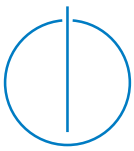
Elias Johannes Berthold Stehle

Massively Parallel Algorithms for Data Ingestion on New Hardware

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Massively Parallel Algorithms for Data Ingestion on New Hardware

Elias Johannes Berthold Stehle

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Jan Křetínský

Prüfer der Dissertation:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. Tilmann Rabl,
   HPI, Universität Potsdam

Die Dissertation wurde am 27.05.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.07.2020 angenommen.

"If I have seen further than others it is by standing on the shoulders of Giants."

– Isaac Newton

Abstract

As the creation and transmission of digital data continues to increase at an astonishing pace, data management systems are confronted with an unparalleled influx of data. While data needs to be ingested at an unprecedented rate, processors are seeing only moderate improvements in sequential processing performance. While performance had increased by more than 50% per year for nearly two decades, CPU advancements have drastically decelerated in 2003, declining to annual improvements of only 3.5% in recent years.

In order to continue the trend of exponentially increasing computational throughput, manufacturers have progressively turned towards scaling hardware parallelism. Yet, CPU cores remain complex, devoting a considerable share of chip area to optimising serial execution, which limits the number of cores that can economically be integrated on a single chip. GPUs, in contrast, have been designed for parallel execution ever since. While core counts of modern CPUs are still in the double-digit range, today's GPUs integrate as many as 5 120 cores.

In order to exploit the computational throughput and superior memory bandwidth of GPUs to accelerate data ingestion, however, algorithms must be designed for parallel execution from the ground up. According to Amdahl's law, even a small fraction of serial work considerably limits the speedup that an algorithm can achieve when trying to scale to thousands of cores. Another challenge is posed by the comparably slow interconnect. Data transfers add to the end-to-end processing duration and have to be amortised.

This thesis aims to accelerate data ingestion on GPUs by providing multiple massively parallel algorithms. By considering the essential building blocks that are required to accelerate the ingestion process, our algorithms contribute towards making GPUs self-sufficient and enabling a coalesced ingestion pipeline that passes along intermediary results on the device. The massively parallel algorithms contributed by this work are designed for scalability from the ground up to address the fundamental shift towards increasingly parallel processors and benefit from this new direction of processor advancement. By employing fine-grained data parallelism, we ensure that the presented algorithms are load-balanced and provide robust performance in spite of input variance.

Zusammenfassung

Da die Erzeugung und Übertragung digitaler Daten weiterhin mit erstaunlicher Geschwindigkeit zunimmt, sehen sich Datenverwaltungssysteme mit einem beispiellosen Zufluss an Daten konfrontiert. Während die Daten in einer noch nie dagewesenen Geschwindigkeit aufgenommen werden müssen, können Prozessoren jedoch nur noch mäßige Verbesserungen bei der sequentiellen Verarbeitungsgeschwindigkeit verzeichnen. Während sich die Leistung über fast zwei Jahrzehnten hinweg um mehr als 50% pro Jahr gesteigert hat, sind die Fortschritte bei CPUs seit 2003 drastisch zurückgegangen und sind in den letzten Jahren auf eine jährliche Verbesserungsrate von nur 3.5% gesunken.

Um den Trend des exponentiell steigenden Rechendurchsatzes fortzusetzen, haben sich die Hersteller nach und nach der Entwicklung von Mehrkernprozessoren zugewandt. Dennoch bleiben CPU-Kerne komplex und widmen einen beträchtlichen Teil der Chipfläche der Optimierung der seriellen Ausführung. Dadurch wird die Anzahl der Kerne begrenzt, die wirtschaftlich auf einem Chip integriert werden können. GPUs hingegen sind seit jeher auf parallel Ausführung ausgelegt. Während die Anzahl an Kernen von CPUs immer noch im zweistelligen Bereich liegen, integrieren GPUs bis zu 5 120 Kerne.

Um den Rechendurchsatz und die Speicherbandbreite von GPUs zur Beschleunigung der Datenaufnahme auszunutzen, müssen Algorithmen jedoch von Grund auf für die parallele Ausführung konzipiert werden. Nach dem Amdahlschen Gesetz schränkt selbst ein Bruchteil an serieller Arbeit den Geschwindigkeitszuwachs, den ein Algorithmus bei der Skalierung auf Tausende von Kernen erreichen kann, erheblich ein. Eine weitere Herausforderung stellt die langsame Datenübertragungsrate über den PCIe bus dar.

Diese Arbeit zielt darauf ab, die Datenaufnahme auf GPUs durch die Entwicklung mehrerer massiv-paralleler Algorithmen zu beschleunigen. Durch die Bereitstellung der Algorithmen, die zur Beschleunigung der Datenaufnahme erforderlich sind, tragen wir dazu bei GPUs autark zu machen und eine ineinandergreifende Verarbeitungspipeline zur Datenaufnahme zu ermöglichen. Die massiv-parallelen Algorithmen, die diese Arbeit beisteuert, sind von Grund auf auf Skalierbarkeit ausgelegt, um den Wandel hin zu zunehmend parallelen Prozessoren zu adressieren und von dieser neuen Richtung der Prozessorentwicklung zu profitieren.

Acknowledgments

As with many matters in life, we owe our achievements to the work, effort, and influence of some special people. We owe it to billions of years of evolution, to our ancestors and predecessors, centuries of humanity's achievements and scientific advancements. We owe it to modern society, billions of people taking on their role, so we can focus on ours. Most of all, we owe it to the people in our life, to our family, to our partner, to our friends, colleagues, companions, and teachers. This work is no exception and I would not have been able to pursue it, if it was not for some influential people in my life.

I would like to thank my supervisor Prof. Dr. Hans-Arno Jacobsen for encouraging me to pursue this path, for being available late hours and weekends, for sharing his decade-long experience, and for putting everything in place, with my scholarship, equipment, and the responsibility, to create an environment that allowed me to fully focus on my research.

I also would like to thank the rest of my thesis committee: Prof. Dr. Tilmann Rabl for agreeing to be the second examiner and Prof. Dr. Jan Křetínský for accepting to chair the committee.

To my colleagues at the chair. Thank you, not only for your valuable input and the technical discussions, but also for making my time at the chair an enjoyable one. A huge thank you, in particular, to Amirhesam Shahvarani, Thomas Kriechbaumer, Matthias Kahl, Christoph Doblander, and Daniel Jorde, for the refreshing casual conversations between deep-focused sessions and for the truly great friendship that we have formed.

To my family. My parents, Ursula and Berthold, my brother Emanuel, my grandparents Ruth and Josef Schnee, and my uncle Joachim Schnee. Thank you for being there for me since day one. Thank you for everything that you have done for me, for all your love, for all your care, for all your support. I would not be where I am without you. It is invaluable to have such a loving and supporting family.

To Isabel, the woman in my life. Thank you, for all your love and support. For your understanding, when work would demand weekends and holidays over an extended period. For your contagious optimism and for the smiles that brighten up my day. For the decompressing moments on the mountaintops. When, for a moment, it is just us, and everything else is just the world around us. Thank you for squeezing in time for proof-reading work unrelated to your field. You changed my life for the better. I love you.

Contents

1

Introduction

For many decades, ever increasing clock rates and improved instruction-level parallelism (ILP) allowed software systems to implicitly benefit from hardware advancements and to cope with ever growing workloads. Around 2003, after more than three decades of exponentially improving sequential computational throughput, the race of rising clock rates came to an end and ILP improvements have considerably slowed down. While, for the most time, CPU performance was improving by more than 50% per year (doubling every 21 months), these days they are seeing an improvement of only 3.5% per year (doubling every 23 years) [1]. As a result, manufacturers have progressively turned towards scaling hardware parallelism. Since then, increasing core counts as well as the introduction and extension of single instruction multiple data (SIMD) capabilities have taken the spot of rising clock rates and improving ILP in an effort to continue advancing processor performance. Now that increasing parallelism aims to sustain the trend of exponentially growing computational throughput, algorithms have to be designed for scalability from the ground up in order to benefit from this new direction of advancement.

This work addresses this fundamental shift towards increasingly parallel processors. With the aim of exploiting the parallelism of modern GPUs that, today, integrate as many as 5 120 cores, we develop multiple massively parallel algorithms. In particular, this work focuses on accelerating data ingestion, contributing algorithms that constitute the essential building blocks, such as sorting and parsing.

1.1 Motivation

Today's data management systems are facing unprecedented challenges as they must cope with data that is generated and queried by hundreds of millions of people and devices [2, 3, 4]. In recent years, there is an increasing number of online communities and social networking sites that are counting more than one billion users, allowing them to interact and communicate with one another [2, 3, 5]. *Facebook*'s social graph store, *TAO*, for instance, sustains billions of queries per second [2]. Their data warehouse exceeds 300 petabytes and generates four new petabytes per day [2]. Email and messenger services capture and archive conversations, providing a searchable history of the information that has been exchanged. In total, the top eight services taken together are counting more than seven billion users [3]. *WhatsApp* alone is seeing more than 65 billion messages being sent each day [6].

The ubiquity of sensor-packed smartphones lowers the hurdle to add even more data points to our digital footprint. With more than one billion smartphones shipped each year, smartphones provide a convenient interface that is always at hand to query and contribute to the data of various services [7]. Whether this happens actively, by capturing moments, or passively, by applications running in the background tracking our movement profiles, smartphones are our daily companion and provide us with access to a world of information. This is emphasised by the drastic increase of cellular data usage. According to a report by *Ericsson*, monthly cellular data usage has grown by over an order of magnitude from around two exabytes in early 2013 to more than 25 exabytes by the end of 2018 [8].

Work life is no exception to the trend of increasing digitalisation. Today, a plethora of cloud and software as a service (SaaS) offerings help organisations to plan, manage, execute, and coordinate various tasks. These offerings do not only simplify collaboration and increase efficiency but they also make processes transparent and traceable. The data gathered can be used for further analysis and serve as basis for process optimisations. A strong indication that more and more organisations are relying on these offerings is given by the growing cloud revenue, which is growing in excess of 50% per year throughout the past five years [3].

Following the billions of smartphones, the Internet of Things (IoT) presents the next wave of devices that are getting connected to the internet. With a projected 14.6 billion connections, IoT is expected to represent more than half of all connected devices by 2022 [4]. As form factors of sensors are shrinking, costs are coming down, and, at the same time, device connectivity via Wi-Fi and mobile networks can be integrated at decreasing cost, more and more devices will be able to sense their environment, help with data acquisition, and provide a convenient interface to access information. For instance, micromobility providers with Global Positioning System (GPS)-equipped electric scooters are able to understand movement patterns in urban environments. *Lime*, to mention just one of over a dozen providers, surpassed one million electric scooter rides within its first eleven weeks of operation in Berlin alone [9]. Smart assistants, on the other hand, provide a convenient and ubiquitous interface to access information. Sales of devices equipped with *Alexa*, Amazon's connected smart assistant, have surpassed the landmark of 100 million units [10]. Adding to this the billions of smartphones that have been shipped in recent years with the respective operating system's default smart assistant pre-installed, access to information via smart assistants has become ubiquitous.

The overall trend indicates that, as technology gets increasingly integrated into our daily lives, the growth rate of digital data is not going to slow down any time soon. At the same time, machines, with the help of advancements in the field of artificial intelligence, are able to meaningfully process and interpret a broadening share of that data. While the trend presents unique opportunities, it also poses unprecedented challenges. Data management systems need to cope with the explosive growth of data, ingest it, index it, and provide a consistent view on a rapidly updating database. At the same time, systems are confronted with an increasing load from a growing number of data consumers, as more and more people and devices are dispatching more and more queries on a challengingly large database. These challenges are highlighted by a recent work from *Google, Inc.* on their Tensor Processing Unit (TPU) [11]. In an assessment of future compute requirements, they concluded that, if people were using voice search for three minutes a day, their datacenter would need to double in capacity to meet the compute requirements with conventional CPUs [11].

While data management systems are confronted with an ever increasing amount of data, CPUs are seeing only moderate improvements in sequential processing performance.

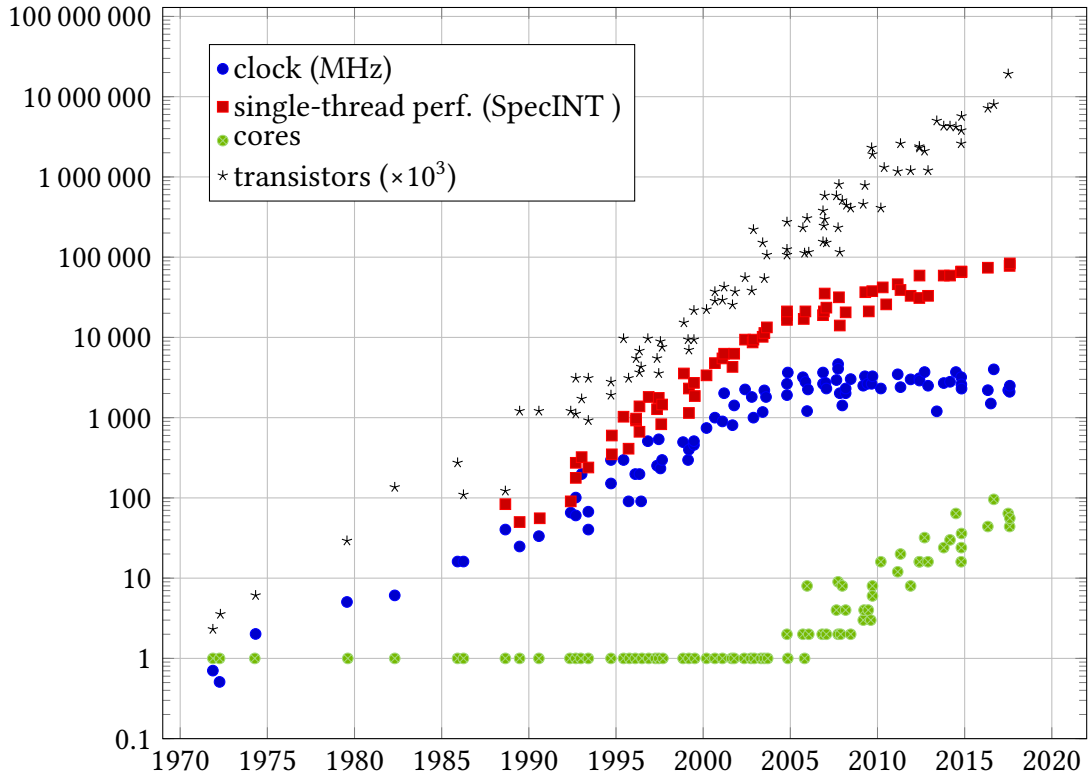


Figure 1.1.1: History of the exponential performance improvements of processors and the drastic deceleration of rising clock rates and single-thread performance around the year 2003. Following the stagnating performance advancements, a fundamental shift to increasingly parallel hardware is taking place (data according to Rupp [12]).

As illustrated in Figure 1.1.1, performance advancements have drastically decelerated in 2003. According to data from Hennessy et al., improvements these days are down to 3.5% per year. This is a drastic drop, after having seen improvements in excess of 50% per year over the course of nearly two decades [1]. Assuming the continued advancement of 50% per year after 2003, processors would have seen a 657-fold performance increase over the subsequent 16 years. In reality, however, processor speed according to Hennessy et al. has increased by only a factor of 9.46 leaving a 69-fold performance gap.

With the aim of closing the gap, manufacturers have progressively turned towards scaling the number of cores and introduced increasingly powerful vector processing capabilities. This constitutes a disruptive shift for software systems. Prior to this transition, software systems implicitly benefitted from hardware advancements over the course of multiple decades. With the shift to increasing hardware parallelism, however, algorithms need to

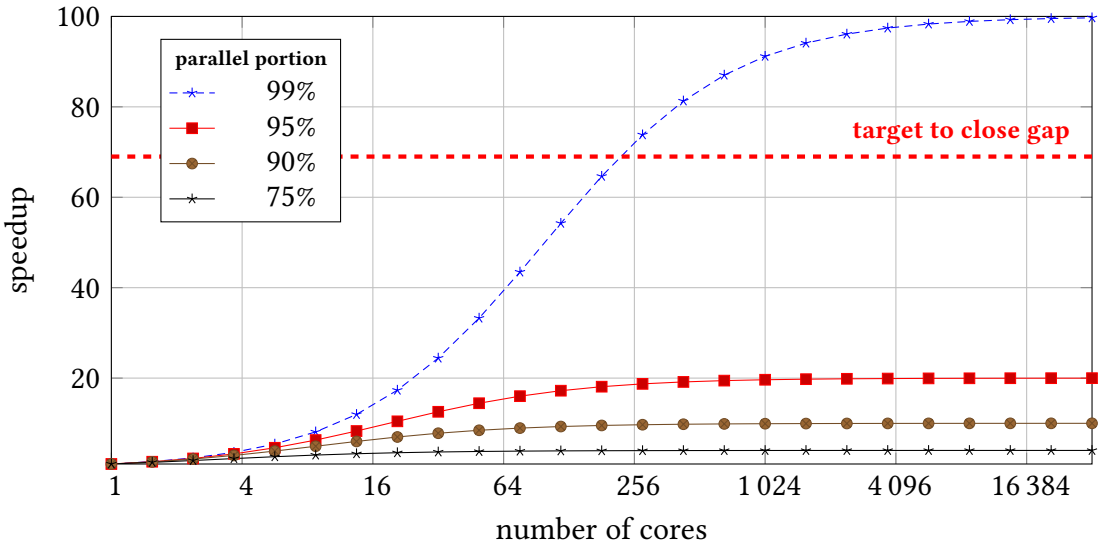


Figure 1.1.2: The maximum speedup according to Amdahl's law that can be achieved for programs with a specific parallel portion that are run on a processor with a given number of cores. A program's *parallel portion* represents the share of the runtime that benefits from parallel execution.

be designed for scalability from the ground up. According to Amdahl's law, even a small fraction of serial work in an algorithm considerably limits the speedup the algorithm can potentially get from a multicore processor. Even if the program's serial portion makes up only 5%, e.g., for synchronising threads or gathering individual results, the maximum speedup is limited to 20. As illustrated in Figure 1.1.2, a program for which 99% of the execution time can be parallelised would require 233 cores in order to make up for the aforementioned 69-fold performance gap (*target to close gap*). However, most software that is run on a CPU struggles to parallelise even considerably smaller fractions of the code. A major implication of these scalability issues of software that was originally designed for the CPU is that manufacturers have to ensure that CPUs still perform exceptionally well when executing serial code. This results in complex cores with deep pipelines that spend a considerable share of chip area optimising ILP, which, in turn, constrains the number of cores that can economically be integrated on a single chip.

GPUs, in contrast, have focused on parallel execution with simpler cores ever since. Originally designed for the graphics domain, today's GPUs with up to 5 120 cores integrate two orders of magnitude more cores than CPUs. Algorithms that are designed from the ground up to exploit parallelism and scale linearly with the number of cores with

only a negligible portion of serial code benefit greatly from the GPUs' massive compute performance. The importance of scalable, parallel algorithms is becoming even more apparent as hardware parallelism is extending beyond a single chip. Today, the availability of CPUs that comprise multiple *chiplets* as well as research that focuses on package-level integration of multiple GPU modules give an indication that future processors are scaling to multiple inherently parallel *chilpets* and GPU modules, respectively, on a package [13].

Another, even more important advantage in the context of data management systems is the GPU's superior memory bandwidth. With a low arithmetic intensity, many operations of data management systems are rather bound by the available memory bandwidth than by compute. Being equipped with High Bandwidth Memory (HBM), today's GPUs achieve a memory bandwidth of up to 900 GB/s, which is about an order of magnitude more throughput than that of a top-of-the-line *Intel Xeon* CPU.

1.2 Problem Statement

As the sequential processing performance improvements of CPUs are stagnating, GPUs present a promising, future-proof alternative to cope with the continuously growing amount of data that needs to be ingested, filtered, stored and analysed. In order to exploit the computational throughput and superior memory bandwidth of GPUs, however, algorithms must be designed for scalability from the ground up. According to Amdahl's law, even a small fraction of serial work considerably limits the speedup that an algorithm can achieve when trying to utilise the thousands of cores of GPUs. Moreover, today's GPUs are equipped with no more than a few tens of gigabytes of device memory. Compared to system memory that may comprise a few terabytes, a GPU's device memory provides only a fraction of that capacity. This requires careful consideration when developing a memory-efficient algorithm and, in some cases, may require extending the algorithm with a heterogeneous approach that involves the CPU to alleviate the limited memory capacity. Another challenge is posed by the comparably slow Peripheral Component Interconnect Express (PCIe) bus, which imposes an upper bound on the achievable throughput and adds to the end-to-end processing duration.

This work addresses these challenges in the context of data ingestion. Considering the limited interconnect bandwidth as well as the latency introduced by data transfers via the PCIe bus, not all operations are feasible for being accelerated on GPUs. In particular, the time taken for transferring the input as well as for returning the result over the PCIe bus has to be amortised. Moreover, the operation has to be computationally expensive enough to avoid being limited by the bandwidth of the interconnect. Comprising a sequence of complex, batch-like tasks, such as parsing and index generation, data ingestion constitutes a well-suited candidate for being accelerated on GPUs.

Even though data ingestion comprises compute-intense stages that often incur a considerable amount of memory transfers, the underlying algorithms pose specific challenges when trying to exploit the massive parallelism of GPUs. As illustrated in Figure 1.1.2, even small fractions of serial code considerably limit the potential speedup one can expect to see from a processor with thousands of cores. As a consequence, algorithms have to be designed for scalability from the ground up, while keeping synchronisation between threads at a minimum.

1.2.1 Massively Parallel Sorting

Sorting, for instance, is a problem with global data dependency. That is, an item's rank in the sorted output sequence depends on all other items. Hence, a thread that processes a subset of the input's items has to coordinate with other threads, which are processing different partitions of the input, to identify the ranks of its items. Since sorting is at the core of a wide range of operations and an essential building block for many massively parallel algorithms, it is a well-studied problem. Over the course of more than two decades a lot of effort was put into developing efficient sorting algorithms on GPUs. In recent years, the least-significant digit first (LSD) radix sort gained popularity as it requires only limited coordination between threads. The LSD radix sort was continuously improved by considering more bits per sorting pass. Even though considering a wider radix digit, i.e., more of a key's bits per sorting pass, is an important endeavour, it comes with its drawbacks and challenges on GPUs. In particular, the amount of *shared memory* (fast on-chip memory) required to maintain each thread's histogram limits the number of bits that can be considered with each sorting pass. The histogram constitutes a fundamental

data structure for the LSD radix sort. For a parallel radix sort that considers d bits with each sorting pass, each thread maintains a histogram over the 2^d bins. Threads use the histogram to count the number of items they intend to write to each bin. By summing over the threads' histograms, threads can identify the offsets into the bins for their items. However, as each thread has to maintain its own histogram and the on-chip memory requirements grow exponentially with the bits being considered with each pass, GPU-based radix sort implementations are bound by the available on-chip memory. As a result, a GPU-based radix sort can consider only a limited number of bits per pass, requiring more sorting passes than their CPU-based counterpart.

1.2.2 Parsing of Delimiter-Separated Formats

Another challenge arises when parsing complex delimiter-separated data formats, such as being encountered when ingesting various log file formats. While a data parallel approach that splits the input into multiple chunks with the aim of being able to process the chunks independently is desirable, there are various challenges that must be addressed. Firstly, for non-trivial input formats, symbols have to be interpreted differently, depending on the context they appear in. For instance, the comma-separated values (CSV) format described in RFC 4180 specifies that delimiters, such as commas and line breaks, have to be interpreted as part of the field, instead of being interpreted as actual field or record delimiters, if they appear within a field that is enclosed in double-quotes [14]. In addition to double-quotes, many formats use more special symbols, introducing even more contexts amongst which the parser has to differentiate. For instance, some formats indicate comments or directives using a symbol like `'#'`, following which, all symbols until the end of line have to be interpreted differently, yet again. Since the context depends on all symbols preceding the symbol currently being interpreted, it is impossible for a thread that processes a chunk somewhere in the middle of the input to be aware of that symbol's context and to meaningfully interpret it. Similarly, a thread lacks information about the record and the field that a sequence of symbols within its chunk belongs to. Lastly, even if a thread was aware of the context as well as the records and columns, it still has to coordinate and possibly collaborate with other threads to assemble field values that span multiple threads.

1.3 Approach

By providing multiple massively parallel algorithms, this thesis aims to accelerate data ingestion on GPUs. With an efficient sorting algorithm, we do not only support accelerating index generation on the GPU, but we also develop an essential building block for many massively parallel algorithms. With an algorithm for massively parallel parsing of delimiter-separated formats, we address a fundamental algorithm for data ingestion that is often considered a major bottleneck [15].

The massively parallel algorithms contributed by this work are designed for scalability from the ground up to address the fundamental shift towards increasingly parallel processors and benefit from this new direction of processor advancement. By employing fine-grained data parallelism, we ensure that the presented algorithms are load-balanced and provide robust performance in spite of input variance.

1.3.1 Massively Parallel Sorting

With an efficient sorting algorithm that provides superior performance, we lay the foundation for our endeavour of accelerating data ingestion on GPUs. Sorting is not only essential for index generation, but it is also an essential building block for many massively parallel algorithms. Sorting, for instance, constitutes an indispensable component for our approach to massively parallel parsing of delimiter-separated formats.

This thesis considers a completely new approach to sorting on GPUs with an MSD-based hybrid radix sort. By deviating from the popular approach of using an LSD radix sort, our approach alleviates the need for stable sorting passes, which, in turn, enables the use of native shared memory atomic operations that became available on recent GPU architectures. Our algorithm exploits this new hardware feature to maintain a single histogram that is shared by multiple threads (i.e., a thread block) instead of having each thread maintain its own local copy. This considerably lowers the required amount of shared memory, allows us to consider a wider radix digit, and therefore to reduce the total number of sorting passes. Other than previous approaches that build on an LSD radix sort,

the MSD radix sort is very challenging, as it may produce millions of subproblems for some distributions of input data and only few for others. Tackling the extremely varying degrees of segmentation with a hybrid approach on the GPU is particularly demanding, as GPUs require very fine-grained parallelism (tens of thousands of threads), lack efficient global synchronisation mechanisms, and have only very limited amount of scratchpad memory. The latter, for instance, requires to have tiny subproblems (no larger than a few KB) before being able to efficiently sort them in local memory, which in turn implies being confronted with extremely high degrees of segmentation. Even though deviating from the established approach of using an LSD radix sort poses all these challenges, we are able to address all of them, providing a scalable, load-balanced approach that provides robust performance.

By extending our fast on-GPU hybrid radix sort with a heterogeneous approach, we aim to allow the efficient sorting of inputs that exceed the GPU's device memory. The heterogeneous approach uses pipelining to mitigate the overhead associated with PCIe data transfers. That is, overlapping the transfer of chunks of unsorted input to the GPU, while simultaneously sorting chunks on the GPU and returning sorted results to the CPU. The CPU uses a multiway merge sort to reduce the amount of memory transfers while bringing the sorted chunks returned from the GPU into global order.

1.3.2 Parsing of Delimiter-Separated Formats

Our approach for parsing delimiter-separated formats is designed for scalability, flexibility and general applicability from the ground up. `ParPaRaw`, our algorithm for massively **par**allel **pa**rsing of delimiter-separated **raw** data on GPUs, overcomes the scalability issues of prior work without compromising applicability or constraining supported input formats. `ParPaRaw` employs a data parallel approach with fine-grained parallelism. The algorithm enables concurrency even beyond the granularity of a single record and ensures load balancing by splitting the input into small chunks of equal size that threads can process independently. Since using a data parallel approach raises the aforementioned challenges, we present an efficient solution for correctly identifying the parsing context of a thread's chunk, its records, and columns.

In order to provide a flexible approach that is applicable to a wide range of inputs, we allow specifying the parsing rules in the form of a deterministic finite automaton (DFA). The massively parallel algorithm for simulating DFAs is at the core of `ParPaRaw` and helps to keep track of the parsing context, which is represented by the DFA's state. While a thread iterates over its symbols, it transitions the states of its DFA instance according to the parsing rules specified by the DFA for a given format. Since a thread, starting to parse somewhere in the middle of the input, cannot simply infer the state it is supposed to start in, we let it assume every possible state. For each thread, this generates a mapping from every possible starting state s_i, to the state it would end up in, had it started parsing its chunk beginning in s_i. Using a parallel prefix scan that computes the composition over these mappings yields the actual starting state for each thread's DFA instance. While this approach increases the overall effort by a small constant factor, it enables a load-balanced, fully concurrent approach that can scale linearly with the number of cores.

Having identified the starting state for each thread's DFA instance, threads can correctly interpret the symbols from their chunk. Yet, threads remain unaware of the records and columns each of their symbols belongs to, as a chunk's record and column offset depend on the input preceding the chunk. In order to resolve the record offset, each thread counts the number of record delimiters it encounters. Computing the prefix sum over the threads' record counts yields the record offset for each chunk. The column offset of a chunk corresponds to the preceding chunk's number of column delimiters following that chunk's last record delimiter. In the absence of any record delimiter within a chunk, a column offset is said to be *relative*. In order to resolve *relative* column offsets, we use a prefix scan, similar to the prefix sum used to resolve the record offsets. Being aware of the parsing context, the records, and columns for all its symbols, each thread can correctly interpret its chunk. Since all the individual steps are performed using a massively parallel algorithm, we are able to address the aforementioned challenges for parsing delimiter-separated formats with a load-balanced, scalable approach.

We extend the massively parallel algorithm running on the GPU with a heterogeneous streaming approach. Our streaming approach is able to exploit the full-duplex capabilities of the PCIe bus and lowers the end-to-end parsing latency. In particular, it parses data on the GPU, while simultaneously transferring raw data to, and parsed data from the GPU.

1.4 Contributions

With the goal of accelerating data ingestion on GPUs, this work develops multiple massively parallel algorithms. By considering the essential building blocks that are required to accelerate the data ingestion pipeline, we help to make GPUs self-sufficient, and we allow GPUs to pass along intermediary results between consecutive stages of the ingestion process. The massively parallel algorithms contributed by this work are designed for scalability from the ground up to address the fundamental shift towards increasingly parallel processors and benefit from this new direction of processor advancement. In pursuit of accelerating data ingestion on GPUs, the main contributions of this work are:

1. We consider a novel approach to sorting on GPUs with an MSD-based hybrid radix sort. With our approach, we alleviate the need for stable sorting passes, which enables the use of native shared memory atomic operations. Using this new feature, we are able to drastically lower the amount of on-chip memory requirements, allowing us to consider a wider radix digit and reduce the number of sorting passes, which ultimately results in considerable performance improvements.

2. We address inputs that either do not reside on the GPU or exceed the available device memory using a pipelined heterogeneous sorting algorithm that mitigates the overhead associated with PCIe data transfers. In order to efficiently exploit the limited device memory, we propose an in-place replacement strategy that improves the overall performance for large inputs.

3. We present an approach to massively parallel parsing of delimiter-separated data formats that is designed for scalability without sacrificing applicability and flexibility. At the core of our approach, we present a massively parallel algorithm for simulating DFAs, which ensures flexibility towards the multifaceted formats that parsers are confronted with.

4. We extend our on-device parsing algorithm with a heterogeneous streaming approach that is able to exploit the full-duplex capabilities of the PCIe bus. Our approach lowers the end-to-end latency and allows parsing data on the GPU, while simultaneously transferring raw data to, and returning parsed data from the GPU.

Parts of the content and contributions of this work have been published in:

- E. Stehle and H.-A. Jacobsen. "A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD. 2017, pp. 417–432. DOI: 10.1145/3035918.3064043
- **2019:stehleparparaw**

1.5 Organization

The rest of this document is organised as follows. Chapter 2 presents our methodology for developing massively parallel algorithms for the GPU. Chapter 3 presents a short summary of the publications that this thesis comprises. The chapter outlines the approach of our sorting algorithm and the massively parallel parsing algorithm for delimiter-separated data formats. For each publication, we summarise the key achievements and highlight the author's contributions. The publications are attached to this thesis and can be found in Appendices A and B. Chapter 4 discusses our results in the larger context of ongoing research on massively parallel algorithms for GPUs. Chapter 5 presents the conclusions and an outlook for future work.

2

Methodology

This chapter gives an overview of the relevant hardware details and presents our methodology for developing a GPU-accelerated approach on heterogeneous systems. Section 2.1 introduces the GPU's architecture, presents the essential concepts of the CUDA programming model, and explains how the software concepts are scheduled on the GPU's hardware components. Section 2.2 presents our methodology for developing a GPU-accelerated system. Based on the characteristics of the system's hardware components, we derive guiding design principles and motivate our decision for choosing data ingestion as a suitable problem for being accelerated on GPUs. Moreover, we reason about the influence that certain hardware characteristics have on our implementation decisions.

2.1 Background

This thesis focuses on *NVIDIA* GPUs and the CUDA computing platform. *NVIDIA* has established a strong ecosystem with an elaborate software stack, which drives the adoption of their GPUs in the datacenter. Moreover, CUDA has been widely adopted and allows to tailor implementations to specific hardware characteristics. This section gives an overview of *NVIDIA*'s GPU architecture and the CUDA programming model.

2.1.1 GPU Architecture

With more than a handful of different GPU architectures, the history of CUDA-capable GPUs already spans more than a decade. Even though, over time, the GPU's architecture has undergone several iterations, they all share many of the underlying concepts. In the interest of comprehensibility and to give a better understanding about the scale and proportion of the key components, we will focus on *NVIDIA*'s most recent GPU architecture called *Turing* [17]. While many of the presented concepts are shared by *Turing*'s predecessors, the configuration for some components may vary and there are subtle differences, which we try to address within a reasonable scope.

At the core of the GPU is the Streaming Multiprocessor (SM). For different GPU models of the same architecture, the number of SMs are scaled up and down as the manufacturer sees fit to meet the demand in compute performance of different market segments. A *Turing* GPU for professional use, the *Quadro RTX 6000*, for instance, comprises 72 SMs. The consumer-grade *GeForce RTX 2080* is counting 46 SMs. As depicted in Figure 2.1.1, an SM of the *Turing* architecture is partitioned into four processing blocks, each of which comprises a *warp scheduler* along with a *dispatch unit*, a *register file*, numerous *FP32* and *INT32* cores, a few *tensor cores*, *load/store units*, and the *special functions units (SFUs)* [17]. Before *Turing*, instead of discrete *FP32* and *INT32* cores, SMs used to have *CUDA cores*, which integrated both, the integer arithmetic logic unit (ALU) as well as the floating-point unit (FPU) [18]. When, on previous architectures, *CUDA cores* were either performing floating point math or integer arithmetic, with *Turing*, the core execution datapath has changed, allowing for parallel execution of floating point math and integer arithmetic [17]. The *warp scheduler* is in charge of issuing instructions to the cores, *load/store units*, and *SFUs*. The *register file* is the backing memory for each thread's context, such as intermediary results and thread-local variables. On *Turing*, the *register file* of an SM provides 65 536 32-bit registers for a total of 256 KB. While *FP32* and *INT32* cores are performing floating point math and integer arithmetic, respectively, *tensor cores* are a relatively new addition, specialised for supporting matrix and tensor operations with reduced precision, i.e., FP16 and more recently also INT8 and INT4. On *Turing*, each SM comprises 64 *FP32* cores, 64 *INT32* cores, and eight *tensor cores*. The *load/store units* load and store data to cache or dynamic random-access memory (DRAM). The *SFUs* implement transcendental instructions, such as sine, cosine, and square root. Further, each SM comes

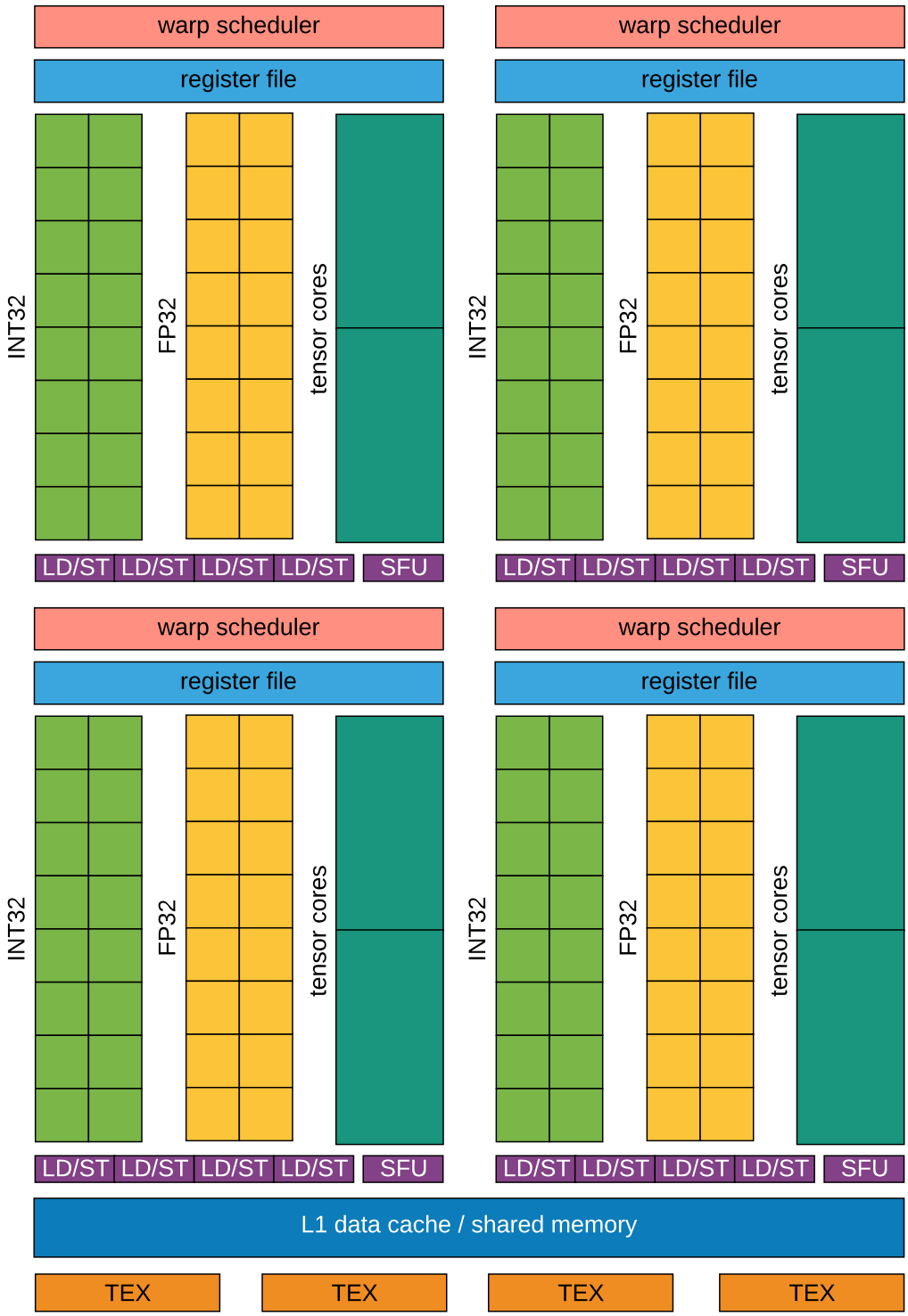


Figure 2.1.1: Schematic of a Streaming Multiprocessor (SM) of the *Turing* GPU architecture based on the NVIDIA whitepaper [17].

with its own *L1 cache* and *shared memory*. *Shared memory* is fast, addressable, on-chip scratchpad memory. *Turing* is using a unified architecture, where the *L1 cache* and *shared memory* are put together for a total of 96 KB [17]. The GPU can be configured for 64 KB and 32 KB of *L1 cache* and *shared memory*, respectively. The amount allocated to the *L1 cache* can be reduced to 32 KB, assigning the remaining 64 KB to *shared memory*. SMs integrate *Texture units*, to support texture sampling and filtering operations. Another addition with *Turing* are *RT cores*, to perform hardware-accelerated ray tracing operations with high efficiency [17]. In the interest of simplicity, *RT cores* have been omitted in the schematic in Figure 2.1.1. On top of numerous SMs, GPUs integrate an *L2 cache* as well as memory and bus controllers. Similar to the number of SMs, the size of the *L2 cache* is also depending on a specific GPU model. The *Quadro RTX 6000*, for instance, comes with six megabytes, while the *GeForce RTX 2080* has access to four megabytes of *L2 cache*.

2.1.2 CUDA Programming Model

Kernels, threads, thread blocks and grids constitute the basic concepts of the CUDA programming model. A kernel is a user-defined function that can be executed by thousands and thousands of threads on CUDA-capable GPUs. The threads that execute a kernel are organised hierarchically. Threads are grouped into thread blocks. Thread blocks, in turn, are organised in a grid. Each thread within a thread block is uniquely identified by its `threadIdx` and each thread block within a grid by its `blockIdx`. The `threadIdx` is a zero-based enumeration of the threads within a block. Similarly, the `blockIdx` is a zero-based enumeration of the thread blocks within the grid. A thread that executes a kernel can query its `threadIdx` as well as its block's `blockIdx`, making it easy to implement data parallel algorithms, where threads are working on different partitions of the input data. When a kernel is launched, the user specifies the execution configuration, which determines the number of thread blocks and the number of threads per thread block that execute the kernel. The number of threads per thread block is configurable up to a size of 1 024 threads per block. All threads that belong to the same thread block share access to a common *shared memory* allocation. That is, writes to *shared memory* by one thread can be observed by another thread that belongs to the same thread block.

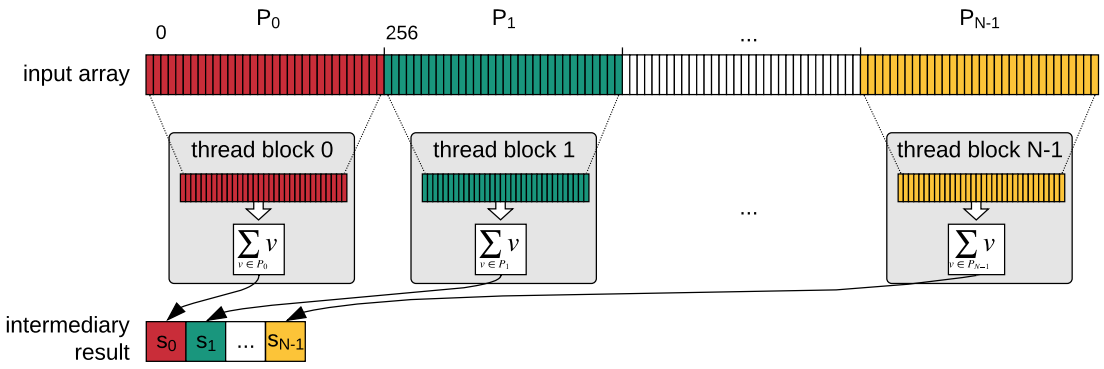


Figure 2.1.2: An example for a data parallel reduction algorithm that computes the sum over a given input array. The input is subdivided into partitions of 256 items and partitions are processed independently.

The concept of subdividing the input into multiple partitions and mapping those partitions to different thread blocks is illustrated in Figure 2.1.2. With the algorithm in Figure 2.1.2, we aim to exemplify a parallel algorithm that computes the sum over all numbers of a given input array. As illustrated in Figure 2.1.2, each thread block computes the sum over 256 items and writes the sum over its items back to device memory. Since each thread block is working on a partition of 256 items, we need to launch a grid of $\lceil n/256 \rceil$ thread blocks to process a given input array of size n. As each thread block computes the sum for its partition, the algorithm outputs $\lceil n/256 \rceil$ intermediary sums after one invocation. Another kernel invocation that uses the $\lceil n/256 \rceil$ intermediary sums output by the first invocation can then be used in a next pass to further reduce the number of sums. In order to compute the sum over n items, the kernel has to be repeatedly invoked $\lceil \log_{256} n \rceil$ times. A similar approach is used within a thread block. Figure 2.1.3 illustrates how the partition, which is assigned to *thread block 0*, is split into smaller chunks that are mapped to the threads of that thread block. In this example, each partition is processed by 64 threads and each thread computes the sum over four input items ($4 \times 64 = 256$). In a first pass, each of the 64 threads reads its four items from global memory into thread registers, computes the sum over its four items, and writes that sum to shared memory, such that the i-th thread outputs its sum to the i-th index in shared memory. The usage of shared memory allows the GPU to keep intermediary results in fast on-chip memory. This reduces the amount of memory traffic to global device memory, which, in turn, improves the algorithm's performance. For an algorithm that exhibits low arithmetic intensity, where the ratio of computations for each byte being loaded or written is comparably low, this is an important optimisation technique.

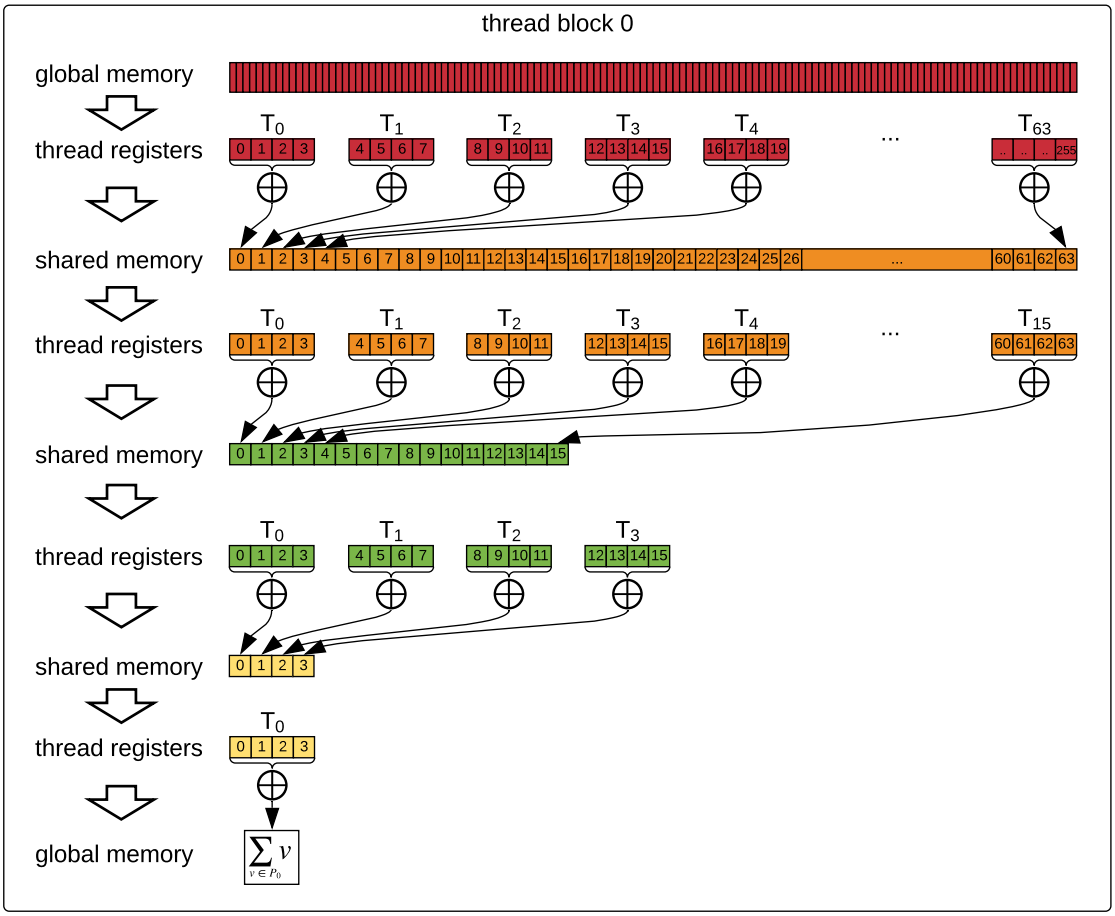


Figure 2.1.3: The processing steps involved by a single thread block to compute the sum over a single partition of 256 items. The algorithm performs multiple passes. With each pass, each thread reads in four items into thread registers, computes the sum over its items, and writes the sum back to shared memory.

In a subsequent pass, each of the first $\lceil 64/4 \rceil$ threads of that thread block reads in four items that were written to shared memory in the first pass. That is, the i-th thread reads the sums that have been written to shared memory by the threads $[i \times 4, i \times 4 + 3]$ in the preceding pass. Similar to the first pass, each thread computes the sum over its four items and writes its sum back to shared memory. It is important to note that threads of the same thread block have to synchronise at the end of each pass in order to avoid race conditions. Considering the second pass, for instance, we have to ensure that thread T_0 will not read the values from shared memory before T_1, T_2, and T_3 have written their values. In order to control the progression of threads in a thread block, the CUDA programming model introduces the *__synchtreads* barrier, which ensures that no thread progresses beyond this barrier before all other threads of the same thread block have reached the

barrier. In the example in Figure 2.1.3, this barrier ensures that all threads have written the intermediary result from the preceding pass before any thread reads in its values from shared memory. Given that we control the threads' progression, we can reuse shared memory between passes and therefore reduce the amount of shared memory allocated to each thread block to only a single shared memory allocation of 64 items per thread block. Overall, the algorithm requires four passes ($\lceil \log_4 256 \rceil = 4$) to compute the sum over the 256 items of a thread block. After the fourth pass, the thread that computes the final sum writes that sum back to global memory.

2.1.3 Scheduling & Parallel Execution Model

After covering the hardware architecture and the programming model, this section explains how software concepts like thread blocks are assigned to hardware entities, such as SMs. In general, scheduling on the GPU is happening on two levels of granularity. On a more coarse-grained level, a set of thread blocks are assigned to SMs. On a more granular level, for every thread block that has been assigned to an SM, threads have to be scheduled and executed.

There are multiple parameters that are decisive when scheduling thread blocks on SMs. To a large degree, the scheduling is determined by the resources required by each thread block and the resources available on each SM. In order to execute a kernel, each thread block requires a certain amount of shared memory and each thread of that thread block requires a certain number of registers. The amount of shared memory required by a kernel for each thread block is apparent, as the user explicitly states the amount of shared memory to allocate for each thread block. For instance, in the example presented in Figure 2.1.3, we make use of shared memory to hold the sum computed by each thread, allocating four bytes per value for each of the 64 threads, for a total of 256 bytes of shared memory per thread block. Compared to the amount of shared memory, the number of registers required by each thread is less obvious. When the kernel is compiled to binary microcode that can natively be executed on the GPU, the number of registers required by each thread executing that kernel becomes known. Multiplying the number of required registers per thread with the number of threads per thread block yields the total number of registers required by a thread block.

In order to improve the latency-hiding mechanism employed by the GPU, the GPU tries to maximise the number of thread blocks it assigns to each SM. Since the GPU is keeping the context of its threads in fast on-chip memory, context switches are seamless, as no registers or other state must be swapped. Hence, the more threads reside on an SM, the larger the pool of threads the scheduler can choose from and therefore the better the latency-hiding. While the GPU aims to maximise the number of thread blocks assigned to each SM, it is constrained by the available resources, particularly the register file size and amount of available shared memory, as well as some additional architecture-specific characteristics, such as the maximum number of threads per SM and the maximum number of thread blocks per SM. For *Turing*, the number of thread blocks per SM is limited to 32 and the maximum number of threads per SM is 1 024. For instance, assuming that our summation kernel requires 70 registers per thread and 256 bytes of shared memory, while the GPU is configured with 64 KB of shared memory and comprises 65 536 registers, the GPU, being bound by the register file size, can assign at most 14 thread blocks per SM at a time. With 64 threads per thread block, this corresponds to 896 threads and a theoretical occupancy of $\frac{896}{1\,024} = 0.875$. The theoretical occupancy represents an important high-level metric for kernels. It reflects the ratio of threads that can reside on an SM, i.e., due to resource constraints, to the maximum number of threads supported by the underlying hardware.

In contrast to the assignment of thread blocks to SMs, which depends on many parameters that can be tuned and influenced by the user, scheduling of threads that reside on an SM is less transparent to the user. Threads of thread blocks that have been assigned to an SM are scheduled in a group of 32 threads that is referred to as a *warp*. Threads that belong to the same *warp* progress in a lock-step manner and execute the same instruction. In case of branch divergence within a warp, where some threads execute a conditional code path while, for other threads, that condition may not be met, the portion of threads that do not meet the condition remain disabled, executing `no-ops`, until the code paths converge again. Given that the execution along divergent branches within a *warp* is serialised and resource utilisation is limited, with only a portion of the threads actually executing instructions, it is important to avoid branch divergence within a *warp*.

2.2 Data Ingestion on GPUs

There are many aspects that have to be taken into account when designing and developing GPU-accelerated systems. With a loosely-coupled heterogenous system that integrates one or more GPUs, data transfers via the interconnect become an important factor that impacts the system's performance. Communication and synchronisation between the CPU and the GPU introduces further overhead, stifles the progression of CPU threads, and degrades the overall system performance. This section presents the characteristics of state-of-the-art hardware and analyses ongoing hardware trends, which may give an indication of future advancements and influence the design and development of software systems. Based on our analysis, we present guiding design principles for heterogeneous systems and motivate our decision for accelerating data ingestion on GPUs. Lastly, we present our methodology for designing and developing massively parallel algorithms for GPUs.

2.2.1 Hardware Characteristics & Trends

While GPUs provide considerable performance improvements over CPUs with respect to compute performance as well as memory bandwidth, the limited device memory capacity and comparably slow interconnects impede the adoption of GPUs for data management use cases.

In contrast to system memory that may comprise hundreds of gigabytes or even a few terabytes, memory capacity of GPUs is still in the order of a few gigabytes (i.e., low double-digit range). However, it is important to note that the trend of exponentially growing memory capacity that holds for main memory also holds for the device memory of GPUs. As the amount of memory per USD is doubling approximately every 2.5 years, data management systems are able to keep more and more data in memory. In the case of main memory, this trend has driven the adoption of in-memory databases for more and more use cases, as the demand in storage capacity for frequently accessed data was not growing as quickly as the memory capacity.

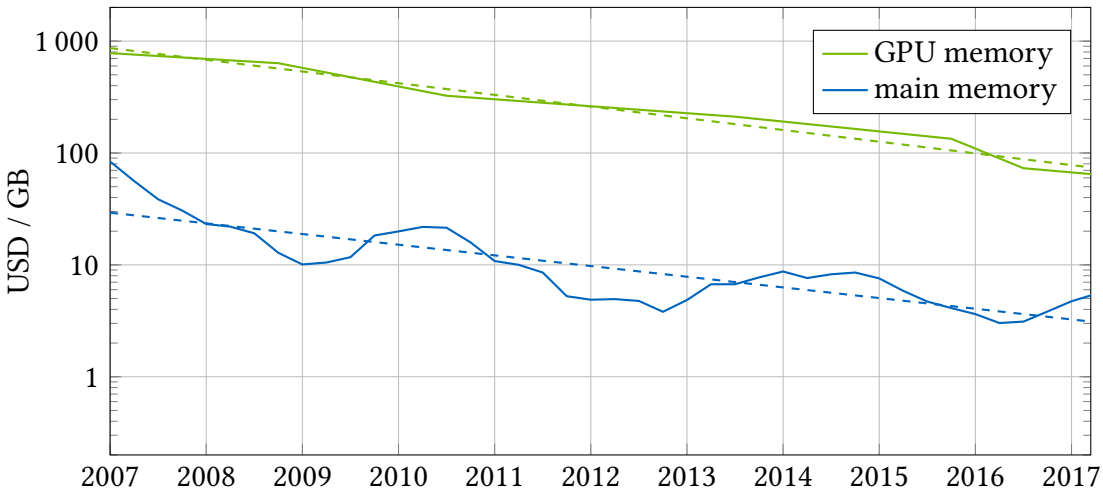


Figure 2.2.1: GPU memory capacity is increasing at a compound annual growth rate (CAGR) of 30%, doubling roughly every 2.5 years. Even though, still about an order of magnitude more expensive, every gigabyte of GPU memory is considered "programmable memory", i.e., including the processor. GPU data was collected for high-end consumer-grade GPUs that would retail in the band of USD 499 to USD 700. Prices are based on the manufacturer's suggested retail price (MSRP) at the time of release of new models. Main memory prices according to data from McCallum [19].

As illustrated in Figure 2.2.1, the memory capacity per USD of GPUs is following the trend of main memory. Lagging roughly a decade behind main memory, GPU memory is approaching the price point of main memory at the time when interest in in-memory databases began to accelerate. It is worth to emphasise that the GPU memory in Figure 2.2.1 is considered *"programmable memory"*, i.e., including the processor. GPUs provide the compute performance and memory bandwidth to scan through all data residing on the device within a few milliseconds (5 - 25 ms), being able to apply filter operations or compute aggregates at peak memory bandwidth. Hence, a GPU together with its device memory can be considered *programmable memory* that employs near-data processing capabilities. Similar to the concept of in-storage computing, the idea is to move processing closer to data storage, in order to avoid superfluous data movement over longer distances, which involves slower, less energy-efficient buses. Given that memory capacity of GPUs is about an order of magnitude more expensive than main memory, keeping data on the device is interesting for very frequently accessed data. The ability to perform complex operations in only a few milliseconds is particularly relevant for use cases like interactive analytics and stream processing, where windows of ephemeral stream data are kept on the device.

Table 2.2.1: The two fundamentally different use cases of GPUs.

GPUs as "programmable memory"	GPUs as accelerators
• increasingly relevant as memory capacity is doubling approximately every 2.5 years	• increasingly relevant with improving interconnect speed (*PCIe 4.0*, *NVLink*)
• hot data fitting on device	• arbitrary data size from various sources
• processing all on-device data in ~5 – 25 ms	• processing at the speed of the interconnect

• use cases:
  • interactive analytics
  • stream processing
	• use cases:
	  • data ingestion
	  • in-situ querying of raw data

⇒ **low-latency processing**	⇒ **high-throughput processing**

As outlined in Table 2.2.1, a fundamentally different perspective for GPUs is using them as *accelerators* to exploit their computational capabilities. Instead of expecting data to be stored on the GPU, compute-intense operations are delegated to the GPU, in order to utilise the GPU's superior compute performance. While this lifts the restrictions imposed by the expensive and limited memory capacity, data has to be transferred via the interconnect. Today, PCIe 3.0 is the most common interconnect. PCIe 3.0 supports full-duplex communication with a theoretical bandwidth of up to 16 GB/s. Given that data needs to be moved to the GPU, the limited interconnect bandwidth may impose an upper bound on the achievable processing throughput of GPUs. As a consequence, the *accelerator* use case is only feasible for complex operations, where data transfers can be amortised by the GPU's superior performance. Comprising a sequence of complex, batch-like tasks, such as parsing and index generation, data ingestion constitutes a well-suited candidate for being accelerated on GPUs. It is worth noting that this use case becomes increasingly relevant as faster interconnects are emerging. PCIe 4.0 doubles the throughput of its predecessor and is already supported by *AMD Zen 2*-based processors [20]. Another alternative is presented by *NVLink*. The interconnect provides full-duplex communication with a bandwidth of up to 25 GB/s per direction and link. GPUs of the *NVIDIA Volta* architecture, for instance, integrate six links per GPU for an aggregate bandwidth of 300 GB/s. *NVLink* provides a fast interconnect between GPUs and, being integrated with *IBM POWER9* CPUs, also between CPUs and GPUs. First applications that investigated the use of *NVLink* to overcome the transfer bottleneck have shown promising results [21].

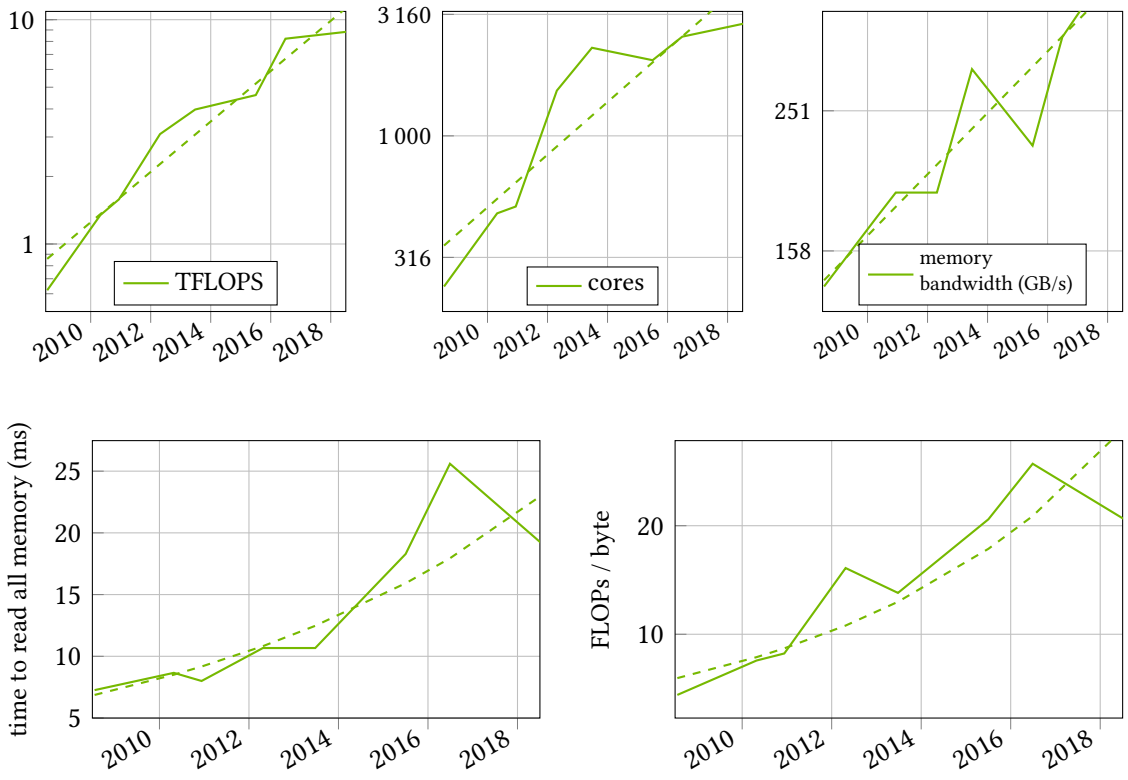


Figure 2.2.2: Evolution of the key characteristics of GPUs over the course of a decade.

Apart from the interconnect and memory capacity, which are essential considerations for the overall system design, we also analyse characteristics that are more influential for the development of massively parallel algorithms on GPUs. Figure 2.2.2 presents how the GPUs' computational performance, measured in trillion floating point operations per second (TFLOPS), the number of cores, and the memory bandwidth evolved over a timespan of ten years. In the given timespan, compute performance increased 14.34-fold, the number of cores increased 12.27-fold, and memory bandwidth increased 3.18-fold. It is worth to highlight that improvements in compute performance were mostly driven by increasing parallelism. While the number of cores was increasing 12.27-fold, corresponding to a CAGR of 28%, per-core performance only increased 1.17-fold with a CAGR of a mere 1.5%. This, once again, emphasises that it is essential to design algorithms for scalability and parallel execution from the ground up in order to ensure they are able to benefit from future hardware advancements. Another important insight is that, over time, compute performance is outgrowing memory bandwidth. Dividing the compute

performance by the memory bandwidth yields a ratio of 4.4 floating point operations (FLOPs) per byte back in 2008. This ratio has grown to 19.9 FLOPs/byte by the end of 2018. Extrapolating this trend with a fitted exponential growth function yields an increase from 6.0 FLOPs/byte in 2008 to 36.3 FLOPs/byte by the year 2020. As memory bandwidth is growing slower than compute performance, reducing the amount of memory traffic is an important measure. Hence, saving memory traffic in exchange for extra computation is becoming increasingly relevant. Another important implication of the comparably slow advancement in memory bandwidth is an increase in the time it takes to scan through all data stored on the GPU. Similar to the compute to memory bandwidth ratio, the memory capacity to memory bandwidth ratio is also increasing over time. Compared to 2008, when the time it takes to read through all on-device memory was down to 7.26 ms, numbers increased 2.5-fold to 18.3 ms by late 2018. A fitted exponential growth function yields a four-fold increase by 2020, growing from 6.9 ms in 2008 to 27.4 ms by the year 2020.

2.2.2 System Design Considerations

Data transfers, communication, and synchronisation between processors in a heterogeneous system are expensive, yet, sometimes inevitable. Given that these factors influence the system performance, it is essential to consider processing pipelines beginning to end and optimise for the end-to-end processing performance, rather than considering individual processing steps in isolation.

Many GPU-accelerated approaches comprise multiple processing stages, where one stage depends on an intermediary result from a preceding stage. As the kernel execution of dependent stages needs to be configured based on the outcome of a prior stage, approaches tend to synchronise CPU execution before a dependent stage. Consider, for instance, the simple parallel algorithm presented in Section 2.1.2, which computes the sum over a given array of items. The number of thread blocks that have to be launched to compute the sum depends on the size of the input. If, however, the size of the input depends on a preceding stage that is executed on the GPU, the CPU has to wait for the preceding stage to finish, copy back the result from the GPU, and configure the number of thread blocks correctly based on the result that has been retrieved from the GPU.

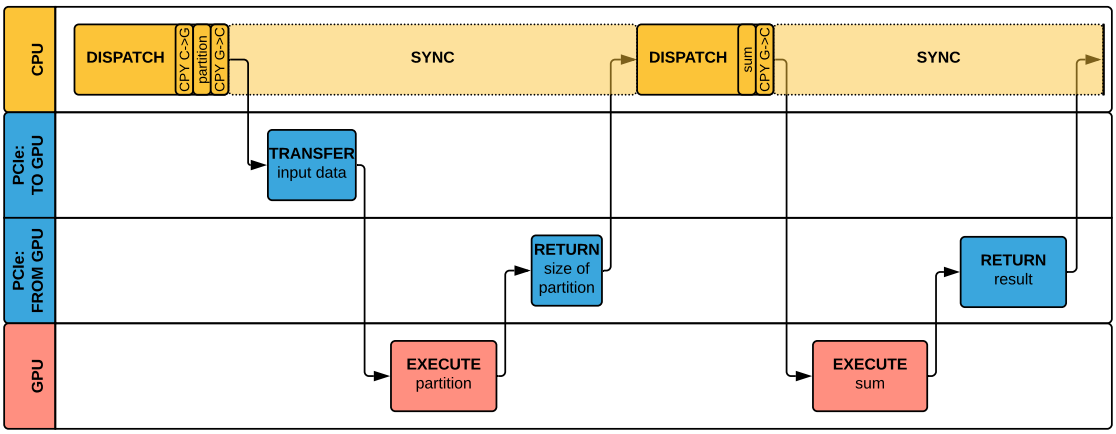


Figure 2.2.3: An illustration of a multi-stage algorithm that first partitions the input on some criteria and then computes the sum over the items of the first partition. The presented algorithm constitutes an inefficient processing pipeline that incurs a superfluous synchronisation and data transfer between the CPU and the GPU. After the GPU has finished executing the partition kernel, the CPU is copying back the size of the first partition to determine the number of thread blocks that need to be launched for computing the sum over the items of the first partition.

Figure 2.2.3 exemplifies these steps of an inefficient processing pipeline that incurs superfluous synchronisation and data transfers between the CPU and the GPU. The depicted processing pipeline exemplifies a two-stage algorithm. The first stage partitions a given input (e.g., a set of employees) based on some criteria (e.g., by gender). The second stage computes the sum over some attribute of the first partition (e.g., the salaries of all female employees).

In a first step, the CPU dispatches work to the GPU. This includes the data transfer of the input from the CPU to the GPU (*CPY C⟶G*), the kernel launch that partitions the input based on some criteria (*partition*), and the memory transfer of an intermediary result back from the GPU to the CPU (*CPY G⟶C*). The intermediary result that is copied back represents the size of the first partition that is required in order to configure the second stage of the algorithm (i.e., computing the sum over the items from the first partition). After dispatching the work, the CPU has to wait for the GPU's *partition* kernel execution to finish, as well as for the memory transfer of the intermediary result to complete. Only once the size of the first partition is known to the CPU, it can dispatch the work for the second stage of the algorithm. That is, dispatching the kernel launch for computing the sum with the correct number of thread blocks and the memory transfer for returning the sum from the GPU.

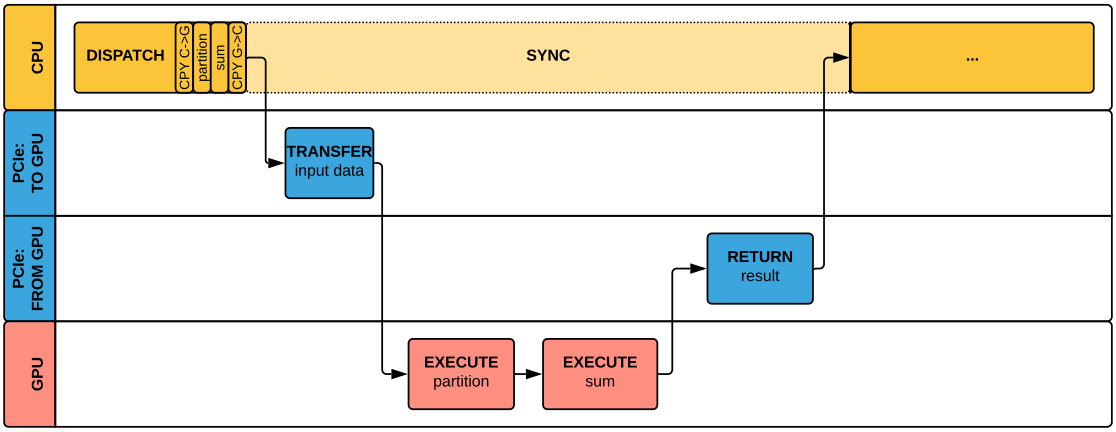


Figure 2.2.4: An example for an efficient design of a multi-stage algorithm that dispatches all work to the GPU at once, keeps intermediary results on the GPU, and avoids superfluous synchronisation between the CPU and the GPU.

The superfluous synchronisation and data transfer between the CPU and the GPU are an obvious shortcoming of the approach illustrated in Figure 2.2.3. Even though the relevant data from the first stage (*partition*) already resides on the GPU, the CPU has to copy back data from the GPU in order to be able to configure the kernel launch of the second stage (i.e., *sum* kernel).

This shortcoming is addressed in the processing pipeline exemplified in Figure 2.2.4. The multi-stage algorithm in Figure 2.2.4 dispatches all work to the GPU at once, keeps intermediary results on the GPU, and avoids superfluous synchronisation between the CPU and the GPU. Once the CPU has dispatched the work to the GPU, both processors can proceed independently. Only once the CPU has to use the result from the GPU, it has to synchronise with the GPU.

In order to circumvent the superfluous synchronisation and data transfer, the CPU needs to be able to configure the kernel launch of the second stage (i.e., *sum* kernel) already upfront. To do so, we fix the number of thread blocks for that kernel launch to the number of SMs available on the GPU multiplied by the number of thread blocks that can reside on an SM. As discussed in Section 2.1.3, the number of thread blocks that can reside on an SM depends on the resources required by each thread block and the resources available on each SM. With this adaptation, however, we also need to change the assignment from chunks to thread blocks.

Instead of mapping exactly 256 items to one thread block and having the number of thread blocks scale with the input size, each thread block now processes multiple chunks of 256 items. That is, each thread block iterates over multiple chunks until reaching the end of the input. For instance, in one possible mapping of chunks to thread blocks, the i-th thread block iterates over the chunks $i + (k \times |B|)$, with $k \in \mathbb{Z}^+$, $k < \lceil n/256 \rceil$, where $|B|$ represents the fixed number of thread blocks that have been launched and n represents the number of items in the array. This algorithm design helps making GPUs self-sufficient and avoids overhead due to superfluous synchronisations and data transfers.

2.2.3 Implementation Considerations

Having looked at the overall design of heterogeneous systems and how GPUs can be efficiently integrated into processing pipelines in Section 2.2.2, this section looks at the GPU in isolation, addressing the aspects that should drive the design, development, and implementation of algorithms for GPUs.

Looking at a processor like the GPU, the need for massively parallel algorithms is apparent. The *NVIDIA V100*, for instance, integrates 80 SMs, each of which supports up to 2 048 threads. Considering the 163 840 threads required to fully occupy the GPU emphasises that algorithms have to be designed for massive parallelism from the ground up. While today's GPUs already require a massive degree of concurrency, the trend indicates that future processors will provide even more hardware parallelism, requiring that algorithms are able to scale to even more cores in order to benefit from this advancement. Another implication of the massive parallelism is that, even with a fine-grained data parallel approach, batches must be reasonably large. If each thread is processing tasks of as little as eight bytes, the batch must comprise at least 1.3 MB. This also provides the motivation for addressing the problem of data ingestion in this work, as data ingestion often involves processing vast amounts of data, allowing for reasonably large batch sizes.

Apart from designing algorithms for parallel execution, load-balancing is another important concern. Unless being designed to provide robust performance, algorithms may achieve reasonable performance for some inputs, while performance may severely degrade for others. Considering, for instance, a simple strategy for a parallel string-matching

algorithm that returns all lines containing a specific pattern. A simple approach may simply split the given input by lines and simultaneously process the lines independently. A shortcoming of such an approach is that some lines may be considerably larger than others, resulting in huge runtime discrepancy. An effective measure to ensure robust performance is to minimise variance in the execution time between the tasks. Considering that the execution time is often a function of the problem size, employing a data parallel approach with fine-grained parallelism is an effective measure.

3

Summary of Publications

This chapter provides a summary of the individual contributions that this publication-based dissertation comprises. Overall, this thesis focuses on our two accepted peer-reviewed publications. For each publication, we highlight the key idea behind our approach, outline the key achievements, and summarise the author's contributions.

Section 3.1 outlines our publication on the design and development of an efficient sorting algorithm on the GPU. The sorting algorithm is extended with our pipelined heterogeneous approach to address inputs that exceed the GPUs's device memory or do not reside on the GPU in the first place. Section 3.2 summarises our massively parallel parsing algorithm for delimiter-separated data formats. Similar to our sorting algorithm, this publication also addresses inputs exceeding the device memory. With a streaming, heterogeneous approach, we do not only lift the memory capacity constraints, but also reduce the end-to-end processing latency.

3.1 A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs

Reference: E. Stehle and H.-A. Jacobsen. "A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD. 2017, pp. 417–432. DOI: 10.1145/3035918.3064043

Full-text version enclosed: Appendix A

Summary:

Sorting is not only a fundamental operation for many database operations, such as index creation, sort-merge joins, or user-requested output ordering, but it is also an indispensable building block for many massively parallel algorithms. With an efficient sorting algorithm that provides superior performance, this publication lays the foundation for our endeavour of accelerating data ingestion on GPUs.

Building on an MSD-based hybrid radix sort, our algorithm considers a completely new approach to sorting on GPUs. By deviating from the established approach of using an LSD radix sort, we lift the requirement for stable sorting passes. This enables the usage of native shared memory atomic operations that became available on recent GPU architectures [22]. Our algorithm exploits this new hardware feature to maintain a single data structure (histogram) that is shared by multiple threads, instead of having each thread maintain its own copy. As a consequence, on-chip memory requirements are drastically reduced, allowing us to consider a wider radix digit and reduce the number of sorting passes, which ultimately leads to considerable performance improvements.

At the time of submission for peer review, our approach outperformed the state-of-the-art GPU-based sorting algorithm by a factor of no less than 1.6. For key-value pairs comprising 64-bit keys and 64-bit values, our algorithm sees as much as a four-fold improvement.

Author's contributions: Conceived, developed, and implemented the approach. Devised optimisations. Conducted analysis and experimental evaluation. Wrote the paper.

3.2 ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data

Reference: 2019:stehleparparaw

Full-text version enclosed: Appendix B

Summary:

Parsing takes a central role in many data ingestion pipelines and is essential for a wide range of use cases, such as stream processing and in-situ querying of raw data. Exhibiting high arithmetic intensity, however, the compute-intense step often constitutes a major bottleneck in many processing pipelines. In this publication, we are able to considerably accelerate parsing with a scalable, massively parallel parsing algorithm for GPUs without sacrificing applicability and flexibility.

Instead of tailoring our approach to a single format, we are able to perform a massively parallel DFA simulation, which is more flexible and powerful, supporting more expressive parsing rules with general applicability. In order to enable a parallel approach for DFA simulation, we exploit the fact that there are only few different states in an DFA that represents the parsing rules of a delimiter-separated format. While our approach increases the overall effort by a single-digit constant factor, it enables a fully concurrent approach and allows us to scale linearly to thousands of cores and beyond.

Parsing a challenging dataset that comprises 4.8 GB in the CSV format in as little as 0.44 seconds, including data transfers, the end-to-end performance of our approach provides an order-of-magnitude improvement over the state-of-the-art GPU-based parsing algorithm [23]. Compared to the best-performing CPU-based parsing algorithm, our algorithm is seeing as much as a 100-fold improvement [24].

Author's contributions: Conceived, developed, and implemented the approach. Devised optimisations. Conducted analysis and experimental evaluation. Wrote the paper.

4

Discussion

This chapter discusses our results in the larger context of massively parallel algorithms for GPUs. Analysing our GPU-accelerated parsing algorithm, we assess the importance of fundamental massively parallel algorithms, such as sorting and the prefix scan, and review related approaches. We conclude the chapter highlighting that, given the level of parallelism provided with modern GPUs, it pays off to accept an increase in the algorithm's overall complexity in order to enable a fully concurrent approach that scales linearly with the number of cores.

Our work on a massively parallel parsing algorithm for delimiter-separated formats highlights the importance of efficient and scalable parallel primitives. That is, recurring parallel algorithms that constitute building blocks for more specialised and complex algorithms. Our approach to parsing delimiter-separated formats on the GPU, for instance, builds on *sorting* and the *prefix scan*. Taken together, these algorithms account for roughly one third of the overall runtime of our GPU-accelerated parsing algorithm, with the majority of that time being spent on sorting. Given the influence of these fundamental algorithms on the performance of more specialised algorithms, in the following, we review approaches for these algorithms, putting a particular focus on sorting. We discuss the impact of these parallel primitives in the context of our massively parallel parsing algorithm.

The parallel prefix scan is amongst the most frequently recurring building blocks for data parallel algorithms, as it addresses one of the core challenges. That is, threads that are concurrently processing independent chunks from the input lack information about input that is preceding their chunk. The parallel prefix scan provides a means of propagating lacking information from preceding chunks. For a given binary reduction operator (e.g., addition), the prefix scan takes an array of input elements and returns an array, where the i-th output element is computed by applying the reduction operator to all input elements up to and including the i-th element [25]:

$$y_i = \bigoplus_{k=0}^{i} x_k$$

Over the years many approaches for a parallel prefix scan have been proposed [25, 26, 27, 28, 29, 30, 31, 32, 33, 34]. As the prefix scan is bound by the available memory bandwidth on modern GPUs, a more recent approach is focusing on lowering the amount of memory movements [25]. While previous approaches incurred roughly three times the input size in memory transfers, Merrill et al. were able to lower the memory movements by one third, providing considerable performance improvements [25]. The prefix scan used by our massively parallel parsing algorithm builds on the approach by Merrill et al. [25].

Even though the prefix scan is indispensable for many massively parallel algorithms, its runtime makes up only a fraction of the overall processing duration of more complex algorithms. In the case of our massively parallel parsing algorithm, the prefix scan's complexity is linear in the total number of chunks. As revealed by our experimental evaluation, the prefix scan therefore contributes only about 2% of the total runtime for a reasonably large chunk size of 15 bytes or more.

In contrast to the prefix scan, sorting is considerably more complex, accounting for a major portion of the total runtime of our massively parallel parsing algorithm. While, for an input of size n, the latest GPU-based approach for a prefix scan requires only a single pass, incurring ~$2n$ memory movements, sorting requires a multitude of the memory transfers, taking roughly an order of magnitude more time. This highlights the importance of an efficient sorting algorithm and motivates our work on a memory-bandwidth efficient hybrid radix sort for GPUs.

Given its importance, there is an extensive body of work that addresses sorting on GPUs. Early approaches were building on sorting network-related algorithms like the bitonic mergesort [35, 36, 37, 38]. Other merge-based approaches have been presented by Satish et al. [39, 40], Davidson et al. [41], Green et al. [42], and Tansic et al. [43]. He et al., as well as Sintorn et al. have considered a partition-based sorting approach [44, 45]. In recent years, the LSD radix sort gained popularity as it requires only limited coordination between threads [40, 46, 47]. The LSD radix sort was continuously improved by considering more bits per sorting pass. Even though considering a wider radix digit, i.e., more of a key's bits per sorting pass, is an important endeavour, it comes with its own drawbacks and challenges on GPUs. In particular, the amount of fast on-chip memory requirements, which grow exponentially with the number of bits being considered, imposes an upper bound on the radix digit.

By considering a completely new approach to sorting with our MSD-based hybrid radix sort, we are able to alleviate the excessive on-chip memory requirements and consider more bits with each sorting pass. This reduces the number of required sorting passes and lowers the overall amount of memory traffic. Our experimental evaluation shows that our algorithm is able to consider eight bits per sorting pass at peak memory bandwidth, as our savings in the amount of memory transfers are translating to speedups in a one-to-one ratio. Moreover, building on an MSD-based hybrid radix sort allows us to save even more memory bandwidth for more uniform input distributions. As a result, our sorting algorithm sees considerable performance improvements, particularly for larger keys (e.g., 64-bit keys), as it is able to reduce the number of sorting passes even further. When sorting 64-bit keys, for instance, this effect culminates in a three-fold performance improvement over prior work.

While our massively parallel parsing algorithm spends roughly a third of the runtime on the prefix scan and sorting, two thirds of the runtime are spent on our specialised algorithms, such as the massively parallel DFA simulation and the type conversion. The massively parallel algorithm for simulating DFAs is at the core of our parsing algorithm and helps to keep track of the parsing context. Building on our scalable approach for simulating DFAs has allowed us to both generalise and drastically improve performance. Our results highlight that, even though our algorithm incurs a constant-factor increase in overall complexity, we are able offset the increased complexity with a scalable approach

that is able to utilise the thousands of cores of GPUs. Our approach does not only allow us to exploit the parallelism of modern GPUs but it also enables us to benefit from the ongoing trend of increasing core counts. Considering the end-to-end processing performance, our algorithm parses a dataset of 4.8 GB in the CSV format in as little as 0.44 seconds. This corresponds to a parsing rate of 10.9 GB/s, which shows that we are able to exploit the full-duplex capabilities of the PCIe bus. Our approach provides over an order of magnitude performance improvement over the state-of-the-art GPU-based parsing algorithm [23]. Compared to CPU-based approaches, it sees as much as a two orders of magnitude improvement [24].

5

Conclusions

In order to help data management systems cope with the continuously growing influx of data, this thesis presented multiple massively parallel algorithms to accelerate data ingestion on GPUs. The algorithms developed in this work are load-balanced, robust to input variance and, most importantly, designed for scalability from the ground up, in order to address the shift towards increasingly parallel processors and benefit from this new direction of processor advancements.

With an efficient sorting algorithm that provides superior performance, we have laid the foundation for our endeavour of accelerating data ingestion on GPUs. Sorting is not only essential for index generation, but it is also an essential building block for many massively parallel algorithms. The sorting algorithm, for instance, is an indispensable component for our approach to massively parallel parsing of delimiter-separated formats. Deviating from the popular approach of using an LSD radix sort on the GPU, our novel, MSD-based hybrid radix sort alleviates the need for stable sorting passes, which allows the algorithm to use native shared memory atomic operations to synchronise accesses to a shared data structure in order to drastically reduce the on-chip memory requirements. By lifting the on-chip memory constraints, we are able to consider a wider radix digit and reduce the number of required sorting passes, which ultimately leads to considerable performance improvements over prior work.

Being able to sort two gigabytes of eight-byte records in as little as 50 ms, our approach achieves a sorting rate of more than 40 GB/s for on-device data. When our work was initially submitted for peer review, our approach was seeing as much as a four-fold speedup over the state-of-the-art at the time, always retaining at least a minimum speedup of a factor of 1.6. Considering the latest approaches that have followed the initial submission for peer review, our approach is still seeing as much as a three-fold speedup, retaining at least a 1.2-fold performance advantage.

With a pipelined heterogeneous sorting algorithm that mitigates the overhead associated with PCIe data transfers, we addressed inputs that either do not reside on the GPU or exceed the available device memory. In order to efficiently exploit the limited device memory capacity, our heterogeneous sorting algorithm uses an in-place replacement strategy that improves the overall performance for large inputs. Comparing the end-to-end sorting performance to the state-of-the-art CPU-based radix sort, our heterogeneous approach achieves a 2.1-fold and a 1.5-fold improvement for sorting 64 GB key-value pairs with a skewed and a uniform distribution, respectively.

While the evaluation of our sorting algorithm has focused on the performance aspect, there are also more fundamental improvements made possible by our novel approach that are beyond the scope of this thesis and remain yet to be explored. A major advantage of our MSD-based hybrid radix sort is that it is converging towards the sorted order. That is, with each sorting pass, our algorithm is getting closer to the correct order. This property allows using intermediate results, while the sorting algorithm is still ongoing. Intermediary results can be used for approximate query processing or rendering of live results in data visualisation applications. Exploiting the property of convergence of our MSD-based hybrid radix sort for these applications remains yet to be explored in future work.

Another advantage made possible by a partition-based sorting algorithm like ours is the potential for considerable performance improvements for partial sorting. This aspect of our approach has already been analysed by related work that looked into efficient top-k query processing on massively parallel hardware [48]. Their analysis shows promising results for our approach, which outperforms a specialised top-k algorithm for many evaluations with $k \geq 256$. Another relevant benefit of using a partition-based algorithm

is its ability to efficiently sort strings. Even though the MSD-based hybrid radix sort presented in this work still requires some adaptations for supporting strings, we consider this a viable and interesting direction for future work, given the relevancy of its application for methods like dictionary compression.

With `ParPaRaw`, our massively parallel parsing algorithm for delimiter-separated formats, we addressed one of the core algorithms for data ingestion. Exhibiting high arithmetic intensity that involves multiple instructions for each character, which often comprise only a single byte, parsing is a very compute-intense step that often constitutes a major bottleneck in many processing pipelines.

`ParPaRaw` is able to considerably accelerate parsing with a scalable, massively parallel algorithm for GPUs without sacrificing applicability and flexibility. With our massively parallel algorithm for simulating DFAs, we were able to address the main challenge of identifying the parsing context (quotation scopes, comments, directives, etc.), without requiring a prior sequential pass. Being designed for scalability from the ground up with a data parallel algorithm that does not require any serial work, `ParPaRaw` does not only provide considerable performance improvements, but is also future-proof and can continue to gain speed-ups, as more cores are being added with future processors. Exploiting the parallelism of a modern GPU with 3584 cores, our approach achieves a parsing rate of as much as 14.2 GB/s for on-device data.

By extending the massively parallel parsing algorithm running on the GPU with a heterogeneous streaming approach, we were able to exploit the full-duplex capabilities of the PCIe bus and lower the end-to-end parsing latency. Instead of waiting for the input to be transferred as a whole, we are streaming smaller partitions via the PCIe bus to the GPU. This allows overlapping the data transfer of raw input data, parsing on the GPU, and returning the parsed data over the PCIe bus, which maximises resource utilisation and reduces latency. Considering the end-to-end processing performance, the algorithm parses a dataset of 4.8 GB in the CSV format in as little as 0.44 seconds, which corresponds to a parsing rate of 10.9 GB/s. `ParPaRaw` provides over an order of magnitude performance improvement over the state-of-the-art GPU-based parsing algorithm [23]. Compared to CPU-based approaches, `ParPaRaw` sees as much as a two orders of magnitude improvement [24].

At the core of `ParPaRaw` is the massively parallel algorithm for simulating DFAs. Building on our scalable approach for simulating DFAs has allowed us to both generalise and drastically improve performance for the well-studied problem of parsing delimiter-separated formats. Our experimental evaluation has shown promising results, highlighting that, given the thousands of cores of modern GPUs, it pays off to accept an increase in the algorithm's overall complexity in order to enable a fully concurrent approach that scales linearly with the number of cores. As DFAs are essential for a wide range of algorithms such as searching and pattern matching, the application of our approach to other problems remains to be evaluated in future work. Similarly, our approach is well-suited to be adapted to other formats like JavaScript Object Notation (JSON) where the order of fields may vary.

Glossary

ALU arithmetic logic unit

CAGR compound annual growth rate

CPU central processing unit

CSV comma-separated values

DFA deterministic finite automaton

DRAM dynamic random-access memory

FLOP floating point operation

FPU floating-point unit

GPS Global Positioning System

GPU graphics processing unit

HBM High Bandwidth Memory

ILP instruction-level parallelism

IoT Internet of Things

JSON JavaScript Object Notation

MSRP manufacturer's suggested retail price

PCIe Peripheral Component Interconnect Express

SaaS software as a service

SFU special functions unit

SIMD single instruction multiple data

SM Streaming Multiprocessor

TFLOPS trillion floating point operations per second

TPU Tensor Processing Unit

Bibliography

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2018.

[2] N. Bronson, T. Lento, and J. L. Wiener. "Open data challenges at Facebook." In: ICDE. 2015.

[3] Meeker, Mary. *Internet Trends 2019*. https://www.bondcap.com/report/itr19. 2019.

[4] *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017-2022 White Paper*. Tech. rep. Cisco, 2019.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-scale." In: *PVLDB* (2015).

[6] Mubarik Imam. *Facebook's F8 2018 Developer Conference*. URL: https://developers.facebook.com/videos/f8-2018/f8-2018-day-1-keynote/. 2018.

[7] Statista Inc. *Number of smartphones sold to end users worldwide from 2007 to 2020*. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/. 2019.

[8] Ericsson. *Ericsson Mobility Report Q4 2018*. https://www.ericsson.com/en/mobility-report/reports/q4-update-2018. 2018.

[9] lime. *Ericsson Mobility Report Q4 2018*. https://www.li.me/second-street/berlin-sets-record-fastest-1-million-electric-scooter-rides. 2019.

[10] Dieter Bohn. *Amazon Says 100 Million Alexa Devices Have Been Sold*. https://www.theverge.com/2019/1/4/18168565/amazon-alexa-devices-how-many-sold-number-100-million-dave-limp. 2018.

[11] N. P. Jouppi, C. Young, N. Patil, et al. "In-datacenter performance analysis of a tensor processing unit." In: ISCA. 2017.

[12] Karl Rupp. *Microprocessor Trend Data*. https://www.github.com/karlrupp/microprocessor-trend-data. 2018.

[13] A. Arunkumar, E. Bolotin, B. Cho, et al. "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability." In: *SIGARCH* (2017).

[14] Y. Shafranovich. *RFC4180 - Common format and MIME type for comma-separated values (CSV) files.* `https://tools.ietf.org/html/rfc4180`. 2005.

[15] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. "DBMS data loading: An analysis on modern hardware." In: *DaMoN* (2016).

[16] E. Stehle and H.-A. Jacobsen. "A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD. 2017, pp. 417–432. DOI: 10.1145/3035918.3064043.

[17] *NVIDIA Turing GPU Architecture. Whitepaper.* Tech. rep. NVIDIA, 2018.

[18] *NVIDIA Fermi Architecture. Whitepaper.* Tech. rep. NVIDIA, 2009.

[19] J. C. McCallum. *Memory Prices 1957+*. `https://jcmit.net/memoryprice.htm`. 2020.

[20] *The 2nd Generation AMD EPYC Processor. Whitepaper.* Tech. rep. TIRIAS Research, 2019.

[21] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects." In: SIGMOD (2020).

[22] *NVIDIA GeForce GTX 980. Whitepaper.* Tech. rep. NVIDIA, 2014.

[23] RAPIDS. *RAPIDS - Open GPU Data Science*. `https://rapids.ai`. 2012.

[24] P. A. Boncz, M. Zukowski, and N. Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. 2005.

[25] D. Merrill and M. Garland. "Single-pass parallel prefix scan with decoupled look-back." In: *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[26] P. M. Kogge and H. S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations." In: *IEEETransComp* (1973).

[27] W. D. Hillis and G. L. Steele Jr. "Data parallel algorithms." In: *CACM* (1986).

[28] J. Sklansky. "Conditional-Sum Addition Logic." In: *IRE Transactions on Electronic Computers* (1960).

[29] R. P. Brent and H. T. Kung. "A regular layout for parallel adders." In: *IEEETransComp* (1982).

[30] G. E. Blelloch. "Scans as primitive parallel operations." In: *IEEETransComp* (1989).

[31] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. "Fast Scan Algorithms on Graphics Processors." In: ICS. 2008.

[32] D. Merrill and A. Grimshaw. *Parallel scan for stream architectures*. Tech. rep. 2009.

[33] S. Ha and T. Han. "A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment." In: *TPDS* (2013).

[34] S. Yan, G. Long, and Y. Zhang. "StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization." In: PPoPP. 2013.

[35] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management." In: SIGMOD. 2006.

[36] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. "High performance comparison-based sorting algorithm on many-core GPUs." In: IPDPS. 2010.

[37] P. Kipfer and R. Westermann. "Improved GPU sorting." In: *GPU gems* 2 (2005), pp. 733–746.

[38] M. Harris, S. Sengupta, and J. D. Owens. "Gpu gems 3." In: *Parallel Prefix Sum (Scan) with CUDA* (2007).

[39] N. Satish, C. Kim, J. Chhugani, et al. "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort." In: SIGMOD. 2010.

[40] N. Satish, M. Harris, and M. Garland. "Designing efficient sorting algorithms for manycore GPUs." In: IPDPS. 2009.

[41] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. "Efficient parallel merge sort for fixed and variable length keys." In: InPar 2012. 2012.

[42] O. Green, R. McColl, and D. A. Bader. "GPU merge path: a GPU merging algorithm." In: ICS 2012. 2012.

[43] I. Tanasic, L. Vilanova, M. Jordà, et al. "Comparison Based Sorting for Systems with Multiple GPUs." In: GPGPU. 2013.

[44] E. Sintorn and U. Assarsson. "Fast parallel GPU-sorting using a hybrid algorithm." In: *Journal of Parallel and Distributed Computing* (2008).

[45] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. "Efficient Gather and Scatter Operations on Graphics Processors." In: SC '07. 2007.

[46] L. Ha, J. Krüger, and C. T. Silva. "Fast Four-Way Parallel Radix Sorting on GPUs." In: *Computer Graphics Forum* (2009).

[47] D. Merrill and A. Grimshaw. "High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for Gpu Computing." In: *Parallel Processing Letters* (2011).

[48] A. Shanbhag, H. Pirk, and S. Madden. "Efficient top-k query processing on massively parallel hardware." In: SIGMOD (2018).

[49] E. Stehle and H.-A. Jacobsen. "ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data." In: *PVLDB* 13.5 (2020), pp. 616–628. DOI: 10.14778/3377369.3377372.

[50] B. Gu, A. S. Yoon, D.-H. Bae, et al. "Biscuit: A framework for near-data processing of big data workloads." In: *ACM SIGARCH Computer Architecture News* (2016).

[51] I. Jo, D.-H. Bae, A. S. Yoon, et al. "YourSQL: a high-performance database system leveraging in-storage computing." In: *PVLDB* ().

[52] IBM POWER9 NPU team. "Functionality and performance of NVLink with IBM POWER9 processors." In: *IBM Journal of Research and Development* (2018).

[53] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner. "Applicability of GPU Computing for Efficient Merge in In-Memory Databases." In: *ADMS@ VLDB*. 2011.

[54] *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015-2020 White Paper*. Tech. rep. Cisco, 2016.

[55] *NVIDIA Tesla P100. Whitepaper*. Tech. rep. NVIDIA, 2016.

[56] M. Herf. *Radix tricks*. `http://stereopsis.com/radix.html`. 2001.

[57] D. Merrill and NVIDIA Corporation. *CUB*. `https://github.com/NVlabs/cub`. 2016.

[58] K. Thearling and S. Smith. "An improved supercomputer sorting benchmark." In: SC '92. 1992.

[59] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. "Quickly generating billion-record synthetic databases." In: *ACM SIGMOD Record*. 1994.

[60] G. Graefe. "Implementing sorting in database systems." In: *ACM Computing Surveys (CSUR)* (2006).

[61] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors." In: PACT '07. 2007.

[62] J. Chhugani, A. D. Nguyen, V. W. Lee, et al. "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture." In: *PVLDB* (2008).

[63] J. Wassenberg and P. Sanders. "Engineering a Multi-core Radix Sort." In: Euro-Par 2011. 2011.

[64] C. Kim, J. Park, N. Satish, et al. "CloudRAMSort: Fast and Efficient Large-scale Distributed RAM Sort on Shared-nothing Cluster." In: SIGMOD. 2012.

[65] O. Polychroniou and K. A. Ross. "A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort." In: SIGMOD. 2014.

[66] H. Inoue and K. Taura. "SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures." In: *PVLDB* (2015).

[67] M. Cho, D. Brand, R. Bordawekar, et al. "PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort." In: *PVLDB* (2015).

[68] M.-C. Albutiu, A. Kemper, and T. Neumann. "Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems." In: *PVLDB* (2012).

[69] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. "Multi-core, main-memory joins: Sort vs. hash revisited." In: *PVLDB* (2013).

[70] B. Chandramouli and J. Goldstein. "Patience is a virtue: Revisiting merge and sort on modern processors." In: SIGMOD. 2014.

[71] C. Kim, T. Kaldewey, V. W. Lee, et al. "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs." In: *PVLDB* (2009).

[72] A. Shahvarani and H.-A. Jacobsen. "A Hybrid B+-tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms." In: SIGMOD. 2016.

[73] N. Leischner, V. Osipov, and P. Sanders. "GPU sample sort." In: IPDPS. 2010.

[74] F. Dehne and H. Zaboli. "DETERMINISTIC SAMPLE SORT FOR GPUs." In: *Parallel Processing Letters* (2012).

[75] S. Baxter. *Modern GPU*. https://github.com/moderngpu/moderngpu. 2016.

[76] J. Hoberock and N. Bell. *Thrust: A parallel template library*. https://thrust.github.io. 2016.

[77] S. Ashkiani, A. A. Davidson, U. Meyer, and J. D. Owens. "GPU Multisplit." In: *CoRR* abs/1701.01189 (Jan. 2017). URL: http://arxiv.org/abs/1701.01189.

[78] M. Najafi, M. Sadoghi, and H. A. Jacobsen. "Configurable hardware-based streaming architecture using Online Programmable-Blocks." In: ICDE. 2015. DOI: 10.1109/ICDE.2015.7113336.

[79] A. Luotonen. *Logging Control In W3C httpd*. https://www.w3.org/Daemon/User/Config/Logging.html. 1995.

[80] P. M. Hallam-Baker and B. Behlendorf. *Extended Log File Format*. https://w3.org/TR/WD-logfile. 1996.

[81] Taxi and Limousine Commission. *TLC Trip Record Data*. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. 2016.

[82] Kaggle. *Kaggle Datasets*. https://www.kaggle.com/datasets. 2018.

[83] Sumo Logic. *Press Release*. https://www.sumologic.com/press/2018-02-27/growth-milestones. 2018.

[84] *NVIDIA Tesla V100 GPU Architecture. Whitepaper*. Tech. rep. NVIDIA, 2017.

[85] R. Johnson and I. Pandis. "The bionic DBMS is coming, but what will it look like?" In: *CIDR* (2013).

[86] C. N. Fischer. *On parsing context free languages in parallel environments*. Tech. rep. 1975.

[87] A. Shahvarani and H.-A. Jacobsen. "A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms." In: SIGMOD. 2016.

[88] T. Mühlbauer, W. Rödiger, R. Seilbeck, et al. "Instant loading for main memory databases." In: *PVLDB* 6.14 (2013), pp. 1702–1713.

[89] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, and D. Kossmann. "Speculative Distributed CSV Data Parsing for Big Data Analytics." In: SIGMOD. 2019.

[90] Alenka. *Alenka - A GPU Database Engine*. https://github.com/antonmks/Alenka. 2012.

[91] Simantex. *Simantex - CSVImporter*. https://github.com/Simantex/CSVImporter. 2012.

[92] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. "Mison: A Fast JSON Parser for Data Analytics." In: *PVLDB* 10.10 (2017), pp. 1118–1129.

[93] D. Bonetta and M. Brantner. "FAD.Js: Fast JSON Data Access Using JIT-based Speculative Optimizations." In: *PVLDB* 10.12 (2017), pp. 1778–1789.

[94] G. Langdale and D. Lemire. "Parsing Gigabytes of JSON per Second." In: *CoRR* (2019). URL: http://arxiv.org/abs/1902.08318.

[95] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. "Here are my Data Files. Here are my Queries. Where are my Results?" In: *CIDR* (2011).

[96] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. "Adaptive Query Processing on RAW Data." In: *PVLDB* 7.12 (2014), pp. 1119–1130.

[97] T. Azim, M. Karpathiotakis, and A. Ailamaki. "ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data." In: *PVLDB* 11.3 (2017), pp. 324–337.

[98] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. "NoDB: Efficient Query Execution on Raw Data Files." In: SIGMOD. 2012.

[99] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. "Fast Queries over Heterogeneous Data Through Engine Customization." In: *PVLDB* 9.12 (2016), pp. 972–983.

[100] M. Ivanova, Y. Kargin, M. Kersten, et al. "Data Vaults: A Database Welcome to Scientific File Repositories." In: SSDBM. 2013.

[101] R. Hai, S. Geisler, and C. Quix. "Constance: An Intelligent Data Lake System." In: SIGMOD. 2016.

[102] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. "Parallel Data Analysis Directly on Scientific File Formats." In: SIGMOD. 2014.

[103] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. "Filter Before You Parse: Faster Analytics on Raw Data with Sparser." In: *PVLDB* 11.11 (2018), pp. 1576–1589.

[104] B. Chandramouli, D. Xie, Y. Li, and D. Kossmann. "FishStore: Fast Ingestion and Indexing of Raw Data." In: *PVLDB* 12.12 (2019), pp. 1922–1925.

[105] W. Zhao, Y. Cheng, and F. Rusu. "Vertical Partitioning for Query Processing over Raw Data." In: SSDBM. 2015.

[106] Y. Cheng and F. Rusu. "Parallel In-situ Data Processing with Speculative Loading." In: SIGMOD. 2014.

[107] A. Mycroft. *String Processing Instruction*. `https://groups.google.com/forum/embed/#!topic/comp.lang.c/2HtQXvg7iKc`. comp.lang.c. 1987.

[108] Apache Software Foundation. *Apache Arrow*. `https://arrow.apache.org`. 2019.

[109] Yelp Inc. *Yelp Dataset Challenge*. `www.yelp.com/dataset/challenge`. 2019.

Appendix A

A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs

A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs

Elias Stehle
Technical University of Munich (TUM)
Boltzmannstr. 3
85748 Garching, Germany
stehle@in.tum.de

Hans-Arno Jacobsen
Technical University of Munich (TUM)
Boltzmannstr. 3
85748 Garching, Germany
jacobsen@in.tum.de

ABSTRACT

Sorting is at the core of many database operations, such as index creation, sort-merge joins, and user-requested output sorting. As GPUs are emerging as a promising platform to accelerate various operations, sorting on GPUs becomes a viable endeavour. Over the past few years, several improvements have been proposed for sorting on GPUs, leading to the first radix sort implementations that achieve a sorting rate of over one billion 32-bit keys per second. Yet, state-of-the-art approaches are heavily memory bandwidth-bound, as they require substantially more memory transfers than their CPU-based counterparts. Our work proposes a novel approach that almost halves the amount of memory transfers and, therefore, considerably lifts the memory bandwidth limitation. Being able to sort two gigabytes of eight-byte records in as little as 50 milliseconds, our approach achieves a 2.32-fold improvement over the state-of-the-art GPU-based radix sort for uniform distributions, sustaining a minimum speed-up of no less than a factor of 1.66 for skewed distributions. To address inputs that either do not reside on the GPU or exceed the available device memory, we build on our efficient GPU sorting approach with a pipelined heterogeneous sorting algorithm that mitigates the overhead associated with PCIe data transfers. Comparing the end-to-end sorting performance to the state-of-the-art CPU-based radix sort running 16 threads, our heterogeneous approach achieves a 2.06-fold and a 1.53-fold improvement for sorting 64 GB key-value pairs with a skewed and a uniform distribution, respectively.

1. INTRODUCTION

Many of today's database systems are facing unprecedented loads as they must cope with data that is generated by hundreds of millions of people, devices, and sensors [7, 9]. Analysing, filtering, and querying the enormous amount of data in a timely manner becomes increasingly difficult. In an endeavour to keep systems responsive, a lot of effort is put into adapting database systems to modern hardware trends [21, 6, 23, 1, 3, 33, 5, 22, 30, 36]. The availability of low-cost memory, for instance, has given rise to the wide adoption of in-memory databases [35, 26, 24, 8]. In many cases, this has shifted the bottleneck from I/O to memory bandwidth and compute performance. Moreover, the rise of multi-core architectures, vector processing capabilities, and growing cache sizes requires to rethink many parts of database systems.

Sorting is no exception to this effort. As a fundamental operation in database systems, sorting finds its application in index creation, user-requested output sorting, and sort-merge joins [13]. Moreover, sorting can speed up duplicate removal, ranking, and grouping operations [13]. Therefore, a lot of research has been devoted to identifying efficient sorting algorithms that utilise modern hardware features and scale well across multiple cores, processors, and even nodes [21, 6, 35, 40, 24, 33, 22, 8]. After having recently achieved sorting rates of over one billion keys per second [28], Graphics Processing Units (GPUs), featuring thousands of cores and a memory bandwidth of several hundred gigabytes per second, emerged as a promising platform to accelerate sorting. Besides approaches that are based on sorting networks [25, 12], merge sort [37, 34, 35], and sample sort [27, 11], the most promising results for larger problem sizes have been shown for implementations using a radix sort [18, 16, 34, 35, 28].

A major challenge arising when trying to make use of the massive parallelism of GPUs for sorting is the fact that a key's position within the output sequence depends on all other keys. Previous work has addressed this issue by using a least-significant-digit-first radix sort (LSD radix sort) that iterates over the keys' bits from the least-significant to the most-significant digit, considering an implementation specific number of consecutive bits at a time. With each sorting pass, a stable counting sort is used to partition the keys into buckets according to the bits being considered with the current pass [16, 34, 35, 28]. The stable counting sort computes each key's offset by counting the number of keys with a smaller digit value and, as it needs to be stable, the keys with the same digit value preceding the key in the input sequence. To achieve concurrency, GPU-based implementations split the input into a sequence of small blocks (a few thousand keys) that are processed in parallel. For each block, a local histogram over the keys' digit values is computed, and the prefix-sum over these histograms is used to determine a key's position within the output sequence. Since the whole input has to be read twice and written once with each sorting pass, radix sort implementations aim to

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
 ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064043

increase the number of bits being considered with each sorting pass, in order to lower the number of passes and the amount of memory transfers. However, as the size of the histogram grows exponentially with the number of bits being considered with each sorting pass, the growing complexity of the prefix-sum computation and the small on-chip memory impose a limit on the number of bits per digit. Due to these limitations, state-of-the-art approaches are restricted to consider only five bits at a time. Incurring a considerable amount of memory transfers, such as reading or writing the input 39 times in the case of 64-bit keys, the sorting rate is ultimately bound by the available memory bandwidth.

In order to lift the memory bandwidth limitation, this work presents a novel, hybrid radix sort that is able to efficiently sort on eight bits with each pass. This reduces the number of sorting passes and therefore the total amount of memory transfers by a factor of at least 1.6. In contrast to an LSD radix sort that is used by state-of-the-art implementations (e.g., CUB), the presented approach does not rely on stable sorting passes [29]. Therefore, it is not restricted to respecting the order of preceding sorting passes for keys falling into the same bucket. Lifting this constraint enables our approach to use native shared memory atomic operations that became available with recent GPU microarchitectures to mitigate the downside of considering more bits with each sorting pass [31, 32]. Our hybrid approach starts from the most-significant bit and proceeds towards the least-significant bit, partitioning the keys into smaller and smaller buckets. It avoids running into situations where the partitioning of the input into too many buckets would negatively impact performance, by finishing with a local sort as soon as a bucket is small enough to fit into on-chip memory. As the local sort performs the sorting in on-chip memory, it needs to access the device memory only twice, once for reading and once for writing the keys, no matter how many sorting passes it requires. This further saves essential memory bandwidth and boosts performance for favourable distributions. While a typical parallel most-significant-digit-first radix sort (MSD radix sort) may incur load balancing issues for skewed distributions that result in buckets of greatly varying size, we efficiently utilise the low-overhead scheduling mechanisms of the GPU to avoid any load imbalance, by subdividing every bucket into tiny, fixed-size blocks that can be evenly distributed amongst the GPU's Streaming Multiprocessors (SMs).

To circumvent the overhead associated with a large number of kernel invocations, we use only a constant number of invocations during each sorting pass. Rather than using at least one invocation for each bucket, passing the memory offset of the bucket's keys and its size as arguments, we generate that information as a byproduct of a sorting pass and place it in device memory, from where it can be read in the subsequent pass to determine the work assignments. Moreover, we show that the use of shared memory atomic operations is highly efficient for almost any key distribution and introduce measures to mitigate performance degradation for highly skewed distributions.

In order to address inputs that either do not reside on the GPU or exceed the available device memory, we present a pipelined heterogeneous sorting algorithm that mitigates the overhead associated with PCIe data transfers. By splitting the input into multiple sub-problems, we are able to interleave several processing stages, allowing us to exploit the full-duplex capability of the PCIe bus while simultaneously sorting on the GPU. In order to max out the limited device memory, we propose an in-place replacement strategy that immediately reuses memory by returning a sorted run while concurrently replacing the contents with the next sub-problem. This allows us to support larger sub-problems, which improves the overall performance for sorting large inputs of tens of gigabytes.

We evaluate the hybrid radix sort for various key and value sizes over twelve different, increasingly skewed distributions and compare it to the state-of-the-art GPU-based radix sort (CUB)[29]. Our experimental results demonstrate that the hybrid radix sort efficiently capitalises on the 1.6-fold reduction in the amount of memory transfers, seeing no less than a 1.58-fold improvement over CUB. Being able to sort two gigabytes of 64-bit keys with an associated 64-bit value in as little as 56 milliseconds, our approach peaks out at a four-fold speed-up. Building on the results of our hybrid radix sort, we evaluate the end-to-end performance for our heterogeneous sort and compare it to the state-of-the-art CPU-based radix sort running 16 threads [8]. Being able to sort 16 GB comprised of key-value pairs with a skewed distribution in as little as 3.37 seconds, the heterogeneous sort outperforms PARADIS by a factor of 2.64 [8]. Sorting an input of 64 GB with a skewed distribution, we still see a 2.06-fold improvement over PARADIS, despite the fact that our CPU-side processing on a weaker processor with only six-cores contributes more than 9.3 seconds to the 16 second total.

Overall, the contributions of this work are five-fold:

1. We present a novel, hybrid radix sort for GPUs that proceeds from the most-significant to the least-significant bit to circumvent the downside of considering more bits with each sorting pass. Not relying on stable sorting passes allows our approach to efficiently sort on eight bits at a time, and therefore reduce the number of passes and the amount of memory transfers by no less than a factor of 1.6.
2. We successfully address the challenges arising from implementing an MSD-based radix sort on GPUs, such as load balancing and congestion issues for skewed distributions and performance degradation due to bucket handling.
3. Using a local sort for sorting small buckets, we are not only able to avoid running into situations with an overwhelmingly large number of buckets, but also to considerably boost the performance for favourable key distributions, culminating in a four-fold speed-up.
4. As an MSD-based radix sort may result in millions of buckets that need to be kept track of, we establish an analytical model that is used to calculate the upper bounds on the number of buckets and analyse the memory requirements. The model shows the feasibility of our hybrid radix sort, indicating that the additional memory overhead, such as for keeping track of buckets, does not exceed a mere 5% of an LSD radix sort.
5. We address inputs that either do not reside on the GPU or exceed the available device memory using a pipelined heterogeneous sorting algorithm that mitigates the overhead associated with PCIe data transfers. In order to efficiently exploit the limited device memory, we propose an in-place replacement strategy that improves the overall performance for large inputs.

Table 1: Notation

symbol	description
k	number of bits per key
d	number of bits per digit
KPT	number of keys per thread
KPB	number of keys per block
$\hat{\partial}$	threshold for local sorting
$\underline{\partial}$	threshold for merging buckets

This paper is organised as follows. In Section 2, we introduce the basics of radix sorting and present the fundamental concepts of general-purpose computing on GPUs. Section 3 analyses the state-of-the-art approaches for sorting on GPUs. Section 4 presents the hybrid radix sort, how it is realised and how performance drops are mitigated. Section 5 addresses our heterogeneous sorting algorithm that aims to mitigate the overhead introduced with PCIe data transfers. Section 6 evaluates the performance of the presented approach and compares it to the state-of-the-art.

2. BACKGROUND

This section gives a quick introduction to radix sorting followed by an overview of recent GPU microarchitectures. This work focuses on NVIDIA GPUs and the *CUDA* computing platform. CUDA has been widely adopted for general purpose computing on GPUs and allows to tailor implementations to specific hardware characteristics. The notation used throughout this work is presented in Table 1.

2.1 Radix Sorting

Radix sorting relies on the reinterpretation of a k-bit key as a sequence of d-bit digits, which are considered one at a time. The basic idea is, that splitting the k bits of the keys into smaller d-bit digits results in a small enough radix $r = 2^d$, such that the keys can efficiently be partitioned into r distinct buckets. As sorting on each digit can be done with an effort that is linear in the number of keys n, the whole sorting can be achieved with a total complexity of $\mathcal{O}(\lceil k/d \rceil \times n)$. Iterating over the keys' digits can be performed in two fundamentally different ways. Either by proceeding from the most-significant to the least-significant digit (MSD radix sort), or vice versa (LSD radix sort).

The MSD radix sort starts with the most-significant digit and partitions the keys into a sequence of r distinct buckets, according to their digit value. This can be done using a counting sort, which starts computing the histogram over the keys' most-significant digit. As the histogram reflects the number of keys that shall be put into each of the r buckets, computing the exclusive prefix-sum over these counts yields the memory offsets for each of the buckets. Finally, the keys are scattered into the buckets according to their digit value. Recursively repeating these steps on subsequent digits for the resulting buckets ultimately yields the sorted sequence.

In contrast, the LSD radix sort starts with the least-significant digit and performs a stable sort in subsequent passes. That is, if there is a tie on the digit's value of any two keys, the original order of the preceding pass is preserved. Hence, during a sorting pass, a key's position is given by the number of keys with a lower digit value plus the number of keys that have the same digit value and precede the key in the input sequence.

2.2 GPU Architecture

GPU architectures have been steadily scaling up their core counts over time, proliferating in thousands of simple cores today. Moreover, discrete GPUs feature their own device memory that provides transfer rates of up to 750 GB/s [32]. The basic building block of a GPU is a SM. Each SM consists of a set of cores (e.g., 64, 128, or 192), a *register file*, *shared memory*, and an L1 cache. The register file is used to hold the registers of all threads that reside on an SM. An important limitation of registers is that they cannot be addressed dynamically. Hence, declaring an array and accessing it based on an index that cannot be resolved at compile time, would render the use of registers impossible. In contrast, shared memory is dynamically addressable and shared by a whole group of threads, referred to as *thread block*. A thread block is the atomic unit that is scheduled on an SM. It is defined by the amount of shared memory that is required, a function (the kernel), and the number of threads that execute the given function. It is possible, and even desired, that several thread blocks reside on an SM at any given time, increasing the *occupancy*. For every thread block that resides on an SM, the required number of registers and the amount of shared memory is allocated to the thread block. Thus, the maximum number of blocks that can possibly reside on a single SM is implied by the resources a thread block requires and the resources that are available on an SM. For example, an SM with 96 KB of shared memory and 65 536 registers, could accommodate up to eight thread blocks of 256 threads, if each block requires eight KB of shared memory and 16 registers per thread (a total of 4 096 registers per block). Each thread block is subdivided into a set of warps, currently comprising 32 threads. All threads of a warp are executed in a lockstep manner. With several thread blocks and therefore several warps residing on an SM, the scheduler can choose from the set of resident warps that are ready for being executed rather than waiting for a single warp to get ready (e.g., for hiding latency from memory accesses).

3. RELATED WORK

Over the years many different approaches have been pursued for sorting on GPUs. Kipfer et al. have proposed a solution that is based on the odd-even sorting network and an approach using a bitonic merge sort algorithm [25]. *GPUTeraSort*, introduced by Govidaraju et al., aims to address larger keys as well as larger problem sizes that previously have been limited to the GPU's device memory [12]. Moreover, they used an index sort that uses the CPU to rearrange the key-value pairs based on the key-index pairs that are sorted and returned by the GPU. To reduce the overall complexity of a sorting network-based approach, which exhibits a complexity of $\mathcal{O}\left(n \log^2 n\right)$, Harris et al. propose a solution that divides the input sequence into smaller subsequences, sorts them locally using a binary radix sort, i.e., a radix of two, and merges the chunks using a parallel bitonic sort [17]. Similarly, Ye et al. proposed *Warpsort*, which sorts the chunks using a bitonic sorting network [41]. In addition, their approach avoids costly synchronisation by exploiting the synchronous execution of a warp's threads. Other merge-based approaches have been presented by Satish et al. [34, 35], Davidson et al. [10], Green et al. [15], and Tansic et al. [38].

Apart from merge-based approaches, promising results

were shown for implementations building on a distribution-based sort, such as a radix sort. As part of their introduction of a multi-pass scatter operation that aims to coalesce memory writes, He et al. present an MSD radix sort that uses a fixed number of partitioning passes [18]. The MSD radix sort partitions the input, considering five bits at a time. After performing a fixed number of partitioning passes, a bitonic sort is used to sort each of the partitions. The approach works for a uniform distribution, which is assumed when the fixed number of required partitioning passes is calculated. For skewed distributions, however, their sort would not gain a big advantage from the partitioning passes. For instance, assuming an input that, according to the algorithm's logic, would be considered for two partitioning passes. If the keys' bits are all zero on their most-significant ten bits, the algorithm would spend time on the two partitioning passes, while it still ends up with one single partition. Sintorn et al. present a hybrid approach that starts with a partitioning pass, using either a quicksort or a bucket sort, before sorting each of the resulting partitions with a merge sort [37]. The bucket sort uses an initial set of heuristic splitters, counts the keys belonging to each of the partitions defined by the splitters, and, if required, refines the splitters. Once the splitters have been examined, the keys are scattered into 1 024 partitions, which, in turn, are sorted using the proposed merge sort.

Satish et al., as well as Ha et al., propose an LSD radix sort, which coalesces writes to device memory by performing the key scattering in the local shared memory, prior to writing the local partitions to device memory [34, 16]. While Ha et al. sort on only two bits at a time, Satish et al. manage to use digits of four bits by repeatedly using a binary split within shared memory on each single bit, before writing the partitions to device memory. Satish et al. provide a thorough evaluation of comparison and non-comparison sorts on different architectures [35]. They examined that their radix sort, which is based on the approach presented by Satish et al. [34], is compute-bound, and make a case for their merge sort. To avoid the computational effort associated with the binary split, and save the amount of data being transferred, Merrill et al. present a tuned radix sort that achieves a sorting rate of over one billion 32-bit keys per second, yet, reaches its optimum for sorting four-bit digits [28]. The approach of Merrill et al. has been integrated into the CUB header library, which is developed and maintained by NVIDIA Research [29]. As part of CUB, the radix sort is able to efficiently sort on five bits at a time.

4. ON-GPU HYBRID RADIX SORTING

This section describes our approach to radix sorting on GPUs. We give an overview of our sorting algorithm, introduce its two fundamental components, the *counting sort* and the *local sort*, and explain how we designed the hybrid radix sort for GPUs. We first limit the presentation of the approach to the sorting of unsigned integer keys before explaining how it can be extended to sort keys and key-value pairs of any primitive data type (e.g., *int*, *float*, *double*).

4.1 The Hybrid Radix Sort

The proposed algorithm is based on an MSD radix sort, which recursively partitions the keys into smaller and smaller buckets until the buckets are eventually small enough to be sorted in on-chip shared memory. We distinguish between

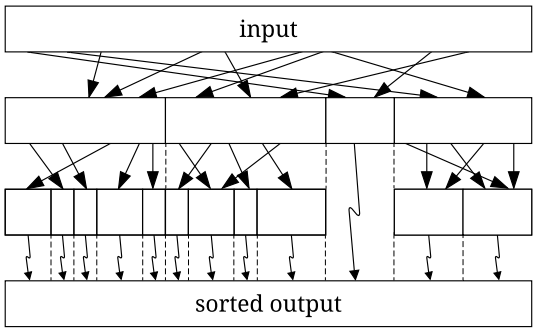


Figure 1: The hybrid radix sort

a *counting sort*, which performs the aforementioned partitioning of a bucket into sub-buckets, and a *local sort*, that brings all keys of a small bucket into sorted order. The algorithm starts with a counting sort on the most-significant digit (the d most-significant bits) and produces a sequence of $r = 2^d$ sub-buckets, each containing a partition of the keys that share the same value on their most-significant digit. With every subsequent sorting pass, each sub-bucket that resulted from the partitioning of the buckets in the preceding pass is either further partitioned using another counting sort, or sorted using a local sort. While proceeding to the next sorting pass, the digit according to which the counting sort partitions the buckets into sub-buckets is advanced by one towards the least-significant digit. The algorithm is finished once all keys are sorted up to and including the least-significant digit, or, if all buckets have been sorted with a local sort. The general workflow is illustrated in Figure 1. It depicts a local sort as a waved arrow pointing from a single bucket to a location in memory for the sorted output, and a counting sort as a set of arrows that point from a single bucket to a sequence of sub-buckets.

While the local sort works in-place, the counting sort requires auxiliary memory to which the partitioned keys are written. In order to reuse memory, we are using double-buffering for the whole sorting algorithm. With every sorting pass, memory for the input and the output is exchanged, such that the memory for the output of the preceding pass becomes the input of the current pass, and the previous pass's input memory is reused for the output. As the memory for the input and the output is alternating with each pass, we return the final sorted sequence within the memory of the original input if the number of digits, $\lceil k/d \rceil$, is even, and within the auxiliary memory otherwise. Since the algorithm might finish early, i.e., if all buckets have been sorted using a local sort prior to reaching the least-significant digit, we make sure that a local sort always places the sorted key sequence in the memory being used to return the final sorted output.

As the local sort is sorting a bucket's keys within on-chip shared memory, it is limited to sort a maximum of $\hat{\partial}$ keys, which is implied by the key size and the available hardware resources. To take advantage of the fact that preceding counting sort passes have already sorted the bucket's keys up to a certain digit, we can tune an LSD radix sort to only sort on the remaining digits.

Buckets that exceed the local sort threshold, $\hat{\partial}$, are partitioned into sub-buckets using a counting sort. The counting sort reads the keys starting at the bucket's offset from the

Table 2: Hybrid radix sorting example: sorting 16 keys of k=4 bits with d=2 bits and a radix of r=4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys (radix 4)	31	12	01	23	12	22	12	00	11	10	10	31	03	13	12	03
histogram	4	8	2	2												
prefix-sum	0	4	12	14												
sort (radix 4)	bucket 0				bucket 1								bucket 2		bucket 3	
	01	00	03	03	12	12	12	11	10	10	13	12	23	22	31	31
histogram	1	1	0	2	2	1	4	1					local		local	
prefix-sum	0	1	2	2	0	2	3	7					local		local	
sort (radix 4)	b_0	b_1	b_3		b_0	b_1	b_2					b_3	local		local	
	00	01	03	03	10	10	11	12	12	12	12	13	22	23	31	31

input memory, partitions them into sub-buckets according to the specified digit and writes the sequence of sub-buckets cohesively into the output memory, such that the sub-bucket holding the keys with the smallest digit value starts at the same offset as the input bucket. An implementation of a counting sort for a single bucket follows these steps:

(1) *Compute the histogram over the digit values of all keys in the bucket to determine the size of each sub-bucket.*

(2) *Compute the exclusive prefix-sum over the histogram to get the offset for each of the r sub-buckets.*

(3) *Scatter the keys into the sub-buckets according to the keys' digit values.*

The presented approach is exemplified in Table 2, which shows the algorithm for 16 keys of a length of four bits. The radix sort is performed using two-bit digits with a radix of $r = 4$, requiring exactly two passes to fully sort the keys. The keys are represented in a base four notation. In the example, we set the threshold for local sorting to $\hat{\partial} = 3$, turning to a local sort for buckets of three keys or less.

4.2 Fine-Grained Parallelism on GPUs

While the presented algorithm allows to process individual buckets in parallel, the level of parallelism may not suffice to have enough threads in flight to hide the latency from memory accesses. Therefore, we introduce a higher degree of concurrency for the counting sort by splitting the n keys of each bucket into a sequence of $\lceil n/KPB \rceil$ *key blocks*, each comprised of up to KPB keys. Each key block is processed once during the computation of the histogram and once during the scattering step.

In order to decrease the overhead associated with kernel invocations, we use only a constant number of invocations per sorting pass, independent of the number of buckets being sorted. A kernel invocation instructs the GPU to execute a given kernel (function) by a specified number of thread blocks, each comprised of a given number of threads. Rather than adjusting the arguments (e.g., pointer to a bucket's keys, number of keys) for each bucket individually, using multiple invocations, we put that information into device memory as a byproduct of the prefix-sum computation and launch just enough thread blocks to have one for each key block of each bucket. During the computation of the histogram and the key scattering step, each thread block looks up the bucket and the block of keys it is assigned to by reading that information from device memory.

We proceed similarly for the local sort, where we assign exactly one thread block to each bucket. However, there is a downside to using only a single kernel invocation for all buckets that are sorted using a local sort. That is, there are just as many threads being assigned for processing a large bucket that has close to $\hat{\partial}$ keys, as there are for sorting a relatively small bucket of only a few keys. Thus, with many threads being over-provisioned for small buckets, this introduces additional overhead. We address this issue in two ways.

Firstly, we start merging tiny neighbouring sub-buckets whose total number of keys falls below a certain threshold $\underline{\partial}$. That is, after a counting sort has partitioned a bucket into r sub-buckets, we merge any sequence of sub-buckets as long as their total number of keys is less than $\underline{\partial}$, with $\underline{\partial} \leq \hat{\partial}$. This further reduces the upper bound on the total number of buckets and avoids having too many tiny buckets, for which the scheduling of an own thread block would introduce considerable overhead, compared to the time that is spent on the sorting.

Secondly, instead of using a single kernel invocation that sorts all buckets whose size falls into the interval $[1, \hat{\partial}]$, we distinguish between different bucket sizes in that interval, e.g., bucket sizes of $[1, 128]$, $(128, 256]$, $(256, 512]$, ..., $(..., \hat{\partial}]$ keys, respectively. For each of these subintervals, a kernel is invoked with each thread block provisioning just enough threads to process the respective number of keys. We refer to each of these as a *local sort configuration*, which represents the combination of a kernel, a number of threads per thread block, and the supported bucket size. In addition to adjusting the number of threads per thread block, this allows to specify a certain kernel that is optimised for sorting the given number of keys. Hence, for small buckets, a configuration with a sorting network or another comparison-based sorting algorithm could be devoted, turning to an LSD radix sort for configurations supporting buckets of a larger size.

4.3 Histogram

One of the key advantages of the proposed approach is, that, in contrast to an LSD radix sort, the hybrid radix sort does not rely on stable sorting passes. Therefore, it is not restricted to respecting the order of preceding sorting passes for keys falling into the same sub-bucket. Lifting this constraint enables our approach to use native shared memory atomic operations for the histogram computation and the key scattering step to mitigate the downside of considering more bits with each sorting pass.

Our histogram computation aggregates one histogram per block in shared memory. Every thread reads KPT keys from device memory, iterates over them, and uses an *atomicAdd* operation to increment the counter in shared memory for the respective digit value. Once all threads of a block are done, the histogram that has been accumulated in shared memory

is added to the global histogram by adding the respective counters in device memory.

Since all threads of a thread block share the same counters for the local histogram, highly skewed distributions with only few digit values potentially degrade the performance, as this causes all threads to simultaneously access the same counters in shared memory. In order to be able to max out the available memory bandwidth, each SM must achieve a processing rate of $\frac{8 \times BW}{k \times |SMs|}$ keys per second, where BW denotes the peak memory bandwidth in bytes per second and $|SMs|$ the number of available SMs. Based on the number of SMs and the theoretical peak memory bandwidth of recent GPUs, this gives a required throughput of $3-4.5$ billion 32-bit keys per SM per second [31, 32]. For a constant distribution, however, our experiments show an average throughput of only 1.7 billion 32-bit keys per SM per second on an *NVIDIA Titan X (Pascal)*, due to competing updates to only one single shared memory location. This performance drop is shown for the *atomics only* approach in Figure 2, which depicts the memory bandwidth utilisation relative to the peak throughput of 369.17 GB/s (determined using a micro-benchmark for a read-only workload). In contrast, for a uniform distribution over q distinct digit values, with $q \geq 3$, the approach that uses atomics only, sees as much as 3.3 billion updates per SM per second, almost achieving peak memory bandwidth.

In order to avoid such a performance drop for highly skewed distributions, we use the available compute resources for a new approach (*thread reduction & atomics*) that reduces each thread's updates to shared memory. For the simple approach (*atomics only*), the computation for each key is limited to bit-shifting the desired digit to the least-significant digit, masking it, and atomically incrementing the counter for the resulting value in shared memory. Instead, with our improved approach, each thread stores its masked digit values in registers, uses a sorting network to bring them into sorted order and combines the counter updates for subsequent registers sharing the same value into a single *atomicAdd* operation. To limit the complexity of the sorting network, we sort runs of up to nine values at a time using a sorting network that involves 25 comparisons. Once the runs of digit values are in a sorted order, the algorithm iterates over them, combining any sequence of identical digit values into a single *atomicAdd* operation. As shown in Figure 2 (*thread reduction & atomics*), the reduced number of atomic updates now effectively mitigates the performance drop for a very skewed distribution.

Since, the block's histogram needs to be recomputed during the key scattering step, the algorithm stores each block's histogram in device memory to save compute resources later on. This slightly increases the utilised memory bandwidth of this step by a factor of $1 + \frac{r*4}{KPB \times k/8}$, given that the histogram uses counters of four bytes. Assuming a reasonable number of KPB, such as 6 912, this adds less than 4% to the data being transferred in the case of 32-bit keys, while saving essential compute resources during the key scattering step.

4.4 Key Scattering

For the scattering of a bucket's keys into its r sub-buckets, we use the same subdivision of buckets into key blocks as for the histogram computation. This allows to reuse the histograms that have already been computed and stored in

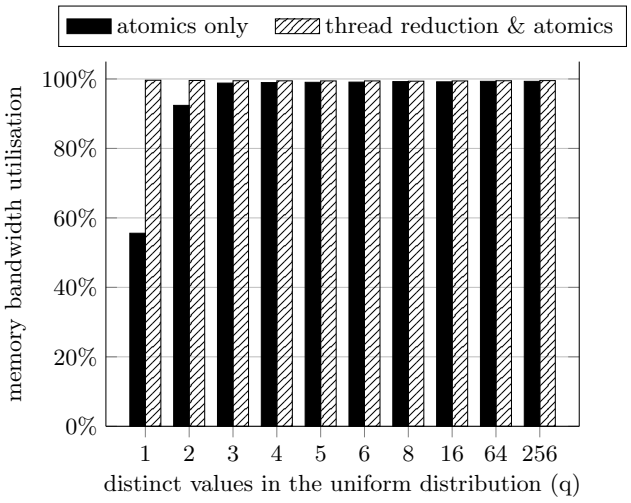


Figure 2: Achieved memory bandwidth utilisation for the histogram computation of a uniform distribution amongst a varying number of values using a non-optimised (atomics only) and an optimised approach (thread reduction & atomics)

device memory for each block. Each of these histograms indicates the number of keys that are going to be scattered from the key block into each of the sub-buckets. It can therefore be used to determine the size of the chunk of memory within each sub-bucket that needs to be reserved for the block's keys. A chunk of memory for storing n keys within a sub-bucket is reserved by performing a single *atomicAdd* operation that reads the sub-bucket's offset and adds n to it. Adding the number of keys, n, to the sub-bucket's offset guarantees that subsequent memory reservations are made beyond this chunk's memory reservation. The original value that has been read before n was added, can therefore be used as the starting offset in memory for the chunk.

Once up to r chunks of memory have been reserved for the block, its keys can be scattered into the reserved memory locations. However, simply scattering the keys to the chunks suffers from irregular memory accesses, as all threads of a thread block write the keys into different chunks residing at distant locations within device memory. To address this issue and coalesce writes to device memory, the keys of each block are first partitioned into the r sub-buckets within shared memory, before writing the whole sub-bucket of a block to the reserved chunk in device memory. Figure 3 illustrates this for a single key block. The top row depicts an excerpt of the device memory holding the input, the middle row represents the local shared memory, and the bottom row shows the device memory for the sub-buckets. The block's keys are read from device memory, partitioned locally into the sub-buckets in shared memory from where the local sub-buckets are finally copied to the chunks that have been reserved within the respective sub-buckets in device memory.

Compared to immediately scattering individual keys to irregular locations in device memory, this considerably improves the memory performance. Yet, depending on the granularity of memory transactions, the choice of r and the number of keys per block, KPB, may have considerable implications on the memory efficiency. For memory transac-

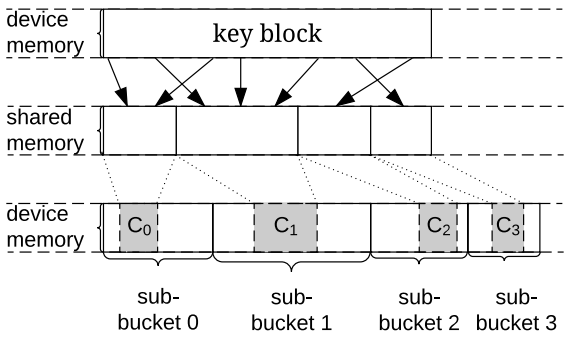


Figure 3: Using shared memory for write combining

tions that read or write T bytes at a time, the lower bound of required memory transactions for a block of k-bit keys is given by $\lceil \frac{KPB \times k}{T \times 8} \rceil$. That is, for each memory transaction, T bytes are written, with the exception of the last transaction, which possibly only writes the remainder that does not make up T bytes. However, the worst case may require one additional transaction for the remainder of each sub-bucket, totaling r additional memory transactions (neglecting inefficiencies due to misaligned writes). Since the local shared memory is limited to a few tens of kilobytes and has to fit all keys of a block, we are limited to a few thousand keys per block. One possible choice for a key block size would be 32 768 bytes, requiring a minimum of 1 024 transactions for $T = 32$ bytes. Calculating the worst case memory efficiency as the ratio of the lower to the upper bound on the number of memory transactions yields 80% for using eight-bit digits with a radix of 256. Further increasing the digit size to nine, ten, or eleven bits, would further decrease the efficiency to 66.66%, 50%, or 33.33%, respectively. We therefore choose $d = 8$ bits as an optimum trade-off between reducing the number of required sorting passes and the worst case memory efficiency.

The partitioning of a block's keys within shared memory makes use of the shared memory atomics to coordinate writes to the local sub-buckets. Similar to the mechanism being used for reserving chunks within the sub-buckets in device memory, we maintain one write counter in shared memory for each sub-bucket. Prior to writing a key into a local sub-bucket, a thread reads the value from the sub-bucket's write counter and adds the number of keys it intends to write. The original value that is read from the write counter serves as the thread's write offset within the sub-bucket in shared memory. Similar to our histogram approach, this makes extensive use of shared memory atomics. Hence, the key scattering suffers a similar performance drop for skewed distributions as the basic histogram implementation. However, compared to the histogram computation, the key scattering is not limited to just reading the keys from device memory, but also requires writing the keys back, resulting in twice the amount of data being transferred. In order to fully utilise the available memory bandwidth, it is therefore sufficient to achieve only half the processing throughput.

In order to mitigate the performance drop for very skewed distributions, we use an implementation that tries to combine writes of multiple keys to the same local sub-bucket. Instead of writing the keys one by one to the respective local sub-buckets, each thread looks at several keys at a time, writing any sequence of up to three keys sharing the same digit value at once. We refer to this approach as a *look-ahead of two*, since each thread considers the two following keys, in addition to the one it is currently looking at. We chose a look-ahead of two as it provides a reasonable trade-off for maximising the probability of combining writes for the highly skewed distributions, which we are trying to address, without wasting too many compute resources.

In order to avoid the overhead for distributions lacking the skewness to benefit from using a look-ahead due to an insufficiently high probability of finding keys destined for the same sub-bucket, we only consider the look-ahead for highly skewed distributions. Having the block's histogram at hand (from the preceding histogram computation), the algorithm can determine the skewness of the key distribution and only turn to the approach using a *look-ahead* for highly skewed distributions.

4.5 Analytical Model

One of the core challenges of the MSD-based hybrid radix sort is that the algorithm may end up with millions and millions of buckets that need to be maintained in memory. This section aims to seize the algorithm's complexity by deducing upper bounds on the maximum number of buckets, blocks, and memory requirements.

The following list presents the most important rules for the sorting algorithm:

(R_1) Any bucket of size n, with $n \leq \hat{\partial}$, is sorted within on-chip shared memory using a local sort.

(R_2) Any bucket of size n, with $n > \hat{\partial}$, is partitioned into r sub-buckets using a counting sort.

(R_3) Any sequence of sub-buckets is merged as long as the total number of keys falls short of the merge threshold $\underline{\partial}$, with $\underline{\partial} \leq \hat{\partial}$.

(R_4) Any bucket of size n, with $n > \hat{\partial}$, consists of exactly $\lceil n/KPB \rceil$ blocks and each block holds a sequence of keys from exactly one bucket.

Based on the presented list of rules, the following bounds can be deduced for sorting an input comprised of n keys:

(I_1) Following from R_1, at any given time, there are at most $\lfloor n/\hat{\partial} \rfloor$ buckets that cannot be sorted with a local sort.

(I_2) Following from I_1 and R_2, at any given time, there are at most a total of $r \times \lfloor n/\hat{\partial} \rfloor$ buckets. This can be deduced, as there are at most $\lfloor n/\hat{\partial} \rfloor$ buckets that are partitioned using a counting sort and each of those buckets is partitioned into at most r sub-buckets.

(I_3) Considering R_3, the upper bound given by I_2 can be refined to $\min(\lfloor 2 \times n/\underline{\partial} \rfloor + \lfloor n/\hat{\partial} \rfloor, r \times \lfloor n/\hat{\partial} \rfloor)$. Following from R_3, we conclude that any two subsequent sub-buckets must have at least $\underline{\partial}$ keys, as they would have been merged otherwise. Yet, as we can only merge sub-buckets originating from the same bucket, there may be one sub-bucket per bucket that cannot be merged.

(I_4) Following from R_4 and I_1, at any given time, there are at most $\lfloor n/KPB \rfloor + \lfloor n/\hat{\partial} \rfloor$ blocks. This follows from the fact that there are at most $\lfloor n/KPB \rfloor$ blocks with KPB keys. Adding to that up to one block for the remaining keys of each bucket gives an upper bound on the number of blocks.

Having determined the upper bound on the number of buckets and blocks, the memory requirements can easily be inferred. We are using unsigned integers of four bytes for

the counters of the histograms, as well as for the offsets of sub-buckets and key blocks. This can be easily adjusted to support more than $2^{32} - 1$ keys by using a larger data type. For the assignments of thread blocks to key blocks, we are using the following data structure: `{k_offs:uint, k_count:uint, b_id:uint, b_offs:uint}`, holding information on the starting offset of the keys, the number of consecutive keys, the bucket's unique identifier, and its offset. Memory required for these assignments needs to be allocated twice, once to keep track of the assignments of the current pass, and once for the assignments of the subsequent pass. Similarly, we store the following information for the assignment of a bucket whose size falls short of the local sort threshold: `{b_id:uint, b_offs:uint, is_merged:bool}`. In addition to storing one histogram for each bucket exceeding the local sort threshold, we allocate memory for each of its blocks' local histograms. This allows the algorithm to write the local histograms during the histogram computation and reuse the blocks' histograms in the subsequent scattering step.

Apart from the negligible amount of constant memory in the order of a few bytes for the synchronisation between thread blocks, the amount of memory (in bytes) that is required for sorting n keys comprised of k bits is given by:

(M_1) Input and auxiliary memory: $2 \times n \times k/8$

(M_2) Bucket histograms: $4 \times r \times \lfloor n/\hat{\partial} \rfloor$

(M_3) Block histograms: $4 \times r \times (\lfloor n/KPB \rfloor + \lfloor n/\hat{\partial} \rfloor)$

(M_4) Block assignments: $2 \times 16 \times (\lfloor n/KPB \rfloor + \lfloor n/\hat{\partial} \rfloor)$

(M_5) Local sort sub-bucket assignments: $12 \times \min(\lfloor 2 \times n/\underline{\partial} \rfloor + \lfloor n/\hat{\partial} \rfloor, r \times \lfloor n/\hat{\partial} \rfloor)$

For 32-bit keys, for instance, the total amount of memory required by M_2 through M_5 is bound by a mere 5% of M_1, given a reasonable configuration, such as $KPB = 6\,912$, $\hat{\partial} = 9\,216$, $\underline{\partial} = 3\,000$, and $r = 256$.

4.6 Sorting Pairs & Other Data Types

In order to support key-value pairs that are stored in a decomposed layout, the hybrid radix sort is extended to rearrange the values along with the keys they are associated with. Therefore, it is sufficient to adapt the key scattering step and the local sort, which are the only components involved in the permutation of keys. We extend the implementation of the key scattering step to keep track of the memory locations to which the individual keys have been written. Hence, while partitioning a block's keys within shared memory, each thread stores the offsets at which its keys have been placed. Once all keys have been rearranged and the block's local sub-buckets have been copied to device memory, the shared memory can be reused for the values. Each thread reads the values its keys are associated with from device memory and writes them to shared memory according to the offsets that have been stored in the thread's registers during the local partitioning of the keys. Finally, the local sub-buckets holding the values are copied to the respective locations in device memory. The local sort is extended by taking advantage of CUB's *BlockRadixSort* that comes with support for sorting key-value pairs [29]. For key-value pairs that are stored coherently in memory, keys and values need to be decomposed into a key and a value part, recomposing them once the sorting is done. Our experiments have shown that the de- and recomposition can be achieved at peak memory bandwidth, adding only negligible overhead to the sorting procedure.

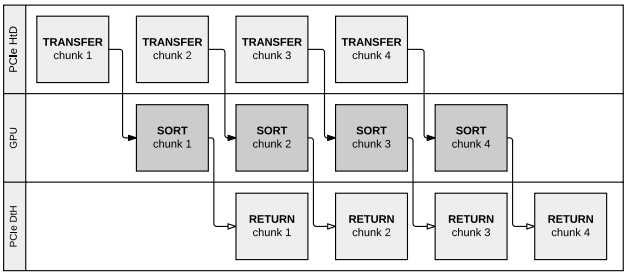


Figure 4: Pipelined sorting exploiting the available resources to mitigate the data transfer overhead

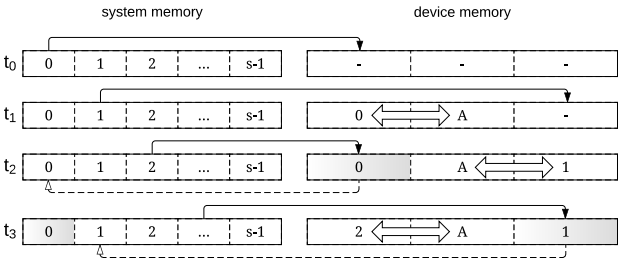


Figure 5: Efficient device memory utilisation for interleaving sorting with data transfers

While the presentation of the proposed hybrid radix sort has been limited to sorting unsigned integer keys, it can be easily extended to cover further primitive data types, such as `int`, `float`, and `double`. Support is added by using a bijective mapping from the input's data type to an order-preserving bit-string. This is as simple as flipping the sign-bit for signed integers and a little bit more involved for floats, where all bits have to be flipped if the sign bit was set, and only the sign bit is flipped otherwise [19]. We transform the input during the scattering step of the first counting sort and recover the original representation either during a local sort or the last counting sort pass.

5. HETEROGENEOUS SORTING

Having presented an efficient approach for sorting inputs within GPU's device memory, this section builds on that component with a heterogeneous sorting algorithm that addresses inputs that either do not reside on the GPU or simply do not fit into the available device memory. In either case, data has to be transferred over the comparably slow Peripheral Component Interconnect Express (PCIe) bus from the CPU to the GPU and vice versa, adding a considerable amount of overhead to the end-to-end sorting performance. Hence, in addition to the time taken for sorting a given input on the GPU (T_S), the time taken for transferring the whole input to the GPU (T_{HtD}) as well as the time taken for returning the sorted sequence from the GPU (T_{DtH}) have to be considered.

In order to support arbitrarily large inputs and mitigate the overhead that is introduced with the data transfers, we split the input into s chunks and treat them as a set of sub-problems that can be processed concurrently. As illustrated in Figure 4, this allows to overlap the processing stages of multiple sub-problems. For instance, while transferring the

data of the third chunk, the GPU can concurrently sort the second chunk and return the sorted run of the first chunk. Since the PCIe bus allows for full-duplex communication, we are able to accelerate data transfers without sacrificing throughput in either direction. With the sorted chunks being returned by the GPU, the CPU is left with the task of merging the s chunks into one final sorted sequence. Denoting the time taken for merging with T_M, the end-to-end sorting duration is given by:

$$T_{EtE} = \frac{T_{HtD}}{s} + \max(T_{HtD}, T_S, T_{DtH}) + \frac{T_{DtH}}{s} + T_M$$

Hence, for large enough s, the time taken for transferring the input to the GPU, sorting the chunks there and writing the sorted runs back to system memory is now almost down to the time taken for transferring the input over the PCIe bus one single time, or sorting the input on the GPU, whichever takes longer. This carves out a considerable amount of time that the CPU can spend on merging the s chunks. In order to improve the merging performance and avoid being bound by the available memory bandwidth, we use the parallel multiway merge that merges multiple chunks in a single pass from the parallel extension of *stdlibc++*. Moreover, to lower the number of merging passes for larger inputs, we max out the limited device memory with our in-place replacement strategy. That is, rather than allocating memory that can host four chunks: one for sorting, one for the auxiliary memory, one for the chunk being returned from the GPU, and one for copying the next chunk to the GPU, we only require enough memory for three chunks. As depicted in Figure 5 for the first few time-steps, we immediately reuse the memory that is used to hold a sorted chunk by replacing it with the input of the next chunk. At time step t_2 in Figure 5, for instance, we return the sorted run for *chunk* 0, while replacing it with the contents of *chunk* 2. This allows supporting larger chunks that may take up almost one third of the available device memory. Assuming a system with sufficient compute power to efficiently merge up to 16 chunks at a time and a GPU with 12 GB of memory, we could sort an input of up to 64 GB using only a single merging pass.

6. EXPERIMENTAL EVALUATION

The experiments were conducted on a system running *Ubuntu 16.04* with kernel version 4.4. The system is equipped with 128 GB DRAM (quad-channel, DDR4-2400) and a *Xeon E5-1650 v4* processor with six physical cores, clocked at 3.60 GHz. The source code was compiled with the *O*3 flag using release 8.0.44 of the CUDA toolkit. We used driver version 367.48 for an *NVIDIA Titan X (Pascal)* with 12 GB device memory, 3 584 cores, and a base clock of 1 417 MHz. The performance numbers were averaged over 25 runs. We used the CUB header library in version 1.5.1 to compare the presented approach to the state-of-the-art GPU-based radix sort [29]. CUB is developed as an open-source project by NVIDIA Research. The radix sort provided by CUB builds on the approach presented by Merrill et al. [28]. Moreover, we include comparisons to the radix sort implementation of Thrust [20], the merge sort presented by Baxter [4], and the radix sort from Satish et al. [34]. Similarly, we compare the end-to-end sorting performance of our heterogeneous sorting algorithm on the aforementioned system with a six-core CPU to the results that were reported on a stronger system with 32 cores for PARADIS (CPU-based radix sort) [8].

Table 3: Our default configurations

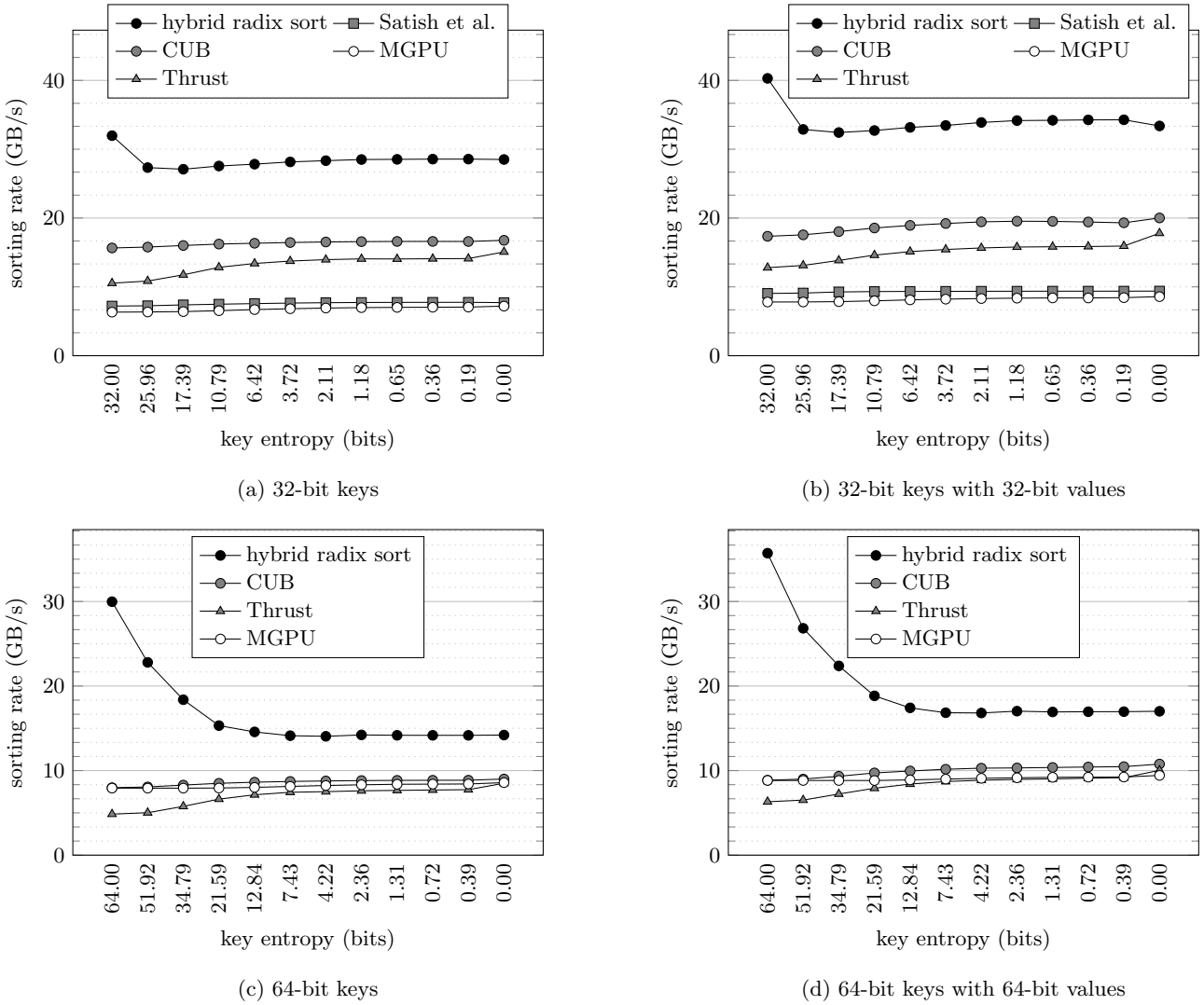
key/value size	KPB	threads	KPT	$\hat{\partial}$
32-bit keys	6 912	384	18	9 216
64-bit keys	3 456	384	9	4 224
32-bit/32-bit pairs	3 456	384	18	5 760
64-bit/64-bit pairs	2 304	256	9	3 840

For the counting sort, we used $d = 8$ bits per digit. In order to improve the occupancy, we determined the number of threads as well as the number of keys per thread (KPT) based on the amount of shared memory and the number of registers being required by the kernels, which, in turn, depends on the key and value size. Similarly, these factors impose an upper bound on the local sort threshold $\hat{\partial}$, where the kernel's on-chip memory requirements for processing $\hat{\partial}$ elements must not exceed the available resources of a single SM. The values that were determined for these parameters are depicted in Table 3.

Other than comparison-based sorting algorithms, the hybrid radix sort is not prone to the order of the input but rather sensitive to the key distribution. Hence, in order to generate distributions with varying skewness, we implement the benchmark proposed by Thearling et al. [39], which uses the Shannon entropy as a measure of data distribution. Data is generated by repeatedly applying the *bitwise AND* operation to uniform random distributions, which increasingly skews the distribution towards keys with fewer bits set. For 32-bit keys, for instance, an entropy of 32 bits corresponds to a uniform distribution with each single bit of a key having a 50% probability of being set. Repeatedly *AND*ing random keys with such a uniform distribution once, twice, or three times, generates distributions with entropies of 25.96, 17.39, and 10.79 bits, respectively. In order to compare the end-to-end performance to the numbers that have been reported for PARADIS, we also ran experiments with a Zipfian distribution [14, 8].

6.1 On-GPU Sorting

We have evaluated the sorting performance for key distributions with varying degrees of skewness, starting from a uniform distribution (32-bit and 64-bit entropy) up to all keys having the same value (zero-bit entropy). Comparing the sorting rates for 32-bit keys (see Figure 6a), the hybrid radix sort shows an improvement of no less than a 1.69-fold speed-up over CUB. Compared to Thrust's radix sort (Thrust), Baxter's merge sort (MGPU), and the radix sort proposed by Satish et al. (Satish et al.), the results show a minimum speed-up of 1.89, 3.96, and 3.66, respectively. Being able to save one sorting pass by finishing early with a local sort, the hybrid radix sort achieves its peak performance for a uniform distribution with more than a two-fold speed-up over CUB, sorting 500 million keys in only 62.6 milliseconds. As shown in Figure 6c, the effect of the local sort becomes even more apparent for 64-bit keys. Sorting a uniformly distributed input of two gigabytes in as little as 66.7 milliseconds, for instance, almost matches the hybrid radix sort's processing duration for 32-bit keys. In contrast, CUB requires roughly twice as many sorting passes for 64-bit keys as for 32-bit keys and therefore sees a 49% performance drop. Starting out with a 3.75-fold speed-up over CUB for



(a) 32-bit keys

(b) 32-bit keys with 32-bit values

(c) 64-bit keys

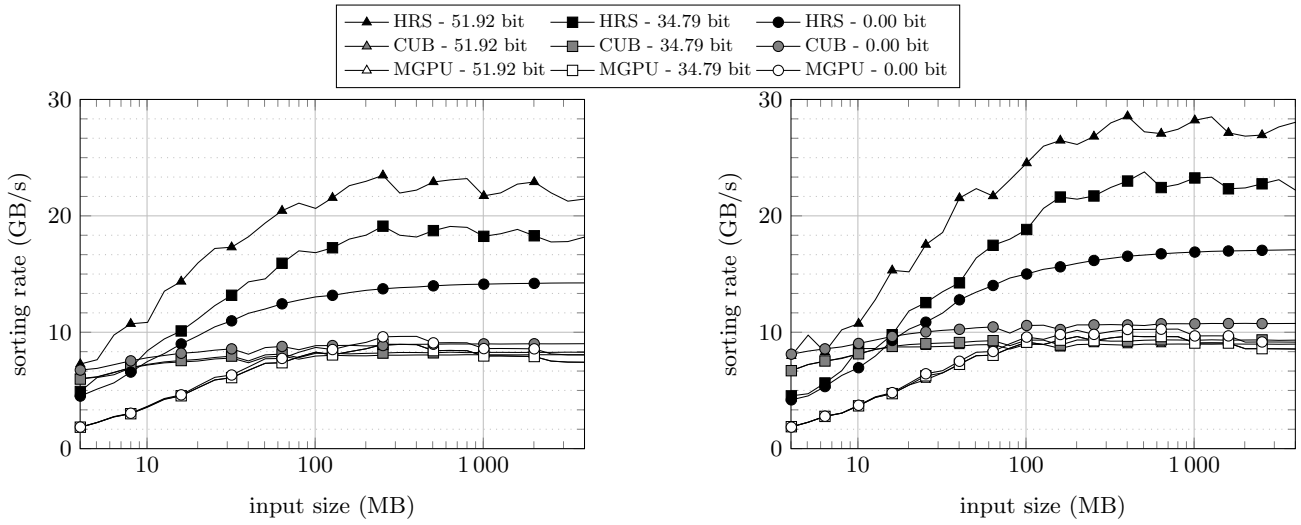
(d) 64-bit keys with 64-bit values

Figure 6: Performance for sorting a 2 GB input with varying data skewness on the GPU

uniformly distributed 64-bit keys, the performance surplus due to the local sort declines for increasingly skewed distributions, flattening out for a distribution with an entropy of zero bits. For such a distribution, all keys have to be run through all counting sort passes. Hence, the performance gain over CUB boils down to the reduced number of counting sort passes and the lower amount of memory transfers. Given keys and key-value pairs that comprise 64-bit keys, an achieved speed-up of the hybrid radix sort with a factor of 1.58 over CUB for such a distribution is in line with the improvements we expect from our 1.625-fold reduction in the amount of memory transfers (13 versus eight sorting passes). Similarly, the 1.7-fold speed-up seen for 32-bit keys closely matches the 1.75-fold improvement over CUB we anticipated as a result of reducing from seven to only four sorting passes. This illustrates that the proposed hybrid radix sort is able to efficiently mitigate the downsides of considering more bits with each sorting pass, achieving more than 97% of the expected theoretical speed-up.

Comparing the hybrid radix sort's performance for sorting key-value pairs to the performance shown for sorting keys only, we see a 20% increase in the amount of data being sorted per second, which matches the reduced amount of memory transfers. Since half the input consists of keys, the hybrid radix sort is reading only half the input during the histogram computation, while still reading and writing the whole input once during the scattering phase. For a total of reading and writing the input only 2.5 times instead of three times, we end up with a 1.2-fold lower amount of memory transfers, which directly translates to a 20% performance increase. This culminates in a sorting rate of up to 40.2 GB/s for 32-bit keys with an associated 32-bit value and up to 35.7 GB/s for 64-bit keys with 64-bit values (see Figure 6b and Figure 6d). Compared to CUB, this corresponds to a 2.32-fold and a four-fold improvement for 32-bit/32-bit key-value pairs and 64-bit/64-bit key-value pairs, respectively.

We also analysed the sorting performance for inputs ranging from 250 000 to 500 million elements with key distributions of varying skewness, i.e., considering an entropy of 64.00, 51.92, 34.79, 21.59, 12.84, 7.43, 4.22, 2.36, 1.31, 0.72,



(a) 64-bit keys

(b) 64-bit keys with 64-bit values

Figure 7: Comparison of the hybrid radix sort (HRS), the CUB radix sort (CUB), and merge sort (MGPU) for different distributions with an entropy of 51.92, 34.79, and 0.00 bits

0.39, and 0.00 bits. Being able to save several sorting passes for a uniform key distribution, the hybrid radix sort outperforms CUB for all of the evaluated input sizes. Yet, incurring a slightly lower constant overhead, CUB has an edge for very small and highly skewed inputs that are sorted in the order of hundreds of microseconds (see Figure 7a and Figure 7b). Considering the hybrid radix sort's worst-case key distribution, however, the hybrid radix sort still outperforms CUB for inputs larger than 1.9 million keys and 1.6 million key-value pairs, independently of the key distribution. Given that the input size is a function parameter, we could easily default to CUB's sorting algorithm using a simple case distinction for small inputs that fall short of these thresholds. Compared to Thrust and the GPU-based merge sort (MGPU), our hybrid radix sort is superior for any of the evaluated problem sizes. For reasons of clarity, however, we decided to only present the performance results gathered from the merge sort implementation.

6.2 Heterogeneous Sorting

This section analyses the end-to-end sorting performance of the pipelined heterogeneous sorting algorithm and compares it to the numbers reported for the CPU-based radix sort PARADIS [8].

Figure 8 compares the heterogeneous sort to a naïve approach that simply transfers the input to the GPU (*PCIe HtD*), sorts the input there (*on-GPU sorting*), and returns the sorted result over the PCIe bus (*PCIe DtH*). The naïve approach was evaluated for two variants. Firstly, using the state-of-the-art radix sort for the on-GPU sorting (CUB), and secondly, using the hybrid radix sort (HRS). We analysed the performance of the heterogeneous sort for several choices of s (the number of chunks). The figure shows the processing duration of the heterogeneous sort broken down into the *chunked sort* and the *CPU merging*. The *chunked sort* represents the time taken for splitting the input into s chunks, transferring the chunks to the GPU, sorting them on the GPU, and returning the sorted runs over the PCIe bus. The time taken for merging s sorted chunks on a six-core CPU is depicted by *CPU merging*. Figure 8 shows that, as the number of chunks increases, the time taken by the chunked sort is approaching the time taken for transferring the input one single time over the PCIe bus (cf. Section 5). For $s = 16$ chunks, for instance, the time of the chunked sort is down to 629 milliseconds, which corresponds to a mere 16% more time than it takes to transfer the whole input to the GPU one single time (540 milliseconds). Noticeably, the chunked sort even outperforms the on-GPU sorting time of CUB (636 milliseconds), even though the chunked sort includes the PCIe data transfers to the GPU and back. While we see the performance of the chunked sort improving for a larger number of chunks, our parallel multiway merge lacks the compute power to efficiently merge more than four chunks at a time. For our six-core CPU, we therefore see

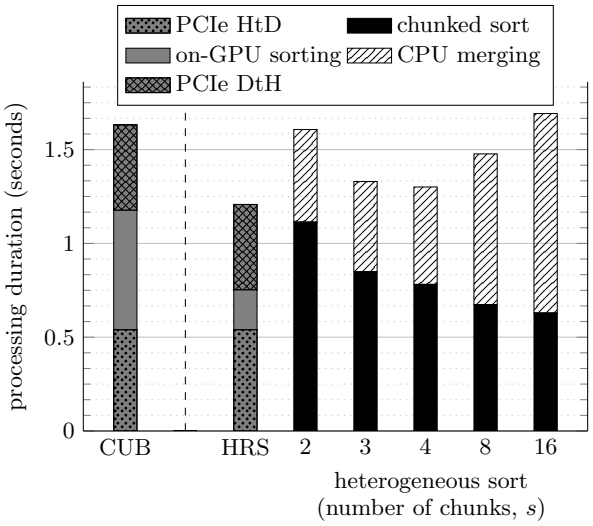
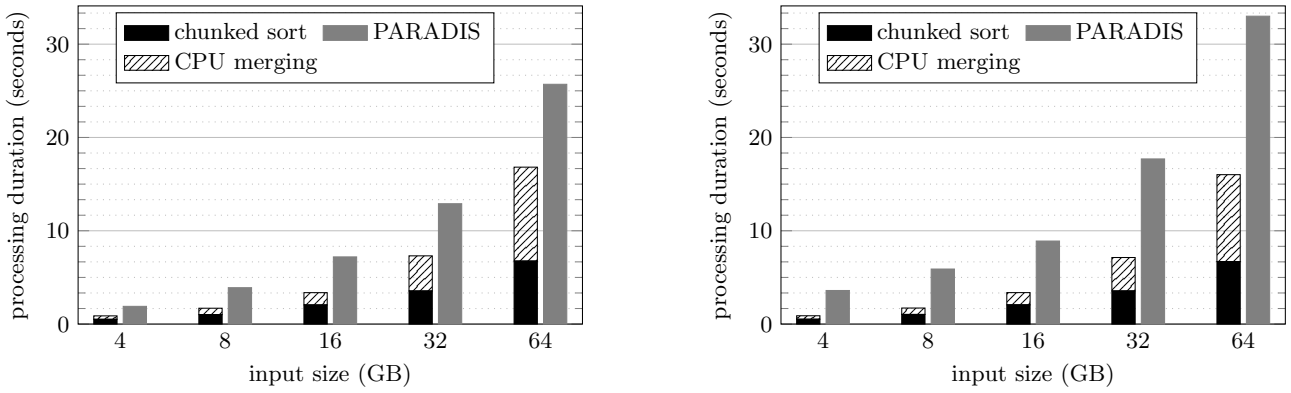


Figure 8: Comparing the end-to-end time for sorting 375 million 64-bit keys with 64-bit values (6 GB)



(a) uniform distribution

(b) skewed distribution (zipf, $\theta = 0.75$)

Figure 9: Comparing the end-to-end sorting performance of the heterogeneous sorting algorithm to the state-of-the-art CPU-based radix sort (PARADIS) for inputs comprising 64-bit keys with 64-bit values

a minimum for the overall end-to-end sorting time for four chunks. While these performance numbers are representative for our system, using our merge-based approach, a more powerful host system will see a lower minimum for a higher number of s, given that it efficiently merges eight, 16, or even more chunks at a time. Similarly, a more efficient multiway merge implementation or an approach building on partitioning rather than merging may also move the optimum towards a higher number of chunks.

Figure 9a and Figure 9b compare the performance of the heterogeneous sort to the numbers reported for PARADIS running 16 threads on a system with 32 CPU cores [8]. For a skewed distribution, our heterogeneous sorting algorithm achieves a four-fold speed-up, sorting four gigabytes in 895 milliseconds. Even though we see our CPU-based parallel multiway merge slightly degrading the overall performance for larger inputs, the heterogeneous sort still shows more than a two-fold speed-up over PARADIS for an input of 64 GB. While the GPU completes sorting and returning all sorted runs after only 6.7 seconds, it takes the parallel multiway merge on a six-core CPU another 9.3 seconds to merge the sorted runs. Compared to PARADIS, which suffers from skewed distributions, the performance of our approach is almost distribution agnostic, varying by no more than 5% between the uniform and the Zipfian distribution. PARADIS, running 32 threads, takes 19.8 and 25.4 seconds for an input of 64 GB with a uniform and a skewed key distribution, respectively. Even though the heterogeneous sort is only running on a six-core CPU, these results are still up by a factor of 1.18 and 1.59 from the time taken by the heterogeneous sort for a uniform and a skewed distribution, respectively.

7. CONCLUSIONS

This work presented a novel approach to radix sorting on GPUs. Instead of building on the common LSD radix sort approach for GPUs that relies on stable sorting passes, we took a different route with our efficient implementation of an MSD radix sort. Proceeding from the most-significant to the least-significant digit allows our algorithm to drop the requirement of stable sorting passes. By lifting this constraint, we were able to substantially reduce the number of required sorting passes and the amount of memory transfers. For the memory bandwidth-bound radix sort, we achieve a baseline of a 1.6-fold reduction in the amount of memory transfers, which directly translates to an achieved minimum speed-up of a factor of 1.58. This shows that our approach is successfully addressing the challenges arising from implementing an MSD radix sort on GPUs, such as load balancing issues for skewed distributions and performance degradation due to bucket handling, while still being able to max out the high memory bandwidth of GPUs. Moreover, sorting small buckets in on-chip memory rather than running them through subsequent partitioning passes enables additional performance improvements, culminating in a four-fold speed-up over the state-of-the-art approach.

In addition, we presented a heterogeneous sorting algorithm that uses the CPU on powerful host systems to mitigate the overhead introduced with PCIe data transfers and sort arbitrarily large inputs. Using pipelining, we were able to exploit the full-duplex communication of the PCIe bus, while interleaving the process of sorting and data transfers. Transferring an input to the GPU, sorting it into runs of up to four gigabytes each, and returning the sorted runs is now almost as fast (i.e., 9.55 GB/s) as transferring the input in one direction, one single time over the PCIe bus (i.e., 12 GB/s). Comparing the end-to-end sorting performance of our heterogeneous sort (including the time taken for merging the runs on a six core CPU) to the numbers reported for PARADIS using 16 threads on a 32 core system, we see a 2.2-fold and a four-fold speed-up for an input of four gigabytes with a uniform and a Zipfian distribution, respectively. Even though being bound by the merging performance of the CPU for larger inputs, like 64 GB, we still see an improvement of a factor of 1.52 and 2.07 for a uniform and a Zipfian distribution, respectively.

8. ACKNOWLEDGMENTS

This research has been supported by the Alexander von Humboldt Foundation. We would also like to thank Saman Ashkiani and the other authors of GPU Multisplit for sharing their implementation with us.

9. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 2012.

[2] S. Ashkiani, A. A. Davidson, U. Meyer, and J. D. Owens. GPU Multisplit. *CoRR*, abs/1701.01189, January 2017.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 2013.

[4] S. Baxter. Modern GPU. https://github.com/moderngpu/moderngpu, 2016.

[5] B. Chandramouli and J. Goldstein. Patience is a virtue: Revisiting merge and sort on modern processors. SIGMOD, 2014.

[6] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 2008.

[7] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 2015.

[8] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri. Paradis: An efficient parallel algorithm for in-place radix sort. *PVLDB*, 2015.

[9] Cisco visual networking index: Global mobile data traffic forecast update, 2015-2020 white paper. Technical report, Cisco, 2016.

[10] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. InPar 2012, 2012.

[11] F. Dehne and H. Zaboli. Deterministic sample sort for GPUs. *Parallel Processing Letters*, 2012.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. SIGMOD, 2006.

[13] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 2006.

[14] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *ACM SIGMOD Record*, 1994.

[15] O. Green, R. McColl, and D. A. Bader. Gpu merge path: a gpu merging algorithm. ICS 2012, 2012.

[16] L. Ha, J. Krüger, and C. T. Silva. Fast four-way parallel radix sorting on gpus. *Computer Graphics Forum*, 2009.

[17] M. Harris, S. Sengupta, and J. D. Owens. Gpu gems 3. *Parallel Prefix Sum (Scan) with CUDA*, 2007.

[18] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. SC '07, 2007.

[19] M. Herf. Radix tricks. http://stereopsis.com/radix.html, 2001.

[20] J. Hoberock and N. Bell. Thrust: A parallel template library. https://thrust.github.io, 2016.

[21] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. PACT '07, 2007.

[22] H. Inoue and K. Taura. Simd- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 2015.

[23] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2009.

[24] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster. SIGMOD, 2012.

[25] P. Kipfer and R. Westermann. Improved gpu sorting. *GPU gems*, 2:733–746, 2005.

[26] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner. Applicability of gpu computing for efficient merge in in-memory databases. In *ADMS@ VLDB*, 2011.

[27] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. IPDPS, 2010.

[28] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 2011.

[29] D. Merrill and NVIDIA Corporation. CUB. https://github.com/NVlabs/cub, 2016.

[30] M. Najafi, M. Sadoghi, and H. A. Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. ICDE, 2015.

[31] NVIDIA GeForce GTX 980. Whitepaper. Technical report, NVIDIA, 2014.

[32] NVIDIA Tesla P100. Whitepaper. Technical report, NVIDIA, 2016.

[33] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. SIGMOD, 2014.

[34] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. IPDPS, 2009.

[35] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. SIGMOD, 2010.

[36] A. Shahvarani and H.-A. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. SIGMOD, 2016.

[37] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 2008.

[38] I. Tanasic, L. Vilanova, M. Jordà, J. Cabezas, I. Gelado, N. Navarro, and W.-m. Hwu. Comparison based sorting for systems with multiple gpus. GPGPU, 2013.

[39] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. SC '92, 1992.

[40] J. Wassenberg and P. Sanders. *Engineering a Multi-core Radix Sort*. Euro-Par 2011. 2011.

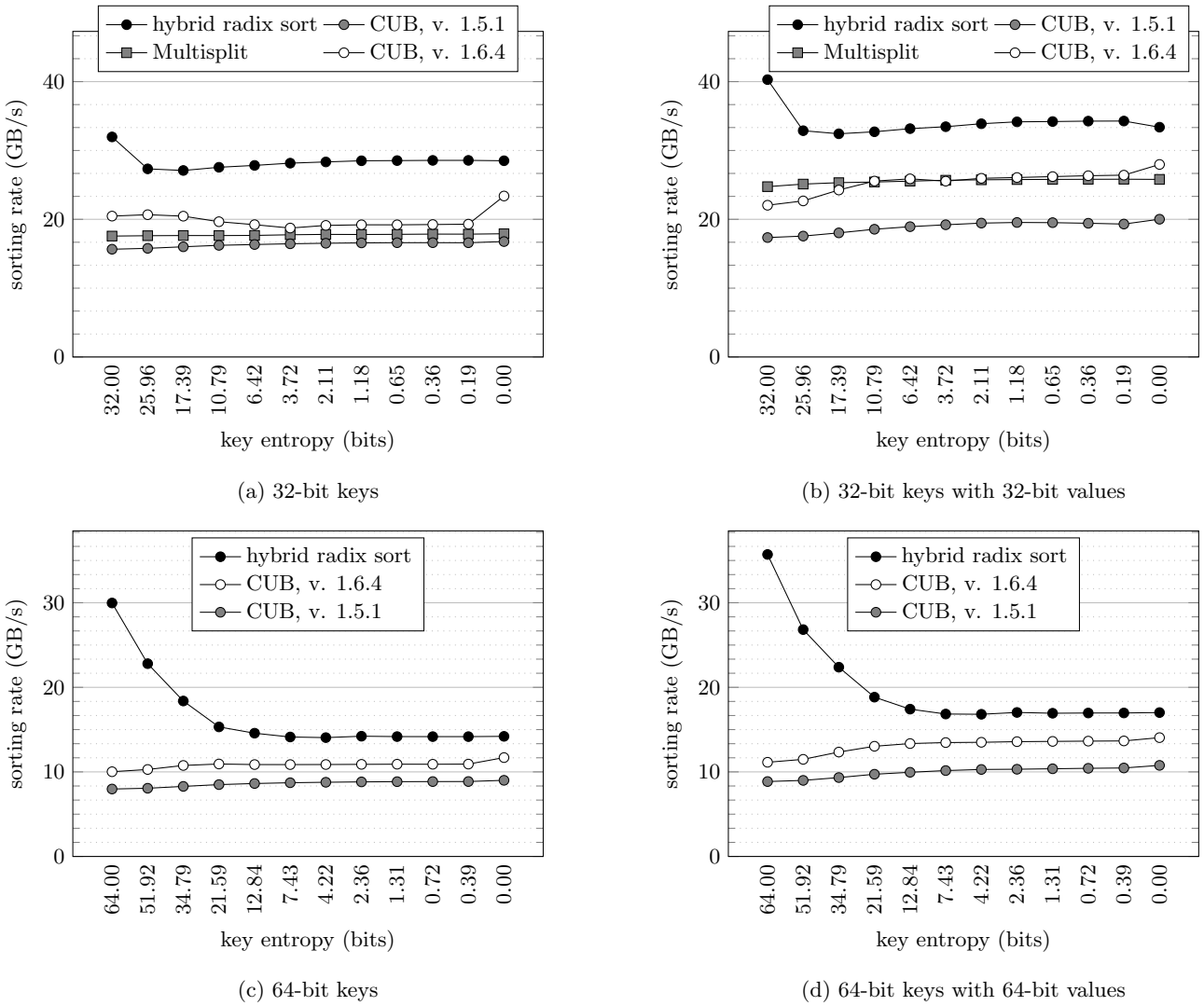
[41] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core gpus. IPDPS, 2010.

APPENDIX

A. ADDENDUM ON THE LATEST WORK

As a fundamental operation that finds its application in many fields, GPU-based sorting algorithms receive a lot of attention. Given the strong interest in efficient sorting algorithms, available implementations are continuously improved and new approaches are regularly published. With this addendum we aim to meet the rapid advancements that are made in this field, covering up to date work, which followed our initial submission and the completion of the peer review process, with preliminary and non-exhaustive results that we were able to obtain just in time with the authors' support. In particular, that is the work of Ashkiani et al., who present an improved version of their multisplit primitive (GPU Multisplit) that can be used for the partitioning passes of a radix sort as well as an update of the CUB library, which in version *1.6.4* enables specific GPU architectures to support up to seven bits per sorting pass [2, 29]. While CUB is maxing out shared memory at the cost of lower occupancy, GPU Multisplit makes use of the warp-synchronous execution and warp-wide intrinsics for the efficient data exchange between threads of the same warp to mitigate excessive on-chip memory requirements.

Figure 10 shows a performance comparison of the hybrid radix sort and the two latest approaches, putting their advancements into context by adding the prior state-of-the-art baseline (CUB, version 1.5.1) to the evaluation. For sorting 32-bit keys, the hybrid radix sort still achieves as much as a 56% improvement over CUB's latest version. For any non-constant distribution, it retains a minimum improvement of 32% over CUB (version 1.6.4), with an edge of 21% for a constant distribution (0 bits entropy). For 32-bit keys, GPU Multisplit is superior to CUB (version 1.5.1), yet, inferior to CUB (version 1.6.4). The hybrid radix sort outperforms GPU Multisplit by no less than a factor of 1.53 for 32-bit keys (see Figure 10a). As shown in Figure 10b, GPU Multisplit and CUB in its latest version are roughly on a par for sorting key-value pairs (32-bit keys with 32-bit values). While GPU Multisplit has an edge over CUB of up to 12%



(a) 32-bit keys

(b) 32-bit keys with 32-bit values

(c) 64-bit keys

(d) 64-bit keys with 64-bit values

Figure 10: Performance for sorting a 2 GB input with varying data skewness on the GPU

for more uniform distributions, CUB (version 1.6.4) has an edge of up to 8% for highly skewed distributions. Compared to GPU Multisplit, the hybrid radix sort achieves as much as a 1.62-fold improvement, with a minimum speed-up of 1.29. Compared to CUB's latest version, the hybrid radix sort shows an improvement of up to 82% and no less than 28% for any non-constant distribution. Similarly, the hybrid radix sort provides a minimum speed-up over CUB (version 1.6.4) of 1.29 for 64-bit keys over any non-constant distribution, showing as much as a 2.99-fold improvement for a uniform distribution (see Figure 10c). For key-value pairs (64-bit keys with 64-bit values), the hybrid radix sort outperforms CUB (version 1.6.4) by a factor of 3.21 for a uniform distribution, while still showing no less than a 21% improvement for any of the remaining distributions (see Figure 10d).

B. IMPACT OF OPTIMIZATIONS

While building on an MSD-based hybrid radix sort enables the performance improvements with considerable speed-ups in the first place, it also makes the algorithm highly sensitive to the input distribution. To ensure that the algorithm provides relatively constant performance results, even for challenging input distributions that are highly skewed or that would otherwise require handling millions and millions of buckets, this work has developed several optimisations. In order to show the impact of individual optimisations, we rerun our experiments with single optimisations being switched off. For our evaluation, we distinguish between independent optimisations that are analysed by disabling them individually and a group of synergistic optimisations. The performance impact of disabling a combination of independent optimisations can easily be approximated by multiplying the relative performance impact of the individual optimisations. Disabling a combination of optimisations within the group of synergistic optimisations (i.e., *single local sort config* and *no bucket merging*), in contrast, may have a more drastic effect than their multiplicative performance impact, since the lack of one optimisation may boost the impact of the absence of the other optimisation. Therefore, in addition to switching off individual optimisations within the group, we also evaluated the performance impact for disabling the combination of synergistic optimisations.

For the group of synergistic optimisations, our analysis

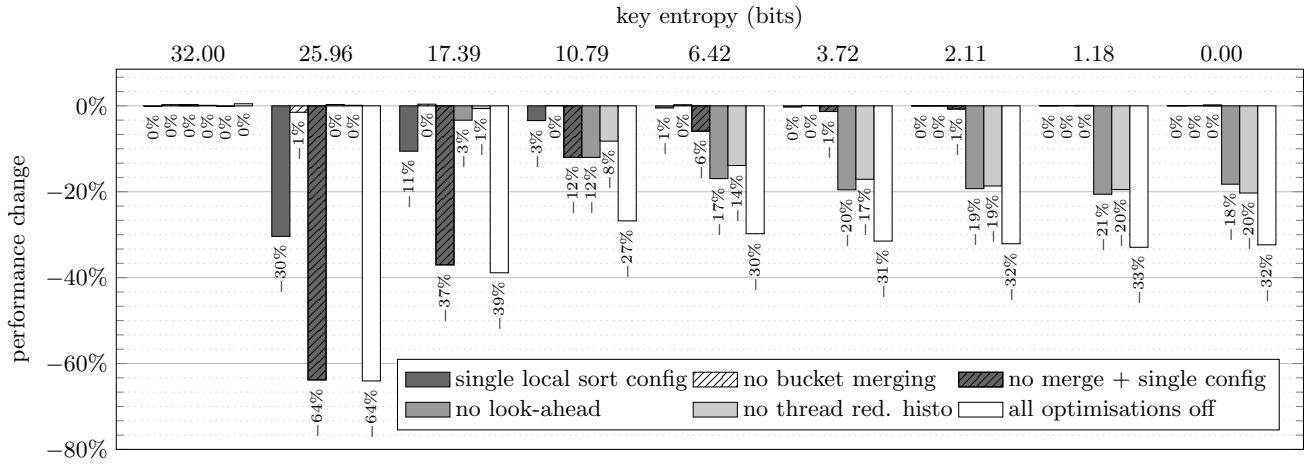


Figure 11: Performance impact on the sorting rate of 32-bit keys, when switching off individual optimisations

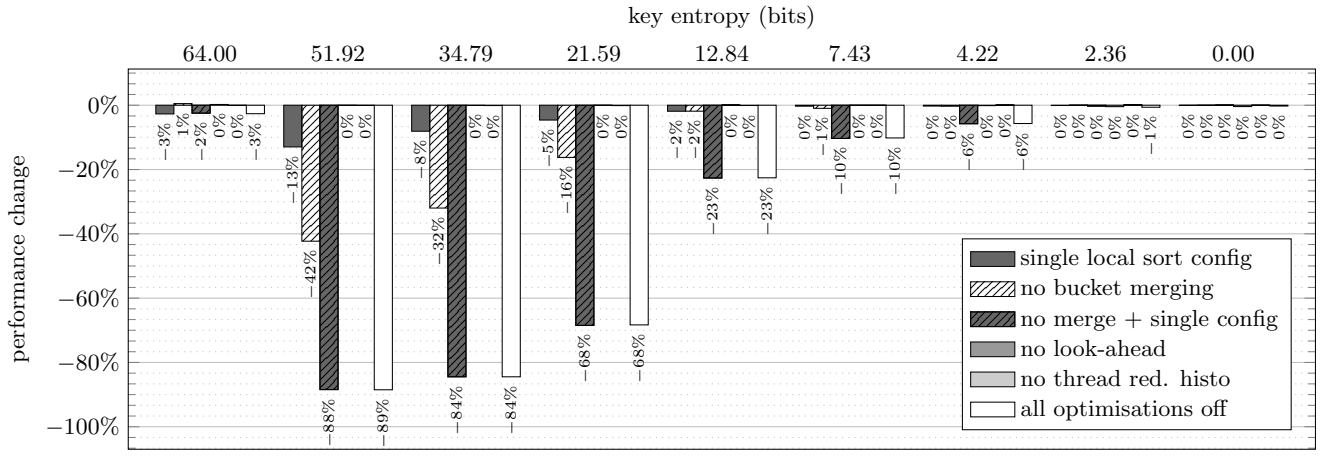


Figure 12: Performance impact on the sorting rate of 64-bit keys, when switching off individual optimisations

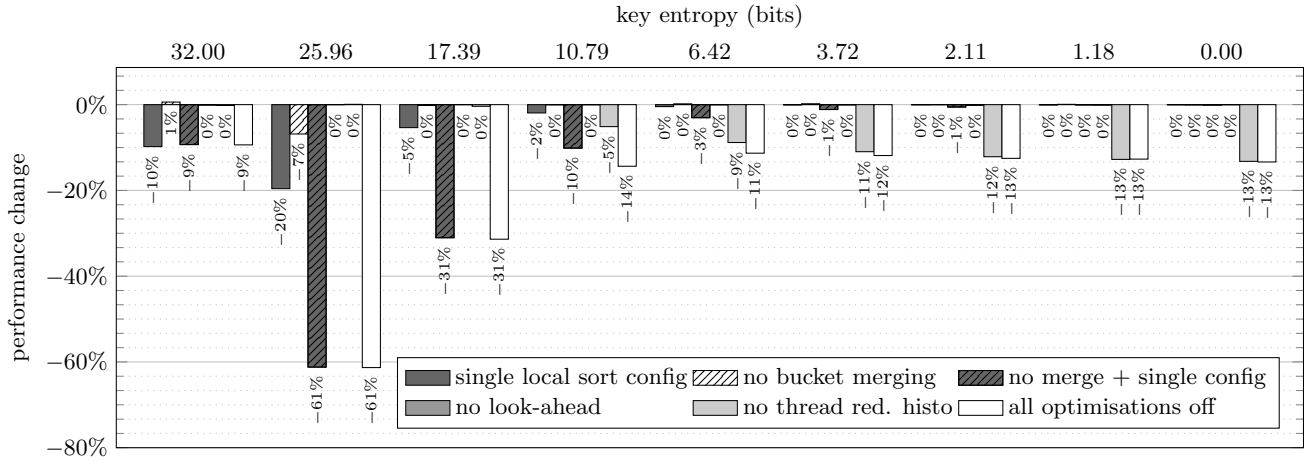


Figure 13: Performance impact on the sorting rate of 32-bit keys with 32-bit values, when switching off individual optimisations

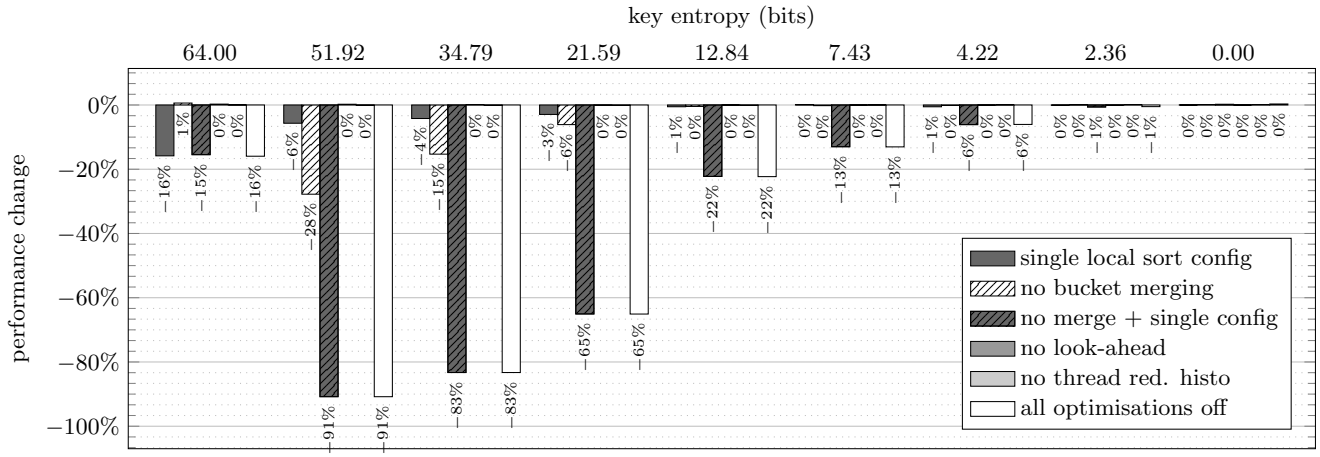


Figure 14: Performance impact on the sorting rate of 64-bit keys with 64-bit values, when switching off individual optimisations

considers using only a single local sort configuration (*single local sort config*) that sorts any bucket of up to $\hat{\partial}$ keys, not merging tiny buckets (*no bucket merging*), as well as the combination of both (*no merge + single config*). Amongst the independent optimisations, we considered not using the look-ahead during the scattering step (*no look-ahead*) and not using the thread reductions during the histogram computation (*no thread red. histo*).

Figure 11 shows the performance impact of switching off individual optimisations when sorting 32-bit keys. The performance impact is depicted as a performance delta, with the percentage denoting the performance increase or drop, after switching off an optimisation, compared to the performance achieved with all optimisations in place. Similarly, Figure 12, Figure 13, and Figure 14, depict the same information for sorting 64-bit keys, 32-bit keys with 32-bit values, and 64-bit keys with 64-bit values, respectively, showing the performance impact of individual optimisations.

Appendix B

ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data

ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data

Elias Stehle
Technical University of Munich (TUM)
stehle@in.tum.de

Hans-Arno Jacobsen
Technical University of Munich (TUM)
jacobsen@in.tum.de

ABSTRACT

Parsing is essential for a wide range of use cases, such as stream processing, bulk loading, and in-situ querying of raw data. Yet, the compute-intense step often constitutes a major bottleneck in the data ingestion pipeline, since parsing of inputs that require more involved parsing rules is challenging to parallelise. This work proposes a massively parallel algorithm for parsing delimiter-separated data formats on GPUs. Other than the state-of-the-art, the proposed approach does not require an initial sequential pass over the input to determine a thread's parsing context. That is, how a thread, beginning somewhere in the middle of the input, should interpret a certain symbol (e.g., whether to interpret a comma as a delimiter or as part of a larger string enclosed in double-quotes). Instead of tailoring the approach to a single format, we are able to perform a massively parallel finite state machine (FSM) simulation, which is more flexible and powerful, supporting more expressive parsing rules with general applicability. Achieving a parsing rate of as much as 14.2 GB/s, our experimental evaluation on a GPU with 3 584 cores shows that the presented approach is able to scale to thousands of cores and beyond. With an end-to-end streaming approach, we are able to exploit the full-duplex capabilities of the PCIe bus and hide latency from data transfers. Considering the end-to-end performance, the algorithm parses 4.8 GB in as little as 0.44 seconds, including data transfers.

PVLDB Reference Format:
Elias Stehle, and Hans-Arno Jacobsen. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *PVLDB*, 13(5): 616-628, 2020.
DOI: https://doi.org/10.14778/3377369.3377372

1. INTRODUCTION

Massive amounts of data from a wide range of data sources are made available using delimiter-separated formats, such as comma-separated values (CSV), and various log file formats like the Common Log Format and the Extended Log Format [37, 30, 19]. The relevancy of the CSV format, for instance, is highlighted by the plethora of public datasets, some in excess of hundreds of gigabytes in size, that are provided using the CSV format [43, 24]. Log files are another origin of data in a delimiter-separated format that constitute an important source for many analytical workloads. For instance, *Sumo Logic*, a cloud-based log management and analytics service, recently announced that it analyses more than 100 petabytes of data and 500 trillion records daily [42]. With an ever increasing amount of data, there is also a growing need to provide and maintain rapid access to data in delimiter-separated formats. This is also emphasised by ongoing research on in-situ processing of raw data and similar efforts that aim to lower the time to insight [11, 35, 6, 18, 25, 46, 7, 12, 26, 22, 2, 21].

While systems face an ever increasing amount of data that needs to be ingested and analysed, processors are seeing only moderate improvements in sequential processing performance. In order to continue the trend of providing exponentially growing computational throughput, manufacturers have therefore progressively turned towards scaling the number of cores as well as extending single instruction, multiple data (SIMD) capabilities. Graphics Processing Units (GPUs), which have focused on parallelism ever since, now integrate as much as 5 120 cores on a single chip [1]. Further, CPUs comprising multiple *chiplets*, as well as research focusing on package-level integration of multiple GPU modules, give an indication that hardware parallelism moves even beyond a single chip, scaling to multiple inherently parallel *chiplets* and GPU modules, respectively, on a package [5].

In order to leverage the current degree of hardware parallelism and benefit from the ongoing trend of an ever growing number of cores, algorithms have to be designed for massive scalability from the ground up [23]. Parsing, as a fundamental and compute-intense step in the data ingestion pipeline is no exception to this.

Parallel parsing of non-trivial delimiter-separated data formats, however, poses a great challenge, as symbols have to be interpreted differently, depending on the context they appear in. For the CSV format, for instance, RFC 4180 specifies that delimiters (i.e., commas and line breaks), which appear within a field that is enclosed in double-quotes, have to be interpreted as part of the field, instead of being interpreted as actual field or record delimiters [37]. In addition, many formats use a symbol to indicate comments or directives (e.g., `'#'`), following which, all symbols until the end of line have to be interpreted differently, yet again. Since the context depends on all symbols preceding the symbol

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377372

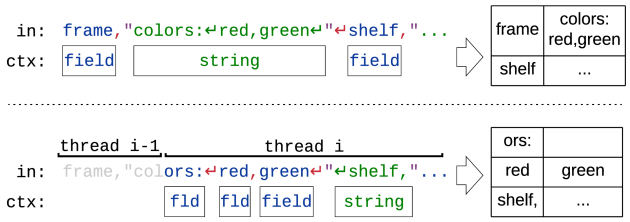


Figure 1: Challenges for parallel parsing: lacking context leads to misinterpretation

currently being interpreted, it is impossible for a thread to simply begin parsing somewhere in the middle of the input. Hence, the input cannot simply be split into multiple chunks that are processed independently.

This is exemplified in Figure 1, where *thread i* begins parsing in the middle of the input. The thread is not aware of the double-quote preceding its chunk that indicates the beginning of a larger string, changing the parsing context. As a result, the thread misinterprets subsequent commas and line breaks as delimiters, while they were actually supposed to be considered as part of the field's string. A similar challenge arises for determining the records and columns that a chunk of the input belongs to, which, again, depends on all the input preceding the current chunk being interpreted. Finally, threads have to coordinate and possibly collaborate in order to assemble field values that span multiple threads. This may also involve converting symbols to a binary type (e.g., `int`, `float`).

Previous work on parallel loading of delimiter-separated data formats has addressed the challenge of determining a thread's parsing context by either performing an initial sequential pass over the input or by completely dropping support for inputs with different parsing contexts, such as inputs containing enclosing symbols (e.g., double-quotes), comments, or directives [33, 3]. Another alternative is to tailor the approach to one specific format and exploit the format-specific characteristics, which, however, limits the approach's flexibility and applicability [29, 28, 39, 36, 16]. One such exploit for a simple CSV format, for instance, is to count the number of double-quotes, inferring the beginning and end of enclosed strings depending on whether the count is odd or even, respectively. More recently, Ge et al. presented an approach for distributed CSV parsing [16]. They aim to circumvent an initial sequential pass by exploiting CSV-specific characteristics to speculate about the parsing context. While such a tailored approach works well with CSV as long as it strictly complies with the format expected by the algorithm, it requires designing a completely new approach from the ground up once the input format deviates. Parsing other delimiter-separated formats, such as log files and their multifaceted formats, poses a challenge for an approach that is tailored to CSVs. Another important characteristic of state-of-the-art approaches is that they are designed for coarse-grained parallelism of distributed and multi-core systems, which renders them infeasible for the fine-grained parallelism required by GPUs [33, 16].

While constraining the input limits the applicability and flexibility, performing a sequential pass over the input contributes a substantial portion of sequential work that limits its scalability and, following Amdahl's law, precludes any speed-ups beyond a certain point. Given the ongoing trend of increasing hardware parallelism on the one hand and the diversity of data sources that today's OLAP systems are confronted with on the other hand, addressing these shortcomings is a viable endeavour.

We present `ParPaRaw`, an algorithm for massively **par**allel **pa**rsing of delimiter-separated **raw** data on GPUs that overcomes these scalability issues without compromising applicability or constraining supported input formats. `ParPaRaw` is designed from the ground up to scale linearly with the number of cores, providing robust performance despite the huge diversity of inputs it is confronted with, by employing a data parallel approach with fine-grained parallelism. `ParPaRaw` is designed to leverage the specifics of GPUs. It enables parallelism even beyond the granularity of a single record and ensures load balancing by splitting the input into small chunks of equal size that threads can process independently. Since using a data parallel approach raises the aforementioned challenges, we present an efficient solution for correctly identifying the parsing context of a thread's chunk, as well as its records and columns. In order to provide a flexible approach that is applicable to a wide range of inputs, we allow specifying the parsing rules in the form of a deterministic finite automaton (DFA). In order to exploit the full-duplex capabilities of the Peripheral Component Interconnect Express (PCIe) bus and lower the end-to-end latency, we present a streaming approach, which parses data on the GPU, while simultaneously transferring raw data to, and parsed data from the GPU.

With a generally applicable approach that does not impose constraints on the input, we are able to parse as much as 14.2 GB/s on the GPU. For end-to-end workloads, including data transfers via the PCIe bus, `ParPaRaw` parses 4.8 GB from the yelp reviews dataset in as little as 0.44 seconds.
In summary, the contributions of this paper are four-fold.

1. We present an approach to massively parallel parsing of delimiter-separated data formats that is designed for scalability without sacrificing applicability and flexibility. The approach develops a scalable, data parallel algorithm that addresses three challenges: a) determining a thread's parsing context without requiring a prior sequential pass, b) determining the records and columns that a thread's symbols belong to, and c) efficiently coordinating threads to collaboratively generate field values.
2. We address the major challenges that arise when mapping our algorithm to GPUs, which provide only very limited addressable on-chip memory (tens of KB) and, due to their limited register file size, require lightweight threads with only very limited context.
3. We show how to exploit the full-duplex capabilities of the PCIe bus with a streaming extension. This lowers the end-to-end latency and allows parsing data on the GPU, while simultaneously transferring raw data to, and returning parsed data from the GPU.
4. Our experimental evaluation highlights that, given today's level of hardware parallelism, it is worth to design algorithms for scalability from the ground up, even if it implies a significant increase in the overall work being performed.

This paper is organised as follows. Section 2 gives an overview of related approaches. Section 3 presents the algorithm, its building blocks, and the processing steps. Section 4 introduces optimisations, extensions, and implementation details. Section 5 evaluates the presented approach.

2. RELATED WORK

Even though parsing is fundamental for in-situ processing of raw files and constitutes a major bottleneck in the data ingestion pipeline, there is only limited work on accelerating the process. This is also highlighted by Dziedzic et al., who show that modern Database Management Systems (DBMSs) are unable to saturate available I/O bandwidth [14]. Using a variety of hardware configurations and datasets, Dziedzic et al. provide an extensive analysis of the data loading process for multiple state-of-the-art DBMSs [14]. Their evaluation reveals that data loading is CPU-bound [14].

A notable advancement for parsing delimiter-separated formats is made by Mühlbauer et al. who present improvements along two lines [33]. On the one hand, they introduce optimisations to reduce the number of control flow branches by utilising SIMD instructions for the identification of delimiters. On the other hand, they present an approach for parallel parsing. Their approach splits the input into multiple chunks of equal size that are processed in parallel. Threads start parsing their chunk only from an actual record boundary onward, i.e., the first record delimiter in their chunk. Threads continue parsing beyond their chunk until encountering the end of their last record. This ensures that threads always process complete records, yet makes the approach sensible to the chosen chunk size and the input's record sizes. For instance, the majority of threads, which work on a record that spans multiple chunks, unsuccessfully search for the beginning of their first record, without performing actual parsing work. Another shortcoming is that threads are not aware of the actual parsing context of their chunk. That is, whether to interpret a field or record delimiter as an actual delimiter or as part of a field's value. To address this, they introduce a *safe mode* for formats that may contain more involved parsing rules. In *safe mode* a sequential pass over the input is performed, which keeps track of the parsing context, such as quotation scopes, splitting chunks only at actual record delimiters. *Safe mode*, however, introduces a considerable portion of serial work, which, according to Amdahl's law, precludes any speedup beyond a certain point. By exploiting CSV-specific characteristics, Ge et al., who look at distributed CSV parsing, are able to bypass that initial pass by speculating about the parsing context [16]. Similar to the approach of Mühlbauer et al., Ge et al. require coarse-grained parallelism (distributed parsing), as threads begin parsing only from a chunk's first record boundary onward. As a result their approach is also sensible to the chosen chunk size and the input's record sizes. The authors highlight that the performance of their approach degrades with decreasing chunk sizes, once a chunk approaches the size of a record [16]. Moreover, both solutions do not provide parallelism beyond the granularity of an individual record, which makes them susceptible to load-balancing issues, particularly for small chunks and large, varying record sizes. These circumstances render the two approaches infeasible for the fine-grained parallelism required by GPUs [33, 16].

Apart from work addressing delimiter-separated formats, multiple approaches tailored to processing JSON have been proposed. Li et al. present *Mison*, a JSON parser that supports projection and filter pushdown by speculatively predicting logical locations of queried fields based on previously seen patterns [29]. Mison deviates from the classic approach of using an FSM while parsing, which allows it to use SIMD vectorisation to identify structural characters, such as double-quotes, braces, and colons. Whenever a structural character is encountered, its occurrence is recorded in the respective bitmap index (e.g., the double-quotes bitmap index). The beginning and end of a string enclosed in double-quotes can be inferred from looking at the odd and even number of set bits, respectively. While this enables SIMD vectorisation and avoids branch divergence, circumventing the use of an FSM and, hence, tailoring the approach specifically to the JSON format, limits the approach's applicability to formats with more involved parsing rules. Bonetta et al. introduce *FAD.js*, a runtime system for processing JSON objects that is based on speculative JIT compilation and selective access to data [9]. Palkar et al. propose a technique referred to as *raw filtering*, which is applied on the data's raw bytestream before parsing [35]. Langdale et al. recently introduced *simdjson*, a standard-compliant, highly-efficient JSON parser that makes use of SIMD instructions. Similar to *Mison*, they focus on the JSON format, which allows them to avoid the use of an FSM while parsing [28].

The parallel prefix scan is a fundamental algorithm of `ParPaRaw` and a frequently recurring building block for data parallel algorithms. Over the years many approaches for a parallel prefix scan have been proposed [31, 44, 17, 32, 13, 8, 20, 10, 27, 40]. For a given binary reduction operator (e.g., addition), it takes an array of input elements and returns an array, where the i-th output element is computed by applying the reduction operator to all input elements up to and including the i-th element [31]: $y_i = \bigoplus_{k=0}^{i} x_k$

A prefix scan that excludes the i-th input element is called exclusive prefix scan. The prefix scan using addition is called prefix sum. The following table shows an example of the inclusive and exclusive prefix sum, respectively:

x_i	3	5	1	2	9	7	4	2
y_i (incl.)	3	8	9	11	20	27	31	33
y_i (excl.)	0	3	8	9	11	20	27	31

It is worth noting that all efficient parallel approaches require the binary operator to be associative. The prefix scan used in `ParPaRaw` builds on the more recent work from Merrill et al., who propose a single-pass prefix scan [31]. Using the parallel prefix scan has also been considered for parallel parsing of regular languages. In fact, the theory for parallel parsing dates back as long as four decades. In particular, Fischer presents an algorithm that instantiates one FSM for each state defined by the FSM [15]. Hillis et al. illustrate the use of the parallel prefix scan computation by presenting an algorithm that is similar to earlier work from Fischer [20]. Even though the theory dates back several decades, it became feasible only more recently with modern hardware, i.e., GPUs with thousands of cores, that would set off the cost of running multiple FSM instances. We believe that this is the main reason why, to the best of our knowledge, this idea has not been considered for identifying the parsing context when parsing delimiter-separated formats on GPUs. Our approach reconsiders the idea behind the work from Fischer and Hillis et al. to address the sub-problem of identifying the parsing context. For identifying the parsing context, we devise a solution that respects the characteristics of GPUs. We show that, given todays degree of hardware parallelism, the cost of running multiple FSM instances can be set off with our efficient and scalable solution.

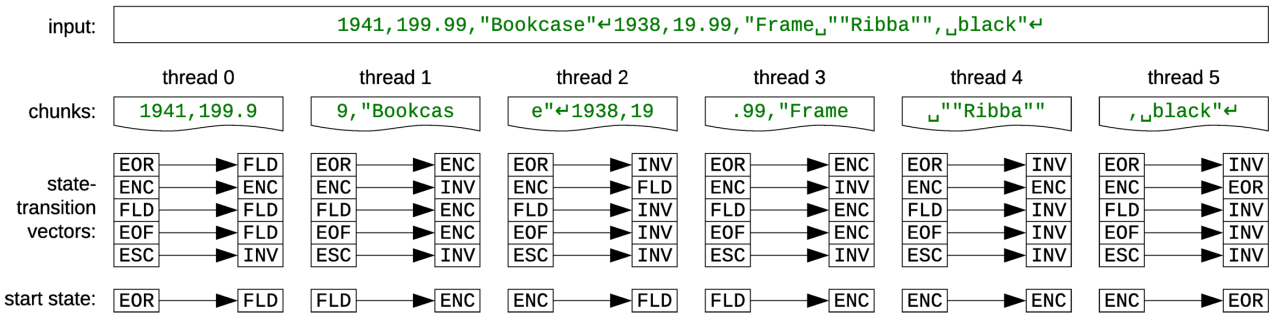


Figure 2: Determining the parsing context

3. MASSIVELY PARALLEL PARSING

In order to achieve scalability, even beyond thousands of cores, we pursue a data parallel approach, which splits the input into multiple chunks (e.g., 32 bytes per chunk) that can be processed independently by the threads. While a data parallel approach allows for massive scalability, there are three key challenges to overcome:

1. Determining the parsing context of a thread's chunk. That is, how a thread is supposed to interpret the symbols within its chunk (Section 3.1).
2. Determining the records and columns that the symbols of a thread's chunk belong to (Section 3.2).
3. Efficient coordination and collaboration between threads to transform a sequence of symbols to the data type of the respective column, e.g., `float` (Section 3.3).

`ParPaRaw` addresses these challenges in multiple steps. With each step, `ParPaRaw` gains additional information about each thread's chunk. This information is captured in meta data that subsequent steps can build on.

3.1 Parsing

The first step addresses the challenge of identifying the parsing context of a thread's chunk, allowing a thread to meaningfully interpret its symbols. That is, distinguishing whether a symbol is a control symbol (e.g., delimiting a field or a record) or whether it is part of a field's value.

It is important to note that without constraining the supported input formats and therefore sacrificing the approach's applicability, it is impossible to determine a thread's parsing context without considering all symbols preceding its chunk. However, if a thread is supposed to consider all symbols preceding its chunk, the approach has to either perform an initial sequential pass over the input or wait for all threads working on preceding chunks to finish. Considering all symbols preceding a thread's chunk introduces severe implications on the approach's scalability.

`ParPaRaw`, however, aims to neither constrain the input nor to introduce sequential work. In order to achieve this, we exploit the fact that there are only few different contexts to consider while parsing. While this increases the overall effort by a constant factor, it enables a fully concurrent approach and allows to scale linearly with the number of cores.

In pursuit of a flexible approach that is generally applicable, `ParPaRaw` uses a DFA while parsing. The current parsing context is represented by the DFA's state. While a thread iterates over its symbols, it transitions the states of its DFA according to its transition table. One example of a DFA for parsing a simple CSV format is shown in Figure 3 (for simplicity it omits the invalid state (`INV`) used to track invalid formats, e.g., reading quotes in `FLD` state). A sequential approach would simply set the starting state of its DFA and read the symbols of the input beginning to end, always being aware of the current state when reading a symbol. For a data parallel approach, however, a thread, starting to parse somewhere in the middle of the input, cannot simply infer the state it is supposed to start in.

In order to perform meaningful work despite lacking the correct starting state, each thread instantiates one DFA for every state, $s_i \in S$, defined by the DFA, setting the starting state of the i-th DFA-instance to state s_i (see *state-transition vectors* in Figure 2). For reasons of clarity, in the following we assume $s_i = i$, i.e., representing a state by its index, to avoid the intricate differentiation between a state and a state index. An efficient implementation uses the same mechanism during preprocessing to ensure efficient lookups into data structures like the transition table. While the thread is reading the symbols of its chunk, it transitions the states of all its DFA-instances accordingly. Once all the symbols of a chunk have been read, the final state of each DFA-instance is noted in a state-transition vector. We maintain one state-transition vector per thread, with each state-transition vector holding $|S|$ elements. The i-th entry of the state-transition vector represents the final state of the i-th DFA-instance (i.e., the DFA-instance that has originally started in state s_i). Hence, the algorithm can infer that if a thread had started parsing in state s_i, it would end up in the state given by the i-th entry of that thread's state-transition vector after the thread has read all its symbols (see Figure 2).

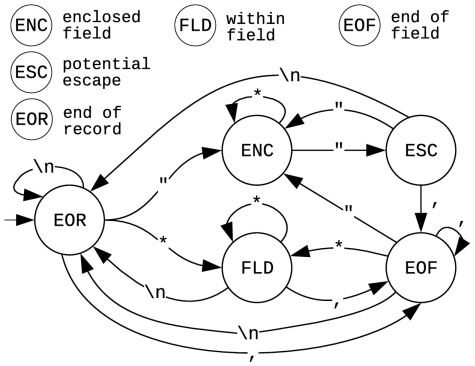


Figure 3: Example for a simple DFA parsing CSVs

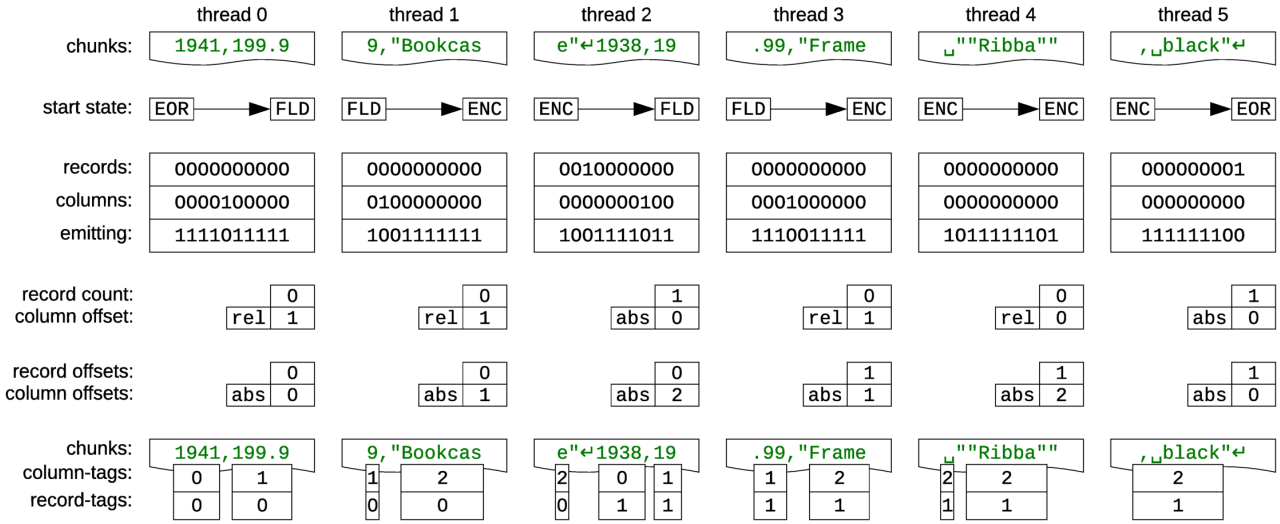


Figure 4: Identifying columns and records

By computing the composite of these state-transition vectors, the algorithm can deduce the starting state for every thread. We define the composite operation $a \circ b$ of two state-transition vectors a and b as:

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{|S|-1} \end{bmatrix} \circ \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{|S|-1} \end{bmatrix} = \begin{bmatrix} b_{a_0} \\ b_{a_1} \\ \vdots \\ b_{a_{|S|-1}} \end{bmatrix}$$

Since the composite operation is associative, the algorithm can compute a parallel exclusive scan using the composite operation, which is seeded with the identity vector. After the exclusive scan, the i-th entry of each thread's resulting vector now corresponds to the state that the thread's DFA is supposed to start in, if the sequential DFA's starting state was s_i. For instance, if the sequential DFA's starting state was s_3, each thread finds its starting state by reading the element at index three from its resulting vector.

Once each thread is aware of its starting state, threads can correctly interpret the symbols from their chunk by simulating a single DFA-instance. While iterating over its symbols, a thread identifies field delimiters, record delimiters, and other control symbols (e.g., an escape sequence) according to the parsing rules specific to the current format being parsed. Since the algorithm addresses delimiter-separated formats, the relevant meta data for each symbol can be represented using three bitmap indexes: one marking symbols that are delimiting a record, one flagging symbols that are delimiting a field, and one indicating whether a symbol is a control symbol (e.g., escape symbol) or whether it is part of the field. Subsequent steps can build on these bitmap indexes without requiring to repeatedly simulate the DFA-instance.

3.2 Identifying Columns and Records

The bitmap indexes from the previous step are used to identify the column and record offset. That is, the record and column that the first few symbols of a thread's chunk belong to, until it encounters the first delimiting symbol. Determining the column and record offsets requires two steps.

First, each thread computes the offset that its chunk adds to the preceding chunk's offset. For the records, the relative offset can easily be computed by counting the records (i.e., the number of set bits of a thread's record delimiter bitmap index using `POPCNT`). For columns, however, this is slightly more involved. If a thread encounters a record delimiter and therefore the beginning of a new record, it can infer the **absolute** column offset for the subsequent chunk (e.g., *thread 2* in Figure 4). Otherwise, all it can infer is that it has seen k field delimiters and, therefore, the next chunk's column offset has an additional offset of k, **relative** to the preceding chunk's column offset. In Figure 4, for instance, *thread 1* encounters one column delimiter but no record delimiter. As *thread 1* is not yet aware of its own column offset, it can only infer that the subsequent thread's column offset increases by one, relative to its own column offset. We distinguish between an absolute and a relative column offset, which are denoted as `abs` and `rel`, respectively, in Figure 4. A column offset is absolute, if there is at least one set bit in the thread's record delimiter bitmap index. The column offset can be computed by zeroing all bits of the column delimiter bitmap index that precede the last set bit in the record delimiter bitmap index, counting the remaining set bits, i.e.: `POPCNT( (~BLSMSK(rec_bidx)) & col_bidx )`

In a subsequent step, the algorithm computes the exclusive prefix sum over the record counts, which yields each thread's record offset. In order to retrieve the column offsets, we perform an exclusive scan using the following operation, where, for a column offset x, x_t denotes whether a column offset is relative (`rel`) or absolute (`abs`) and x_o denotes the offset value:

$$\begin{bmatrix} a_t \\ a_o \end{bmatrix} \oplus \begin{bmatrix} b_t \\ b_o \end{bmatrix} = \begin{cases} \begin{bmatrix} b_t \\ b_o \end{bmatrix} & \text{if } b_t \text{ is } \texttt{abs} \\ \begin{bmatrix} a_t \\ a_o + b_o \end{bmatrix} & \text{if } b_t \text{ is } \texttt{rel} \end{cases}$$

Once all absolute column and record offsets have been calculated, threads can correctly identify the column and record that each of its symbols belongs to. In preparation for the next step, which transforms the row-oriented input to a columnar format and, if applicable, converts strings of symbols to the data type of the corresponding column, we tag the symbols with the column and record they belong to, as illustrated at the bottom of Figure 4.

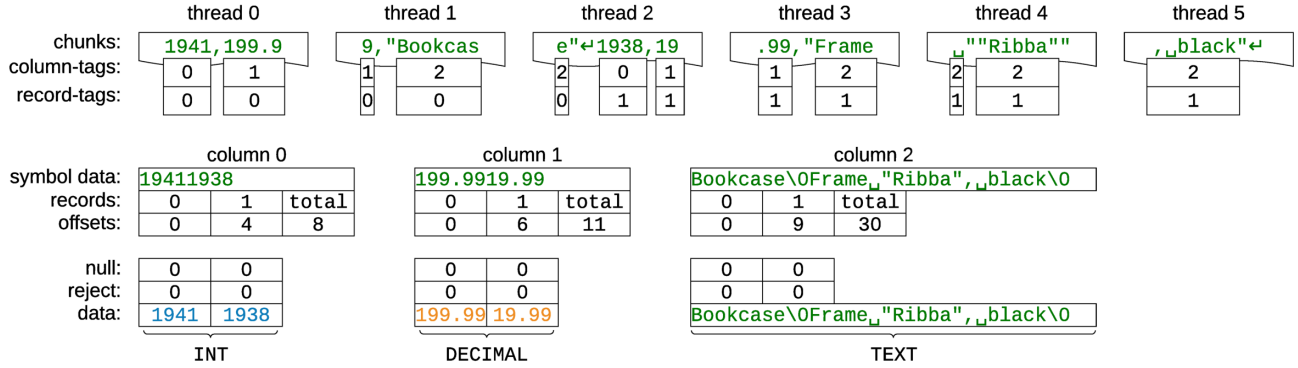


Figure 5: Preparing data in a columnar format and type conversion

3.3 Columnar Format & Type Conversion

Now, that each thread is fully aware of the associated columns and records, threads still need to generate the individual field values in a columnar format. Depending on the column that a string of symbols belongs to, this may require converting to the respective column's type (e.g., `int`, `float`). As shown at the top of Figure 5, symbols belonging to the same field may still span multiple chunks, requiring involved threads to collaborate on generating a single field value. To circumvent the collaboration between threads entirely, one option would be to change the assignment of threads, assigning exactly one thread to exclusively process all symbols required for generating a single field value. This approach, however, may cause considerable load-balancing issues, as the number of symbols per field may be subject to high variance. In particular, values of columns with variable-width types, such as text or Binary Large Objects (BLOBs), may be arbitrarily large.

Another challenge arises due to the fact that symbols are still in a row-oriented format. That is, two threads working on subsequent chunks may be parsing two different columns of two different types. Hence, they may require different rules, executing completely different code segments for parsing the fields' values. For instance, one thread may require generating an integer value, while the next one is extracting a date. The behaviour of different threads executing different code paths is particularly punishing on GPUs, where all threads within a warp (e.g., a group of 32 threads) are executing the same instruction in lockstep.

`ParPaRaw` addresses these challenges by first partitioning all symbols by the column they are associated with. During partitioning, `ParPaRaw` ensures that symbols within a column maintain their order by using a stable radix sort that uses the symbols' column-tags as the sort-key. While sorting, the symbols and the record-tags are moved along with the associated sort-key. The radix sort iterates over the bits of the column-tags, performing a stable partitioning pass on the sequence of bits considered with a given pass. A single partitioning pass involves (1) computing the histogram over the number of items that belong to each partition, (2) computing the exclusive prefix sum over the histogram's counts, and (3) scattering the items to the respective partition.

After partitioning, all symbols belonging to the same column lie cohesively in memory. We refer to all symbols belonging to the same column as the concatenated symbol string (CSS) of a column. The histogram that is maintained while sorting is used to identify the offsets of the columns' CSSs. Similar to the symbols, all the symbols' record-tags lie cohesively in memory, indicating which record a symbol belongs to.

Having all symbols in a columnar format allows the algorithm to efficiently process each of the columns. This may include type inference, validation, identifying `NULL`s, and converting symbol strings to the column's type. First, `ParPaRaw` uses the record-tags to generate an index into the CSS. The index is used to identify the offsets and lengths of the fields' symbol strings. To generate the index, the algorithm performs a run-length encoding on the symbols' record-tags, which yields each field's record and its number of symbols. Computing the exclusive prefix sum over the fields' symbol counts yields the offsets into the CSS, as shown in Figure 5. The symbol count of a field can be inferred using the difference of the successive field's offset and the field's own offset.

Building on the index, `ParPaRaw` can now start generating the fields' values by interpreting the strings of symbols, if that is required for a given column (e.g., numerical or temporal types). In order to address possible load-balancing issues due to having high variance in the number of symbols per field, we use three different collaboration levels: thread-exclusive, block-level, and device-level collaboration. By default, a thread tries to exclusively generate a field value, looking up the offset and number of its symbols in the index. Once the thread has identified the symbols, it starts converting the symbol string to the column's type (e.g., `int`, `float`). If, during lookup, a thread detects that its string of symbols exceeds a certain threshold, it will defer generating that field value for the block- or device-level collaboration. The threshold depends on the on-chip memory of a GPU's streaming multiprocessor and its number of cores. If there are fields left for the block-level collaboration, all threads of a thread-block (e.g., 64 threads) collaborate on generating a field value. Fields that exceed the on-chip memory available to a thread-block (typically in the order of tens of kilobytes) are addressed by the device-level collaboration. Block- and device-level collaboration use the same data parallel approach as the overall approach presented for parsing delimiter-separated inputs. Hence, the same technique for determining a thread's parsing context is employed.

4. EXTENSIONS & IMPLEMENTATION

Having presented the fundamental processing steps for a robust approach to massively parallel parsing in Section 3, this section focuses on optimisations, extensions, and implementation details. We develop two optimised specialisations that can be applied if a given input meets certain conditions (see Section 4.1). Section 4.2 addresses symbols crossing chunk boundaries, such as being encountered when dealing with variable-length encodings. To highlight that not only efficiency but also the approach's applicability was of great importance to this work, we present a few more capabilities in Section 4.3. With an end-to-end streaming extension, we aim to hide the latency of data transfers via the PCIe bus (see Section 4.4). Finally, Section 4.5 presents how we address the major challenges of mapping the algorithm to the GPU.

4.1 Alternative Tagging Modes

`ParPaRaw`, as presented in Section 3, focuses on robustness. It is even resilient to inputs that contain records with a varying number of field delimiters per record (e.g., "`1,Apples\n2\n`"). This section focuses on presenting two optimised specialisations that are chosen, if the input provides a constant number of columns per record or if the user prefers to reject records that have an inconsistent number of field delimiters.

Since many of the presented processing steps work at peak memory bandwidth, reading and writing record-tags of four bytes increases the amount of memory transfers and degrades performance. Hence, we aim to lower the amount of memory transfers by reducing the memory footprint of the record-tags. As illustrated in Figure 6, we provide two alternatives to record-tags.

The *inline-terminated CSS* replaces delimiters with a terminator during the tagging phase. Just like the null character for null-terminated strings, the terminator is a unique character that indicates the end of a field's symbols. Good candidates for terminators are various separators specified by the `ASCII` standard, such as the *record separator* (`0x1E`) or the *unit separator* (`0x1F`). To generate the CSS's index, the algorithm simply writes the offsets of all occurrences of the terminator symbols to the index. The *inline-terminated CSS* requires that the terminator is not part of a column's CSS, as those symbols would otherwise get confused for a terminator.

The *vector-delimited CSS* can address this scenario by devoting its own auxiliary boolean vector that delimits the fields within a column. The CSS's index is generated the same way as for the *inline-terminated CSS* with the minor difference that the algorithm identifies non-zero values in the auxiliary vector instead of terminators from the CSS.

4.2 Variable-Length Symbols

So far, we have not addressed the challenge of symbols crossing chunk boundaries. While this can be easily prevented for fixed-size symbols spanning multiple bytes by adjusting the chunk size to be an integer multiple of the symbol size, it is more involved for variable-length symbols. For instance, if inputs are encoded using a variable-length Unicode Transformation Format (UTF), such as UTF-8 or UTF-16, symbol boundaries become unpredictable and some symbols might be crossing chunks. If a symbol crosses chunk boundaries, the thread working on the chunk at which the symbol

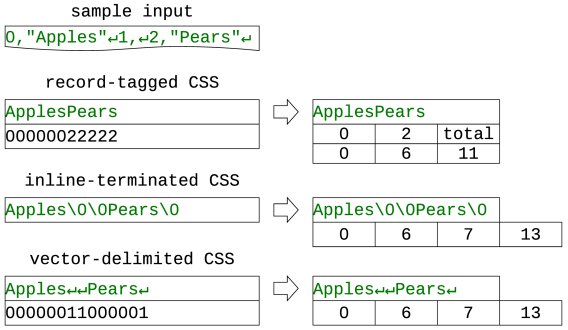


Figure 6: Alternative tagging modes

begins (i.e., the symbol's leading bytes) is in charge of reading that symbol and transitioning the state of its DFA accordingly. Threads working on subsequent chunks that only read trailing bytes of a symbol skip those bytes. For the variable-length encodings UTF-8 and UTF-16, threads can identify whether the first bytes of a chunk are only trailing bytes of an encoded code point (a code point is a numerical value and most code points are assigned a character). UTF-8 encodes code points using one, two, three, or four bytes. Unless a single byte is used, all trailing bytes have the common binary prefix of `0b10XX XXXX`. Hence, for UTF-8 encoded inputs, threads simply ignore a chunk's first few bytes with that binary prefix. UTF-16 uses either two bytes to encode code points ranging from `0x0000` to `0xD7FF` and from `0xE000` to `0xFFFF`, and four bytes for code points beyond `0x010000`. If four bytes are used, the two high order bytes, referred to as high surrogate, are in the range of `0xD800` to `0xD8FF`, and the low order bytes, referred to as low surrogate, are in the range of `0xDC00` to `0xDFFF`. Since unicode does not assign any characters in the range of `0xD800` to `0xDFFF`, there is no two-byte combination in that range. Hence, similar to UTF-8, a thread ignores a chunk's first two bytes if their value is in the range of `0xDC00` to `0xDFFF`.

4.3 Capabilities

This section focuses on pointing out a few more capabilities to highlight `ParPaRaw`'s applicability to real-world requirements.

Validating format — One notable strength of `ParPaRaw` is its ability to simulate an FSM while parsing, which makes it widely applicable and enables more expressive parsing rules. With the presented massively parallel approach for simulating a DFA, `ParPaRaw` is always aware of the DFA's current state when reading a symbol. Hence, invalid state transitions as well as a non-accepting end state can easily be detected.

Skipping records and selecting columns — `ParPaRaw` is able to ignore a user-specified set of records and columns. While tagging symbols with their associated column and record, all symbols that belong to records or columns that are supposed to be ignored are identified and marked as irrelevant. Irrelevant symbols can be ignored following the partitioning step.

Skipping rows – It is worth noting that rows are different from records, as some records may span multiple rows. Since ignoring rows may interfere with the assignment of symbols to columns and records, `ParPaRaw` has to ensure that rows are ignored early on. Hence, `ParPaRaw` ignores a set of rows

by performing an initial parallel pass over the input, pruning symbols of ignored rows (i.e., parallel stream compaction).

Inferring or validating number of columns — If no schema is provided and therefore the number of columns is not known a priori, `ParPaRaw` can infer the input's number of columns. Similarly, if `ParPaRaw` is supposed to reject records that do not conform to the expected number of columns the same technique is applied. In either case, during DFA simulation threads need to track three values in addition to the relative or absolute column offset handed over to the subsequent chunk. Firstly, every thread keeps track of the number of field delimiters encountered before reading its very first record delimiter, which subsequently is referred to as *relative min/max*. Further, every thread maintains the minimum and maximum number of columns it counted per record for all records following the chunk's first record delimiter. We use an extra bit to denote if no minimum and maximum was determined, i.e., the chunk does not contain any record delimiter. After the prefix scan of the column offsets, `ParPaRaw` can resolve the *relative min/max*, turning it into an absolute column offset. The absolute column offset is then incorporated in the respective chunk's minimum and maximum column count. A subsequent reduction over the maximum is then used to infer the number of columns. Comparing the identified minimum and maximum column counts indicates whether a given chunk conforms to the expected number of columns per record.

Default values for empty strings — If the input has a consistent number of field-delimiters per record, the default value for empty strings is set during type conversion. That is, when field values are parsed, the empty string is parsed as the column's default value. If the input does not have a consistent number of field-delimiters per record, the column's data is pre-initialised with the user-specified default value and later overwritten for non-empty fields.

Type inference — `ParPaRaw` is comparably efficient when identifying a column's type, as, prior to type conversion, all of a column's symbols lie cohesively in memory. During an initial pass over the column's symbols, threads identify the minimum numerical type being required to back their field value. A subsequent parallel reduction over the minimum type yields the inferred type of a column. `ParPaRaw` currently only considers type inference for numerical types, but can be extended to cover temporal types.

4.4 End-to-End Streaming

This section provides an extension to `ParPaRaw`'s on-GPU parsing algorithm presented in Section 3 to address inputs that do not reside on the GPU or exceed its available device memory. In order for the GPU to be able to process the input, the input first needs to be transferred via the comparably slow PCIe bus and, once processed, the parsed data has to be returned. It is worth noting that the PCIe bus allows for full-duplex communication, enabling simultaneous data transfers in either direction at peak bandwidth. While the PCIe bus does not necessarily limit the throughput, waiting for the data transfer to complete before and after parsing, respectively, adds a considerable amount of latency to the end-to-end processing time. Hence, rather than waiting for the input to arrive on the GPU, before the GPU begins processing it and, once finished, starts returning the parsed data, we make use of a streaming approach. The streaming approach splits the input into multiple partitions. Each partition, at some point, is transferred to the GPU, processed, and its data is returned. Having multiple partitions allows to overlap these stages for subsequent partitions, similar to the pipelined approach in [38, 41]. That is, transferring a partition, while processing its predecessor on the GPU and simultaneously returning parsed data via the interconnect.

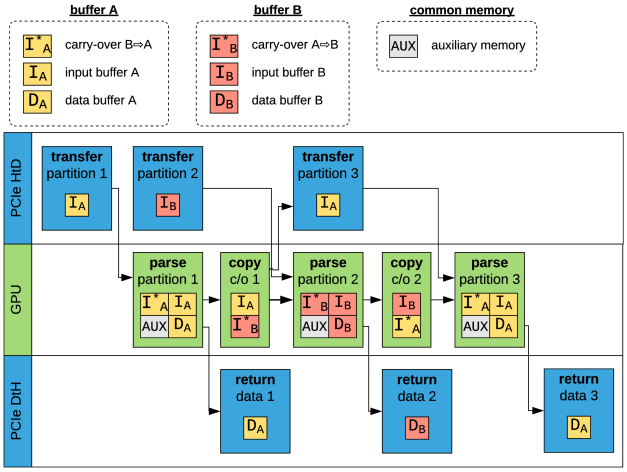
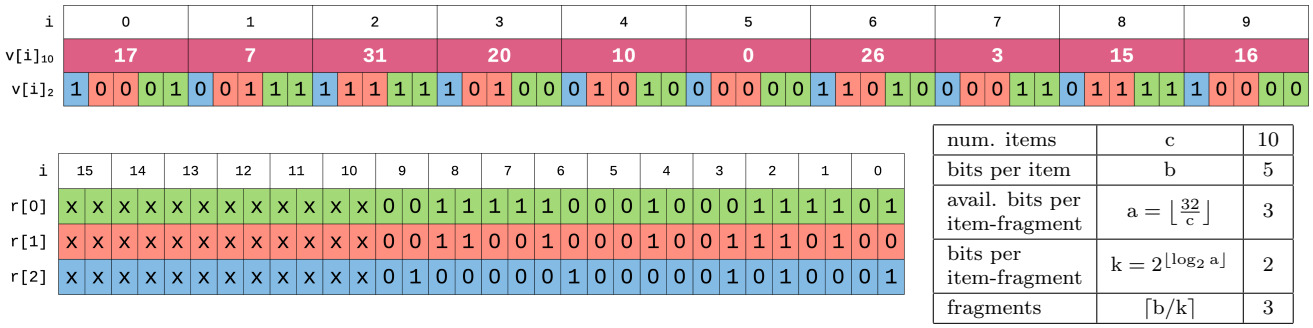


Figure 7: End-to-end streaming

For the end-to-end streaming approach, we allocate a double-buffer and some auxiliary memory on the GPU (see top of Figure 7). Each buffer comprises memory for the raw input and the parsed data. One buffer's raw input allocation is used as input for parsing on the GPU, while the opposing buffer's raw input allocation is receiving data of the next partition. Similarly, one buffer's data allocation is used to output parsed data, while data is being returned via the interconnect from the opposing buffer's data allocation. In addition, we prepend additional memory for a carry-over to the memory allocated for the input of each buffer. The carry-over is used for prepending the last, incomplete record at the end of one buffer's input to the opposing buffer's input.

Figure 7 exemplifies the processing steps of the streaming parsing approach. The stages of a partition are (1) *transfer*: transferring the raw input of a partition from the host to the GPU, (2) *parse*: parsing the input of a given partition, including the prepended carry-over and writing the parsed data to the data buffer, and (3) *return*: returning the parsed data from the data buffer to the host. The resources required by each processing step are illustrated by the rectangular symbols within a step (e.g., I_A representing the memory allocated for the input of buffer A). A processing step's dependency on a preceding processing step is depicted by an incoming edge. An important sequence depicted in Figure 7 is when the GPU switches work from one double-buffer to the other. For instance, after the GPU has finished parsing the input of the first partition (raw input provided by *input buffer A*), the last incomplete record is prepended to the second partition by copying it to the memory of the *carry-over* of *buffer B*. Since copying the carry-over is reading from *input buffer A*, the algorithm ensures that the transfer of the third partition to *input buffer A* does not take place before the carry-over has been copied, as the carry-over would otherwise get corrupted.



num. items	c	10
bits per item	b	5
avail. bits per item-fragment	$a = \lfloor \frac{32}{c} \rfloor$	3
bits per item-fragment	$k = 2^{\lfloor \log_2 a \rfloor}$	2
fragments	$\lceil b/k \rceil$	3

Figure 8: Logical and physical view of the multi-fragment in-register array

4.5 Implementation Details

This section addresses the main challenges faced when mapping `ParPaRaw` to the GPU. Specifically, we introduce a new data structure, referred to as multi-fragment in-register array (MFIRA), which provides a workaround for the constraint that threads cannot dynamically address into the register file. Since the register file is extremely fast and provides the most on-chip memory, addressing this shortcoming is a viable endeavour. The presented data structure allows to dynamically index into and access elements of a bounded array. MFIRA is particularly efficient for low-cardinality arrays of small integers. This is a recurring pattern in GPU programming, since the GPU's threads need to be very lightweight, allowing for only very limited context (i.e., using only few registers). Hence, even though MFIRA was designed as an efficient data structure backing various objects when parsing, MFIRA likely would be useful for other use cases as well. Further, we present a branchless algorithm that builds on SIMD within a register (SWAR) to identify the index of a read symbol in the transition table. With that approach, we are able to keep the symbols that the algorithm compares against in the very fast register file. At the same time, it avoids that threads within a warp are executing along different branches.

Multi-fragment in-register array — The idea behind the data structure is that even though thread registers themselves cannot be addressed dynamically, individual bits within a register can be. Specifically, we use the intrinsic functions bit-field insert (BFI) and bit-field extract (BFE), which require only two clock cycles on recent microarchitectures, to efficiently access an arbitrary sequence of bits from a register. We use these two functions to decompose an item that is written to the data structure and distribute the item's fragments (i.e., partitions of its bits) amongst one or more registers. Similarly, when an item is accessed, it is reassembled from its fragments. Figure 8 illustrates this principle, depicting an array containing up to ten items, each five bits wide. For such an array, the data structure could use up to three bits per fragment. To efficiently compute bit-offsets into a register, however, the number of bits actually being used by the data structure is chosen to be a power of two. This allows replacing the expensive integer multiplication with a bit-shift operation. In the example depicted in Figure 8, the data structure would therefore devote two bits per fragment, using a total of three fragments. The individual fragments of the items are colour-coded in Figure 8 to highlight how the logical view (top of the figure) maps to the physical view (bottom of the figure).

Table 1: Transition table example

symbols	groups	states					
		EOR	ENC	FLD	EOF	ESC	INV
\n	0	EOR	ENC	EOR	EOR	EOR	INV
"	1	ENC	ESC	INV	ENC	ENC	INV
,	2	EOF	ENC	EOF	EOF	EOF	INV
*	3	FLD	ENC	FLD	FLD	INV	INV

Symbol matching using SWAR — During DFA simulation, the algorithm uses a transition table to identify the state transition from the DFA's current state and a read symbol to the DFA's new state. The transition table is two-dimensional, with states along one and symbols along the other dimension. To compress the transition table's size, we collapse all the transition table's symbols that have identical state transitions into symbol groups. As illustrated in Table 1, we have one symbol group per row instead of having symbol groups as columns, which allows coalesced access to all state transitions of a read symbol. This is particularly useful when computing the state-transition vectors. A thread reads a symbol from its chunk, identifies its symbol group, and fetches the row of state transitions for the matched symbol group. For each of its DFA instances, it can then efficiently determine the new state from that row.

Having introduced symbol groups, mapping a symbol to its symbol group is an elementary step. To ensure an efficient mapping, we exploit the fact that delimiter-separated formats typically have only a few symbols to distinguish amongst, such as an escaping symbol, field and record delimiters, and enclosing symbols like quotes or brackets (see Table 1). Hence, for the symbols we use a comparison-based approach, rather than devoting a full lookup-table that maps each character value to its group. Since symbols are often only eight bits wide (e.g., `ASCII` and UTF-8-encoded `ASCII` characters), while GPUs implement 32-bit wide arithmetic instructions, we use a branchless SWAR algorithm to perform multiple comparisons at a time (see Table 2). On the one hand, this avoids inefficiencies due to threads executing divergent branches. On the other hand, with the following approach, we are more space-efficient and are able to keep the symbols in the very fast register file. As illustrated in Table 2, we place each of the symbols that we try to match against in the individual bytes of four-byte registers. We refer to these registers as lookup-registers (LU-registers). For later comparison against the LU-registers, whenever a symbol is read, we replicate that symbol in every byte of

Table 2: Identifying a symbol's index using SWAR

byte	7	6	5	4	3	2	1	0
symbol group			3	2	2	2	1	0
lookup (LU)				\t	—	,	"	\n
read symbol (s)	,	,	,	,	,	,	,	,
$c = LU$ XOR s	--	--	--	25	50	00	0E	26
$swar = H(c)$	--	--	--	00	00	80	00	00
bfind($swar$)>>3	1F	FF	FF	FF	00	00	00	02
$idx = \min_{\forall r_i}(x)$								0x00000002
$\min(idx, 5))$								0x00000002

$H(x) = ((x - \texttt{0x01010101}) \;\&\; (\sim x) \;\&\; \texttt{0x80808080})$

a separate register (i.e., the s-register). Computing the exclusive or for each of the LU-registers with the s-register yields a null-byte if the two bytes match. Subsequently applying the bit-twiddling hack to determine a null-byte, as suggested by Mycroft in 1987 [34], sets the most-significant bit for that byte (see definition of $H(x)$ in Table 2). Using the intrinsic function `bfind`, we retrieve the position of the most-significant set bit. If no bit was set, i.e., the read symbol does not match any byte from the LU-registers, `bfind` will return `0xFFFFFFFF`. To retrieve the matching index, we divide the value returned by `bfind` by eight, using bit-shift for efficiency reasons (i.e., shifting it three bits to the right). For LU-registers that contain no match, the matching index is `0x1FFFFFFF`, while for the ones that contain a match, it yields a value between zero and three. To ensure that we consider a match, if present, we compute the minimum over all matching indexes. Finally, in case there was no match, we map the matching index of `0x1FFFFFFF` to the `catch-all` symbol group by using the minimum function. The minimum is computed very efficiently, requiring only one or two cycles on recent microarchitectures and is therefore generally preferable to a conditional expression.

5. EXPERIMENTAL EVALUATION

For the experimental evaluation we use two systems, one to evaluate CPU-only implementations, referred to as *CPU system*, and one system equipped with a GPU (*GPU system*) used for evaluating GPU-based approaches. Both systems are running Ubuntu 18.04. The *CPU system* has four sockets, each equipped with a Xeon E5-4650 clocked at 2.70 GHz. It has a total of 512 GB of DRAM (DDR3-1600). The *GPU system* is equipped with 128 GB of DRAM (DDR4-2400) and a Xeon E5-1650 v4 processor with six physical cores, clocked at 3.60 GHz. The source code was compiled with the `-O3` flag. We used release 10.1.105 of the CUDA toolkit. The *GPU system* hosts an NVIDIA Titan X (Pascal) with 12 GB device memory, 3 584 cores, and a base clock of 1 417 MHz (driver version is 418.40.04).

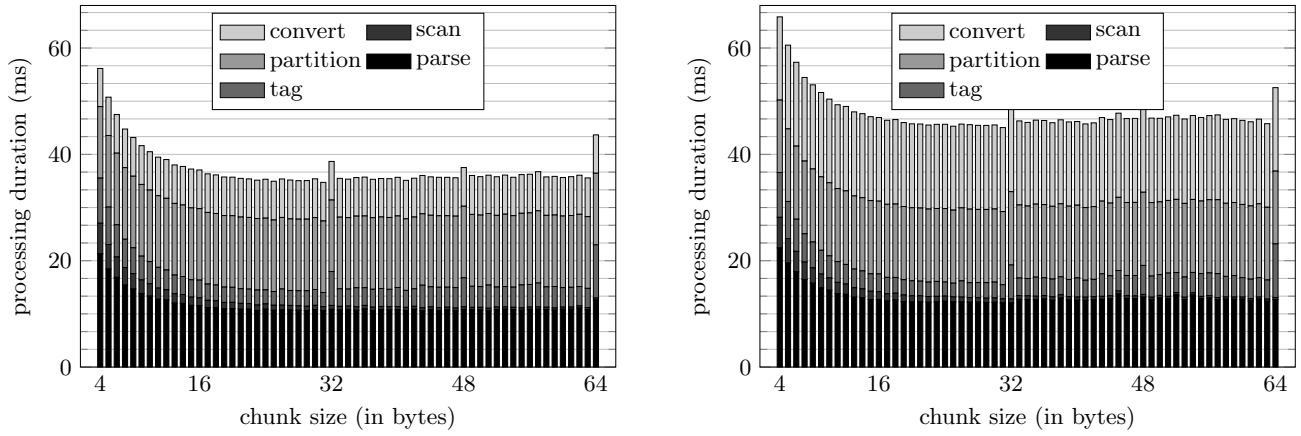
The output of `ParPaRaw` is configured to comply with the format specified by *Apache Arrow*. *Apache Arrow* specifies a columnar memory format for efficient analytic operations [4]. It is used by a multitude of well-known in-memory analytics projects, such as *OmniSci*, *pandas*, and *Apache Spark*. For `ParPaRaw` we use a DFA that is capable of parsing any RFC4180 compliant input [37]. The DFA defines six states, including one state to track invalid state transitions.

To evaluate the systems, we choose the two dissimilar real-world datasets *yelp reviews* and *NYC taxi trips*. The *yelp reviews* dataset comprises 6.69 million reviews from yelp's dataset as CSV, with all fields enclosed in double-quotes [45]. The dataset is 4.823 GB large with an average record size of 721.4 bytes per record. Each record is made up of nine columns, covering text-based, numerical, and temporal types. The dataset is of particular interest due to the text-based reviews that may include field and record delimiters, which poses a challenge for many parallel parsers.

The *NYC taxi trips* dataset is 9.073 GB large and comprises 102.8 million yellow taxi trips taken in the year 2018 provided by the *NYC Taxi & Limousine Commission* [43]. The dataset's 17 columns cover numerical and temporal datatypes. With an average of only 88.3 bytes per record and 5.2 bytes per field, the majority of the fields are very short and of a numerical type, putting the emphasis on data type conversion.

5.1 On-GPU Parsing

This section provides a detailed evaluation of the presented algorithm using on-GPU workloads. Our on-GPU evaluation focuses on identifying efficient configurations and analysing the algorithm's sensibility to input parameters. Time measurements for the on-GPU parsing experiments represent the GPU wall-clock time, measured using CUDA



(a) yelp reviews

(b) NYC taxi trips

Figure 9: Time spent on individual processing steps depending on the chunk size configuration

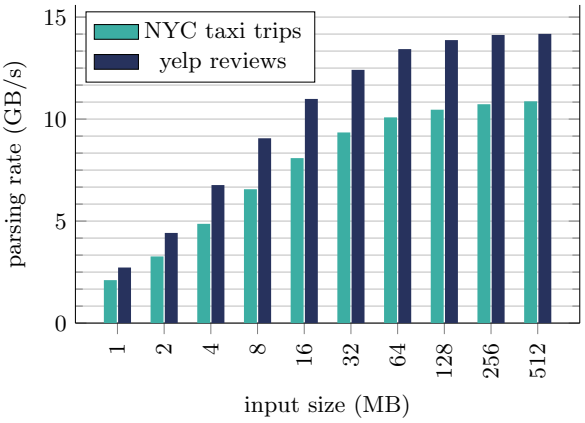


Figure 10: Parsing rate as a function of input size

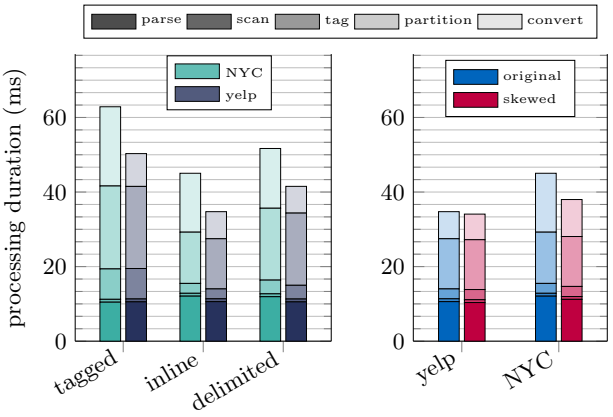


Figure 11: Time breakdown for different tagging modes (left) and skewed input (right)

events. Other than the end-to-end parsing experiments, on-GPU experiments do not include data transfers between host and device. Unless noted otherwise, we use the first 512 MB of each dataset for this evaluation, to be able to evaluate all tagging modes before running out of device memory.

We provide a breakdown of the time spent on the individual processing steps as a function of chunk size in Figure 9. Comparing the breakdown of the two datasets highlights the complexity of converting the many numerical and temporal types of the *NYC taxi trips* dataset, which, on average, make up only 5.2 bytes per value. The type conversion of the *NYC taxi trips* dataset accounts for roughly one third of the total processing time. Type conversion of the *yelp reviews* dataset, in contrast, only contributes approximately 20% to the total processing time, as the text-based reviews make up the majority of the raw record size. The analysis shows that the approach is mostly agnostic to choice of the chunk size, as long as it is reasonably large. Only for tiny chunk sizes of 15 bytes and less, the overhead of initialising and scheduling tens of millions of threads becomes noticeable. For a tiny chunk size, the ratio of actual work being done in relation to the time spent on initialising threads and the amount of meta data being written becomes unfavourable. Choosing a small chunk size is disadvantageous to parsing, tagging, and the prefix scan. As the prefix scan's complexity is linear in the total number of chunks, its share of the processing time becomes noticeable when using very small chunks. The prefix scan takes less than two percent of the total processing time for most choices of the chunk size. The small spikes for parsing and tagging when using 32, 48, and 64 bytes per chunk, respectively, are due to shared-memory bank conflicts and bad occupancy. The best performance is achieved for 31 bytes per chunk, which will be used as default for the remaining evaluations.

Figure 10 shows `ParPaRaw`'s performance for various different input sizes. Parsing ten megabytes of the *yelp reviews* dataset in as little as one millisecond, `ParPaRaw` shows impressive performance even for small inputs, achieving a parsing rate of 9.75 GB/s. For even smaller inputs, `ParPaRaw` is able to process a single megabyte from either dataset in less than 500 μs, corresponding to a parsing rate of more than 2.1 GB/s and 2.7 GB/s for the *NYC taxi trips* and the *yelp reviews* dataset, respectively. Even though the absolute performance is impressive, in particular when compared to available parsers (see Section 5.2), `ParPaRaw`'s efficiency degrades as the input size decreases. When parsing only five megabytes of either of both datasets, `ParPaRaw`'s performance achieves roughly 50% of its peak performance. A major reason for this, especially for inputs that are parsed in less than a millisecond, is the overhead due to the many kernel invocations during the type conversion step. During type conversion, there are multiple kernel invocations per column, required for the CSS-index generation as well as the type conversion itself. Hence, considering the many columns of the two datasets, kernel invocations, each with an estimated overhead in the order of roughly 5 - 10 μs, account for a reasonable share of the few hundred microseconds that are required for parsing those tiny inputs.

We also analyse the performance of the different tagging modes (see Figure 11). Compared to the original inputs, the skewed inputs in Figure 11 (right) contain a single record that is 200 MB in size, while the remaining records remain the same. As expected, the use of record-tags (*tagged*) is noticeably slower than the two other tagging modes. In particular the tagging, partitioning, and type conversion steps take more time, as they depend on the choice of the tagging mode. The analysis also highlights the approach's robustness, providing stable performance for the two dissimilar datasets, even if they are skewed (see Figure 11). On the one hand, the time breakdown shows that, except for the type conversion, all steps take roughly the same time for both datasets. Only type conversion, which involves generating data for more than an order of magnitude more fields in case of the *NYC taxi trips* dataset, shows perceivable performance differences. On the other hand, the approach shows robust performance even for highly skewed inputs.

5.2 End-to-End Parsing

For the end-to-end parsing experiments, we measured the CPU wall-clock time. Measurements include the time for reading the input from RAM and writing the parsed data back to system memory. For `ParPaRaw`, this includes data transfers between the host and the device. The end-to-end parsing approach was compared against *MonetDB*, *Apache Spark*, *pandas*, and the approach presented by Mühlbauer et al. [33] (*Inst. Loading*). In addition, we evaluated the GPU-based parser that is part of NVIDIA's recently introduced open GPU data science project called *RAPIDS*. For *RAPIDS* we provide two evaluations. Firstly, simply read-

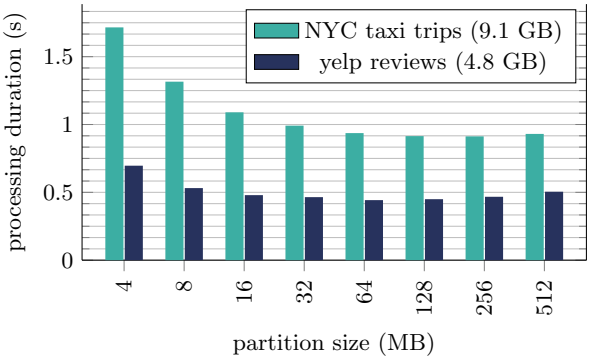


Figure 12: End-to-end parsing performance

ing the input into a GPU-based *DataFrame* called *cuDF* from where the data may be queried and processed with GPU support (*cuDF**). Secondly, exporting the parsed data to the host in the *Apache Arrow* columnar memory format using *cuDF*'s `to_arrow()` method (*cuDF*).

We analyse `ParPaRaw`'s performance depending on the chosen partition size (see Figure 12). Our evaluation shows that `ParPaRaw`'s performance increases with the partition size. Once the partition size grows beyond 128 MB for the *yelp reviews* and 256 MB for the *NYC taxi trips* dataset, however, the end-to-end processing duration starts growing again. This is due to the increased time for copying the very first partition and returning the parsed data of the very last partition (see Figure 7). Larger datasets compensate the effect of larger partitions. It is worth noting that this remains the only noticeable effect for larger inputs, since, with increasing input size, the number of partitions increases linearly, while the time per partition remains the same.

Figure 13 shows the time taken for parsing the respective input end-to-end. The performance numbers reported for parsing the 4.8 GB from the *yelp reviews* dataset highlight the strength of `ParPaRaw`, which takes only 0.44 seconds for the more challenging dataset. Only *cuDF*, which is still roughly 16 times slower than `ParPaRaw`, provides comparable performance. All CPU-based approaches, i.e., *MonetDB*, *Spark*, and *pandas*, are more than two orders of magnitude slower. Unfortunately, the implementation of *Inst. Loading* provided to us by the authors could not handle the yelp dataset due to its incomplete handling of quoted strings in parallel loads. Compared to *yelp reviews*, parsing of the *NYC taxi trips* dataset is easier to parallelise, as all line breaks correspond to record delimiters, making it trivial to identify the parsing context. Hence, even though parsing of the *NYC taxi trips* is computationally more expensive due to its many numerical and temporal fields, all CPU-based approaches benefit from the simpler format and see great improvements in the parsing rate. In particular, *Inst. Loading*, the approach proposed by Mühlbauer et al. [33], is about an order of magnitude faster than any other CPU-based implementation. Even though *Inst. Loading* is able to exploit the parallelism of the 32 physical cores for the *NYC taxi trips* dataset, `ParPaRaw` running on a single GPU is still roughly four times faster, despite the fact that `ParPaRaw` performs a full DFA simulation to keep track of the parsing context. Compared to the remaining approaches, `ParPaRaw` is more than ten times faster than *RAPIDS* loading the data into *cuDF* and over 40 times faster than the next best CPU-based approach.

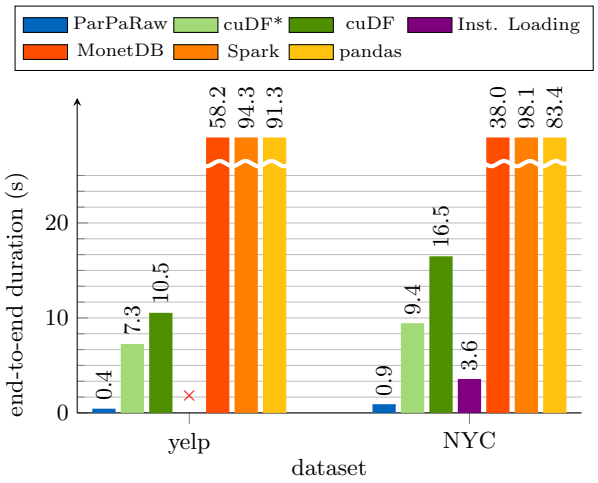


Figure 13: End-to-end performance comparison

6. CONCLUSIONS

This work presents `ParPaRaw`, a novel, massively parallel approach to parsing delimiter-separated formats. Other than the state-of-the-art that targets multicore and distributed systems with a coarse-grained approach, `ParPaRaw` is designed for fine-grained parallelism. Supporting parallelism even beyond the granularity of individual records makes it suitable for GPUs and ensures load balancing despite small chunks or large and varying record sizes. Being designed for scalability from the ground up with a data parallel approach that does not require any serial work, `ParPaRaw` is future-proof and can continue to gain speed-ups, as more cores are being added with future processors. Our approach identifies the parsing context (quotation scopes, comments, directives, etc.) without requiring a prior sequential pass. `ParPaRaw` is flexible and generally applicable. It supports even complex formats with involved parsing rules, as `ParPaRaw` is able to perform a massively parallel FSM simulation. State-of-the-art JSON parsers and the speculative approach by Ge et al., in contrast, have to deviate from the classic approach of using an FSM in order to be able to use SIMD vectorisation and speculation, respectively. This limits their applicability to other formats and requires designing completely different algorithms when confronted with another format (e.g., log files).

We show that `ParPaRaw` provides scalability without sacrificing applicability and flexibility. Achieving a parsing rate of as much as 14.2 GB/s, our experimental evaluation shows that `ParPaRaw` is able to scale to thousands of cores and beyond. With `ParPaRaw`'s end-to-end streaming approach, we are able to exploit the full-duplex capabilities of the PCIe bus while hiding latency from data transfers. For end-to-end workloads, `ParPaRaw` parses 4.8 GB of *yelp reviews* in as little as 0.44 seconds, including data transfers.

7. ACKNOWLEDGMENTS

This research has been supported in part by the Alexander von Humboldt Foundation. We would like to thank the authors of *"Instant loading for main memory databases"*, in particular Thomas Neumann, for providing their implementation [33].

8. REFERENCES

[1] NVIDIA Tesla V100 GPU Architecture. Whitepaper. Technical report, NVIDIA, 2017.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. SIGMOD, 2012.

[3] Alenka. Alenka - a gpu database engine. `https://github.com/antonmks/Alenka`, 2012.

[4] Apache Software Foundation. Apache Arrow. `https://arrow.apache.org`, 2019.

[5] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. *SIGARCH*, 2017.

[6] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.

[7] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. SIGMOD, 2014.

[8] G. E. Blelloch. Scans as primitive parallel operations. *IEEETransComp*, 1989.

[9] D. Bonetta and M. Brantner. Fad.js: Fast json data access using jit-based speculative optimizations. *PVLDB*, 10(12):1778–1789, 2017.

[10] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEETransComp*, 1982.

[11] B. Chandramouli, D. Xie, Y. Li, and D. Kossmann. Fishstore: Fast ingestion and indexing of raw data. *PVLDB*, 12(12):1922–1925, 2019.

[12] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. SIGMOD, 2014.

[13] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. ICS, 2008.

[14] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. Dbms data loading: An analysis on modern hardware. *DaMoN*, 2016.

[15] C. N. Fischer. On parsing context free languages in parallel environments. Technical report, 1975.

[16] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, and D. Kossmann. Speculative distributed csv data parsing for big data analytics. SIGMOD, 2019.

[17] S. Ha and T. Han. A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment. *TPDS*, 2013.

[18] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. SIGMOD, 2016.

[19] P. M. Hallam-Baker and B. Behlendorf. Extended Log File Format. `https://w3.org/TR/WD-logfile`, 1996.

[20] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *CACM*, 1986.

[21] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? *CIDR*, 2011.

[22] M. Ivanova, Y. Kargin, M. Kersten, S. Manegold, Y. Zhang, M. Datcu, and D. E. Molina. Data vaults: A database welcome to scientific file repositories. SSDBM, 2013.

[23] R. Johnson and I. Pandis. The bionic dbms is coming, but what will it look like? *CIDR*, 2013.

[24] Kaggle. Kaggle datasets. `https://www.kaggle.com/datasets`, 2018.

[25] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[26] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on raw data. *PVLDB*, 7(12):1119–1130, 2014.

[27] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEETransComp*, 1973.

[28] G. Langdale and D. Lemire. Parsing gigabytes of JSON per second. *CoRR*, 2019.

[29] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast json parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[30] A. Luotonen. Logging Control In W3C httpd. `https://www.w3.org/Daemon/User/Config/Logging.html`, 1995.

[31] D. Merrill and M. Garland. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

[32] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. Technical report, 2009.

[33] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.

[34] A. Mycroft. String Processing Instruction. `https://groups.google.com/forum/embed/#!topic/comp.lang.c/2HtQXvg7iKc`, 1987. comp.lang.c.

[35] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *PVLDB*, 11(11):1576–1589, 2018.

[36] RAPIDS. Rapids - open gpu data science. `https://rapids.ai`, 2012.

[37] Y. Shafranovich. RFC4180 - Common format and MIME type for comma-separated values (CSV) files. `https://tools.ietf.org/html/rfc4180`, 2005.

[38] A. Shahvarani and H.-A. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. SIGMOD, 2016.

[39] Simantex. Simantex - csvimporter. `https://github.com/Simantex/CSVImporter`, 2012.

[40] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, 1960.

[41] E. Stehle and H.-A. Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. SIGMOD, 2017.

[42] Sumo Logic. Press release. `https://www.sumologic.com/press/2018-02-27/growth-milestones`, 2018.

[43] Taxi and Limousine Commission. Tlc trip record data. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`, 2016.

[44] S. Yan, G. Long, and Y. Zhang. Streamscan: Fast scan algorithms for gpus without global barrier synchronization. PPoPP, 2013.

[45] Yelp Inc. Yelp Dataset Challenge. `www.yelp.com/dataset/challenge`, 2019.

[46] W. Zhao, Y. Cheng, and F. Rusu. Vertical partitioning for query processing over raw data. SSDBM, 2015.