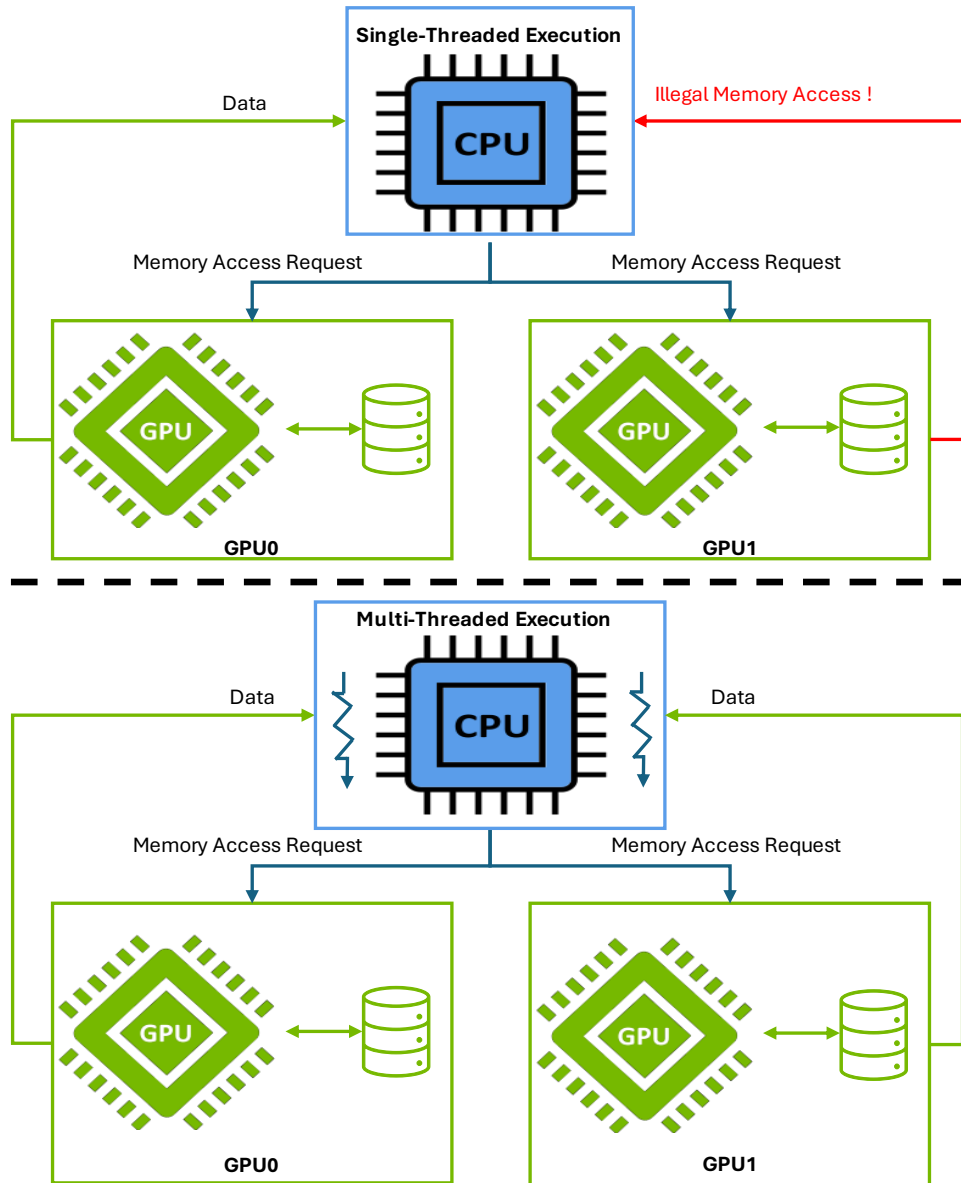


Figure 13. Memory Access Challenges in Multi-GPU Environment

Other multi-GPU support challenges include the nature of the task to be performed, the presence of data dependencies among the GPUs, the size of the data, and the amount of computation to be performed, among other factors that can hinder the performance of a multi-GPU design.

When performed properly, a multi-GPU design can improve the performance of running tasks in parallel over the available GPUs in the system. It can also greatly expand the memory that can be utilized to perform data-intensive tasks, as memory constraints in GPUs are among the greatest impediments of GPU-centric approaches. *FIGURE 14* shows the difference between a single-GPU and multi-GPU execution workflows, where the multi-GPU execution workflow distributes a CSV row-wise into multiple chunks to be processed concurrently on multiple GPUs.

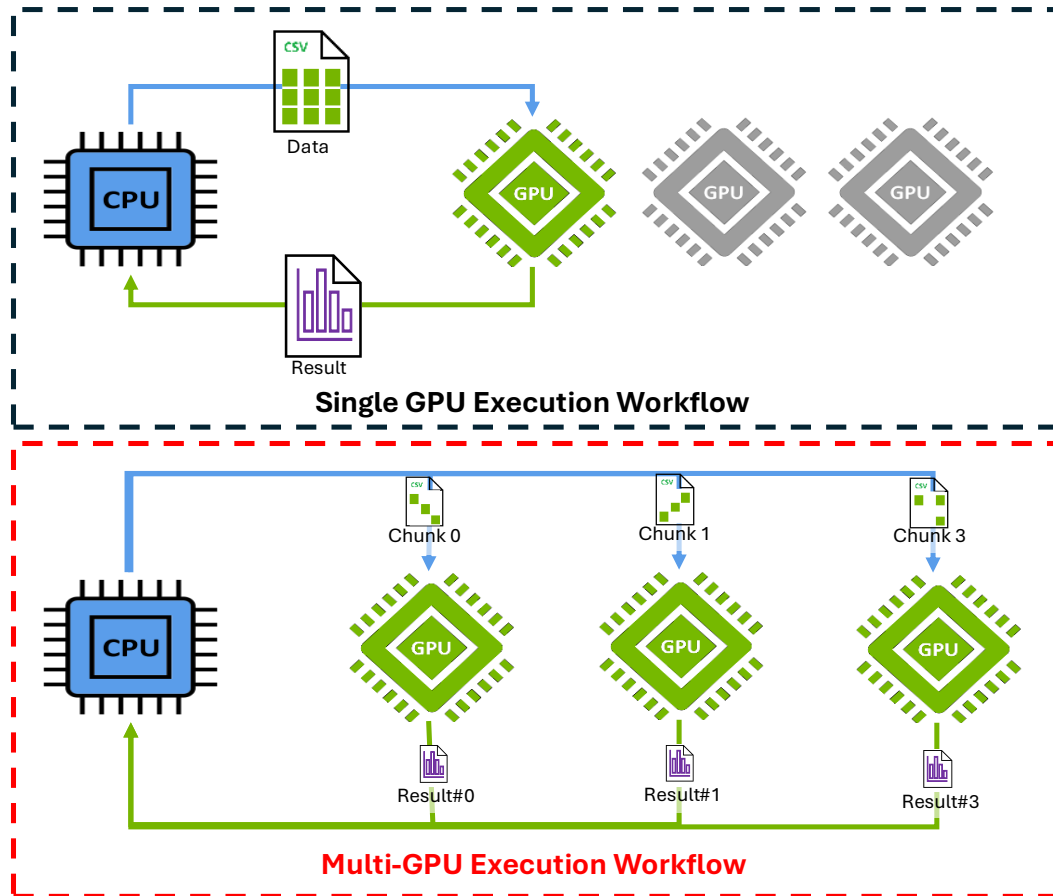


Figure 14. Conceptual Representation of Single-GPU Execution Vs. Multi-GPU Execution

5.1.2 Customized Memory Allocation Strategy

In CUDA programming, Device (i.e. GPU) memory allocation starts by initiating the allocation request from the host (i.e. CPU). Improving the performance workflows intended to utilize CUDA enabled GPUs requires optimal memory allocation. RMM is developed to achieve this optimal performance by providing an interface that allows developers to customize device and host memory allocation. RMM defines a memory resource known as a “DeviceMemoryResource” which is an object that provide a mechanism to allocate and deallocate memory. A memory resource can be customized to achieve different allocation/deallocation strategies that can aid in improving the performance and achieving higher scalability. *FIGURE 15* shows how the use of custom memory management strategies can change the normal workflow of loading CSV data into a cuDF DataFrame on the GPU.

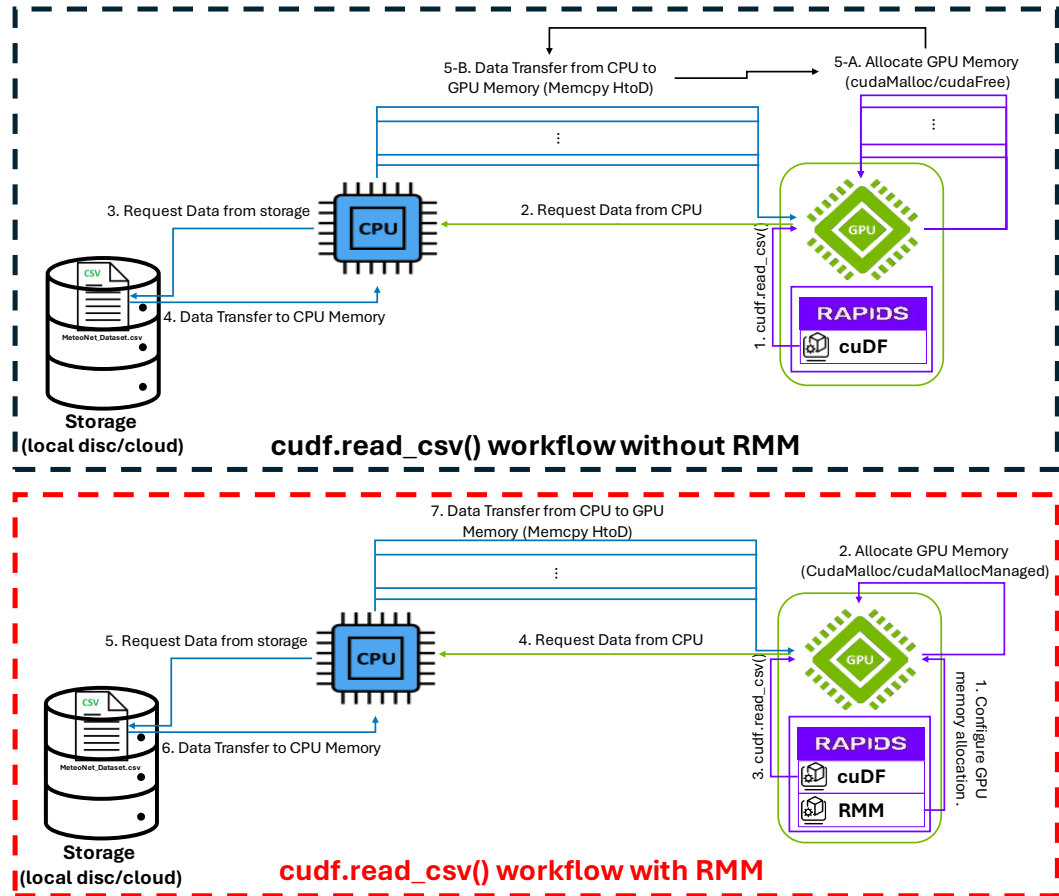


Figure 15. cudf.read_csv() Operation workflow Diagram w/&w/o RMM.

Utilizing those strategies allows us to handle different use cases depending on the data size required to be processed by the GPU. For instance, if the data can fit within the GPU's memory, then using a memory strategy that creates a large pre-allocated memory pool can help in reducing the memory setup overhead. On the other hand, if the data cannot fit in the GPU's memory, using a strategy that can allocate managed memory, that is memory shared by both host and device, can help in improving scalability by handling larger data sizes.

5.1.3 Copy-Compute Overlap

Utilizing CUDA's support for concurrent multi-stream execution, overlapping copy operations with computations can improve the overall performance. By splitting the data

into smaller chunks, where each chunk can be copied by the copy engine while the kernel engine can start its computations independently of the copy engine. Effectively achieving an overlap between the data coping and performing computation on those data as can be seen conceptually in *FIGURE 16*.

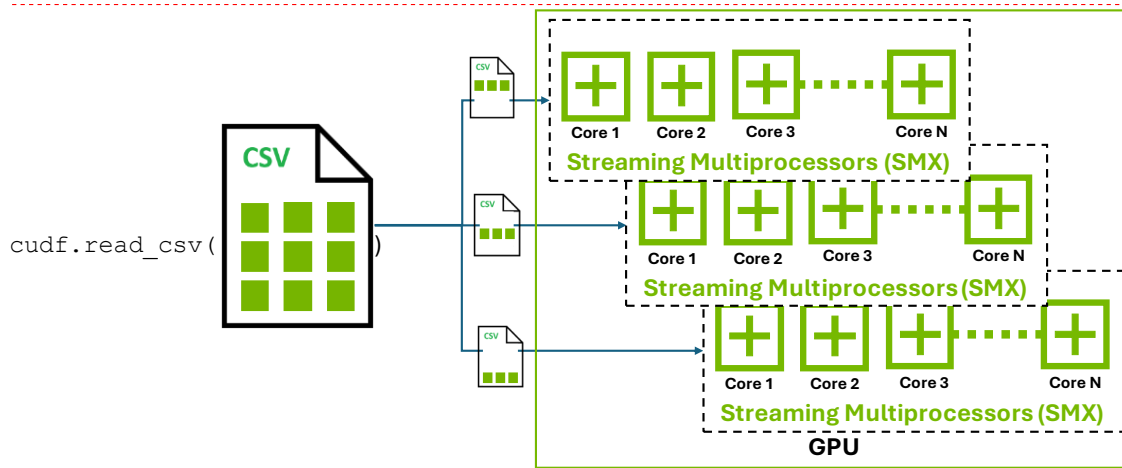


Figure 16. Parallel Multi-Stream Processing Concept

5.1.4 Optimized Utilization of I/O Bandwidth

As all the data resides in the host memory, data needs to be transferred to the device memory for any sort of manipulation of the data. This means How the data is transferred can greatly affect the performance. There are several communication interfaces that allow CPU-GPU communication, such as PCIe and NVLink [67]. Underutilization of the communication interface, especially for large data, will require more data transfers, leading to a degradation in performance. Conversely, when the data size exceeds the available communication interface bandwidth, the interface becomes a bottleneck, leading to performance degradation [68]. *FIGURE 17* gives a general idea of the effect of reading data from memory following an underutilized vs optimally utilized I/O bandwidth.

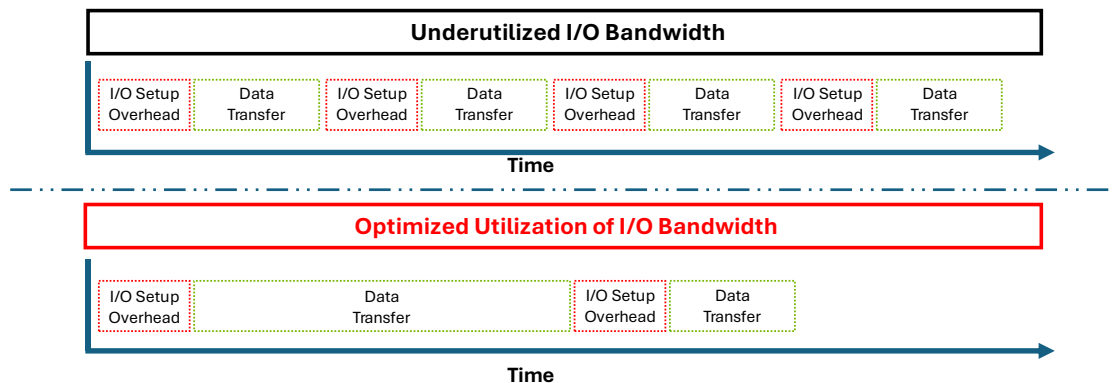


Figure 17. Visualization of the I/O Bandwidth Utilization Effect on Data Transfer Time

5.2 Optimization Implementation

The Optimization Implementation section dives deeper into the technicalities of how the different optimization approaches explored in the OPTIMIZATION APPROACH section are implemented.

The focus of the optimizations mostly relates to loading CSV data operation, as CSV formatted datasets are quite common. Loading a dataset is a must in the preprocessing of any model, unlike any other data processing operation that may or may not be used. This made optimizing `cudf.read_csv()` function of paramount importance in achieving superior performance.

5.2.1 Multi-GPU cuDF

The original implementation of cuDF only utilizes a single GPU. We have modified the cuDF library to support multi-GPU reading of CSV files and to perform some reduction functions on a multi-GPU. The multi-GPU version of reading CSV data, called “`read_csv_mgpu`” is a Breaking Change to cuDF, since it requires a Data Model Change to cuDF. Due to this change, other cuDF functions need to be modified to accommodate

the new structural scheme. As proof of concept and correctness, five reduction functions were modified to handle the new data model change.

Multi-GPUs CSV Data Loading

The multi-GPU version for reading CSV files into cuDF DataFrames is called “`read_csv_mgpu()`”. This function is fully integrated into the cuDF source code as part of the `cudf.io` package as in the original implementation of the cuDF library.

The cuDF user guide API reference defines “`cudf.read_csv()`” function as follows:

```
cudf.read_csv(filepath_or_buffer, sep=",", delimiter=None,
header="infer", names=None, index_col=None, usecols=None, prefix=None,
mangle_dupe_cols=True, dtype=None, true_values=None, false_values=None,
skipinitialspace=False, skiprows=0, skipfooter=0, nrows=None,
na_values=None, keep_default_na=True, na_filter=True,
skip_blank_lines=True, parse_dates=None, dayfirst=False,
compression="infer", thousands=None, decimal=".", lineterminator="\n",
quotechar='"', quoting=0, doublequote=True, comment=None,
delim_whitespace=False, byte_range=None, use_python_file_object=True,
storage_options=None, bytes_per_thread=None)
```

Our developed `read_csv_mgpu` function is defined as follows:

```
cudf.read_csv_mgpu(filepath_or_buffer, num_devices=len(cuda.list_devices
()), sep=",", delimiter=None, header="infer", names=None, index_col=None,
usecols=None, prefix=None, mangle_dupe_cols=True, dtype=None,
true_values=None, false_values=None, skipinitialspace=False, skiprows=0,
skipfooter=0, nrows=None, na_values=None, keep_default_na=True,
na_filter=True, skip_blank_lines=True, parse_dates=None, dayfirst=False,
compression="infer", thousands=None, decimal=".", lineterminator="\n",
quotechar='"', quoting=0, doublequote=True, comment=None,
delim_whitespace=False, byte_range=None, use_python_file_object=True,
storage_options=None, bytes_per_thread=None)
```

The function takes the same arguments as the “`cudf.read_csv`” function with the addition of one more argument called “`num_devices`”. This optional argument can be used to select the number of GPUs on the system to be used to read the CSV file. If not specified by the user, the default value is used, which represents the number of GPUs available in the system. It should be mentioned that the current implementation will use

the first “num_devices” GPUs in the system, starting from GPU0 following the CUDA convention and ending with GPU“num_devices-1”, to read the CSV file into the devices. *FIGURE 18* shows the flowchart that summarizes the read_csv_mgpu logical implementation.

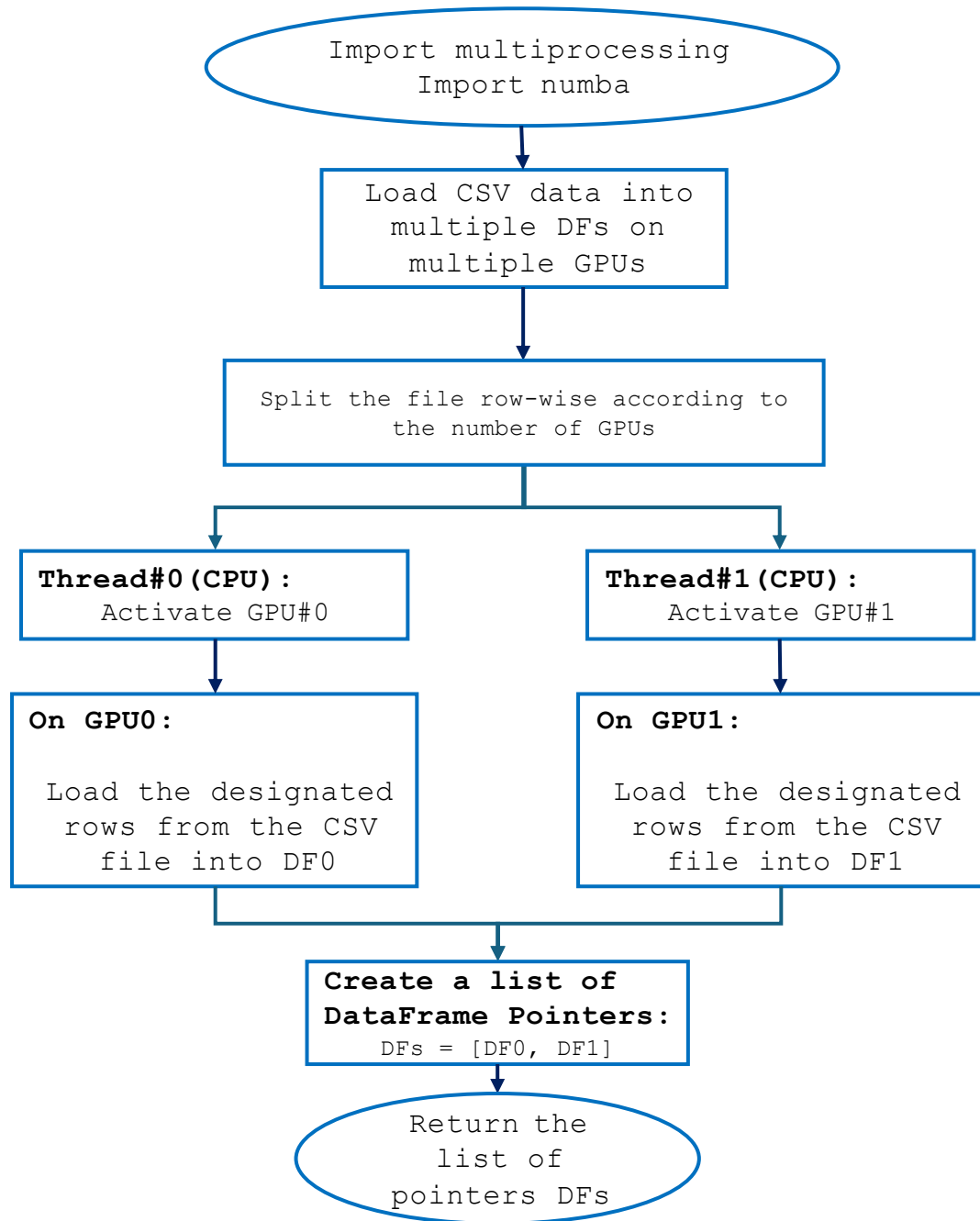


Figure 18. Flowchart illustrating the logical flow of the `cudf.read_csv_mgpu` function

Before implementing the logic of the function we start by importing multiprocessing and numba libraries on top of the other dependencies already used by the original implementation of cuDF for handling CSV files. The multiprocessing library is used to create and control threads processes to achieve parallelism [69]. Numba library allows CUDA GPU programming directly from Python by compiling Python code into CUDA compatible code following CUDA execution model [70].

The `cudf.read_csv_mgpu` function starts by querying the operating system about the size of the file to be read. Based on the file size, the size of the file can be split into multiple chunks to be read, where the number of chunks depends on the number of GPUs specified.

Based on the number of devices specified, a ThreadPool is created to control a pool of worker threads equivalent to the number of devices. Each thread in the pool is assigned a job to run the `cudf.read_csv()` function with the appropriate size of data to be read on each device. The main reason for using multiple threads in this fashion is because of how the CUDA context is managed. Once cuDF is imported, the host process ID is linked to the CUDA context. If we try to access the memory of another device, other than the default device, this will lead to “an illegal memory access was encountered” error. In order to avoid this error, we create a new thread for each device to perform the CSV read operation. After a thread is created, a CUDA context is created using `select_device` function facilitated by numba library. Using threads also gives the benefit of running the operation concurrently, where each GPU starts its sub-operations to read the file concurrently.

In order for each device to read a different section of a CSV file, the `cudf.read_csv` function is changed to accommodate multi-GPU support. The changes include defining the id of the device invoking the function, the number of bytes to read and the overall number of devices used to read the file. Those changes made to the original function were coded in a manner that would not affect the normal behavior of the function if used to abide by the original implementation. This is made possible by setting the new optional arguments introduced by multi-GPU support with default values that would allow skipping over all of the logic amended to the function to support multi-GPU.

Based on the newly introduced arguments to `cudf.read_csv` function, if the number of devices used to invoke the function is more than one, then the logic for multi-GPU support is applied. Multi-GPU support starts by creating a new CUDA context with the `device_id` that calls the function. Then, based on the device ID and the number of bytes the device should read, we utilized an existing argument in the function called “`byte_range`” to manage data distribution among the GPUs. The “`byte_range`” is a list that specifies the offset at which the reading should start and the range of bytes to read. With the `device_id`, the offset can be as follows:

$$offset = number\ of\ bytes\ to\ read * device_id$$

The range of bytes to read is already provided as an argument. For the first device to start reading, i.e. device number 0, its portion of the file will contain the CSV file head. This means the DataFrame that will be created in GPU0 will have the appropriate header, while the other DataFrames in the rest of the GPUs will not. This technicality is addressed in the explaining how the reduction functions are implemented. After that the `cudf.read_csv` code

is kept intact to perform all of the required logic to read CSV data into a cuDF DataFrame and returns that DataFrame of type `cudf.core.dataframe.DataFrame` in the intended GPU.

The “`read_csv_mgpu`” function no longer returns a cuDF DataFrame as each GPU now has its own DataFrame. However, a list of DataFrames is returned, where the index of the DataFrame represents the device associated with it. This means any function amended to utilize this new data model can use the index of the DataFrame as a `device_id`.

Multi-GPUs Reduction Functions

The multi-GPU versions of reduction functions, in the same manner as `cudf.read_csv_mgpu()`, are fully integrated into cuDF source code as part of the `cudf` package. Given that reading a CSV file over multiple GPUs results on a cuDF DataFrame within each GPU, the mechanism to perform any other operation in cuDF needs to be modified. This is to account for the data type change of the returned object of `cudf.read_csv_mgpu` and to access those DataFrames properly. Five reduction functions were created to work on the multi-GPU version: `cudf.max_mgpu()`, `cudf.min_mgpu()`, `cudf.mean_mgpu()`, `cudf.sum_mgpu()`, and `cudf.count_mgpu()`. The adapted reduction functions are defined as follows:

```
cudf.max_mgpu(DFs, col)
cudf.min_mgpu(DFs, col)
cudf.mean_mgpu(DFs, col)
cudf.sum_mgpu(DFs, col)
cudf.count_mgpu(DFs, col)
```

`cudf.max_mgpu()` function is taken as a representative example to explain how reduction functions are designed, with minor differences between them that will be pointed

out. FIGURE 19 shows the flowchart that summarizes the `cudf.max_mgpu` function logical implementation.

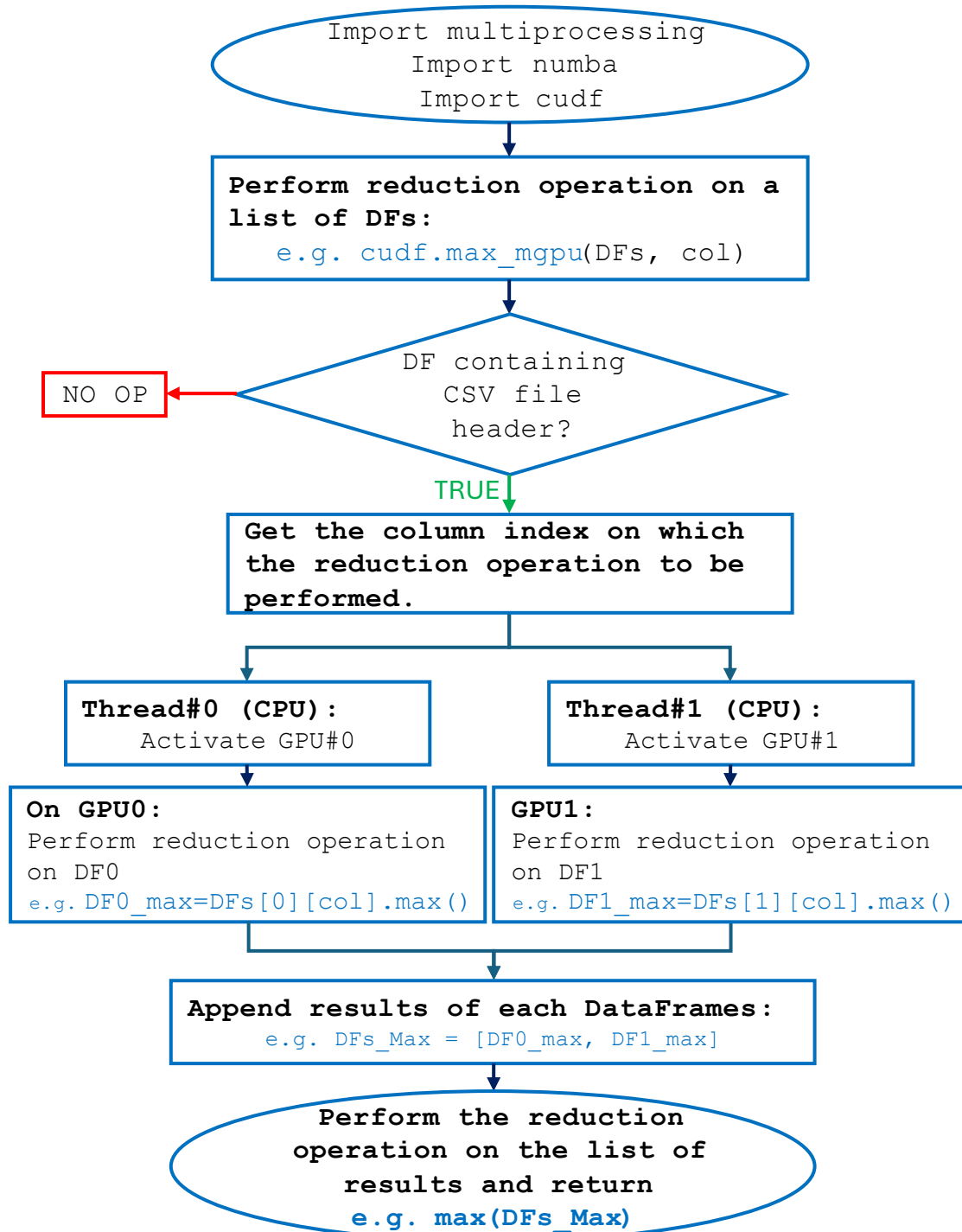


Figure 19. Flowchart illustrating the logical flow of the `cudf.max_mgpu` function

The functions take a list of cuDF DataFrames as the first argument and the name of the series for the function to be applied on. The last argument is optional; if not specified, the reduction function is applied to all DataFrames in the list.

Similar to the approach followed in `cudf.read_csv_mgpu`, a `ThreadPool` is created to control a pool of worker threads equivalent to the number of DataFrames in the list returned after reading on multi-GPUs. The benefit of threading here is twofold: concurrency and creating new threads that can link to the CUDA context on other GPUs to access their memories and process their DataFrame without causing an “an illegal memory access was encountered” error.

The created threads work on a helper function called “`max_mgpu_helper`” that concurrently performs the reduction function on each DataFrame. A new CUDA context is created, and a GPU is selected, based on the index of the DataFrame in the list. Once the CUDA context is created, the DataFrame can be accessed. From the first DataFrame in the list, which is the only DataFrame with the proper header information, the header is translated into the column index for the remaining DataFrames to use. Based on the column index, the DataFrames can select the correct series to perform the reduction. As for the reduction function itself, the function, for example `max`, calls `cudf.Series.max` function which is the original implementation of the `max` function in cuDF. Once each DataFrame is done processing the reduction function, the result is appended to a global list accessible by all threads. After that, the reduction function is applied on the global list to return the final result.

`cudf.mean_mgpu()` function differs slightly from the rest of the function as the mean is calculated differently to ensure correctness. The sum of each DataFrame is appended to a global list along with the number of rows with valid values that participated in that sum, which are appended to another global list. The returned result is the sum of the partial sums of each DataFrame divided by the sum of the valid rows of each DataFrame.

Reduction functions provide the user with the ability to benefit from their intended functionality as well as act as proof of the correctness of the `read_csv_mgpu` implementation. If there is any duplication during the data distribution or improper alignment of the data while loading the data, the results of those reduction functions would be incorrect. As a point of reference for the results of those reduction functions, the same files were read into a Pandas DataFrame, and the reduction operations were applied. The results from Pandas and the multi-GPU versions of the functions were exactly the same, proving the correctness of the implementation of the reduction functions.

5.2.2 Dynamic Memory Management Strategy

Utilizing different memory resources available in RMM can change the behavior of device memory allocation/deallocation. Depending on the size of the data required to be loaded into cuDF, different memory strategies are used in a dynamic manner. This dynamic selection of the memory allocation strategy relieves the user from finding the best memory allocation strategy that can streamline the CSV data loading process or a strategy that can mitigate the GPU memory constraints.

One of the custom memory resources provided by RMM is called "PoolMemoryResource", which allocates a large, coalesced pool of memory. Coalesced memory access is known to reduce the number of memory transactions required [71]. This large pool of memory can then be used for subsequent allocations, reducing the overhead of multiple allocations that would be needed if this allocation strategy is not used. Compound utilization of a large allocation and coalesced memory access reduces the overhead of memory access and leads to improved performance in loading a CSV into cuDF.

Another memory strategy is called "ManagedMemoryResource", which gives another sort of advantage that cannot be achieved with other memory resources offered by RMM. This resource uses the CUDA directive "cudaMallocManaged" to allocate memory shared by both CPU and GPU. Although this memory strategy might be slower, however, it gives the ability to expand the memory capacity beyond what the device memory can handle, allowing handling larger data that would cause a "cudaErrorMemoryAllocation out of memory" error.

If the raw data size on disk is known to cause an "cudaErrorMemoryAllocation out of memory", which after some experiments showed that the file must be around 31% of the free GPU memory size, then the "ManagedMemoryResource" strategy is used to allocate the memory to load the CSV file. Otherwise, the data size can be read into a cuDF DataFrame, then a "PoolMemoryResource" strategy is used to allocate the memory pool before reading the CSV file.

5.2.3 Utilizing Multiple CUDA Streams

At the core of `cudf.read_csv()` function, there are three CUDA kernels that run on the GPU to carry out the required operations to read a CSV into cuDF DataFrame, found in the `"csv_gpu.cu"` file in the source code of cuDF. The kernels are as follows:

- `cudf::io::csv::gpu::convert_csv_to_cudf`
 - This kernel is designed to validate file fields, parse the rows into fields in order to convert them into cuDF DataFrame column data, and ultimately convert raw CSV data into cuDF format.
 - It utilizes GPU parallelism to process multiple fields simultaneously, where each thread in the grid handles one record/row at a time.
- `cudf::io::csv::gpu::data_type_detection`
 - This kernel is primarily designed to parse raw CSV data to infer the datatype of each field in order to assign a datatype to a cuDF DataFrame column.
 - It utilizes GPU parallelism to process multiple fields simultaneously, where each thread in the grid handles one record/row at a time.
- `cudf::io::csv::gpu::gather_row_offsets_gpu`
 - This kernel takes the raw CSV file characters in 16KB chunks (called a block), then it figures out the number of rows within a block and the total number of rows in all of the blocks. After that, the location of each row in a block is calculated.

NVIDIA Nsight Systems profiler gives a report called `"cuda_gpu_kern_sum"`, which summarizes the CUDA kernels running on the GPU and their execution times. Utilizing

this stats report obtained by profiling `cudf.read_csv()` of different file sizes, the first kernel, `"convert_csv_to_cudf"`, accounts for 43.23% of the GPU kernel execution time on average. The `"data_type_detection"` kernel accounts for 38.75% of the GPU kernel execution time on average. This means that, on average, those two kernels account for 81.98% on average of the time taken by the GPU to perform the `cudf.read_csv()` operation.

In the original implementation of cuDF, those kernels are executed in a single stream, which is the default stream. We have modified the source code of `"csv_gpu.cu"` file to utilize multiple streams in order to improve the performance of those two kernels. This performance improvement is achieved by allowing multiple streams to run independently of each other, introducing an overlap in the execution of the streams, and reducing the overall kernel execution time.

To convert the kernels to run on multiple streams, we first start by changing the kernel launch setup. In the kernel launch setup, we start by defining the number of streams to be used. After performing some experimentations on multiple numbers of streams (1, 2, 3, 4, 16, and 32), the optimal performance gain is found to be in using three streams. Then, those streams are created. Based on the number of streams, the CSV file was divided into multiple chunks for each stream to launch the kernel on its intended chunk. The CSV data is represented in span, which gives a lightweight C++ view of the data in memory. The span is divided based on the stream index to point only to the allotted chunk of data the stream should process.

After taking care of the kernel launch setup and ensuring that each stream will launch a kernel on the appropriate chunk of data, the kernel itself is modified to keep a global understanding of the CSV file and the DataFrame it is processing. To achieve this global view, the parameters of the kernels are changed to include the stream index. Based on the stream index and the chunk size, loop tiling technique is followed to improve the locality of reference of each thread in the grid. Processing the data in tiles increases the chances of threads within the thread blocks accessing data available in the cache, reducing cache miss penalties. The ID of the records used to process the CSV data is changed to reflect the global ID of the entry in the full CSV file.

After performing the aforementioned changes to the kernels and functions launching the kernels, we can visually see the change on the launched kernels by profiling the `cudf.read_csv` after the code modifications. The shows the two target kernels are now executing on multiple CUDA streams as see in *FIGURE 20* (size of file read is 9.81GB).

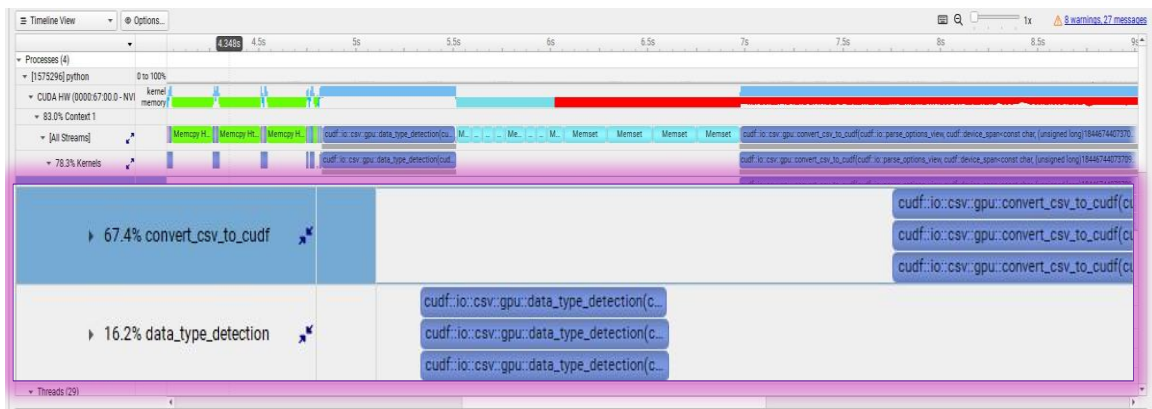


Figure 20. NVIDIA Nsight Systems Profiling showing Multi-Stream Execution of GPU Kernels

The correctness of this optimization was confirmed by ensuring that the new implementation produces the same DataFrame as the original implementation. To do so, a CSV file is read with/without the optimization, then those DataFrames are converted back

to CSV using `cudf.to_csv()`. The newly created CSV files are read into two Pandas DataFrames, then using `"pandas.DataFrame.equals"` function we were able to confirm that all of the elements in the CSV files are the same. We used Pandas DataFrame to perform the last check to make sure that the changes we made to the source code of cuDF did not make any other undesired changes that might have changed the functionality of cuDF without noticing.

5.2.4 Processing Large Chunks of Data

`cudf.read_csv()` function utilizes a cuDF-IO CSV reader implementation defined in `"reader_impl.cu"` file in cuDF source code. This file defines a function called `"load_data_and_gather_row_offsets"`. This function defines a statically set constant variable called `"max_chunk_bytes"` set to handle 64MB chunks of data. This variable is used to set the buffer size used to create a `"device_uvector"` which is responsible for allocating device memory for a specific stream. The `"max_chunk_bytes"` variable is also used to keep track of the section of the CSV data to be processed to load into a cuDF DataFrame. Using such a small chunk size of 64MB will require a huge number of chunks to be created and processed, with each chunk created requiring increasing the `"device_uvector"` to allocate more memory on the device until the file is copied into the device memory to be further processed to produce the DataFrame.

It is expected when dealing with GPU-assisted data processing that the file sizes are sufficiently large to benefit from the massive parallelism offered by GPUs. With large file sizes, the number of chunks will grow quite high. The overhead associated with preparing

each chunk, from memory allocation expansions to data transfer of relatively small data sizes, will cause degradation in the performance.

This led us to try and find an optimal chunk size that would streamline the process of reading the loading the data into the device memory without causing any other bottlenecks. The source code already safeguards against creating a buffer size that exceeds the size of the data. It also ensures that the chunks to be read do not exceed the size of the data. After testing multiple sizes of chunks (64MB, 128MB, 256MB, 512MB, 1GB, 1.5GB, and ~2GB), the improvements saturated at 1GB chunk size as can be seen in *FIGURE 21*. So, “max_chunk_bytes” was set to represent a chunk size of 1GB as the optimal chunk size of data to be used to load CSV data into a cuDF DataFrame. The change made to the chunk size was tested on a low-end commodity, “NVIDIA GeForce GTX 1650 Ti Mobile 4GB Memory” GPU, and it did not cause any errors. This ensures that this optimization is safe to be generalized across the CUDA-enabled GPU lineup.

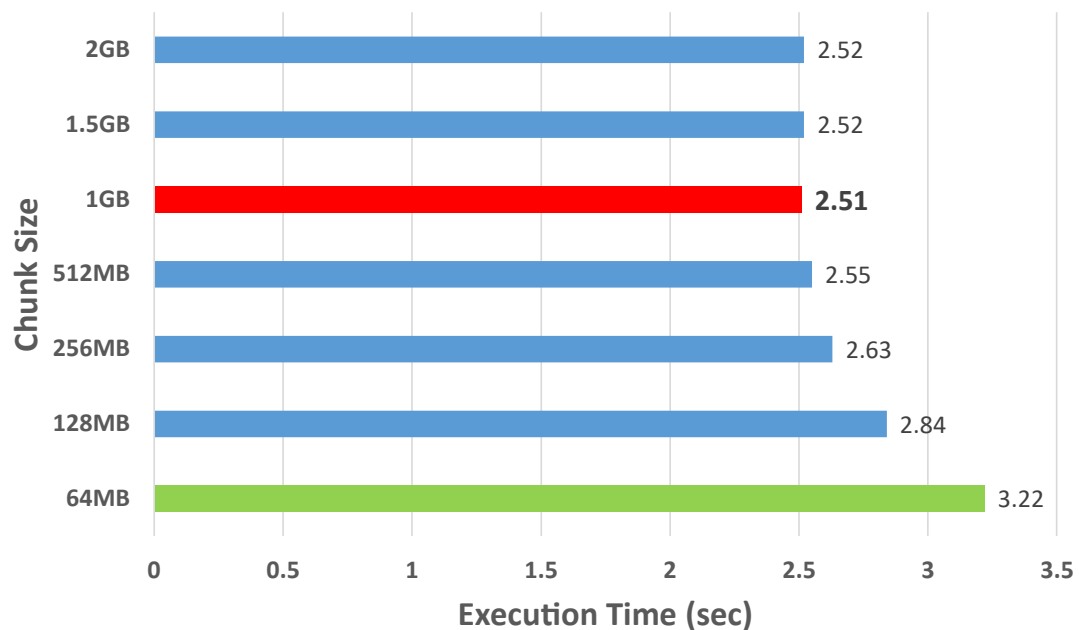


Figure 21. Execution Time of cudf.read_csv with Multiple Chunk Sizes (Tested on ~6GB file)

CHAPTER 6

RESULTS AND DISCUSSION

This section presents the results obtained before and after applying the optimizations discussed in *OPTIMIZATION IMPLEMENTATION*. The convention followed in the legend of the presented figures shows the library being tested followed by the optimization employed [Library-Name_Optimization-Employed]. The *DISCUSSION* part in this section sheds light on “how” and “why” the implemented optimizations work and their effect on the obtained results.

6.1 Results

6.1.1 Baseline Evaluation

FIGURE 22 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames without making any changes to the source code of cuDF, except for the red curve which used the multi-GPU version of cuDF. This figure establishes the baseline that can be used to track the effect of the optimizations. The effect can be seen on the speed of executing the operations using the different libraries, and on the size of the dataset each library can handle.

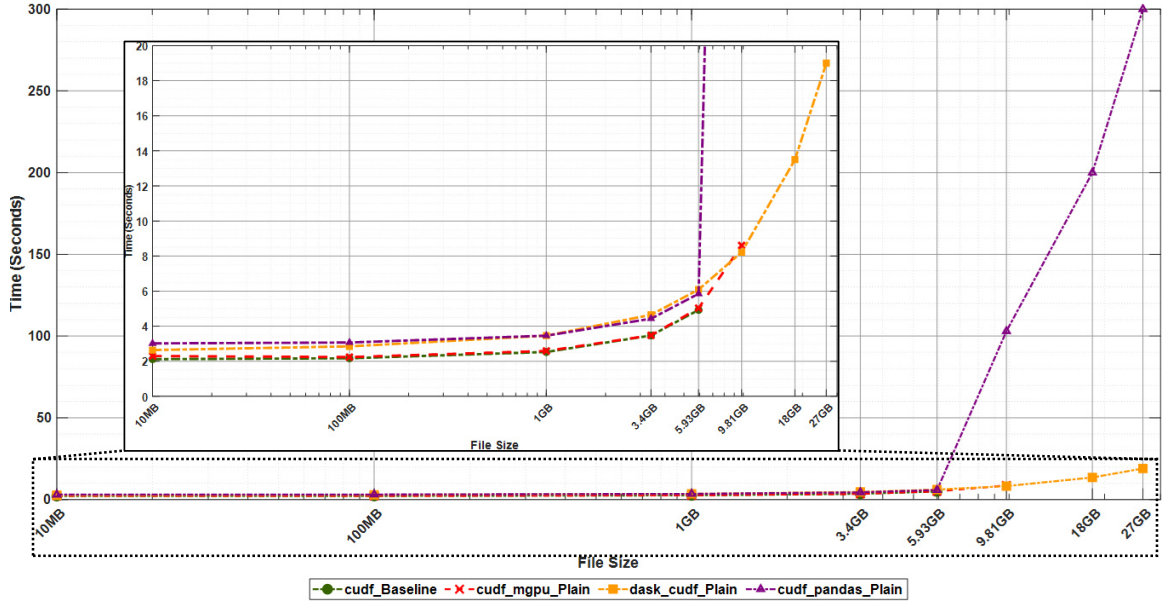


Figure 22. Execution Time of Read CSV & Max (Baseline)

6.1.2 RMM Evaluation

FIGURE 23 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when the dynamic memory management strategy is employed.

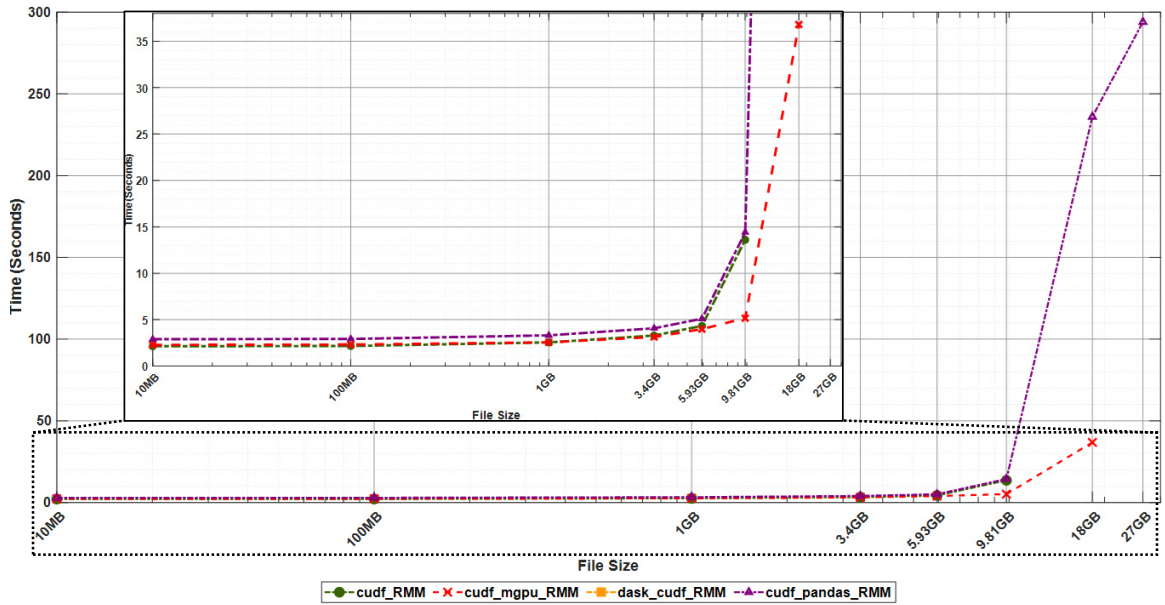


Figure 23. Execution Time of Read CSV & Max (RMM)

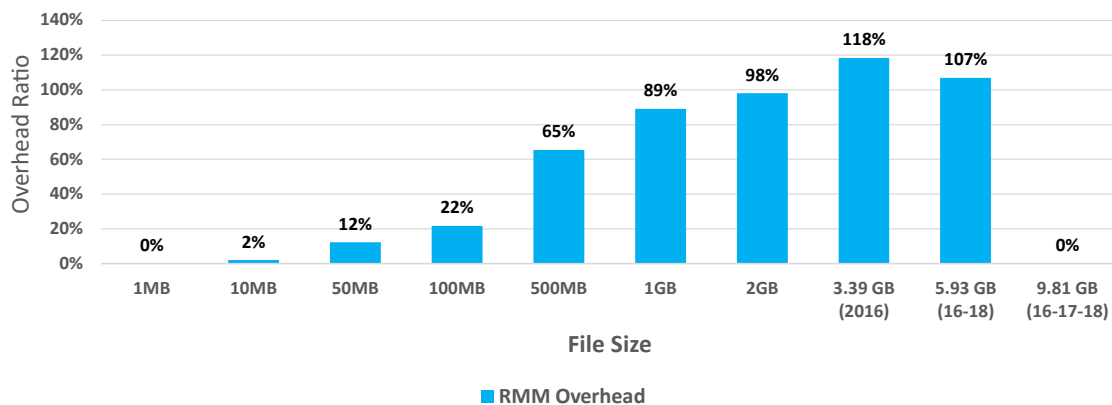


Figure 24. cudf_RMM GPU memory consumption overhead (manual setting of pool size) against memory consumed by cuDF without RMM.

FIGURE 24 shows the overhead of utilizing RMM customized memory allocation strategies on GPU memory utilization after reading each file size against plain cuDF. This figure tests the GPU memory consumption by manually setting the pool size based on the optimal pool size for memory consumption. The readings stop at the 9.81GB file since the original implementation of cuDF could not read of file of this size.

6.1.3 SMs Evaluation

FIGURE 25 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when multiple CUDA streams optimization is employed.

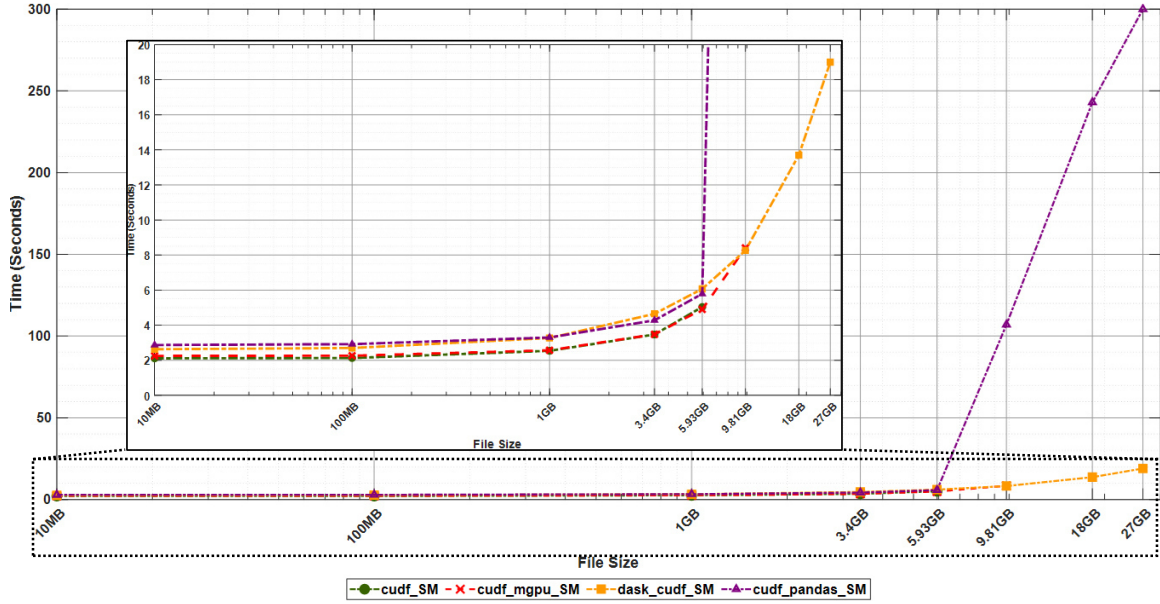


Figure 25. Execution Time of Read CSV & Max (SM)

6.1.4 Optimal Chunk Size Evaluation

FIGURE 26 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when optimal chunk size optimization is employed.

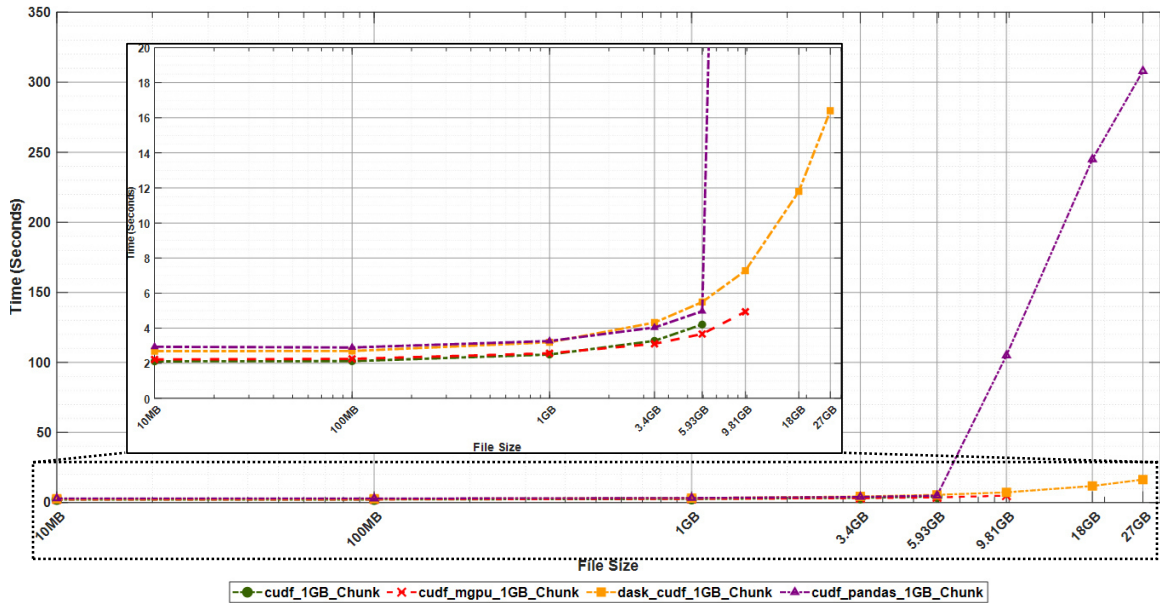


Figure 26. Execution Time of Read CSV & Max (1GB_Chunk)

6.1.5 RMM & SMs Evaluation

FIGURE 27 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when dynamic memory management strategy and multiple CUDA streams optimizations are employed together.

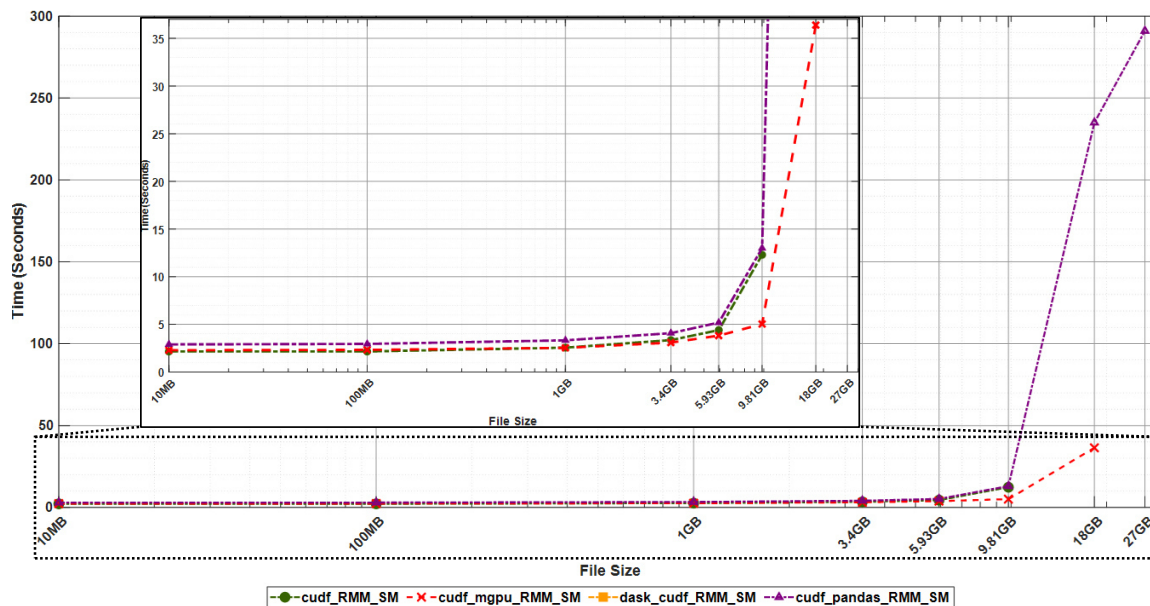


Figure 27. Execution Time of Read CSV & Max (RMM & SMs)

6.1.6 RMM & Optimal Chunk Size Evaluation

FIGURE 28 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when dynamic memory management strategy and optimal chunk size optimizations are employed.

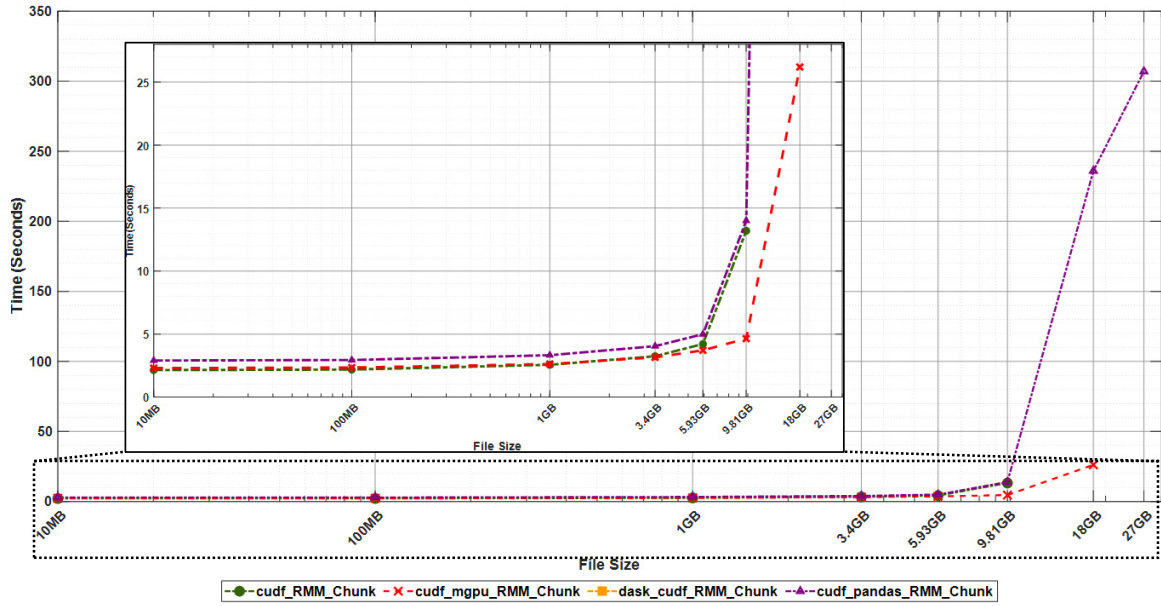


Figure 28. Execution Time of Read CSV & Max (RMM & 1GB_Chunk)

6.1.7 SMs & Optimal Chunk Size Evaluation

FIGURE 29 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when multiple CUDA streams and optimal chunk size optimizations are employed.

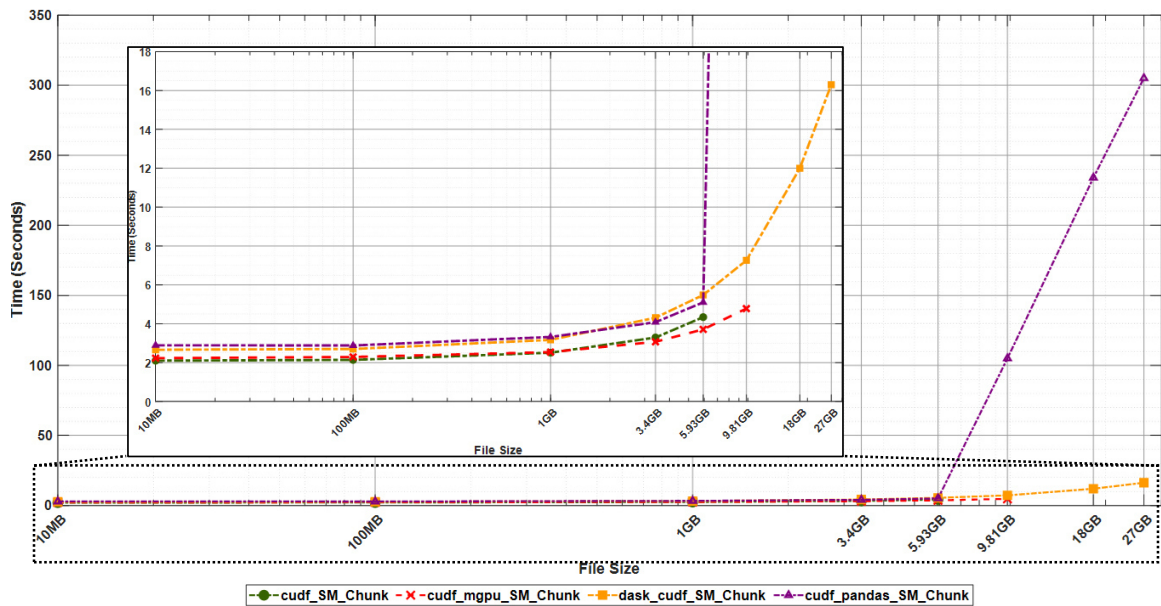


Figure 29. Execution Time of Read CSV & Max (SM & 1GB_Chunk)

6.1.8 RMM & SMs & Optimal Chunk Size Evaluation

FIGURE 30 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames when all of the optimizations, dynamic memory management strategy, multiple CUDA streams, and optimal chunk size are employed.

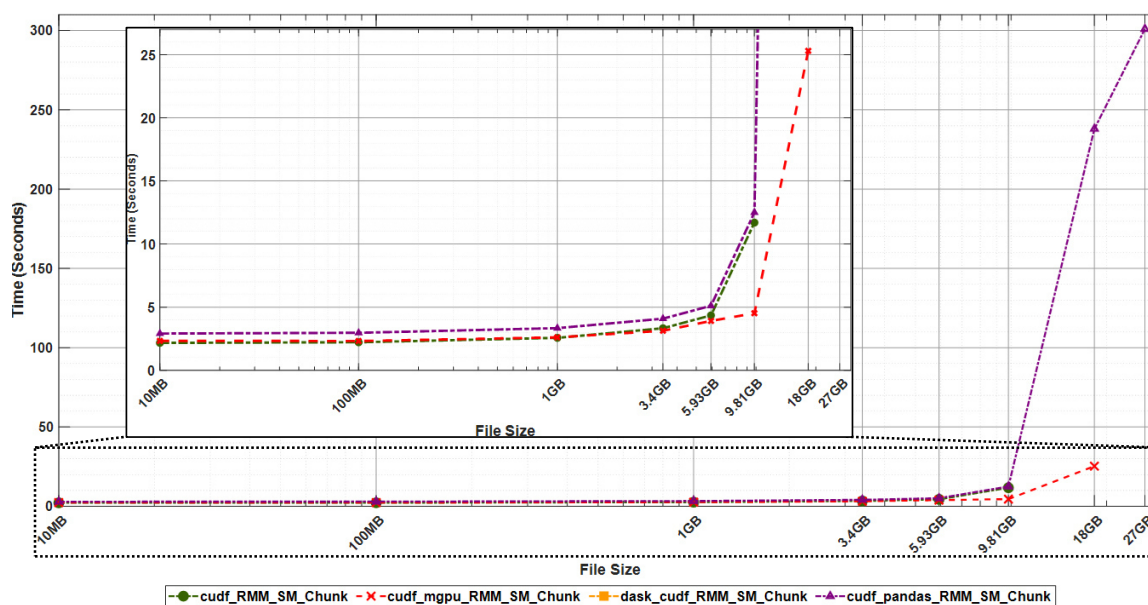


Figure 30. Execution Time of Read CSV & Max (RMM & SM & 1GB_Chunk)

6.1.9 RMM & SMs & Optimal Chunk Size Vs. Baseline Evaluation

FIGURE 31 shows the execution time of running the Python scripts performing `read_csv` & `max` operations using different file sizes on the DataFrames. The figure compares the performance of the baseline plain cuDF against a version of cuDF that employs a managed memory strategy, only on large files that are not read by cuDF represented by the blue curve, against a version of cuDF with all of the optimizations employed on a single-GPU version and the multi-GPU version of cuDF.

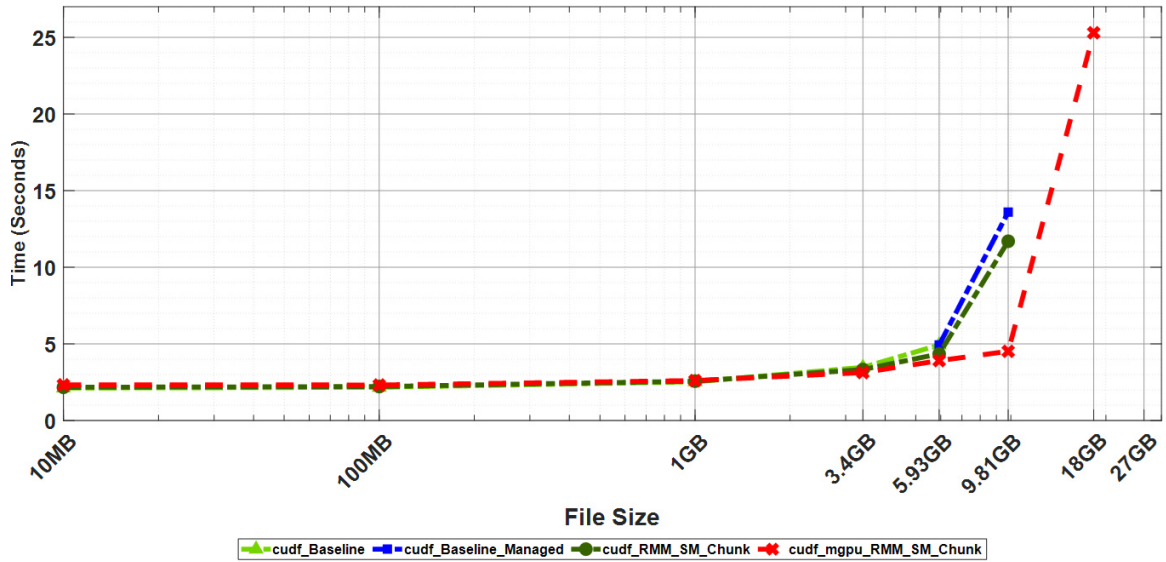


Figure 31. Execution Time of Read CSV & Max - Baseline Vs. Optimized cuDF (RMM & SM & 1GB_Chunk)

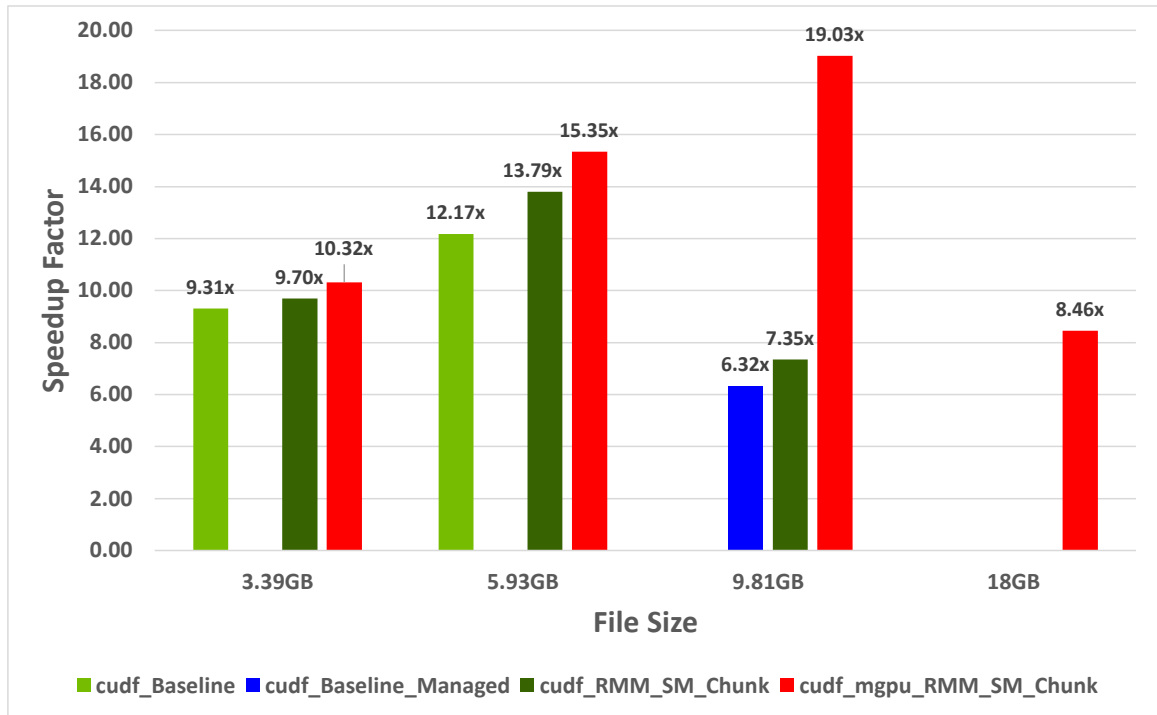


Figure 32. Speedup of Read CSV & Max Against Pandas

FIGURE 32 shows the speedup gains in executing the Python scripts against Pandas. This figure compares GPU-accelerated DataFrames against Pandas DataFrame, which runs on

the CPU. Because large files cannot be read by the original implementation of cuDF, Pandas DataFrame was selected as the baseline in this figure. When the original implementation of cuDF causes an out-of-memory error, managed memory alone is used with plain cuDF to indicate the optimization effect on larger files.

FIGURE 33 shows the speedup gain for each scenario, considering the multi-GPU version only, in the results section against the original implementation of cuDF.

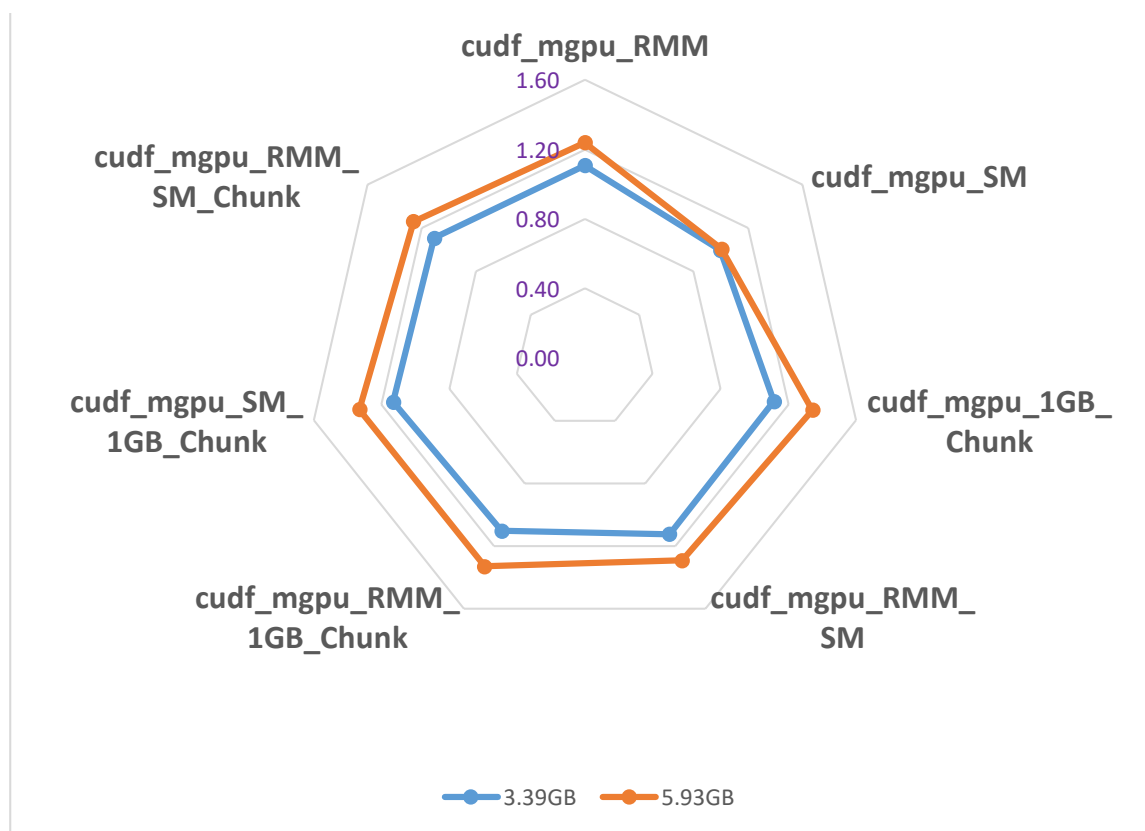


Figure 33. Execution Time Speedup Visualization of the different optimizations Against Plain cuDF

In order to have a bigger picture of the impact those optimizations have on larger files, FIGURE 34 shows the speedup gains of optimized versions of multi-GPU cuDF with RMM against Pandas.

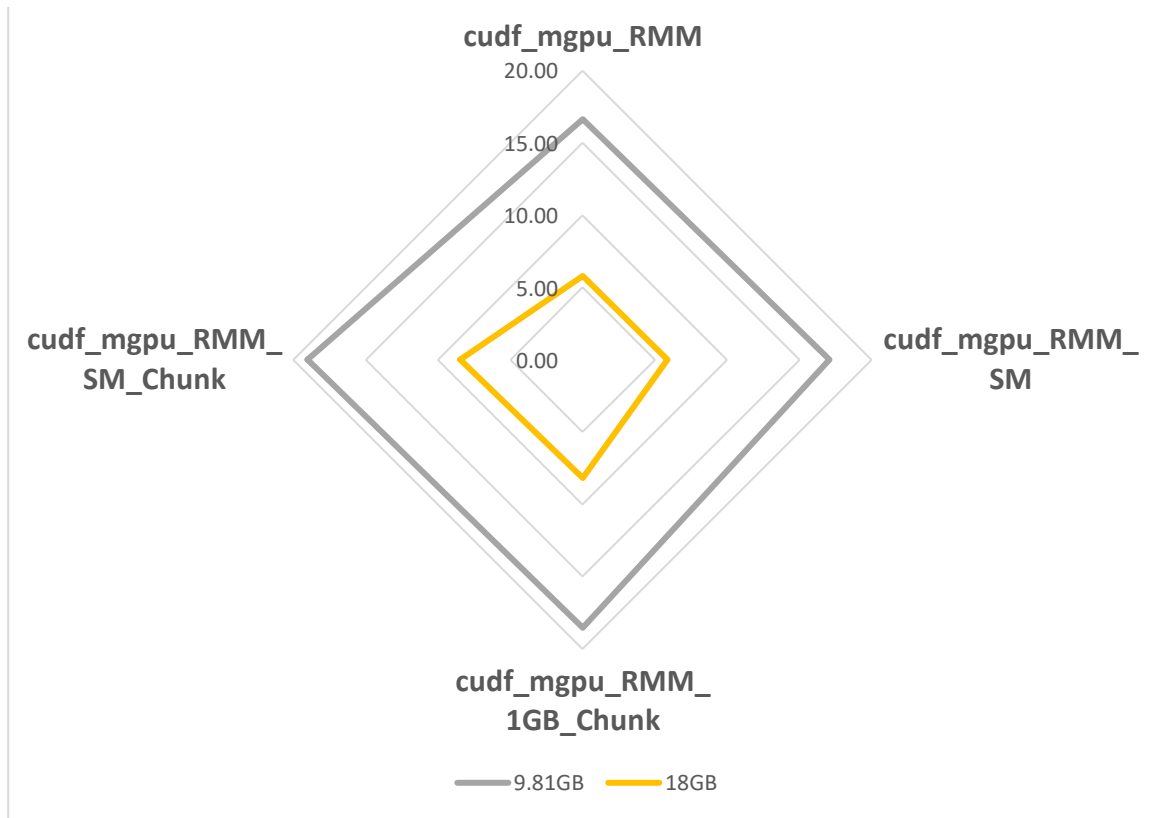


Figure 34. Execution Time Speedup Visualization of the different optimizations Against Pandas

FIGURE 35 shows the load capacity gain in file sizes by the optimized version of cuDF running on a single GPU in dark green and on multi-GPUs shown in red. The gains are in comparison with the original implementation of cuDF.

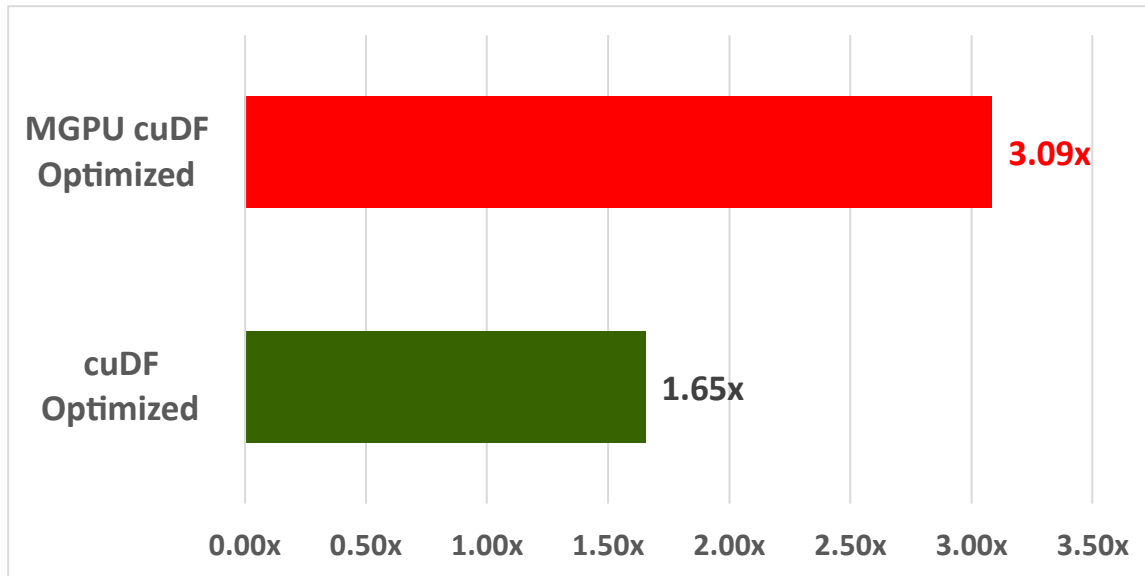


Figure 35. Load Capacity Gain in Readable File Sizes over Original cuDF

6.1.10 Deep Learning Model Data Preprocessing

TABLE 3 shows the speedup in data preprocessing operations used for the model training.

The speedup is measured against the original implementation of cuDF. The size of the data used to obtain those numbers is based on the largest file size plain cuDF was able to handle.

Table 3. Optimized cuDF Data Preprocessing Operations Speedup (tested on 3.39GB file)

Operation	Optimized vs Original cuDF Operation Execution Time Speedup	Operation Execution Time Ratio
<code>cudf.to_datetime</code>	72.47%	8.95%
<code>cudf.set_index</code>	104.20%	0.55%
<code>cudf.resample('D').mean().bfill()</code>	8.33%	90.50%
Overall Preprocessing Speedup	14.60%	100%

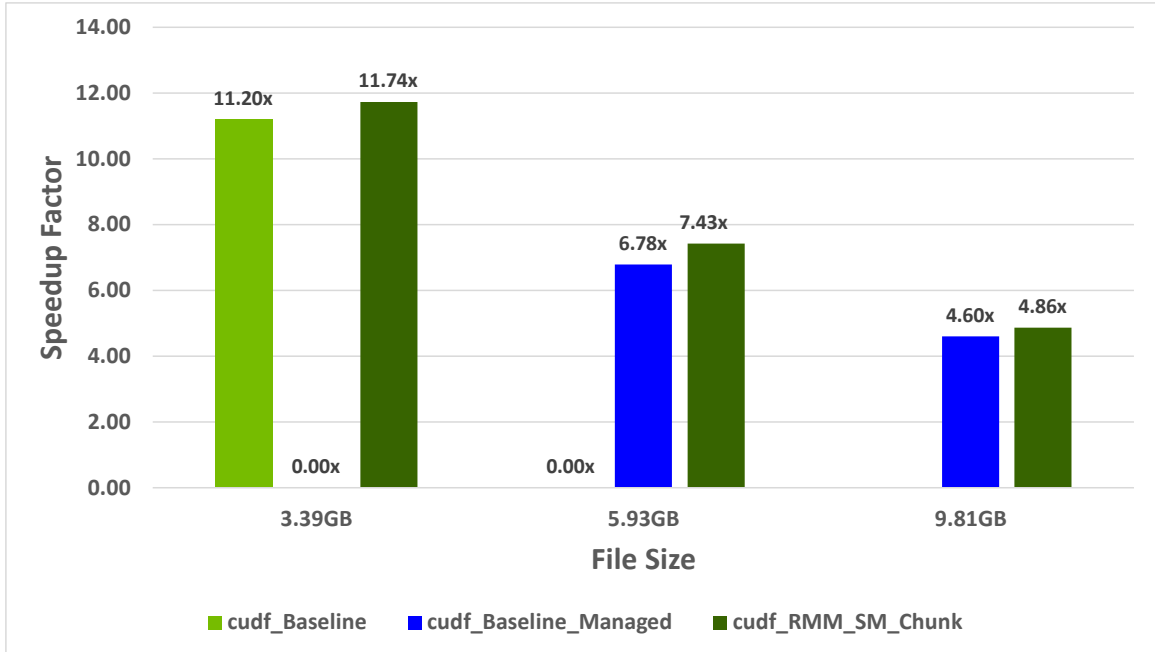


Figure 36. Data Preprocessing Speedup Against Pandas DataFrame.

FIGURE 36 Shows the speedup in performing the data preprocessing operation to be fed to the model. The speedup is measured against performing the same operations using Pandas DataFrame. Because the original implementation of cuDF failed to go beyond the 3.39GB dataset size, Pandas is used as a baseline to measure the speedup. When cuDF fails to read, managed memory (shown in blue) is used with plain cuDF to give an idea of the effect of the rest of the optimizations.

In FIGURE 37, we can see the dataset size that can be handled by the original implementation of cuDF vs. the optimized version of cuDF. The capacity is calculated to emulate a realistic scenario, where a deep learning model is to be trained on the MeteoNet Dataset. Based on the preprocessing operations in the model, the capacity is calculated based on the largest file that allows the model to complete its training.

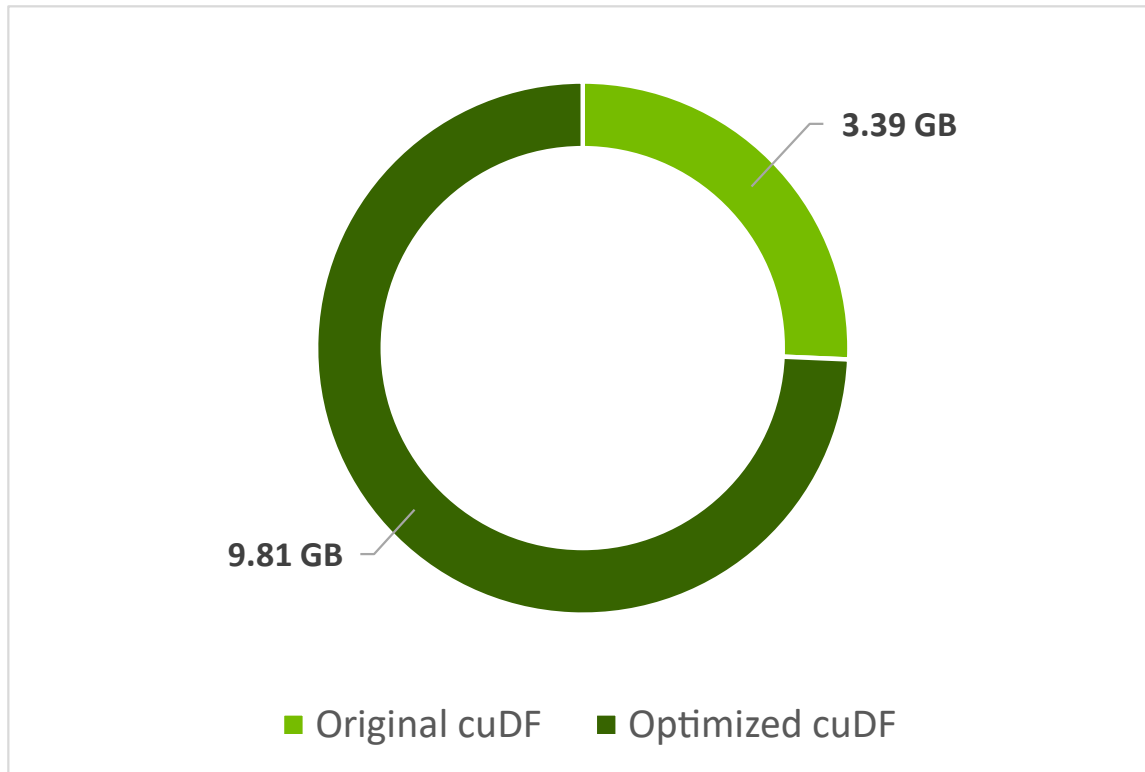


Figure 37. Dataset Size Capacity for Deployed Deep Learning Model Data Preprocessing.

6.2 Discussion

This section discusses the impact of the results obtained on the Execution Time (E.T.), GPU memory consumption, deep learning model data preprocessing, and the profiling results of the experiments.

6.2.1 Execution Time Analysis

The E.T. analysis is performed on the single-GPU and multi-GPU optimized versions of cuDF.

Single-GPU Analysis

Starting with the single-GPU optimized version of cuDF versus the original implementation of cuDF, *TABLE 4* shows the obtained speedups. The table shows the effect of each optimization on the execution of the Python script. The speedups are measured on the file sizes plain cuDF could handle.

Table 4. Speedup in Execution time against Original cuDF (Single GPU)

File Size	3.39GB	5.93GB
cuDF Version		
cudf_RMM	1.05x	1.14x
cudf_SM	1.00x	0.98x
cudf_1GB_Chunk	1.06x	1.17x
cudf_RMM_SM	1.04x	1.12x
cudf_RMM_1GB_Chunk	1.07x	1.17x
cudf_SM_1GB_Chunk	1.05x	1.14x
cudf_RMM_SM_Chunk	1.04x	1.13x

Utilizing RMM yields a speedup of 5-14%, whereas increasing the chunk size to 1GB results in a 6-17% speedup. The version that executes over multiple CUDA streams gave a slightly worse performance than the baseline, however, as will be seen in the following table, as the file size increases, the SM version starts to contribute to a positive speedup in execution. The noticed trend is that as the file size increases, the divergence between the baseline and the optimized versions increases. The best speedup obtained is when utilizing RMM and a 1GB chunk size, which results in a 17% speedup.

Pandas is used as another frame of reference to measure the speedup in files that cannot be read using the original cuDF implementation. *TABLE 5* shows the obtained speedups of the optimized version vs. Pandas. A speedup of 0 indicates that the file cannot be read.

Table 5. Speedup in Execution time against Pandas (Single-GPU)

File Size \ cuDF Version	3.39GB	5.93GB	9.81GB
cudf_Baseline	9.31x	12.17x	0.00
cudf_RMM	9.82x	13.86x	6.32x
cudf_SM	9.31x	11.88x	0.00
cudf_1GB_Chunk	9.85x	14.25x	0.00
cudf_RMM_SM	9.64x	13.64x	6.99x
cudf_RMM_1GB_Chunk	9.97x	14.25x	6.52x
cudf_SM_1GB_Chunk	9.82x	13.82x	0.00
cudf_RMM_SM_Chunk	9.70x	13.79x	7.35x

On the largest file size, a single-GPU-optimized version can read (9.81GB file), SMs give a speedup of ~11-13% when compared with the version that does not use multi-streams versions. Overall, it is noticed that the multi-streams versions perform the best when the file sizes are the largest that can be handled, and a managed memory scheme is used.

Multi-GPU Analysis

As for the multi-GPU supported version of cuDF against the original cuDF version, as seen in the results section in *FIGURE 22*, loading CSV data using the `read_csv_mgpu()` function does not yield any execution time improvements when compared to the original cuDF implementation on the file sizes it can handle. This in itself can be interpreted as a good indicator, as it signifies that the incurred data distribution overhead is relatively countered by the load-balanced execution over multiple GPUs. However, applying the optimizations one by one, we can see improvements in the execution time, especially as the file sizes grow. As seen in *FIGURE 31*, the divergence starts to become clear beyond the 3.39GB file size, where the optimized versions (on single-GPU and multi-GPU) execute their scripts faster.

FIGURE 33 and FIGURE 34 can give us a general idea about the effect of each optimization and how it impacts the performance. We can see that as the file size increases, the speedup also keeps increasing. The test cases are limited to file sizes that can be handled by cuDF on the setup used in the experiments without causing an out-of-memory error.

Table 6. Speedup in Execution time against Original cuDF (Multi-GPU)

File Size cuDF Version	3.39GB	5.93GB
cudf_mgpu_RMM	1.11x	1.24x
cudf_mgpu_SM	1.00x	1.01x
cudf_mgpu_1GB_Chunk	1.12x	1.34x
cudf_mgpu_RMM_SM	1.12x	1.29x
cudf_mgpu_RMM_1GB_Chunk	1.10x	1.33x
cudf_mgpu_SM_1GB_Chunk	1.13x	1.33x
cudf_mgpu_RMM_SM_Chunk	1.11x	1.26x

From TABLE 6, we can see that speedup gains for each optimization against cuDF. The RMM optimization yields 11-24% speedup, while the 1GB chunk size yields 12-34% speedup, depending on the file size. The multi-streamed version yields the lowest speedup of ~1X, meaning no degradation in performance in the case of the 3.39GB file. As the file sizes increase, multiple streams version of cuDF slowly starts to perform better, contributing to ~3.5% improvement in the execution time. The best-case scenario for the execution time speedup for relatively small files is obtained when utilizing multi-GPU with RMM and 1GB chunk size, where the speedup against plain cuDF is 1.33X. Although this speedup is not obtained by the version that implements all of the optimizations, the fully optimized version makes up for it when handling the largest file this version can handle as can be seen next.

Given reading files beyond ~6GB in cuDF using the extermination setup fails, the speedup for all dataset file sizes is tested against pandas, as shown in *TABLE 7*. A speedup of 0 indicates that the file cannot be read.

Table 7. Speedup in Execution time against Pandas (Multi-GPU)

File Size	3.39GB	5.93GB	9.81GB	18GB
cuDF Version				
cudf_Baseline	9.31x	12.17x	0.00x	0.00x
cudf_mgpu_RMM	10.32x	15.08x	16.67x	5.82x
cudf_mgpu_SM	9.28x	12.24x	10.23x	0.00x
cudf_mgpu_1GB_Chunk	10.39x	16.35x	19.15x	0.00x
cudf_mgpu_RMM_SM	10.45x	15.71x	17.06x	5.88x
cudf_mgpu_RMM_1GB_Chunk	10.25x	16.17x	18.53x	8.17x
cudf_mgpu_SM_1GB_Chunk	10.52x	16.17x	17.99x	0.00x
cudf_mgpu_RMM_SM_Chunk	10.32x	15.35x	19.03x	8.46x

The overall best-case scenario for the execution time speedup is when a file cannot fit in a cuDF DataFrame without utilizing managed memory versus a multi-GPU version, where each chunk of the file can fit in the memory of its respective GPU. In such cases, as we can see in *FIGURE 31*, the speedup of the multi-GPU optimized cuDF can reach over 3.01X when compared to a cuDF version that only utilizes a managed memory scheme.

When comparing the multi-GPU optimized version with the single-GPU optimized version on the largest file size a GPU can handle (i.e. 9.81GB), we get a speedup in the execution time of 2.59X. This represents the best-case scenario in comparing multi-GPU vs single-GPU versions because the single-GPU version would be utilizing managed memory to handle the file, which makes it slower, while the multi-GPU version is utilizing memory pooling, which makes cuDF faster, especially with large files.

It is also noticed from the results that Dask and cuDF Pandas DataFrames stand to benefit from the implemented optimizations as they use cuDF as a backbone while performing the `read_csv` operation. Those improvements can reduce the execution time of the Python scripts by a factor reaching 16.5% in the case of Dask against its baseline. cuDF Pandas DataFrame can reduce its execution time by 87% in the best-case scenario compared with its baseline. It must be said that this case occurs when RMM is employed, where the baseline version falls back to CPU execution due to out-of-memory error with cuDF.

Dask_cuDF DataFrame dominates all other GPU DataFrames tested when it comes to large files, i.e. larger than 9.81GB. This is because Dask performs the operation partitions, i.e. chunks, making its operation execute much faster. However, this impressive performance comes at a cost. As stated, `dask_cudf.read_csv()` is lazy, the data is only transferred to the GPU when it is needed, and then the data is ejected from the GPU's memory. Calling the same operation, e.g. `max`, will incur the same cost every time it is called. This makes it inconvenient during data exploration, as the data exploration process can require many operations that need the data to be transferred, and the cost of transfers will accumulate over many operations. There is an option in Dask_cuDF that forces the data to persist in memory. However, even when utilizing the persisting data, an out-of-memory error is encountered on the 18GB file, while the multi-GPU optimized version cuDF was able to handle this file size.

6.2.2 Memory Capacity Analysis

The multi-GPU supported version of cuDF, excluding all other optimizations as seen in *FIGURE 22* shows higher support for handling larger data sizes, given the added memory

utilized on the second GPU. The memory utilization improvements are extended much further when utilizing a managed memory scheme allowing for larger files to be read. Those improvements in load capacity gains can be extended in the single-GPU version of the optimized cuDF and on the multi-GPU version. The load capacity gain as seen in *FIGURE 35* shows that the single-GPU optimized version can allow the user to read files that are 1.65x larger than plain cuDF could read. The gain in multi-GPU cuDF reached more than 3x in comparison to plain cuDF.

Improvements in memory utilization are of paramount importance given that cuDF is quite memory-demanding. The fact that cuDF is memory demanding on top of GPU memory is the main constraint to GPU-centric tasks, making this improvement all the more important. For instance, cuDF DataFrame memory footprint, when reading a CSV file of size 5.93GB, produces a DataFrame that requires around 8.6GB of GPU memory. Whereas reading the same file on a Pandas DataFrame required less than 7.7GB of CPU memory. This means the cuDF requires an extra ~12% of memory to read the same file into a cuDF DataFrame. Add to it, cuDF does not support `inplace` operations as Pandas does. Although cuDF documentation might show the argument to enable `inplace` execution, asserting `inplace` argument in some operations leads to `“NotImplementedError(“`inplace` not currently implemented.”)` error. This rapid memory consumption can be attributed to the architecture and design goals on which cuDF is built and designed. cuDF prioritizes acceleration, i.e. faster execution of operations, over memory efficiency.

RMM memory strategies, although effective in different ways depending on the memory allocation strategy, come at a cost. The cost can be in terms of memory consumption when using memory pooling or in execution time when using managed memory. With memory pooling, we gain the benefit of faster loading of data. However, a large portion of memory is allocated for the process initiating the memory pool. *FIGURE 24* shows the impact of utilizing RMM memory pooling. It must be stated that the figure shows the best-case scenario, where the memory pool bounds are tightly configured. The reason why the largest file size shows 0% overhead is because cuDF runs out of memory, in which case, there is no comparison to be made with cuDF's default memory management strategy. For generalization purposes and to suit the largest number of scenarios, the memory pool size is set to the default settings where the initial size of the pool is set to half the available GPU memory, and the maximum pool size is set to use the total available memory.

6.2.3 Deep Learning Model Data Preprocessing Analysis

The main purpose of the model is to evaluate the performance of cuDF when handling deep learning models and to assess the robustness of the optimizations in facilitating the model handling larger datasets. As can be seen in *TABLE 3*, cuDF operations that are used in the data preprocessing of the model have gained a decent speedup. The CSV data loading operation is not shown in this table since it has been examined extensively. Scaling the speedup based on the execution time ratio of each operation, the data preprocessing portion is faster by a factor of 14.6%. This speedup is solely attributed to the memory management scheme as will be discussed in the profiling analysis section. *FIGURE 36* shows the speedup gains of the full data preprocessing procedure, including data loading.

Based on the data preprocessing operations used by a certain model, the data size that can be processed may differ depending on the operation. For instance, in our test case, the preprocessing routine performs some operations that do not require the creation of a new DataFrame, such as `to_datetime()` & `set_index()` operations. However, resampling the data causes an out-of-memory error if the dataset used is close to the limit of what cuDF can utilize of the GPU memory, which is, as stated previously ~31% of the GPU memory. Such operations are not done `in-place`, which means they might require intermediate data to handle the operation. The accumulation of the intermediate data consumes the remaining memory and causes an out-of-memory error. This approach by cuDF limits the size of the data that can be handled during preprocessing. *FIGURE 37* shows that the optimized version of cuDF can handle much larger datasets going through the same preprocessing routine. This is because of the utilization of managed memory, allowing cuDF to spill data to host memory if it exceeds the GPU capacity. This gives the user the ability to handle a workload nearly 2.9x larger than the original implementation of cuDF.

6.2.4 Profiling Analysis

Profiling a Python script that performs a `read_csv` function of a 5.93GB file gives us the following figure,

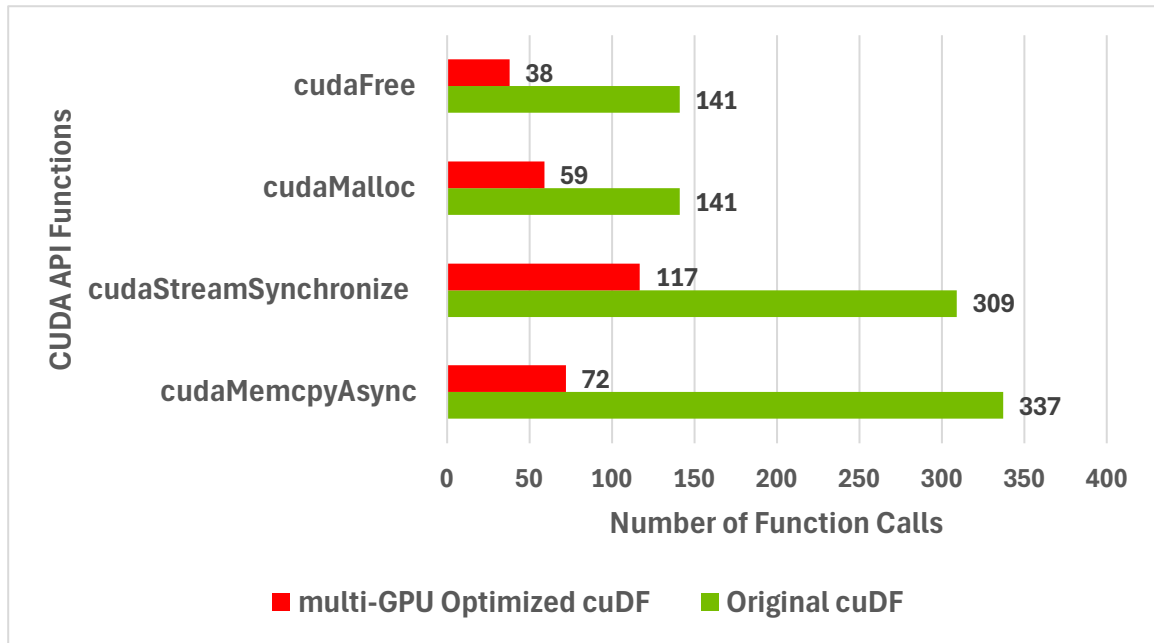


Figure 38. CUDA API Function Calls plain cuDF vs Optimized multi-GPU cuDF

Those CUDA API functions account for over 90% of the execution time of CUDA API functions that were executed during the profiling. Those functions take ~50% less time to execute on the multi-GPU version. “cudaMemcpyAsync” accounts for 70% of the CUDA API execution time on the original cuDF implementation. In contrast, it accounts for 43% only on the multi-GPU version. This function is called for copying data between host and device memories.

The four functions are shown to account for ~85% of the execution time of the script reading a CSV file. This indicates that data transfer between host and device significantly impacts the execution of loading a CSV file. The multi-GPU version stands to better utilize the communication channel between the host and GPUs, contributing to the ~20% reduction in execution time.

As for changing the “max_chunk_bytes”, the profiler shows that this changing this variable reduces the “gather_row_offsets_gpu” kernel instances launched to

gather the necessary information about the rows and their offsets for further processing. Although the number of launches of this kernel is reduced by 92%, yet the kernel execution time differs in less than 3 microseconds only. However, the number of instances of this kernel launches is indicative of the number of buffers created to allocate memory on GPU and the number of data transfers between host and device, which are reduced as well.

Regarding the speedup noticed on other cuDF operations that were not explicitly optimized, as seen in the deep learning model sub-section of the results, this speedup is a byproduct of utilizing memory pooling. Profiling the data preprocessing Python script reveals that the optimized cuDF version executes GPU memory operations much faster as memory allocation operations are already completed. The memory operations in the optimized cuDF version are executed 22% faster, resulting in an improved overall performance.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

This thesis introduces several novel contributions and optimizations to cuDF to improve its performance and scalability. Starting by focusing on optimizing the inner workings of cuDF in terms of memory management, the adaptation of concurrent multiple CUDA stream execution of kernels used to load CSV data into a cuDF DataFrame, and the use of an optimal chunk size for optimal I/O bandwidth utilization. Those optimizations are then extended by changing the single-GPU workflow of cuDF to a novel system-wide multi-GPU execution workflow, allowing optimal utilization of multi-GPU systems resources. the scalability of this novel version of cuDF improves its scalability on two levels, on the “component level” where cuDF on a single GPU can handle larger datasets. The second level is on the “horizontal level”, where cuDF utilizes available CUDA-enables devices to distribute the workload.

Those optimizations resulted in an improved data loading capacity by a factor of 1.65X on a single-GPU version compared with the original cuDF implementation and by a factor of 3.09X on the multi-GPU version on the setup used in this thesis. The optimizations have also resulted in improved execution time performance, where the optimized multi-GPU version of cuDF is faster than the original cuDF by 1.26X. This factor is limited by the data sizes the original cuDF implementation could handle. Comparing the single-GPU optimized version with the multi-GPU version on the largest file size the single-GPU

version could handle gives us a speedup factor of 2.59X in the best-case scenario. Yet, the multi-GPU version could still handle a file double the size of the maximum file the optimized single-GPU version could handle.

Testing the optimized version of cuDF on a weather forecasting deep learning model as a use-case resulted in an improved performance in execution time of data preprocessing operation by a factor of 1.14X in comparison with the original cuDF in a single-GPU environment. It also allowed the optimized version to handle a 2.9X larger dataset compared to the original version.

The performance and scalability of the optimized versions proved to allow for democratized GPU acceleration for deep learning model development on commonly available hardware. Where in certain cases, the optimized versions would allow its users to accomplish tasks that would require a more advanced setup to perform the same tasks.

7.2 Future Work

Going through the source code and the inner workings of cuDF to apply the optimization helped identify possible areas of optimization that would further improve the performance of cuDF. This section describes those areas of optimization that can be addressed in future work.

7.2.1 Multi—GPU Data Distribution

Data distribution over multiple GPUs is implemented by relying on the `cudf.read_csv()` function arguments to define the byte range a GPU should read. This makes the process easier, as the inner workings of cuDF handle CSV file parsing, which utilizes “libcudf”,

the core C++ library powering cuDF. Using “byte_range” takes care of aligning the data within cuDF DataFrame. Thus, each GPU gets the final row in its range fully. However, as we can see in *FIGURE 39*, distributing the data manually between the GPUs can lead to a 37% reduction in the execution time of the Python script.

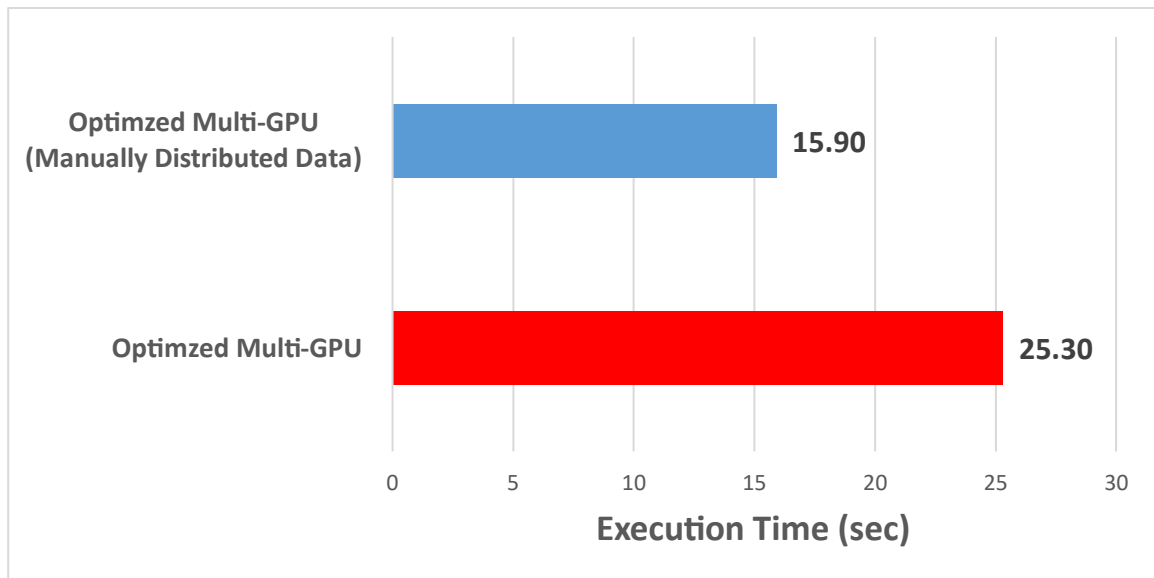


Figure 39. Execution Time of Optimized Multi-GPU cuDF w/o Manually Distributed Data (tested on 18.3GB file)

We have tested distributing the files using “skiprows” & “skipfooter” arguments in the `cudf.read_csv()` function. However, the performance was much worse than using “byte_range” as the logic that relies on those parameters to split the file reads the full file for each GPU and then returns the specified portion of the file into the DataFrame.

Implementing a mechanism that correctly distributes the files more efficiently between GPUs has shown to have good potential in improving the performance of the optimized multi-GPU cuDF quite drastically when handling large files.

7.2.2 Dynamic Chunk Size

Identifying the underlying factors that determine the optimal chunk size used to transfer CSV data from the host can help in the dynamic selection of optimal chunk size based on the hardware capabilities of a system. As seen in the discussion, chunk size can reduce the overhead associated with many operations depending on how many chunks are needed to process a CSV file.

7.2.3 cuDF Memory Consumption

Improving how cuDF handles memory allocation, as we have seen, if the raw file size is over ~31% of the available GPU memory, cuDF would not be able to read the data and it would return “`cudaErrorMemoryAllocation out of memory`”. Possible areas of investigation include altering memory allocation implementation. Another area to look into is minimizing intermediate allocations by supporting splitting files into chunks to minimize the effect of those intermediate allocations similar to Pandas.

7.2.4 Application Adaptation

The cuML library already implements Principal Component Analysis (PCA) and K-Nearest Neighbor (KNN) algorithms to work with cuDF. However, given the new data structure returned by the multi-GPU version of cuDF, those algorithms need to be modified to accommodate those changes. This integration would enable these algorithms to scale efficiently across multiple GPUs, leveraging the distributed memory and computational capabilities to handle larger datasets. PCA could benefit from faster dimensionality reduction on high-dimensional data, while KNN could achieve improved neighbor search performance for massive datasets, making these methods more practical for large-scale machine learning workflows.

REFERENCES

- [1] “Nvidia overtakes Apple as world’s most valuable company | Reuters.” Accessed: Dec. 11, 2024. [Online]. Available: <https://www.reuters.com/technology/nvidia-overtakes-apple-worlds-most-valuable-company-2024-10-25/>
- [2] R. Vuduc and J. Choi, “A brief history and introduction to GPGPU,” *Modern accelerator technologies for geographic information science*, vol. 9781461487456, pp. 9–23, Aug. 2013, doi: 10.1007/978-1-4614-8745-6_2/FIGURES/5.
- [3] “CUDA by Example: An Introduction to General-Purpose GPU Programming - Jason Sanders, Edward Kandrot - Google Books.” Accessed: Dec. 01, 2023. [Online]. Available: https://books.google.com.sa/books?hl=en&lr=&id=49OmnOmTEtQC&oi=fnd&pg=PT16&dq=history+of+CUDA+&ots=tuZbYrSpP_&sig=UBnC13WOGiKbz5hGVCAqRDe7BW0&redir_esc=y#v=onepage&q=history%20of%20CUDA&f=false
- [4] J. Mitura, B. E. Chrapko, and O. Bachanek-Mitura, “MedVoxelHD: Improved CUDA-accelerated morphological Hausdorff distances in medical image analysis,” *SoftwareX*, vol. 26, p. 101744, May 2024, doi: 10.1016/J.SOFTX.2024.101744.
- [5] C. Feng, D. Zhao, and M. Huang, “Image segmentation using CUDA accelerated non-local means denoising and bias correction embedded fuzzy c-means (BCEFCM),” *Signal Processing*, vol. 122, pp. 164–189, May 2016, doi: 10.1016/J.SIGPRO.2015.12.007.

- [6] P. Iványi, “CUDA accelerated implementation of parallel dynamic relaxation,” *Advances in Engineering Software*, vol. 125, pp. 200–208, Nov. 2018, doi: 10.1016/J.ADVENGSOFT.2018.02.008.
- [7] T. Kemmer, S. Hack, B. Schmidt, and A. Hildebrandt, “CUDA-accelerated protein electrostatics in linear space,” *J Comput Sci*, vol. 70, p. 102022, Jun. 2023, doi: 10.1016/J.JOCS.2023.102022.
- [8] N. Perera *et al.*, “In-depth analysis on parallel processing patterns for high-performance Dataframes,” *Future Generation Computer Systems*, vol. 149, pp. 250–264, Dec. 2023, doi: 10.1016/J.FUTURE.2023.07.007.
- [9] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, “THE COMPUTATIONAL LIMITS OF DEEP LEARNING”.
- [10] N. K. Chauhan and K. Singh, “A review on conventional machine learning vs deep learning,” *2018 International Conference on Computing, Power and Communication Technologies, GUCON 2018*, pp. 347–352, Mar. 2019, doi: 10.1109/GUCON.2018.8675097.
- [11] Z. Hu, J. Tang, Z. Wang, K. Zhang, L. Zhang, and Q. Sun, “Deep learning for image-based cancer detection and diagnosis – A survey,” *Pattern Recognit*, vol. 83, pp. 134–149, Nov. 2018, doi: 10.1016/J.PATCOG.2018.05.014.
- [12] R. Gargeya and T. Leng, “Automated Identification of Diabetic Retinopathy Using Deep Learning,” *Ophthalmology*, vol. 124, no. 7, pp. 962–969, Jul. 2017, doi: 10.1016/J.OPHTHA.2017.02.008.

- [13] X. Liu *et al.*, “A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: a systematic review and meta-analysis,” *Lancet Digit Health*, vol. 1, no. 6, pp. e271–e297, Oct. 2019, doi: 10.1016/S2589-7500(19)30123-2/ATTACHMENT/49AB10F2-3AA4-4101-A155-11A5ED6772BC/MMC1.PDF.
- [14] R. Lam *et al.*, “Learning skillful medium-range global weather forecasting,” *Science (1979)*, Nov. 2023, doi: 10.1126/SCIENCE.ADI2336.
- [15] I. D. Mienye and N. Jere, “Deep Learning for Credit Card Fraud Detection: A Review of Algorithms, Challenges, and Solutions,” *IEEE Access*, vol. 12, pp. 96893–96910, 2024, doi: 10.1109/ACCESS.2024.3426955.
- [16] L. R. G. Carrillo and K. G. Vamvoudakis, “Deep-Learning Tracking for Autonomous Flying Systems under Adversarial Inputs,” *IEEE Trans Aerosp Electron Syst*, vol. 56, no. 2, pp. 1444–1459, Apr. 2020, doi: 10.1109/TAES.2019.2930017.
- [17] S. Alshahrani and E. Kapetanios, “Are deep learning approaches suitable for natural language processing?,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9612, pp. 343–349, 2016, doi: 10.1007/978-3-319-41754-7_33/TABLES/1.
- [18] B. P. Agrawal, P. S. Nair, S. B. G. T. Babu, R. Rajprabu, P. R. Pesaru, and L. Natrayan, “Deep Learning Methods for Detecting ImageBased Defects in Manufacturing Processes,” *Proceedings of 9th International Conference on Science, Technology, Engineering and Mathematics: The Role of Emerging Technologies in*

- Digital Transformation, ICONSTEM* 2024, 2024, doi: 10.1109/ICONSTEM60960.2024.10568644.
- [19] Y. Sun, J. Li, Y. Xu, T. Zhang, and X. Wang, “Deep learning versus conventional methods for missing data imputation: A review and comparative study,” *Expert Syst Appl*, vol. 227, p. 120201, Oct. 2023, doi: 10.1016/J.ESWA.2023.120201.
 - [20] F. Del Pup *et al.*, “The more, the better? Evaluating the role of EEG preprocessing for deep learning applications,” Nov. 2024, Accessed: Dec. 19, 2024. [Online]. Available: <https://arxiv.org/abs/2411.18392v1>
 - [21] B. Bala and S. Behal, “A Brief Survey of Data Preprocessing in Machine Learning and Deep Learning Techniques,” *8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud), I-SMAC 2024 - Proceedings*, pp. 1755–1762, 2024, doi: 10.1109/I-SMAC61858.2024.10714767.
 - [22] “Accelerating the Wide & Deep Model Workflow from 25 Hours to 10 Minutes Using NVIDIA GPUs | NVIDIA Technical Blog.” Accessed: Jan. 07, 2025. [Online]. Available: <https://developer-qa.nvidia.com/blog/accelerating-the-wide-deep-model-workflow-from-25-hours-to-10-minutes-using-nvidia-gpus/>
 - [23] A. Géron, “Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (2019, O’reilly),” *Hands-On Machine Learning with R*, p. 510, 2017, Accessed: Jan. 18, 2025. [Online]. Available: <https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>

- [24] “Bioimage Data Analysis Workflows-Advanced Components and Methods Kota Miura Nataša Sladoje Editors Learning Materials in Biosciences”.
- [25] A. Rule, A. Tabard, and J. D. Hollan, “Exploration and Explanation in Computational Notebooks,” 2018, doi: 10.1145/3173574.3173606.
- [26] “pandas download stats.” Accessed: Jan. 28, 2025. [Online]. Available: <https://pepy.tech/projects/pandas?timeRange=threeMonths&category=version&includeCIDownloads=true&granularity=daily&viewType=chart&versions=2.2.3%2C2.2.2%2C2.2.1>
- [27] D. Petersohn *et al.*, “Towards scalable dataframe systems,” *Proceedings of the VLDB Endowment*, vol. 13, no. 11, pp. 2033–2046, Jul. 2020, doi: 10.14778/3407790.3407807.
- [28] “RAPIDS cuDF Accelerates pandas Nearly 150x with Zero Code Changes | NVIDIA Technical Blog.” Accessed: Jan. 28, 2025. [Online]. Available: <https://developer.nvidia.com/blog/rapids-cudf-accelerates-pandas-nearly-150x-with-zero-code-changes/>
- [29] J. M. Chambers and T. J. Hastie, “Chambers, J. M. and Hastie, T. J.,” *Statistical models in S.*, p. 624, 1992, Accessed: Jan. 16, 2025. [Online]. Available: https://books.google.com/books/about/Statistical_Models_in_S.html?id=IBkLQA AACAAJ
- [30] “pandas: a Foundational Python Library for Data Analysis and Statistics.” Accessed: Jan. 16, 2025. [Online]. Available:

https://www.researchgate.net/publication/265194455_pandas_a_Foundational_Python_Library_for_Data_Analysis_and_Statistics?_sg=9QFAzlJkIKuUHp7gUzQmEsp5tQy9ab8gG7FGxLHfmuqwo8SUZu3utL-MnlgoQs_c7Ghs6MT2ifBDRxI&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6ImxvZ2luIiwicGFnZSI6Ii9kaXJlY3QifX0

- [31] “API reference — pandas 2.2.3 documentation.” Accessed: Jan. 28, 2025. [Online]. Available: <https://pandas.pydata.org/docs/reference/index.html>
- [32] “CUDA C++ Programming Guide.” Accessed: Dec. 19, 2024. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- [33] W.-M. Hwu and D. Kirk, “Thread Optimization - The University of Maryland Institute for Advanced Computer Studies.”
- [34] J. R. Glover, “Solving Binary Integer Programming Problems With Balas’ Additive Algorithm Using Graphics Processing Units,” Jun. 2024, doi: 10.32920/26052688.V1.
- [35] A. Anghel, G. Dittmann, R. Jongerius, and R. P. Luijten, “Spatio-Temporal Locality Characterization”.
- [36] A. Noelle and G. Dissertation, “Tackling GPU Memory Size Limitations,” 2023.
- [37] J. Zhao *et al.*, “Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically,” *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 451–466, Oct. 2022, doi: 10.1145/3559009.3569656.

- [38] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, “A Survey on Distributed Machine Learning,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, Mar. 2020, doi: 10.1145/3377454.
- [39] “Wes McKinney - Apache Arrow and the ‘10 Things I Hate About pandas.’” Accessed: Jan. 17, 2025. [Online]. Available: <https://wesmckinney.com/blog/apache-arrow-pandas-internals/>
- [40] A. Mozzillo, L. Zecchini, L. Gagliardelli, A. Aslam, S. Bergamaschi, and G. Simonini, “Evaluation of Dataframe Libraries for Data Preparation on a Single Machine,” 2025, Accessed: Jan. 17, 2025. [Online]. Available: <https://github.com/dbmodena/bento/tree/master#write-a-test-file>
- [41] “RAPIDS | GPU Accelerated Data Science.” Accessed: Dec. 11, 2024. [Online]. Available: <https://rapids.ai/>
- [42] “Welcome to the cuDF documentation! — cudf 23.10.00 documentation.” Accessed: Dec. 03, 2023. [Online]. Available: <https://docs.rapids.ai/api/cudf/stable/>
- [43] “Welcome to dask-cudf’s documentation! — dask-cudf 24.08.00 documentation.” Accessed: Dec. 11, 2024. [Online]. Available: <https://docs.rapids.ai/api/dask-cudf/legacy/>
- [44] “Dask — Dask documentation.” Accessed: Dec. 11, 2024. [Online]. Available: <https://docs.dask.org/en/stable/>
- [45] “cuDF Pandas | RAPIDS | GPU Accelerated Data Science.” Accessed: Dec. 11, 2024. [Online]. Available: <https://rapids.ai/cudf-pandas/>

- [46] “RAPIDS Suite of AI Libraries | NVIDIA Developer.” Accessed: Jan. 07, 2025. [Online]. Available: <https://developer.nvidia.com/rapids>
- [47] B. Hernández *et al.*, “Performance evaluation of python based data analytics frameworks in summit: Early experiences,” *Communications in Computer and Information Science*, vol. 1315 CCIS, pp. 366–380, 2021, doi: 10.1007/978-3-030-63393-6_24/FIGURES/8.
- [48] A. E. Topcu and İ. Omak, “Parallelization of a meteorological model using message passing interface and CUDA: A case study with the inversion estimation algorithm,” *Concurr Comput*, vol. 34, no. 9, p. e6738, Apr. 2022, doi: 10.1002/CPE.6738.
- [49] S. Rabhi, M. R. B. Kristensen, W. Sun, J. Liu, J. Perez, and E. Oldridge, “Accelerating recommender system training 15x with rapids,” *ACM International Conference Proceeding Series*, Sep. 2019, doi: 10.1145/3359555.3359564.
- [50] B. Yogatama *et al.*, “Accelerating User-Defined Aggregate Functions (UDAF) with Block-wide Execution and JIT Compilation on GPUs,” *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023*, pp. 19–26, Jun. 2023, doi: 10.1145/3592980.3595307.
- [51] D. Yang and D. Cheng, “Efficient GPU Memory Management for Nonlinear DNNs,” 2020, doi: 10.1145/3369583.3392684.
- [52] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, “Are dynamic memory managers on GPUs slow?: A survey and benchmarks,” *Proceedings of the ACM*

- SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 219–233, Feb. 2021, doi: 10.1145/3437801.3441612.
- [53] N. Becker *et al.*, “Streamlined and Accelerated Cyber Analyst Workflows with CLX and RAPIDS,” *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019*, pp. 2011–2015, Dec. 2019, doi: 10.1109/BIGDATA47090.2019.9006035.
- [54] Y. Kawakami and B. Hernández, “Early experiences on OpenPOWER architecture: Analysis of Billion-scale Atomistic Datasets”, Accessed: May 30, 2024. [Online]. Available: <https://thrust>.
- [55] O. Green, C. Nolet, and J. Eaton, “Generating Permutations Using Hash Tables,” *2022 IEEE High Performance Extreme Computing Conference, HPEC 2022*, 2022, doi: 10.1109/HPEC55821.2022.9926387.
- [56] A. Lazar *et al.*, “Accelerating the Inference of the Exa.TrkX Pipeline,” *J Phys Conf Ser*, vol. 2438, no. 1, p. 012008, Feb. 2023, doi: 10.1088/1742-6596/2438/1/012008.
- [57] C. J. Nolet *et al.*, “Bringing UMAP Closer to the Speed of Light with GPU Acceleration,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 418–426, May 2021, doi: 10.1609/AAAI.V35I1.16118.
- [58] S. Kang, J. Nke, and B. Rees, “Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank,” *2022 IEEE High Performance Extreme Computing Conference, HPEC 2022*, 2022, doi: 10.1109/HPEC55821.2022.9926341.

- [59] J. Berchtold, “Out-of-Core GPU Path Tracing on Large Instanced Scenes via Geometry Streaming,” *Master’s Theses*, Jun. 2022, doi: 10.15368/theses.2022.65.
- [60] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. M. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 73–82, 2008, doi: 10.1145/1345206.1345220.
- [61] T. Lutz, C. Fensch, and M. Cole, “Helium: A transparent inter-kernel optimizer for OpenCL,” *ACM International Conference Proceeding Series*, vol. 2015-February, pp. 70–80, Feb. 2015, doi: 10.1145/2716282.2716284.
- [62] W. Vanderbauwhede and T. Takemi, “An investigation into the feasibility and benefits of GPU/multicore acceleration of the weather research and forecasting model,” *Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013*, pp. 482–489, 2013, doi: 10.1109/HPCSIM.2013.6641457.
- [63] L. Solano-Quinde, R. Gualan-Saavedra, and M. Zuñiga-Prieto, “Multi-GPU implementation of the horizontal diffusion method of the weather research and forecast model,” *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2016*, pp. 98–103, Mar. 2016, doi: 10.1145/2883404.2883407.
- [64] B. A. Whetstone, V. Limpasuvan, and D. B. Larkins, “GPU acceleration of the Advanced Regional Prediction System (ARPS),” *Proceedings of the 2014 ACM*

Southeast Regional Conference, ACM SE 2014, Mar. 2014, doi: 10.1145/2638404.2735458.

- [65] A. Puranik, “DataFrames on the GPU,” *International Research Journal of Engineering and Technology*, 2020, Accessed: Dec. 04, 2023. [Online]. Available: www.irjet.net
- [66] Gwennaëlle Larvor, Léa Berthomier, Vincent Chabot, Brice Le Pape, Bruno Pradel, and Lior Perez, “MeteoNet, an open reference weather dataset by METEO FRANCE, 2020,” 2020. Accessed: Dec. 07, 2023. [Online]. Available: <https://meteonet.umr-cnrm.fr/>
- [67] A. Li *et al.*, “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, Jan. 2020, doi: 10.1109/TPDS.2019.2928289.
- [68] T. Kim *et al.*, “Accelerating GNN training with locality-aware partial execution,” *APSys 2021 - Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 34–41, Aug. 2020, doi: 10.1145/3476886.3477515.
- [69] “multiprocessing — Process-based parallelism — Python 3.13.1 documentation.” Accessed: Jan. 09, 2025. [Online]. Available: <https://docs.python.org/3/library/multiprocessing.html>
- [70] “Overview — Numba 0+untagged.871.g53e976f.dirty documentation.” Accessed: Jan. 09, 2025. [Online]. Available: <https://numba.readthedocs.io/en/stable/cuda/overview.html>

- [71] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, “Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU,” *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 57–67, Feb. 2013, doi: 10.1145/2517327.2442523.

VITAE

Name :Hamed Aidaros Al-Mehdhar

Nationality :Yemen

Date of Birth :7/22/1991

Email :Almehdhar.Hamed@gmail.com

Address :Saudi Arabia, Jeddah, Al-Fayha District, Al Hassan Ibn
Qutada Street, Postal Code 22246 – Short Address JAFA6558

Academic Background :Bachelor of Science in Computer Engineering

Education:

- 2022 - 2024: Master of Science in Computer Engineering, *King Fahd University of Petroleum and Minerals (KFUPM)* – First Honors.
- 2009 - 2014: Bachelor of Science in Computer Engineering, *King Fahd University of Petroleum and Minerals (KFUPM)* –Honors Degree.

Biography:

An accomplished professional, possessing a unique blend of technical expertise and managerial proficiency. My technical foundation is strong in Artificial Intelligence, Distributed Computing, CUDA Programming, GPU Acceleration, HDL Programming, with a specialized focus on Parallel Programming and Accelerated Computing. This expertise is complemented by extensive experience in business support, project management, and strategic planning, particularly within the Saudi Arabian market. Adept at leveraging my technical and managerial abilities, fostering productivity, and creating goal-oriented environments.

Work Experience:

- Project Manager:
 - *Sustainable Technology & Engineering Solutions (STES)* – Saudi Arabia
 - Apr 2016 – Aug 2021.
- Sales Engineer:
 - *Sustainable Technology & Engineering Solutions (STES)* – Saudi Arabia
 - Oct 2014 – Apr 2016.
- Trainee:
 - *Saudi Arabian Airlines (SAUDIA)* – Saudi Arabia
 - IT business Application Department – Summer Training.
 - Jun 2013 – Aug 2013.

Publications:

- Hamed A. Al-Mehdhar, Mohammed A. Makarem, “DDS as a Crowd Management Systems Integration Platform”, *International Research Journal of Engineering and Technology (IRJET)*, Nov 2023. [IF=8.226]