Technische Universität Berlin
Fakultät IV
DIMA

Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing

Master Thesis
July 2020

Alexander Kumaigorodski
(319735)

Supervisor: Prof. Dr. Volker Markl
Advisor: Clemens Lutz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, 31.07.2020

_____

(A.Kumaigorodski)

Abstract

CSV is a widely-used format for data exchange. Due to its prevalence, virtually all industrial-strength database systems and stream processing frameworks support loading data from CSV input.

However, loading CSV data efficiently is challenging. Modern I/O devices such as InfiniBand NICs and NVMe SSDs are capable of sustaining high transfer rates. Fast interconnects, such as NVLink, provide the GPU high bandwidth to data in main memory. At the same time, the semi-structured and text-based layout of CSV is non-trivial to parse in parallel.

We propose to speed-up loading CSV input using GPUs. We devise a new approach to parallelize parsing that can correctly handle CSV features that are often neglected, in particular quotes. To efficiently load data onto the GPU from network devices, we extend our approach with RDMA and GPUDirect. Our evaluation shows that we are able to load real-world data sets at up to 60 GB/s, thereby saturating high-bandwidth I/O devices.

Zusammenfassung

CSV ist ein weitverbreitetes Format für den Datenaustausch. Auf Grund dieser Verbreitung, unterstützen praktisch alle Hochleistungsdatenbanksysteme und Stream Processing Frameworks das Laden von CSV-Daten.

Dennoch ist das effiziente Parsen von CSV-Daten herausfordernd. Moderne I/O-Geräte, wie InfiniBand-Netzwerkkarten und NVMe-SSDs, sind in der Lage hohe Transferraten zu liefern. Schnelle Interconnects, wie z.B. NVLink, bieten der GPU zu den Daten im Arbeitsspeicher eine hohe Bandbreite. Zugleich ist aber das semi-strukturierte und textbasierte Layout von CSV schwer parallel zu parsen.

Wir beschleunigen das Laden von CSV mit Hilfe von GPUs. Wir entwerfen einen neuen Ansatz zum Parsen, der oft vernachlässigte Eigenschaften von CSV, insbesondere Anführungszeichen, korrekt verarbeitet. Um Daten effizient aus dem Netzwerk zu laden, erweitern wir unseren Ansatz mit RDMA und GPUDirect. Unsere Evaluation zeigt, dass wir in der Lage sind echte Datensätze mit bis zu 60 GB/s zu laden und damit die Bandbreite von schnellen I/O-Geräten ausschöpfen.

Acknowledgments

First and foremost, I would like to thank Clemens Lutz for advising me on my thesis. His deep knowledge on GPUs, related research, and adjacent topics throughout this thesis proved to be a valuable resource for information and insight. He helped, inspired, and lead me. Thank you for all the time you have taken out of your schedule to meet with me every week and for being an excellent advisor.

Further, I would like to thank the primary author of *ParPaRaw*, Elias Stehle, for not only providing their implementation but also taking the time to help make it run with my evaluation data on the test machine.

Table of Contents

1. Introduction

Sharing data requires the source provider and the user of that data to agree on a common file format for exchanging data. *CSV* (comma-separated values) is a file format [1] for tabular data exchange that is widely supported by consumer, business, and scientific applications. It is a plain text file with lines representing rows and commas separating column values. As such, it trades performance and size for simplicity [2] [3]. More efficient file formats exist (e.g. Apache Parquet [4]), but they lack universal compatibility and are usually specific to their domain or platform. With *Big Data* and ever-growing data sizes in data processing, transferring and parsing CSV data increasingly becomes the bottleneck. To deal with this large or complex data gained more interest in the field of Big Data as well [5]. Most of these online data sets grow rapidly because they are constantly and increasingly gathered by cheap and various sensors, such as IoT devices, mobile devices, software logs, cameras, microphones, or RFID readers, and often stored as CSV [6]. The amount of online data has roughly doubled every 40 months since the 1980s, with currently over 2.5 exabytes of new data generated every day [6].

Motivation

Over the past decade, hardware evolved significantly. Previously, data processing or moving data was the bottleneck due to insufficient I/O performance. Thus, parsing data on the CPU was the overall fastest method. However, with new technologies like GPUs, *NVLink 2.0*, *RDMA* (remote direct memory access), and *GPUDirect* RDMA, moving data to the GPU is more efficient and not the bottleneck anymore. Additionally, these new technologies provide new opportunities for data parsing.

CSV parsing has traditionally been done on the CPU, with more advanced applications utilizing *SIMD* instructions [7] [8]. As processing is done in parallel on each of the cores, speeding up parsing further can only be achieved by scaling up CPU cores. For their highly parallel processing capabilities with tremendous computational power in comparison to CPUs [9], GPUs can already be used to

speed up data processing [10] [11]. Additionally, network interfaces can be scaled up to achieve a higher bandwidth than the CPU's interconnect, e.g. *PCIe* 3.0, can provide for reading the CSV data.

In contrast to CPUs, however, GPUs are specialized for throughput instead of latency. GPUs achieve high throughput by massively parallelizing computations. However, CSV's data format is challenging to parse in parallel. Finding line breaks to parallelize by rows requires iterating over the entire data first.

Contributions

Previous work on GPUs does not consider end-to-end parsing from I/O devices. In addition, previous approaches do not parallelize context-awareness but use resource-intensive *late context detection* instead. These do not scale to rates necessary for fast interconnects.

In this work we investigate strategies for parsing and deserializing CSV data in parallel on GPUs. We propose a new *early context detection* approach, that explores a new trade-off to gain efficiency. Our key insight is that the maximum row length has a known upper bound in practice. This insight enables us to parallelize *chunking* from the CPU-based approach by Mühlbauer et al. [7]. We analyze the most efficient memory access patterns in regards to threads and multi-level caches on the GPU. We adapt SSE 4.2 string specific SIMD instructions to GPU-based variants or alternatives and analyze their performance and viability for CSV parsing on GPUs. We implement an end-to-end parsing approach to offload CPU-bound CSV loading to the GPU and thereby saturate I/O bandwidth. The need for faster interconnects is underlined with performance of on-GPU parsing. We also present an implementation to efficiently load data onto the GPU from network devices using RDMA and GPUDirect. We show that even for small sized CSV files, loading data using the GPU can be reasonable. Overall, our contributions are:

- We propose a new approach to parallelize CSV parsing on GPUs.
- We analyze how to make the most efficient use of the CUDA platform and the GPU's architecture for CSV parsing.

- We show how to efficiently load data onto the GPU for end-to-end parsing.
- We evaluate cases in which it makes sense to offload parsing to the GPU.

Outline

Here, we provide an overview of the content and describe the structure of this thesis. In the next Chapter 2, we discuss the background. We give an overview of CSV characteristics and in-memory databases. Then we describe how GPUs are used for highly parallel processing in general and with CUDA specifically. In Chapter 3, we introduce our theoretical approach for parsing CSV data efficiently on GPUs using CUDA. The subsequent Chapter 4 shows the practical implementation and introduces its components. Next, in Chapter 5 we show comparisons and performance evaluations of some of our implementation strategies as well as comparable CPU- and GPU-based CSV parser implementations. In Chapter 6 we then present an overview of related research areas and its related work. Finally, in the last Chapter 7, we conclude and present possibilities for future work.

2. Background

In this chapter we give an overview of technologies and concepts significant to understanding this thesis. We will first describe the CSV file format. Then, we outline in-memory databases and their growth in popularity. We follow this up by introducing GPUs for their highly parallel processing capabilities. An in-depth overview of CUDA and its relevant details of the Volta architecture follows. Finally, we show high-speed NICs with the ability to directly manipulate remote memory.

2.1 CSV

CSV (comma-separated values) is a common file format [1] for tabular data exchange that is widely supported by consumer, business, and scientific applications and commonly used by databases for importing and exporting data. It is a lightweight, plain-text data format that often fulfills the requirement of being the least common denominator of information exchange.

According to the standard RFC 4180 [12], the CSV format can be recursively defined as follows:

```
File = [Header CRLF] Record *(CRLF Record) [CRLF]
Header = Name *(Comma Name)
Name = Field
Record = Field *(Comma Field)
Field = (Quoted | Unquoted)
Unquoted = *Char
Quoted = DoubleQuote *(Char | "" | Comma | CR | LF) DoubleQuote
Char = Any character, except ", CR, LF, and ,
Comma = ,
DoubleQuote = "
```

An example of typical CSV data is shown in Figure 1. A CSV *file* can contain an optional *header* that maps field names to the corresponding columns of the records. Lines separated by *CRLF* (`\r\n`) represent *records* (rows), and commas separate *fields* (columns). Each field is either *quoted* or *unquoted*. A field that

contains quotation marks ("), commas, or newline characters must be enclosed quoted in *double quotes*. Furthermore, an embedded quotation mark must be escaped by preceding it with another quotation mark. Whitespace characters are not ignored and treated as part of the field value, i.e. putting a space after the comma separating the fields, as is common in the English language, would result in a leading space character for the succeeding field.

```
ID,Country,ISOCode,Population\n
1,Germany,DE,83149319\n
2,Spain,ES,47007367\n
3,France,FR,67076431\n
4,Italy,IT,60317116\n
5,Japan,JP,126150745\n
6,China,CN,1427647786\n
7,United States of America,US,328239523\n
8,Canada,CA,37894799\n
...
```

Figure 1: Typical view of a CSV file

While the RFC 4180 is widely considered to be the main reference for CSV parsing, it is not an official specification, as the CSV format itself, despite its popularity, has never been officially standardized. This is most likely due to the fact that CSV was already first used in 1972 [13] and became popular by 1983 [14] but no standardization attempts were made until the RFC 4180 in 2005.

It should be noted that some variations of the CSV format exist. These use other delimiters for separating fields (e.g. tabs, pipes, or semi-colons instead of commas) or use another character to escape embedded quotes (e.g. backslash). It is also common for Linux-based software to export its CSV data with a simple *LF* (`\n`) instead of *CRLF* (`\r\n`).

Massive amounts of data from a wide range of sources and applications is made available this way. The *Common Log Format*[1] and *Extended Log Format*[2] are standardized text formats in CSV that are used by web servers to generate log files (e.g. for every file accessed by clients during page visits) as well as many other applications that use CSV, or a variation thereof, for logging. In 2018, van den Burg et al. [15] estimated that GitHub.com alone contains over 19 million CSV

[1] W3C. Logging Control in W3C httpd. https://www.w3.org/Daemon/User/Config/Logging.html
[2] W3C. Extended Log File Format. https://www.w3.org/TR/WD-logfile.html

files. Open government data repositories make their datasets increasingly more available in the CSV format [16], some in excess of hundreds of gigabytes in size[3] [4].

Advantages of the CSV format include their portability and simplicity but at the sacrifice of performance and file size [2]. More efficient formats exist (e.g. Apache Parquet [4]) but they lack universal compatibility and are usually specific to their problem domains or platforms.

In regards to correctly parsing CSV data, we consider a parser *context-aware* if it can correctly determine whether an encountered comma is an actual field delimiter or a character of a quoted string field. We classify *late context detection* as determining the context only during parsing and *early context detection* as having the context already discovered before parsing begins.

2.2 In-Memory Database Systems & Data Processing

To analyze or mine Big Data, its growth results in an increasing interest in data processing, especially *OLAP* (online analytical data processing), and a need for high-performance data loading and processing. However, in traditional databases the hard disk has been the bottleneck [17]. Meanwhile, decreasing prices of memory chips allowed server systems to be equipped with multiple terabytes of main memory. By the end of 2008, main memory cost fell under $10,000 USD per terabyte for the first time, as shown in Figure 2.

With these decreasing prices, in-memory databases became commercially viable, e.g. MonetDB[5]. In contrast to traditional disk-based database management systems, in-memory database management systems take advantage of the higher bandwidth and lower latency of main memory to increase query performance [18].

[3] Kaggle. https://www.kaggle.com/datasets?filetype=csv
[4] NYC OpenData. https://data.cityofnewyork.us/browse?limitTo=datasets
[5] MonetDB. https://www.monetdb.org/

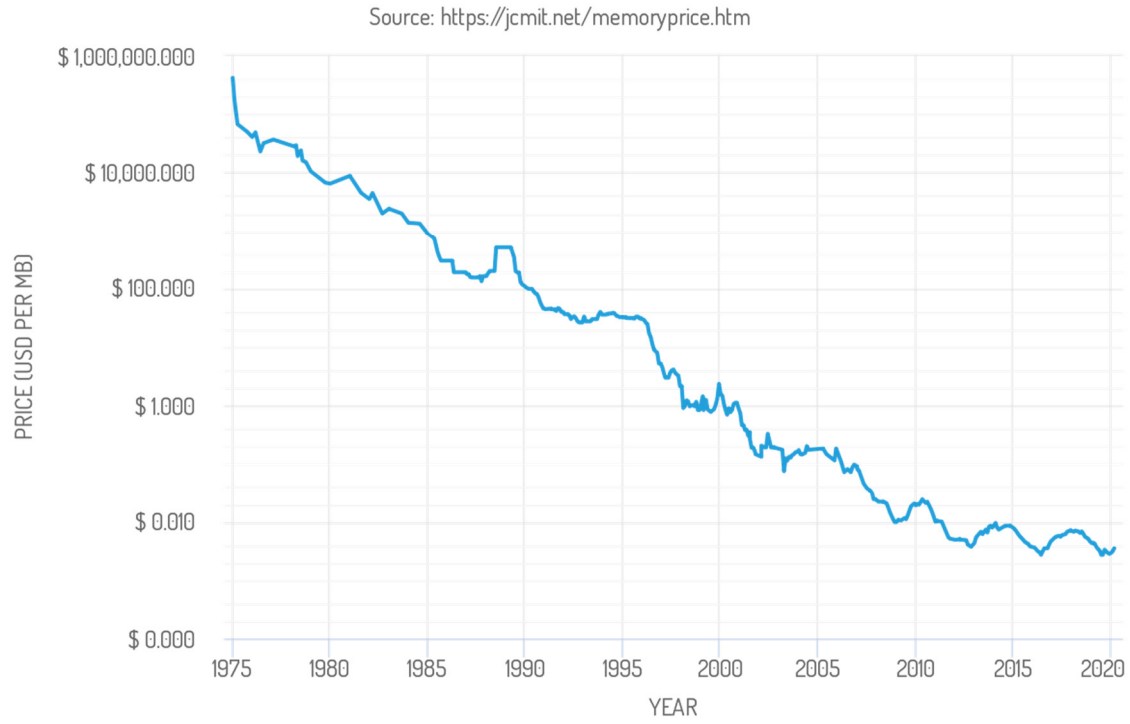

Figure 2: Main memory prices between 1975 and 2020.

OLAP-type workloads typically involve complex analytical queries over the entire data set. For this reason, column-oriented (in-memory) database systems succeeded in their effectiveness and this led to the development of a series of columnar data formats, e.g. Apache Parquet[6], ORC[7], Arrow[8] [19].

Aside from main memory, the CPU is the new bottleneck in these systems [20]. Recent hardware developments, discussed in the next sections, have presented an opportunity to further improve these CPU bottlenecks.

2.3 GPGPU

Computations, including CSV parsing, have traditionally been done on the CPU, with more advanced applications even utilizing SIMD instructions. With the advance of *GPGPU* (general-purpose computing on graphics processing units) for its highly parallel processing capabilities with tremendous computational power in

[6] Apache Parquet. https://parquet.apache.org/
[7] Apache ORC. https://orc.apache.org/
[8] Apache Arrow. https://arrow.apache.org/

comparison to CPUs[9], more and more data processing is being done on GPUs [21]. They are specialized for compute-intensive and highly parallel computations with transistors being devoted to data processing rather than data caching or flow control as is the case for CPUs [9]. Entire database systems are running on GPUs (e.g. Brytlyt[10], SQream DB[11]), and GPUs are used to speed up traditional and in-memory databases by offloading some of their data processing onto GPUs.

Modern GPUs are composed of multiple *SM*s (Streaming Multiprocessor), of which each has dozens of dedicated cores and several types and layers of cache. They are designed to be executed with the *SIMT* (Single Instruction, Multiple Threads) model. On Nvidia GPUs, this is typically done in groups of 32 threads, collectively referred to as *warps*.

Additionally, in comparison to the host machine's main memory, modern GPUs' dedicated memory (*VRAM*) allows for far higher throughput of data between processor and memory. Modern high-bandwidth memory technologies, like HBM2, currently deliver up to 900 GB/s of peak memory bandwidth [22], while modern DDR4 main memory can only provide a theoretical bandwidth of up to 25 GB/s per channel[12].

However, a major downside is that the VRAM's capacity on GPUs is rather limited in comparison to main memory on host machines. Current high-end GPUs at most only provide 32 GB of memory[13]. As this is typically too small to store the data of an in-memory database system, the relevant data needs to be transferred from host memory to GPU memory for processing and, optionally, the results copied back to host memory again. This is done over a comparatively slow interconnect between the CPU and GPU.

The most common interconnect today is PCIe v3.0, providing a theoretical maximum bandwidth of 15.8 GB/s[14]. In practice, however, throughput is typically only around 12 GB/s [21]. A limited amount of GPUs already released with PCIe

[9] "Theoretical GFLOP/s Intel CPUs vs. Nvidia GPUs" https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/floating-point-operations-per-second.png

[10] "Brytlyt: GPU based PostgreSQL Database" https://www.brytlyt.com

[11] "SQream DB: GPU-accelerated data warehouse" https://sqream.com/product/

[12] JEDEC Standard. DDR4 SDRAM. JESD79-4B.

[13] Nvidia V100. https://www.nvidia.com/en-us/data-center/v100/

[14] PCI-SIG. PCI Express Base Specification Revision 3.0, 2010.

v4.0 and a theoretical maximum bandwidth of 31.5 GB/s[15]. With Nvidia's protocol NVLink 2.0 and its own interconnect *NVHS* (Nvidia High-Speed Signaling Interconnect) an aggregated maximum bandwidth of 160 GB/s over four links is supported [23].

2.4 CUDA

CUDA is Nvidia's proprietary framework for their GPGPU pipeline and high-performance computing. In contrast to prior APIs like *DirectX* or *OpenGL*, CUDA provides an API with a focus on parallel programming. It gives developers a software layer that provides direct access to the GPU's virtual instruction set and compute elements to execute GPU functions (referred to as *compute kernels*) with an abstract view of the underlying architecture. In contrast to regular parallel programming, CUDA exposes architectural features, such as memory hierarchies and execution models, directly to the programmer. This enables finer control for better optimization of heterogeneous massively parallel programming tasks. The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, APIs, and extensions to industry-standard programming languages, including C/C++, Fortran, and Python.

The following sections will mostly focus on providing insight into CUDA running on the Volta architecture (V100 accelerator on GV100 GPU). To some extent, previous architectures (e.g. Pascal on GP100) may differ in their implementation.

2.4.1 Thread Abstraction and Organization

A CUDA program consists of a combination of the *host code* that runs on the CPU and *device code* that runs on the GPU. When a kernel is launched on the host side, its device code's statements are executed by the threads on the GPU. Threads within a warp execute the same statement simultaneously. CUDA exposes a two-level thread hierarchy abstraction, decomposed into grids of blocks and blocks of threads, illustrated in Figure 3.

[15] PCI-SIG. PCI Express Base Specification Revision 4.0, 2017.

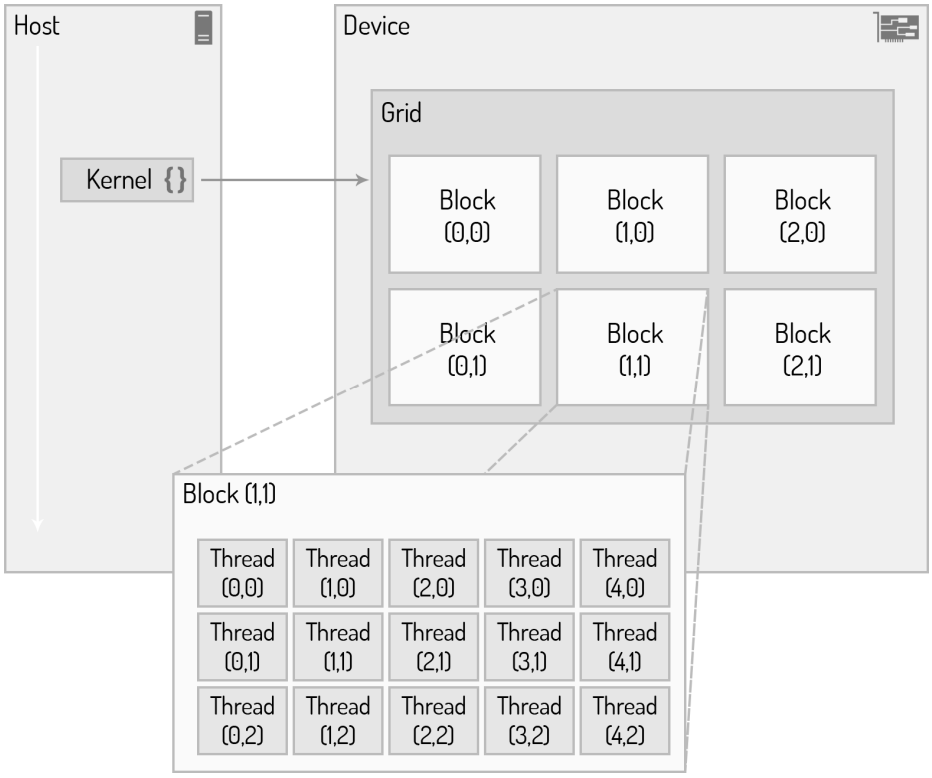

Figure 3: CUDA execution model and its thread organization

CUDA implements the *BSP* (bulk synchronous parallel) model for its thread architecture and requires structured parallelism workloads with much more tasks than available hardware cores on the SMs to scale and run efficiently. This allows CUDA to hide latencies caused by instructions and memory operations, as further discussed in the subsequent sections.

The threads that are launched by a kernel are collectively called a *grid*. All threads belonging to the same *block* can cooperate with each other by synchronizing or using the SM's shared memory space. The grids and blocks represent a logical view of the thread hierarchy of the kernel. Because their dimensionality affects performance, this abstraction allows to further optimize and even efficiently execute the same application code on various devices with different compute and memory resources.

While from a logical point of view it appears threads are executed concurrently, from a hardware point of view not all threads can physically execute at the same time. When a grid of blocks is launched (i.e. a kernel launch), the blocks are distributed among SMs, partitioned further into *warps*, and scheduled for execution. The number of warps per block can be calculated as follows:

$$warpsPerBlock := ceil\left(\frac{threadsPerBlock}{warpSize}\right)$$

Thus, the hardware always allocates a discrete number of warps for a thread block. Conceptually, this is the granularity of work processed simultaneously by an SM in SIMD fashion. If *threadsPerBlock* is not a multiple of 32, threads in the last warp are left inactive. It is therefore important to optimize workloads to fit within these boundaries to maximize utilization of the SM's compute resources.

When a warp stalls (e.g. when waiting on a memory operation to complete), the SM will switch to another eligible warp for execution to hide the latency that would have been otherwise introduced from the stalling warp. Ideally, the *occupancy* of SMs' cores should be kept close to 100% with enough warps to keep the device occupied:

$$occupancy := 100 \times \frac{activeWarps}{maximumWarps}$$

2.4.2 Warp Divergence

CPUs try to keep back pressure in their instruction pipeline for maximum hardware utilization. When encountering flow-control constructs, however, branching occurs and the pipeline can no longer be filled with upcoming instructions since it is not yet clear which path the application's control flow will take. Modern CPUs include complex hardware to try to predict the outcome of these conditional branches for their pipelined architecture (i.e. *branch prediction*). GPUs, however, are comparatively simple devices without complex branch prediction mechanisms. All threads in the warp must execute the same instruction in the cycle. This exposes a significant performance degrading problem when threads in the same warp take different paths (i.e. *warp divergence*). If threads in a warp diverge, the warp will serially execute each branched path while disabling threads that do not partake in that branch. Code like `if(cond){…}else{…}` would essentially cut the performance in half whenever at least one thread evaluates `cond` differently than the other 31 threads. With more conditional branches, the loss of parallelism would be even greater. To obtain the best performance, different execution paths within the same warp should be avoided. This keeps the *branch*

efficiency close to 100%, which is defined as the ratio between non-divergent branches and total branches:

$$branchEfficiency := 100 \times \left(\frac{totalBranches - divergentBranches}{totalBranches}\right)$$

Algorithms often need to be redesigned to achieve this. However, even simple techniques like *loop unrolling* or rearranging data access patterns can reduce or avoid warp divergence. More complex techniques, like thread-data remapping [24], exist as well.

2.4.3 Memory Hierarchy

Data-intensive workloads are bottlenecked by how fast they can read and write data. Thus, having higher bandwidth and lower latency memory would speed up the workload's performance. However, equipping hardware with a large amount of such memory is not always technologically feasible or economically viable. In that case, the memory architecture needs to achieve optimal latency and bandwidth with the underlying memory hardware, including hard disks or flash drives, main memory, caches, and registers.

When moving closer to the processor, the memory in the memory hierarchy becomes progressively faster but also smaller in capacity. For data that is actively being used by the processor, it is kept in that low-latency part of the memory hierarchy. For later use of that data, it is stored in the high-latency/high-capacity part of the memory hierarchy. CPUs and GPUs use similar models in their memory hierarchy design with GPUs allowing for finer control of their behavior. The L1 and L2 caches are examples of non-programmable memory in the CPU memory hierarchy. In contrast, CUDA's model allows many types of memory to be explicitly programmed. CUDA's memory model unifies the host's and the device's memory hardware but still exposes the full memory hierarchy to allow optimizing for highest performance and lowest latency with maximum capacity. We provide an illustration of CUDA's memory hierarchy model in Figure 4.

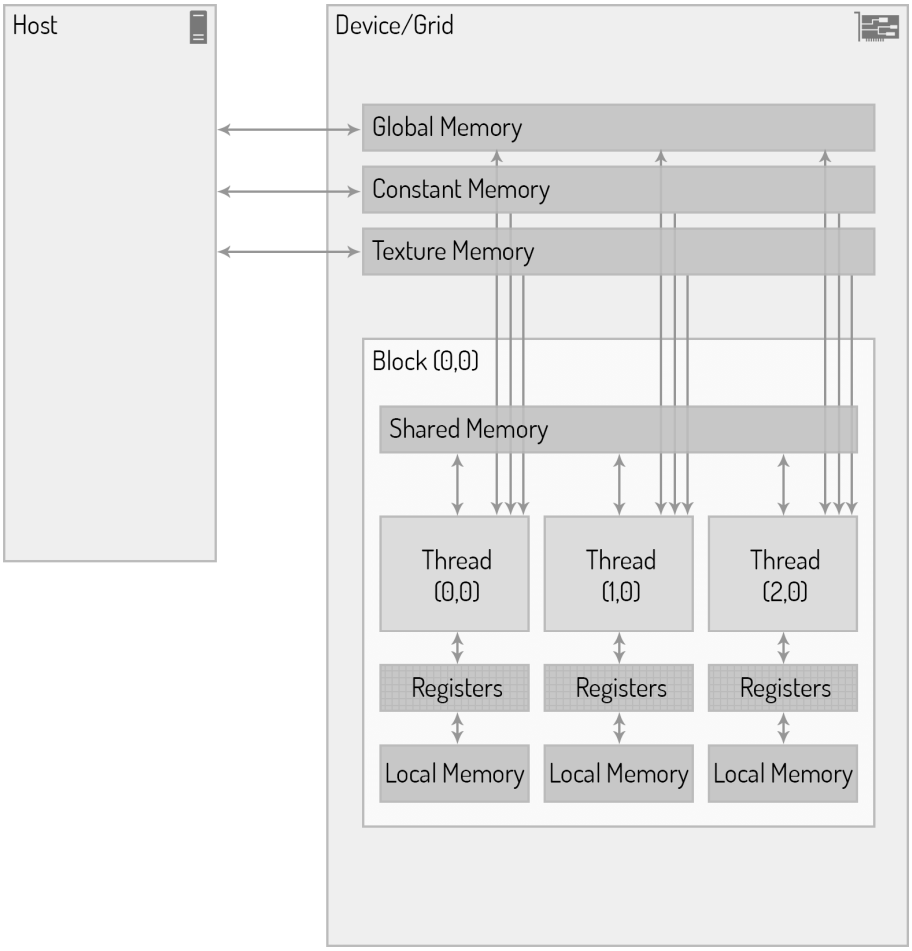

Figure 4: CUDA memory hierarchy model

Each has a different scope, lifetime, and caching behavior. While threads have their own private local memory (registers and spilled register data in global memory), blocks have their own shared memory space that is visible to all threads belonging to that block throughout its lifetime. The principal traits of various memory types are shown in the following table:

Memory	Location	Cached	Access	Scope	Lifetime
Register	on chip	-	R+W	Thread	Thread
Local	VRAM	L1	R+W	Thread	Thread
Shared	L1	-	R+W	Block	Block
Global	VRAM	L2 (L1 opt.)	R+W	All + host	Application
Constant	VRAM	Yes	R	All + host	Application
Texture	VRAM	L1	R	All + host	Application

Figure 5: Salient features of device memory (V100)

Registers. This is the fastest memory space on a GPU. Variables declared in a kernel are generally stored in a 32-bit register. Dynamically allocated arrays are always directly spilled to local memory. While register variables may be shared across threads in a warp using *Warp Shuffle Functions*[16], its contents are otherwise private to their respective thread and can no longer be accessed once a warp is finished with the kernel. Given the large number of threads that may run on an SM, its registers are a rather scarce resource (65,536 per SM on V100) and limited to 255 per thread. Should a thread exceed its limit or a block exceed the SM's resources, registers will be spilled into the much slower local memory. Conversely, threads with few registers used allow for more blocks to reside on the SM, which can increase occupancy and improve performance.

Local Memory. Variables from a kernel that were not eligible for or did not fit into the register space will spill into this memory space. Contrary to its name, it is merely reserved space in the VRAM and as such it is subject to the same high latencies and low bandwidths as global memory. In contrast to global memory, however, it is per default cached in the L1 cache.

Shared Memory. Variables declared `__shared__` in a kernel are stored on the same physical location as the L1 cache. It shares its lifetime with its thread block. Similar to the CPU L1 cache, it has a much higher bandwidth and much lower latency than global memory but is also programmable. Its main purpose is inter-thread communication using synchronized data access (e.g. using `__syncthreads()` or `__syncwarp()`) or as a software-managed cache. On Volta, up to 96 KB can be configured for use as shared memory per SM, limited by the 128 KB unified cache. When a thread block is finished with the kernel, its allocation of shared memory is released for new thread blocks to use.

Global Memory. This is the largest memory space on a GPU (either 16 or 32 GB on V100) with the highest latency and lowest bandwidth. It can be accessed from any kernel and from any SM at any time throughout the application's lifetime. It is usually allocated and managed by the host with address pointers passed to kernels as parameters. Data can be copied from and to the host's main memory in a similar fashion to `memcpy()`, using `cudaMemcpy()` or `cudaMemcpyAsync()` on the host. If the

[16] Nvidia. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions

host's memory is not pinned, CUDA will first allocate pinned memory on the host as a temporary staging area to transfer the data into before copying it to or from the device to avoid unsafe data access in case the operating system needs to physically move that memory page during a transfer. When reading or writing to global memory from a kernel, optimizing memory transactions are vital for obtaining maximum performance (see next section). For better performance, data from global memory is cached in L2 per default. Data that is read-only for the entire lifetime of a kernel can also be cached in L1 using `__ldg()` or hinted to the compiler with `const` and `__restrict__` variable modifiers, if the compiler hasn't already detected the read-only behavior through static code analysis. Addresses for memory operations need to be aligned, i.e. a thread accessing a data type of *n* bytes requires its address in global memory to be a multiple of *n*.

Constant Memory. Global variables declared `__constant__` are stored in the VRAM but cached in a dedicated 2 KB read-only constant cache on the SM, similar to L1 [25]. It is a latency optimized read-only memory space for threads within a warp accessing the same address simultaneously, e.g. when accessing a coefficient for a mathematical formula. Concurrently reading data from different addresses within a warp results in a severe performance penalty. Data in constant memory must be initialized by the host before kernel launch.

Texture Memory. A memory space optimized for 2D spatial locality with support for hardware filtering and interpolation. It is not relevant to this work and we include it here solely for the sake of completeness.

L2 Cache. A fast, global, 6 MB sized cache for all SMs on Volta. It caches global memory reads and writes with a cache line size of 128 bytes, divided into four 32 byte sectors. It is worth noting that contrary to Nvidia's Volta documentation [22], some sectors might be empty in the cache line. A cache miss in one of these sectors will not cause a load of the entire 128 byte cache line from global memory but only the lower or upper two sectors of the cache line, as found by Jia et al. [25] and confirmed by an Nvidia employee[17]. Implicitly, L2 is also used to cache instructions, constant data, and for the *TLB* (translation lookaside buffer).

L1 Cache. A very fast, 128 KB sized unified data cache on every SM on Volta. The memory space is shared with the shared memory space, which can be

[17] Nvidia Developer Forums. https://forums.developer.nvidia.com/t/pascal-l1-cache/49571/20

configured to take up to 96 KB, leaving 32 KB for L1 cache data. When configured to cache global memory reads, it also uses a cache line size of 128 bytes, similarly to L2.

To better understand each memory space's performance impact and put them into perspective, the table below provides latency and bandwidth comparisons filled with data from the official Nvidia Volta Architecture whitepaper [22] and data collected by Jia et al. [25] using micro-benchmarking on a V100-based GPU. For on-chip memory the bandwidth is the combined bandwidth of all 80 SMs on the V100 to make a direct comparison with off-chip memory possible. Even though L1 and shared memory are in the same physical location, we speculate their difference in latency is most likely due to the additional cache management overhead required for L1 data.

Memory	Latency	Bandwidth (Combined)
Registers	(e.g. FMA) 4 cycles	(e.g. FMA) ~58,000 GB/s
L1	~28 cycles	~12,000 GB/s
Shared	~19 cycles	~12,000 GB/s
L2	~193 cycles	~2,155 GB/s
VRAM	~1029 cycles	~750 GB/s

Figure 6: Performance comparison of memory types (V100)

2.4.4 Memory Access Patterns

As shown in the previous section, accessing global memory is relatively slow in regards to latency and bandwidth in comparison to the other memory spaces. Since most data access in applications begins or ends here, it is therefore important for a kernel to fully saturate the GPU's memory bandwidth whenever possible.

The global memory space is segmented into sectors of 32 bytes. Read/write operations are issued per warp, where each thread provides an address. With these addresses, CUDA then calculates how many memory sectors it needs to access and creates memory transactions based off of that. Each memory transaction can consist of one, two, or four consecutive 32 byte memory sectors. Ideally, each and

all of these 32 bytes were requested by the warp, i.e. the number of bytes requested is close or equal to the number of bytes actually moved by the hardware:

$$gld\_efficiency := \frac{requestedGlobalMemoryLoadThroughput}{requiredGlobalMemoryLoadThroughput}$$

Since global memory reads/writes are staged through caches and the fact that L2's cache line size is 128 bytes, a warp requesting these 128 bytes could potentially be served with only a single memory transaction from global memory.

Depending on the warp's distribution of memory addresses, memory access can be categorized into different memory access patterns. The ideal access pattern to global memory is the *aligned and coalesced* access pattern, i.e. Figure 7. *Aligned* access requires the address of the requested memory to be a multiple of 32 as the global memory space is segmented into sectors of size 32 bytes. A misaligned load will cause wasted bandwidth, since irrelevant bytes from the memory sector had to be physically transferred, as illustrated in Figure 8. *Coalesced* access refers to consecutive bytes requested by the threads in a warp. Combining these two patterns yields the best case scenario. In contrast, seemingly random access to memory within the warp yields the worst case scenario, as shown in Figure 10. Similarly, accessing the same data, i.e. Figure 9, does not make good use of the available bandwidth, as bus utilization is very low.

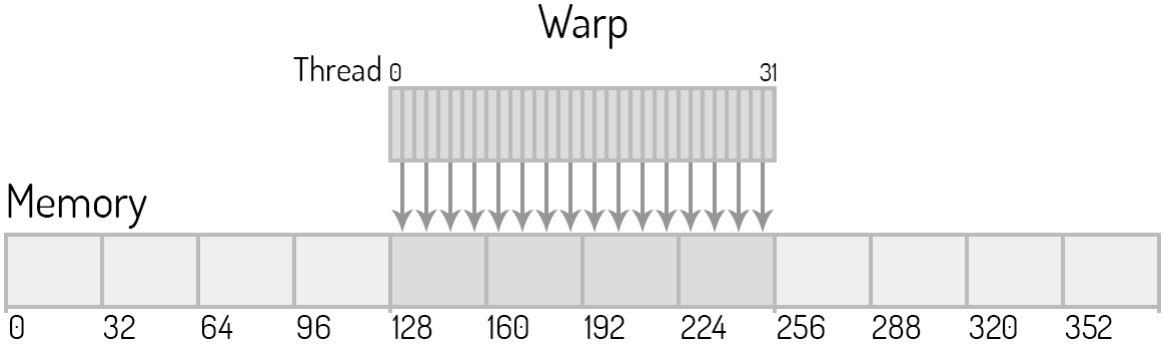

Figure 7: Ideal case, *aligned* and *coalesced* access. Addresses required for the 128 bytes requested fall within four sectors. Bus utilization is 100% with no loads wasted.

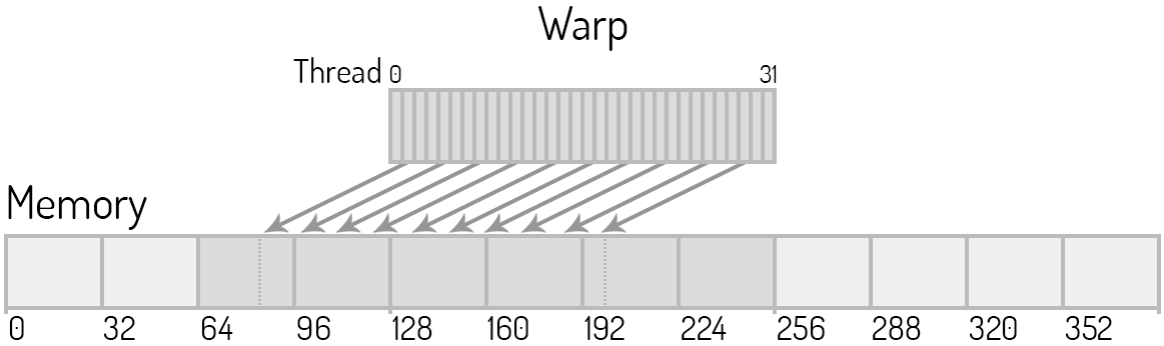

Figure 8: *Coalesced* but misaligned access. Warp requests 32 consecutive 4 byte elements but not from a 128 byte aligned address. The addresses fall within at most five sectors but six sectors are loaded. Bus utilization is 66.67%.

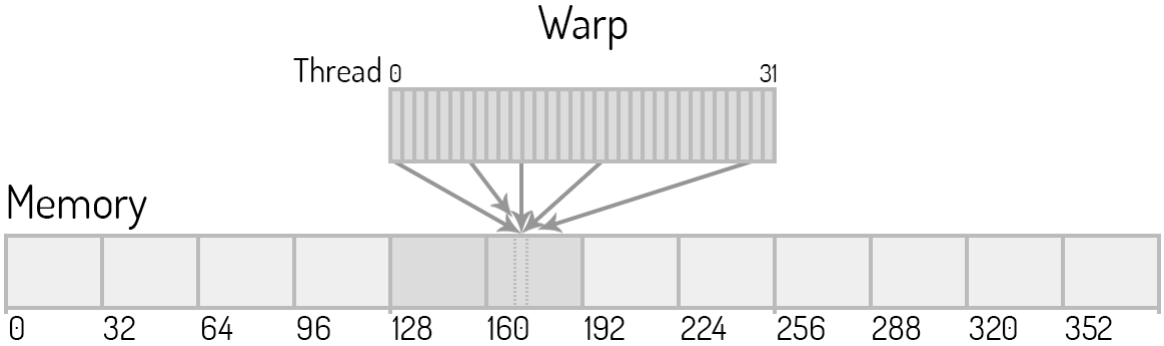

Figure 9: All threads in warp request same 4 byte data. The addresses fall within one sector but two sectors are loaded. Bus utilization is merely 6.25%.

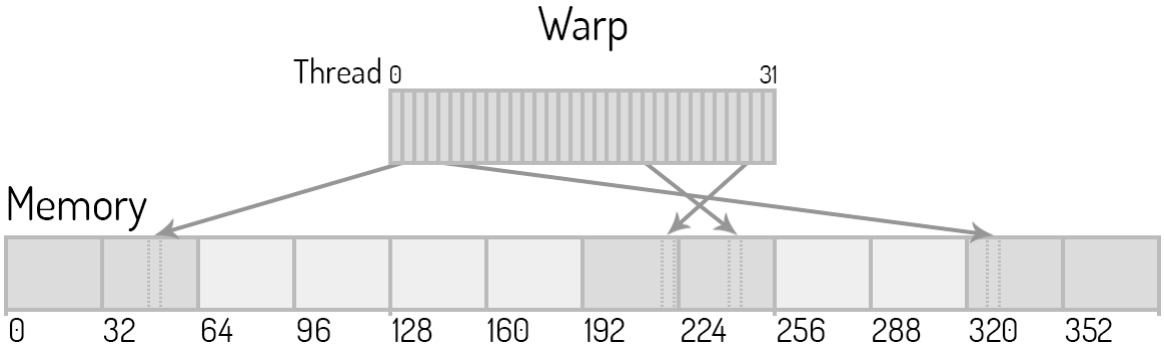

Figure 10: Worst-case scenario. 4 byte loads are scattered across 32 addresses in global memory.

Write operations behave similarly to read operations in regards to access patterns and memory transactions. They are stored in the L2 cache before being sent to the VRAM.

2.4.5 Shared Memory Banks and Access Patterns

When dealing with global memory, good performance can be achieved by using optimized access patterns so they are aligned and coalesced with no wasted memory transactions. For cases where aligned memory access is not possible, the L1 and L2 cache can mitigate performance issues. But memory access that is not coalesced and is scattered throughout global memory will still cause performance degradation and poor bandwidth utilization. Shared memory can help improve global memory access in many such instances.

For example, when transposing a 2D-matrix in global memory directly you will have non-coalesced access, regardless if the data in global memory is laid out as rows-of-columns or columns-of-rows. Using shared memory to cache data from the original matrix, one could avoid this strided global memory access. A column from shared memory can then be transferred to a transposed row in global memory.

Shared memory can also be used for threads within a warp or thread block, to cooperatively operate on temporary data in-memory. For example, a temporary prefix-sum over input data in a preparation step inside the kernel.

The shared memory space is partitioned among all thread blocks on an SM and a critical resource that can limit kernel occupancy. Access operations to shared memory are issued per warp and special care needs to be taken. While physically the shared memory is arranged in a linear manner, its access is divided into 32 four byte wide *memory banks*, as illustrated in Figure 11, that can be accessed simultaneously.

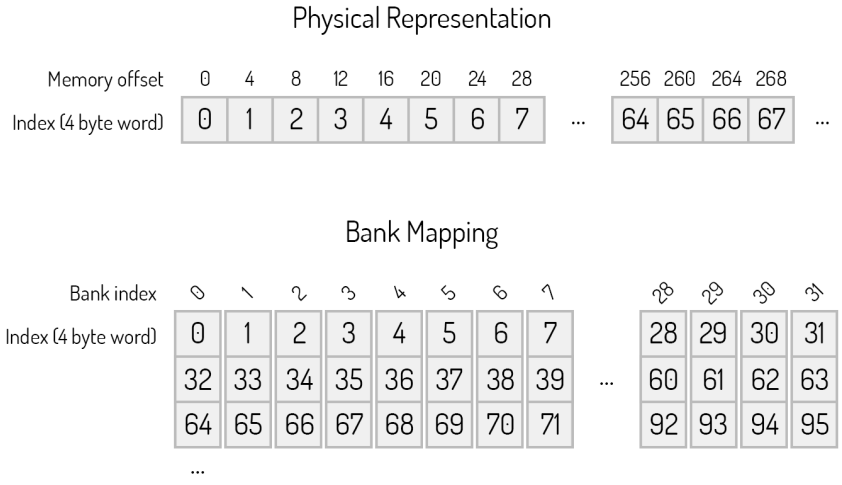

Figure 11: Mapping physical bytes to shared memory bank indexes

If a warp's memory operations do not access more than one memory location per bank, they can be serviced by one memory transaction, as shown in Figure 12 and Figure 13. Otherwise multiple memory transactions need to be issued, which will decrease shared memory bandwidth utilization.

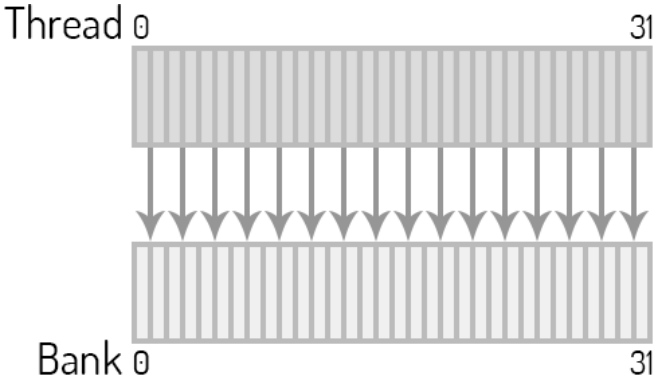

Figure 12: Optimal parallel access pattern. No bank conflicts, every thread accesses a different bank. Maximum bandwidth utilization.

A *bank conflict* occurs when multiple addresses of memory operations within a warp fall into the same memory bank, as illustrated in Figure 14. CUDA will split the memory operations into separate conflict-free memory transactions. With every additional memory transaction, the effective bandwidth is reduced by a factor equal to the number of total transactions. Multiple threads accessing the same address in the same bank can still be served with only a single memory transaction, however, since the accessed data in that bank is simply broadcast to all requesting threads afterwards.

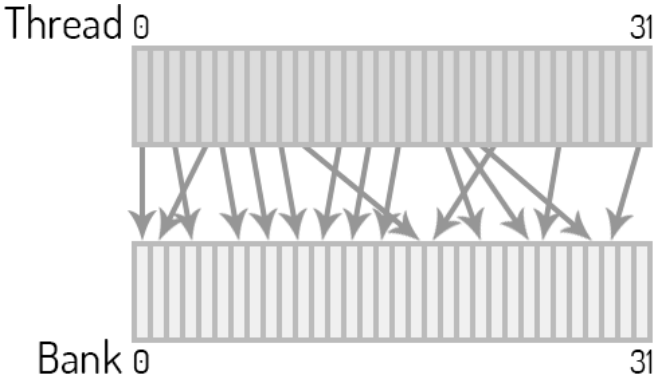

Figure 13: Irregular access pattern. No bank conflicts, because every thread still accesses a different bank. Maximum bandwidth utilization.

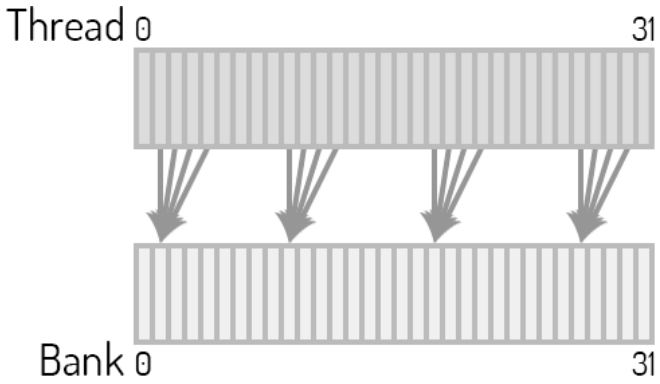

Figure 14: Irregular access pattern. Several bank conflicts with mutliple threads accessing the same bank. Conflict-free broadcast access only possible if threads access the same address within the bank. Poor bandwidth utilization.

The bank a shared memory operation is mapped to can be calculated as follows below. An address is divided by four to convert it to an index, since memory banks are four bytes wide, followed by the modulo operation with the total number of banks, 32:

$$bankIndex(memoryAddress) := \frac{memoryAddress}{4} \bmod 32$$

2.4.6 Streams

A CUDA *stream* (`cudaStream_t`) consists of an ordered sequence of host issued asynchronous CUDA operations to be executed on the device. By default, all CUDA functions implicitly use the NULL-stream. By using additional explicitly specified streams to launch multiple simultaneous kernels, we can implement *grid level concurrency*, allowing us to overlap execution of operations.

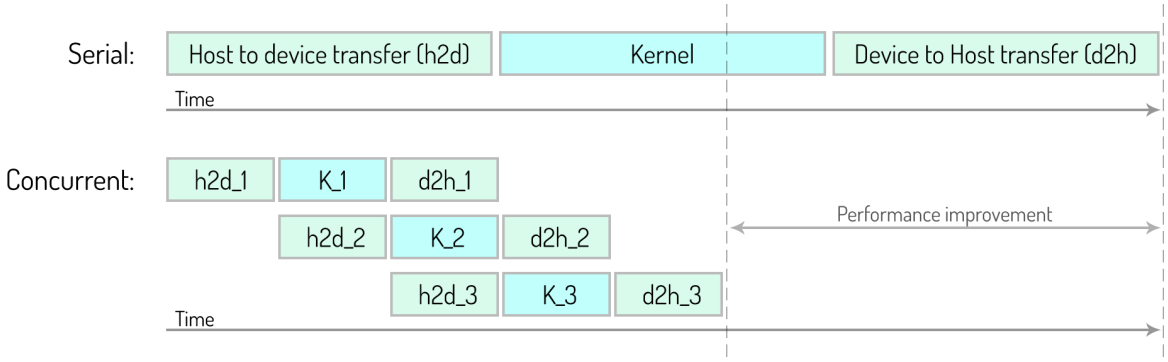

Figure 15: Example of using three CUDA streams to evenly distribute work

A typical pattern in a CUDA program is to first transfer input data from the host to the device, execute a kernel on that data, and then transfer the results back to the host. Instead, we can overlap the data transfer with kernel execution, thus, hiding some of the cost of the data transfer while performing useful work at the same time. We illustrate this pattern in Figure 15.

2.4.7 Code Comparison

The below code shows a shortened example of a CUDA kernel summing up arrays `a` and `b` and storing the results in array `c`:

```
...

void vectorAdd_CPU(int *a, int *b, int *c, int n)
{
    for(int i=0; i<n; ++i)
        c[i] = a[i]+b[i];
}

__global__
void vectorAdd_GPU(int *a, int *b, int *c)
{
    int i = threadIdx.x;
    c[i] = a[i]+b[i];
}

int main()
{
    constexpr int arraySize = 1024;

    ...

    //calculate sums on host
    vectorAdd_CPU(a, b, c, arraySize);

    ...

    //calculate sums on GPU
    constexpr int gridSize = 1;
    constexpr int blockSize = arraySize;
    vectorAdd_GPU<<<gridSize, blockSize>>>(a, b, c);
    cudaDeviceSynchronize();

    ...
}
```

As seen, programming CUDA kernels can be very similar to programming regular C/C++ functions. To allocate or transfer data to the GPU, CUDA provides several functions similar to C's own `malloc()`, `free()`, `memcpy()`, `memset()` etc.

Calling host classes and functions from device code is not possible, including the C's Standard Library or the C++'s Standard Library. While some ported functions are included with CUDA (e.g. `printf()`), data structures and algorithms need to be re-implemented and most likely redesigned to work efficiently in CUDA.

2.5 InfiniBand with RDMA & GPUDirect

InfiniBand is a networking communication standard used in high-performance computing [26]. Similar to Ethernet, it is used as an interconnect between servers or storage systems. It features a very high throughput of up to 50 Gb/s per link with latency below 0.5 µs[18]. Upcoming InfiniBand versions with up to 250 Gb/s per link are already planned[19]. Typically, four links are aggregated on most systems for improved bandwidth. Aggregated links of eight or twelve are possible but normally used for clusters or supercomputers. When compared to the bandwidth of regular main memory, it becomes clear that network bandwidth is no longer the bottleneck in previously network bound applications. In such a scenario, assuming the NIC is a PCIe-based device, PCIe itself can become the new bottleneck [27].

In the past years, *RDMA* (Remote Direct Memory Access) capable network cards have decreased in price and made their way into datacenters. InfiniBand supports RDMA, which is a feature that allows direct access to the main memory of a remote host with little or no CPU overhead. Requests are sent directly to the NIC without involving the kernel and are serviced by the remote NIC without interrupting the CPU.

GPUDirect extends this concept to the GPU. It allows remote hosts to directly access the GPU's memory for reading and writing, thus bypassing the main memory of the host machine.

However, most systems still rely on Berkeley sockets, even though several cloud services already provide RDMA-enabled devices [28]. A reason is believed to be ease of use [28]. For InfiniBand's RDMA, the low level *ib_verbs* API has to be used.

[18] Mellanox Technologies. InfiniBand Essentials Every HPC Expert Must Know. http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1/1_Mellanox.pdf

[19] InfiniBand Trade Association. InfiniBand Roadmap. https://www.infinibandta.org/infiniband-roadmap/

Before being made available for remote access, the memory region also needs to be pinned so the OS kernel does not page out the memory region. Client and server processes need to coordinate and have to be converted to use these RDMA-specific techniques.

Nevertheless, for transferring data between nodes this allows for very efficient zero-copy transfers in comparison to the typically used sockets, which require the data to be copied between buffers and the involvement of the kernel and CPU on both hosts [27].

3. Thesis Approach

In this chapter we introduce a new algorithm for parsing CSV data that is optimized for GPUs. When designing our approach, we focused on the hardware's main advantages and how to make use of them. Optimizing for GPUs is challenging, because parsers typically have complex control flow. However, fast GPU kernels should regularize control flow to avoid execution penalties caused by warp divergence. Therefore, our approach explores a new trade-off: we simplify control flow at the expense of additional data passes and, thus, more memory bandwidth. Overall, our approach adapts CSV parsing to fully utilize the GPU's architecture. This includes its very high memory bandwidth, low cache latencies, and high parallelism. Simultaneously, we focused on the hardware's shortcomings and how to work around them. Mainly branching, warp divergence, and inactive threads.

We first give a conceptual overview of our approach using the flowchart in Figure 16. Each step is then described and discussed in more detail with its challenges and solutions in the following sub-chapters.

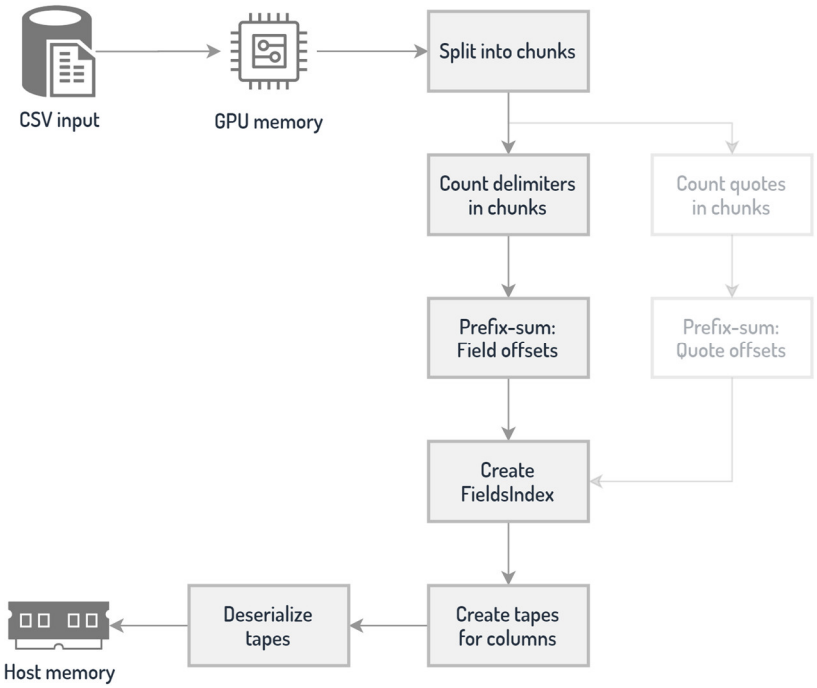

Figure 16: Conceptual overview of our approach

Conceptually, the CSV input data is first transferred onto GPU memory from a data source, such as main memory or an I/O device. The input data is then split into equally sized chunks to be processed in parallel. With the goal to index all field positions in the input data, we first count the delimiters in each chunk and then create prefix sums of those counted delimiters. Using these prefix sums, the chunks are processed again to create the *FieldsIndex*. This allows the input data to be copied to column-based *tapes* in the next step. Tapes enable us to vectorize processing by transposing multiple rows into a columnar format. Finally, each tape is deserialized in parallel. The resulting data is column-oriented and can then be further processed on the GPU or copied to another destination for further processing, e.g. to the host's main memory.

In this default *Fast Mode*, the parser may be unaware of the correct quotation scope when fields are enclosed in quotation marks. Fields may themselves contain field delimiters, resulting in an incorrect FieldsIndex. To mitigate this problem and create a context-aware FieldsIndex, we introduce the *Quoted Mode*. It is an alternative parsing mode with *early context detection*, that additionally keeps track of quotation marks and allows us to parallelize parsing of quoted CSV data but is more processing intensive than the default parsing mode. The main focus of our work is on the default Fast Mode, however, as well-known public data sources indicate that quotes are rarely used in practice[20] [21].

For now, we will assume the CSV input data already resides in GPU memory. In the last sub-chapter, we will present *Streaming*, which allows incoming chunks of data to be parsed without the need for the entire input data to be on the GPU.

Overall, our data-parallel CSV parser solves three main challenges: partitioning the data into chunks for parallel processing, determining each chunk's context, and fast deserializing of fields with their correct row and column number in parallel.

[20] Kaggle. https://www.kaggle.com/datasets?filetype=csv

[21] NYC OpenData. https://data.cityofnewyork.us/browse?limitTo=datasets

3.1 Parallelization Strategy

One of the GPU's strengths lies in highly parallel processing, making it an ideal platform for problems in which data can be split for parallel computations. However, text-based data formats, such as CSV, are challenging to parse in parallel.

Load Balancing Warps. Parallelizing by rows requires iterating over the entire data first and will also result in unevenly sized row lengths. This will cause subsequent parsing or deserialization threads in a warp to stall during individual processing and, thus, not make maximum use of the hardware. Instead, Figure 17 shows how we naively split the input data into equally sized chunks that are independent of each other and individually processed by a warp.

Figure 17: Splitting input into equally sized, independent, chunks

This allows all threads in a block to keep busy as they transfer and process the same amount of data. And because of the chunks' spatial locality and equal sizes, all warps within a block are subject to the similar or even same latencies when transferring their data from global memory or L1/L2, further reducing unnecessary delays within a thread block, thus, allowing a new block to run sooner on the SM.

Parallelization Granularity. The choice of *chunk size* and how a warp loads and reads its chunks impacts the parallelizability of the delimiter-counting process and, ultimately, the entire parsing process. Because a warp has 32 threads, the smallest reasonable chunk size is 32 bytes. This gives every thread in a warp exactly one character to look at and check whether it is a delimiter or not. However, as L1's and L2's cache line size is 128 bytes and a single memory transaction can serve up to four consecutive 32 byte sectors, i.e. 128 bytes, increasing the chunk size to at least 128 is plausible. This will give each thread in a chunk four bytes to look at, either four consecutive bytes or four individual bytes in a 32 byte stride. At first glance, increasing the chunk size further past 128 bytes seems counter-productive, as that could increase resource usage by every thread block and, thus, reduce overall parallelizability. However, common optimization techniques like loop unrolling prove that an application's execution speed can often be reduced at the expense of data size, essentially a space-time tradeoff. As such, having a warp process multiple consecutive 128 bytes in a loop could overall reduce processing times. CUDA's scheduler for warps and blocks will have less overhead, as well as a reduced overhead associated with the launch of every new thread block or warp, e.g. calculating a warp's chunk ID.

Vectorization. To analyze these access patterns and identify chunk sizes that allow for the most optimal loading and processing, we implemented several kernels that each load chunks at different sizes. We defer further details to Chapter 5.2.1. Going forward, the forceful loading of four consecutive bytes as an `int` to a register will be our strategy for loading and accessing the chunks.

One caveat of our chosen strategy is that the last thread of the last chunk will cause a memory access violation for input data that is not a multiple of 128 bytes. While threads within the last warp can simply calculate if they are still in bounds to load their four bytes, the last valid thread would need to first check for the number of remaining bytes and then take an alternative path that accesses the one, two, or three remaining bytes individually. This would not only add complexity to the code but also an additional branch and potential warp divergence, resulting in a drop in performance. Instead, we pad the input data with NULLs to a multiple of 128 bytes during input preparation. Conventionally, strings are NULL-terminated, so any such occurrence in a thread, or even external applications, will simply cause these padded bytes to be ignored during loading or later parsing. This not only

avoids the additional branch to check for the availability of all four bytes but also the branch needed to check whether a thread within a warp is still in bounds for the warp's current loop iteration.

3.2 Indexing Fields

We can now start processing the chunks. Our end goal in this phase is to have all of the field positions of the input data indexed in the *FieldsIndex*. This index will be an integer array of yet unknown size `rows*columns` with a sequence of continuous field positions. This is a three-step approach.

In the first pass over the chunks, every warp counts the number of field delimiters in its chunk. The number of delimiters in each chunk is stored in an array. For optimization purposes, record delimiters are treated as field delimiters, thus, creating a continuous sequence of fields. A field's row and column numbers can later be inferred with:

$$row(fieldIndex) := floor\left(\frac{totalFields}{fieldIndex}\right)$$
$$column(fieldIndex) := fieldIndex \bmod columns$$

In the second phase, we compute the chunks' field offsets with an exclusive prefix sum, as illustrated in Figure 18.

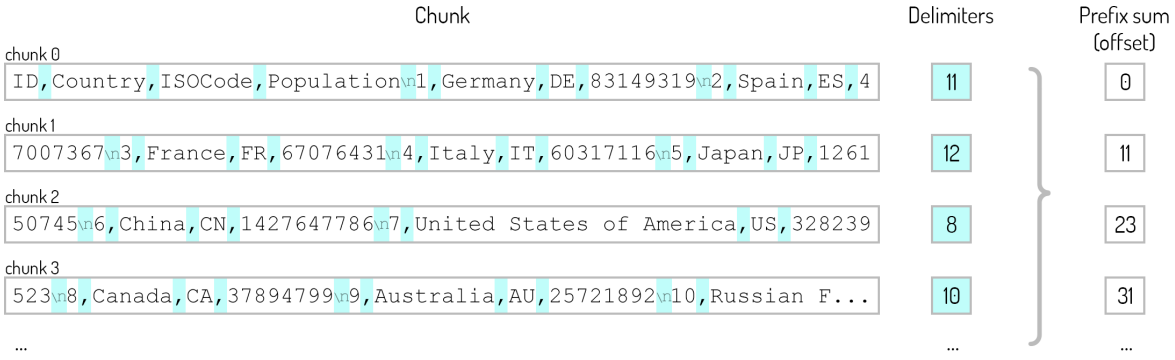

Figure 18: Computing the field offset for every chunk using a prefix sum

At the end of the prefix sum calculation, the total number of fields is automatically available, and consequently rows, so the necessary space in global memory for the FieldsIndex array can be allocated.

In the third and final phase, the FieldsIndex can now be filled in parallel. We perform a second pass over all the chunks and scan for field delimiters again. The

GPU memory's high bandwidth keeps the performance penalty of additional passes over data comparatively low. As shown in Figure 19, using the prefix sum at a chunk's position, the total number of preceding fields in the input data can instantly be inferred.

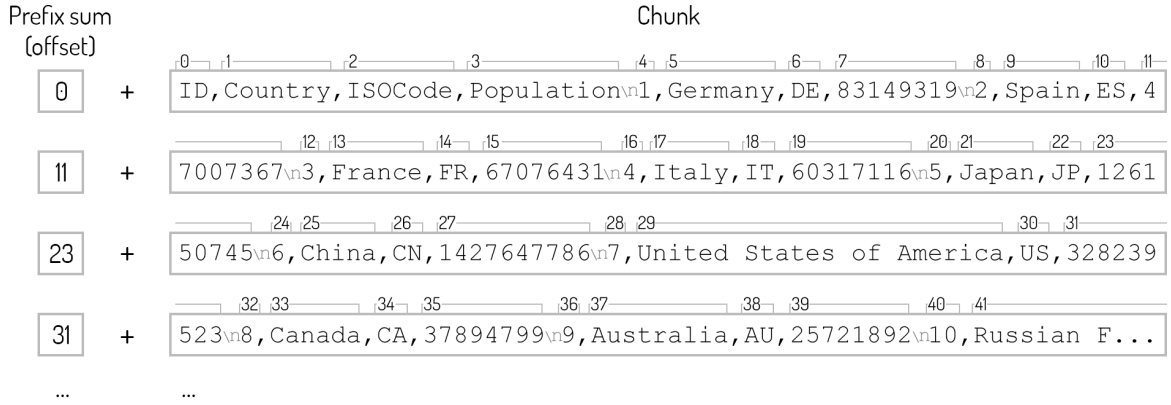

Figure 19: Using the chunk's prefix sum to infer number of preceding delimiters

We efficiently compute the FieldsIndex using millions of warps. To have a thread correctly determine a field's index when encountering its delimiter, it needs to also know how many delimiters the warp's preceding threads already had and will have. So, for every 128 byte loop iteration over the chunk, threads first count how many delimiters they have in their respective four byte sector. Since threads within a warp can efficiently access each other's registers, calculating an exclusive prefix sum of these numbers is fast and will now result in the complete information needed to determine a field's exact position and index to store it in the FieldsIndex array, with i being the i-th byte of the four byte sector:

$$fieldPos := chunkId \times chunkSize + laneId \times 4 + sectorId \times 128 + i + 1^{22}$$
$$fieldIndex_{>0} := chunkPrefixSum[chunkId] + warpPrefixSum[laneId] + di^{23} + 1$$
$$fieldIndex_0 := 0$$

The complete FieldsIndex then allows to not only instantly look up a field's position in the input data, but also its length:

$$fieldLength(fieldIndex) := FieldsIndex[fieldIndex + 1] - FieldsIndex[fieldIndex] - 1$$

[22] Adding one gives us the position of the actual field instead of the encountered delimiter

[23] di is the running count of delimiters in this four byte sector, for cases where it has more than one

Quoted Mode

For the *Quoted Mode*, a few additional steps need to be taken throughout to create a correct FieldsIndex. When counting delimiters in the first phase, quotation marks are also counted in a similar manner simultaneously. After calculating the prefix sums for the delimiters, the prefix sums for the quotation marks are created as well. In the third phase, during the second pass over the chunks, quotation marks are also counted again along the delimiters and prefix sums are created for both within the warp. We can now exploit the fact that a character is considered quoted whenever the number of preceding quotation marks is uneven. So, now before writing a field's position into the FieldsIndex when encountering a field delimiter, first the number of total preceding quotation marks at this position is checked. Should that number be even, the field's position to the FieldsIndex is written as before, regardless of how many preceding quoted field delimiters exist. Otherwise, a sentinel value of 0 is written to the FieldsIndex at the index the field's position otherwise would have been written to, representing an invalid field delimiter. After the FieldsIndex is created, a stream compaction pass is done on the FieldsIndex to remove all invalid, i.e. quoted, field delimiters and remove gaps between valid, i.e. unquoted, field delimiters. We illustrate an example with valid and invalid field delimiters in Figure 20.

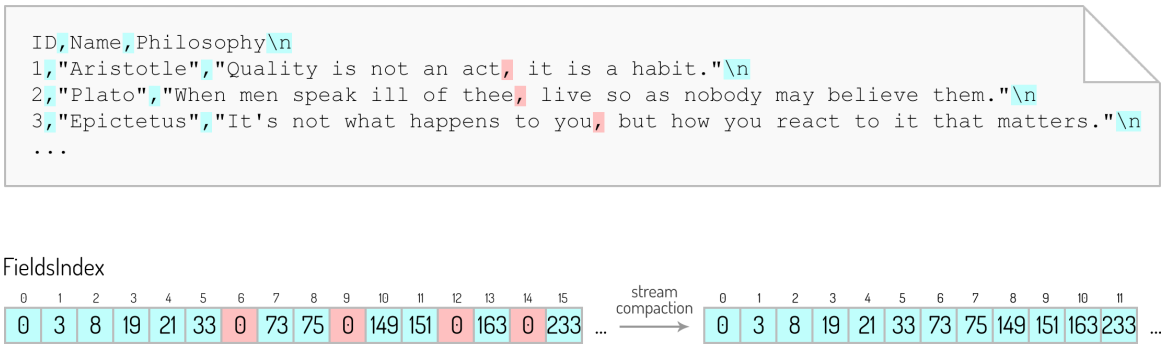

Figure 20: Additional pass in Quoted Mode to remove invalid delimiters

Doing this additional step separately instead of during the actual FieldsIndex creation, removes the complexity from the kernel that would otherwise introduce significant processing delays due to branching, warp divergence, and non-coalesced memory write operations.

We classify this approach as *early context detection*.

3.3 Deserialization

Efficient deserialization on the GPU is a many-sided problem. Not only is the question of how to parallelize deserialization challenging, but also how to keep the entire warp occupied while doing so.

A naive approach is to have every thread deserialize a field. However, we must assume that neighboring columns have different data types. So, constructing a generic kernel that can handle all data types involves many branches, causing warp divergence. Instead, we explore three different approaches to avoid warp divergence: row-based using *Dynamic Parallelism*, column-based with grouped warp lanes, and column-based with maximum column lengths.

Approach: Dynamic Parallelism (Row-Based)

To allow parallelization and have threads in a warp deserialize different data types without warp divergence, we experimented with an approach that involves *Dynamic Parallelism* in CUDA. Dynamic Parallelism allows a kernel to launch another kernel and even synchronize on this newly nested work. In our approach, a generic deserialization kernel reads a field in every thread and, based on its data type, launches a specific deserialization kernel with a grid size of one and a block size of 32. For integers, every thread looks at one character, converts it to a digit, and multiplies it by $10^{fieldLength-laneId-1}$. The warp's sum of these numbers is the resulting deserialized integer. Using this approach, we could potentially skip writing out the FieldsIndex in the previous step and instead just directly launch the appropriate deserializer kernel when encountering a field delimiter.

While this is a very inefficient use of resources, since a field of length five would result in 27 unoccupied threads, initial testing showed performance to be above PCIe v3.0's bandwidth. However, performance drops instantly when the input data grows to a few thousand fields that need to be deserialized. CUDA's scheduler uses a launch buffer in global memory to keep track of pending kernel launches. Once this buffer is full, it uses a virtualized buffer in the host's main memory [29].

This Dynamic Parallelism approach quickly saturates the native buffer with scheduled deserialization kernels for each field, leading CUDA to fall back to the

much slower virtualized buffer. While CUDA offers the possibility to increase the buffer on the GPU, this approach does not scale well.

Approach: Grouped Warp Lanes (Column-Based)

It becomes clear that using any row-based approaches requires adding lots of complexity to work in parallel. Complexity that is likely to cause idle threads. Any approach that is column-based, however, can make use of the fact that all fields in the column have the same data type, thus, giving us an easy pattern to parallelize on. While such an approach would not utilize the available bandwidth very well, given a GPU's high memory bandwidth, it is expected to be the most optimal solution.

Building upon the deserializer from our Dynamic Parallelism approach, we implemented a column-based deserializer kernel that can make use of all threads in a warp. For integers of length eight, this means the warp can be divided into four sub-groups, each deserializing one field of the same column. Instead of 32 threads working on deserializing one number, this approach has these 32 threads working on four numbers simultaneously using the same instructions, thus, keeping the occupancy much higher without warp divergence and making much better use of the available hardware. The cost of calculating each lane's exponentiation of base 10 remains relatively high, however. Using the fast constant memory space for a memoization technique is not viable, since accesses to constant memory need to have a unique address within the warp to be efficient.

Approach: Columns with Maximum Lengths

To remove the cost of determining each lane's exponentiation of base 10 and the cost of cooperatively calculating a sub-group's sum, we extend our above approach with an alternative solution. Every thread in the warp deserializes one field, allowing the entire warp to deserialize 32 fields in parallel. Similar to SQL's *DDL* (Data Definition Language), users of CUDAFastCSV specify a column's maximum length along its type for deserialization purposes. To keep the warp's memory access pattern optimal, every thread first consecutively reads four aligned bytes into a dedicated register until enough bytes were read to satisfy the specified length

of the column. When all column fields are contiguous in memory, this will also cause the warp to read the data in an aligned and coalesced fashion with a high bandwidth utilization, serving the entire warp with just one memory transaction. In a loop equal to the size of the specified column length, every thread can now read and convert each digit from a register while calculating the running sum, as illustrated in Figure 21.

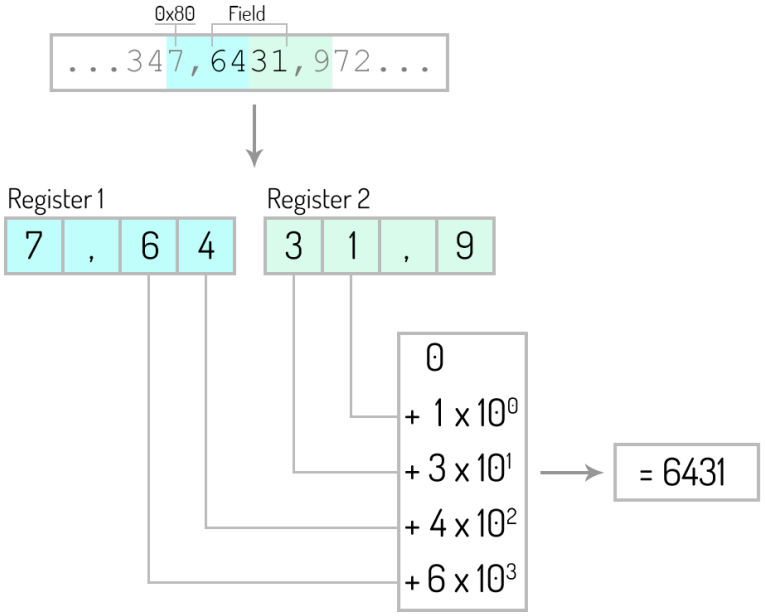

Figure 21: Thread reading aligned bytes to register for looped deserialization

While this approach can still leave some threads in the warp early with no work, i.e. when neighboring fields in a warp are of various length, this approach causes no warp divergence and only uses one branch in its entire kernel.

We identified this approach as the fastest deserialization strategy for our algorithm by implementing all three approaches as kernels for comparison. We defer further details to Chapter 5.2.1.

3.4 Optimizing Deserialization: Transposing to Tapes

Since our deserializer uses a column-based approach, its memory access pattern only allows for a coalesced and aligned memory access with full use of all the relevant bytes when given the optimal circumstances. CSV, however, is a row-oriented storage format. The optimal circumstances would come only into effect when there is just one column in the input data or when the field's data types are

identical along a multiple of 32 wide field count. For columns of various data types performance decreases significantly. In Figure 22 we illustrate this.

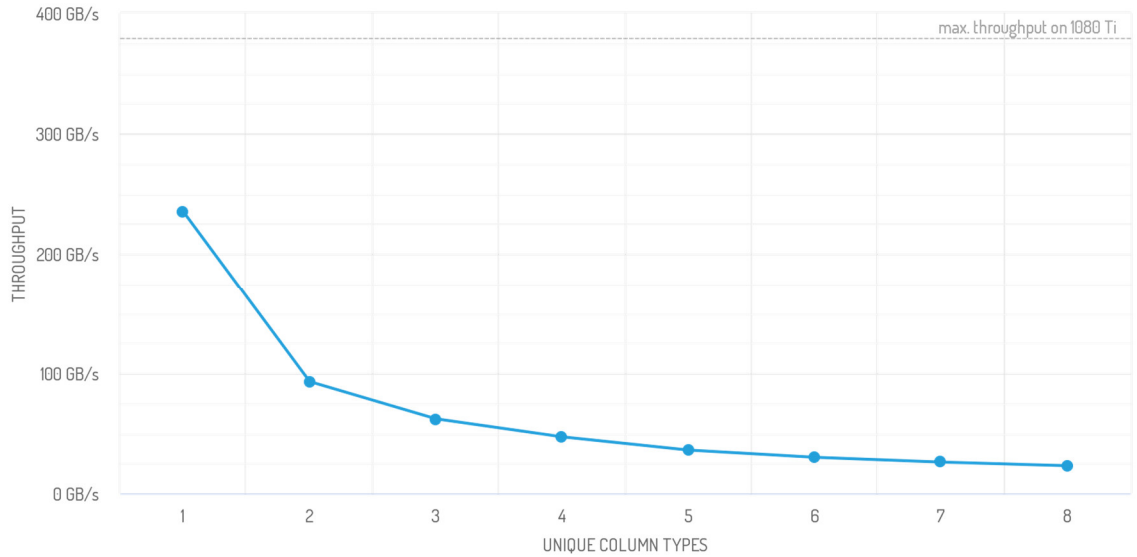

Figure 22: Column-based deserialization performance scaling for unique data types on 1080 Ti

The analysis shows the deserializer works the most efficiently when all its input fields are contiguous in memory. To improve deserialization performance for columns with various data types we make use of that fact and introduce deserialization with *tapes*. Tapes are column-based and enable us to vectorize deserialization by transposing multiple rows into a columnar format.

A separate tape for every column is created in an additional step during the parsing process. Given a column's length, we define a tape's width, *tapeWidth*, equal to its specified column length. The *tapeLength* is equal to the number of fields it will contain, i.e. the number of rows of the input data. Consequently, a *tapeSize* is the size of the tape's buffer in memory and is equal to $tapeWidth \times tapeLength$. For every field in FieldsIndex, the input's field value is copied to its column-individual tape at the following address in memory:

$$tapeAddress(field) := tape_{col(field)} + row(field) \times tapeWidth_{col(field)}$$

Field values that do not fully utilize their tapeWidth are right-padded with NULLs on the tape. We illustrate this approach with an example in Figure 23.

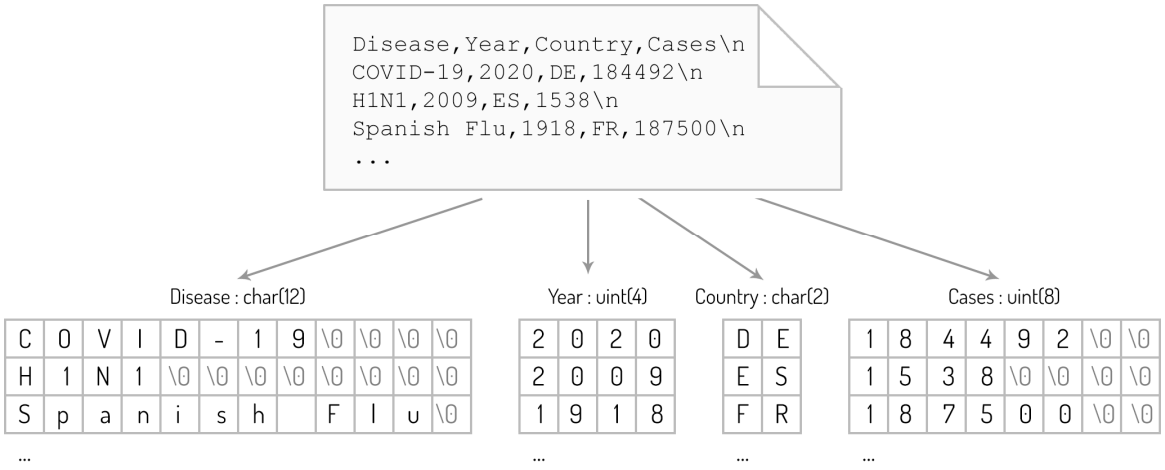

Figure 23: Visual representation of deserialization tapes

We defer its evaluation with further details to Chapter 5.2.1.

For our Fast Mode, writing out the FieldsIndex to GPU memory can be skipped and instead be temporarily written to shared memory. When a chunk's FieldsIndex is complete, the field values can be directly copied onto the tapes, essentially combining two steps of the process into one. However, only having a chunk's own isolated FieldsIndex, the length of the chunk's very last field is not calculable. We work around this obstacle by saving each chunk's first delimiter offset along the chunk's delimiter count during the first step of the parsing process.

Combining these two steps saves us from writing out the huge FieldsIndex and from having to do a third pass over the input data for creating the tapes, as is still the case for the Quoted Mode.

3.5 Streaming

We extend our approach to allow *streaming*. This enables us to start parsing the input data before it is fully copied onto the GPU, i.e. reducing overall latency, and for input data that is too big to otherwise fit into the GPU's memory.

The input data is split into *batches* before being copied to the GPU's memory for individual and independent parsing without the need for the complete input data to be on the GPU. We refer to an individually split part of the input data as a *batch*, representing a batch of aforementioned chunks. The batches are equal in length and of size *streamingBatchSize*.

> *"An orphan has no past, a widow has no future."*
> - Common mnemonic in typesetting

In typesetting, *widows* are lines at the end of a paragraph left dangling at the top from the previous page. *Orphans* are lines at the start of a paragraph left dangling at the bottom for the next page. Both are separated from the rest of their paragraph. Batching our input data creates a similar effect that we need to account for. In a batch, we consider the last row an orphan, unless it is terminated by a record delimiter, making the orphan empty. This row will not be parsed by its batch. Instead we copy the orphan's bytes to a temporary *widowBuffer*. The next batch will prepend available data from the widowBuffer to its batch data before starting the parsing process. Figure 24 illustrates our concept.

The widow buffer's size is pre-defined by the *streamingWidowBufferSize*. We set its default size to 10 KB. This default size is sufficient to handle rows spanning over 10,000 characters. Longer rows would end up being an orphan and not fully fit into the widowBuffer.

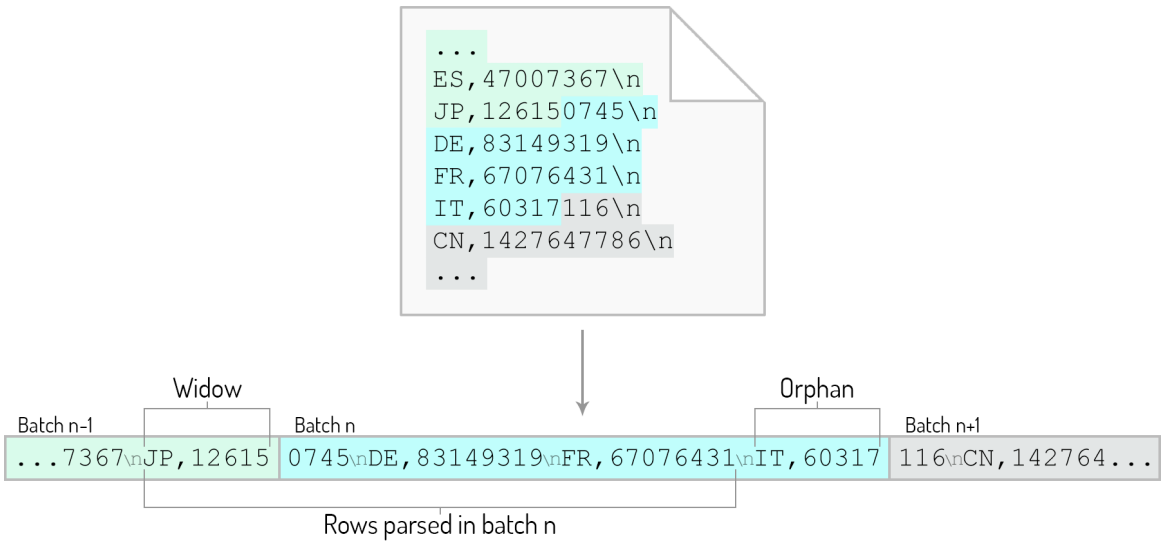

Figure 24: Widows are taken from the previous batch, while orphans are left for the next batch

Since our parser kernel reads data from global memory in four byte pieces from its chunk, its memory access needs to be aligned to a four byte memory address to avoid a memory alignment error. A widow whose size is not a multiple of four, would trigger such an error. We work around this issue by further padding the batch's data with one, two, or three NULL bytes. When parsing, these leading NULLs are then simply ignored by the first lane of the very first chunk.

4. Implementation

In this chapter, we show the practical implementation of the previously discussed approach and introduce the components of *CUDAFastCSV*, our C++ implementation of the presented work.

We will first give an overview of the most relevant components of our architecture, divided into groups. Each group will start with a class diagram of its components, containing each class' most relevant properties and methods. We then shortly describe each component and its function within the architecture.

In the next subchapter we will present how the components interact with each other and give a short breakdown of the processing cost for each step presented in the thesis approach. We end this subchapter by discussing some of the restrictions, optimizations, and challenges we faced during our implementation.

4.1 Components

For simplicity and a better overview of this subchapter, we group the components into six categories: Input Reading, Deserialization, Utilities, Parsing, RDMA, and Main.

4.1.1 Input Reading

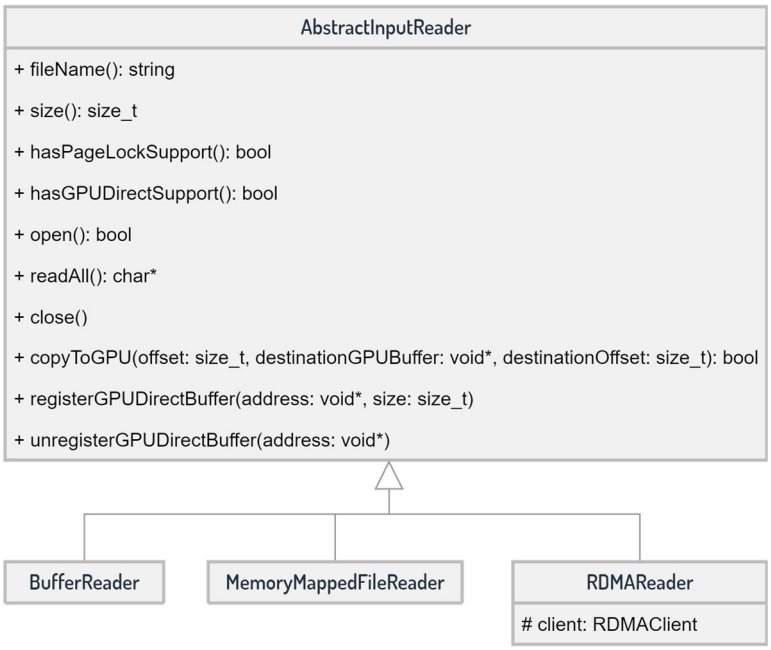

Figure 25: UML class diagram of input relevant components

AbstractInputReader. Since our input CSV data can originate from different sources, including network sockets, GPU memory, hard disk on host machines, or even from other applications, CUDAFastCSV uses an abstract base class to handle its input. The subclasses need to at least implement `size()`, `open()`, `readAll()`, and `close()` and may override the other methods for any unusual behavior.

BufferReader. Simple implementation that uses input data already residing in the host's main memory.

MemoryMappedFileReader. Using `mmap()` on Unix systems and `MapViewOfFile()` on Windows, this class is used for reading files from the host's file system.

RDMAReader. An implementation of an input reader that reads the input data from a remote machine, using RDMA, directly onto GPU memory, using the *RDMAClient* class (described in the RDMA group).

4.1.2 Deserialization

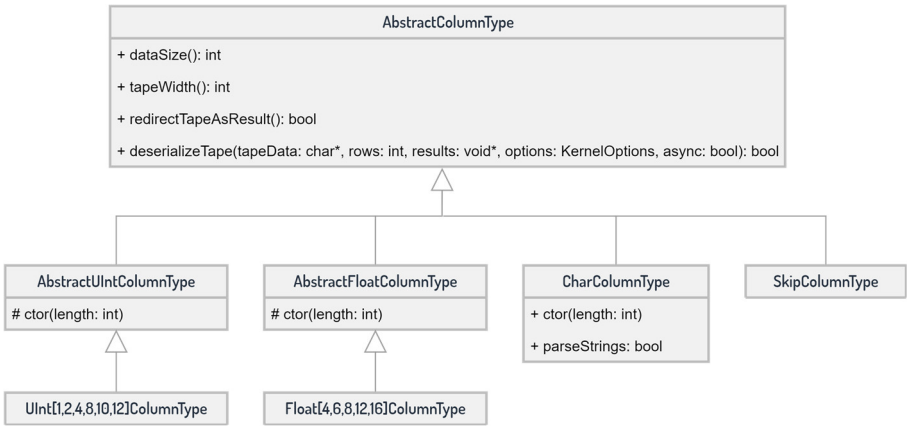

Figure 26: UML class diagram of deserialization relevant components

AbstractColumnType. This is the base class to implement data type specific deserialization logic as a CUDA kernel and provide requirements for the deserialization tape. The `dataSize()` tells CUDAFastCSV how much bytes a deserialized field will occupy to pre-allocate the result buffer, while `tapeWidth()` returns the required tape width needed for the tape buffer. The `deserializeTape()` method will be called to launch the deserialization kernel. In some cases, the tape buffer can be used as a result, e.g. when strings do not need to be further processed and already contain a null-terminated character by nature of the tape's design, so `redirectTapeAsResult()` can be overridden to let CUDAFastCSV skip the deserialization and instead simply copy or use the tape buffer as the result. Using the `DECL_CSVCOLUMNTYPE(className)` macro, subclasses are automatically registered and referenced in CUDAFastCSV.

UInt*ColumnType. Deserializes unsigned integer columns. Similar to SQL's DDL, the specified length refers to the input's maximum possible field length, not the deserialized integer's number of bytes. Fields of length 1 and 2 deserialize to `uint8_t`, while 4 deserializes to `uint16_t`. Longer fields use `uint32_t`.

Float*ColumnType. Deserializes to the `float` data type.

CharColumnType. For strings, the column's length is passed to the constructor. `parseStrings` can be set to true to remove quotation marks.

SkipColumnType. Similar to a projection operator, a special column type that can be used to fully ignore an irrelevant column in the input data to save parsing and processing costs.

4.1.3 Utilities

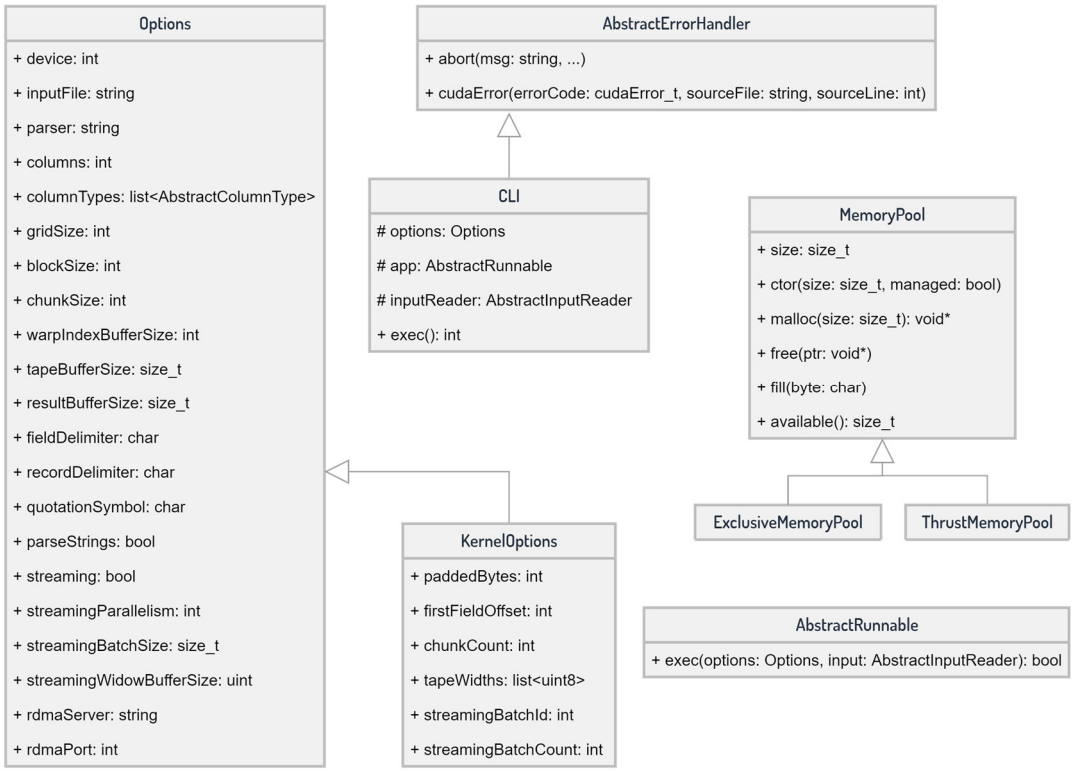

Figure 27: UML class diagram of helper components used throughout CUDAFastCSV

AbstractErrorHandler. An interface to be implemented by a class to handle errors that arise during CUDAFastCSV's execution.

AbstractRunnable. To allow CUDAFastCSV to be used as a stand-alone application for either parsing on the client machine or as an RDMA file server.

Options. Holds all configuration and customization for CUDAFastCSV. All properties can be changed from the command line using parameters.

KernelOptions. A sub-class of Options that is copied to constant memory to be used by the parser and deserialization kernels and holds additional contextual information needed for the current batch during execution.

CLI. Implements a command line interface for CUDAFastCSV.

MemoryPool. To save latency costs from many small `cudaMalloc()` calls of various and changing sizes, we allocate a large buffer on the GPU, whose memory is then managed and reused as needed.

ExclusiveMemoryPool. Distributes dedicated MemoryPools to individual CUDA streams.

ThrustMemoryPool. For device wide prefix-sum calculations and stream compaction in our parser we use *Thrust*, an algorithm library shipped with the CUDA SDK. Internally, Thrust uses memory on the GPU as a helper buffer every time it is used. ThrustMemoryPool provides a custom allocator to Thrust that uses our MemoryPool instead to avoid the overhead from constantly allocating and deallocating these buffers. This would otherwise be especially punishing when streaming, as another stream might be blocking the PCIe bus with a transfer of a batch of input data of several hundred megabytes, causing the parsing thread to stall. Using this custom allocator, we were able to improve overall performance by up to 15%.

4.1.4 Parsing

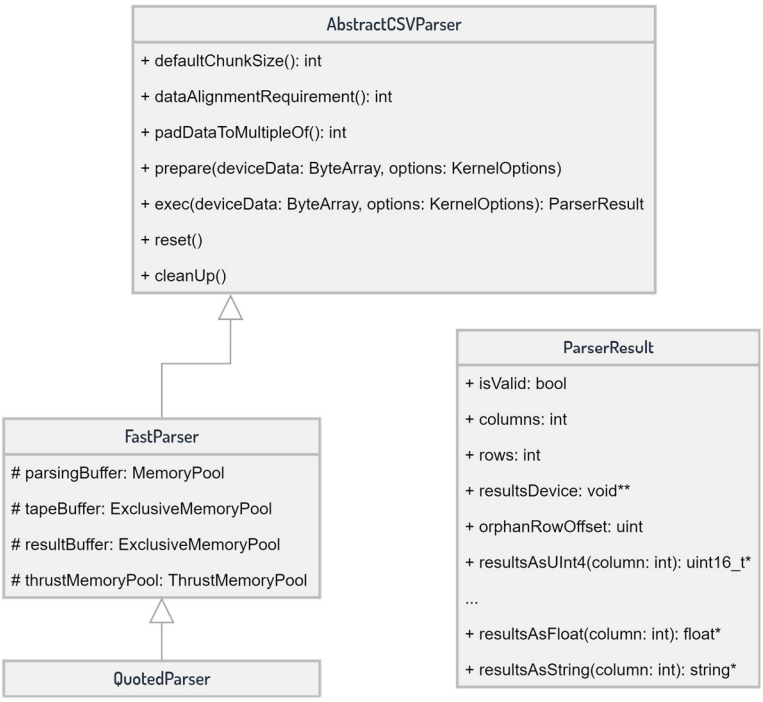

Figure 28: UML class diagram of components relevant to parsing

ParserResult. Result object that holds the deserialized fields from the processed batch with additional helper methods and the batch's contextual information. The `resultsDevice` property is a column-based array of pointers to global memory holding the deserialized values that need to be cast to their underlying data type.

AbstractCSVParser. The base class for all parser implementations of CUDAFastCSV, providing it information about data padding and memory alignment requirements as well as the actual implementation as a kernel. `exec()` is called for every batch and should synchronously return the ParserResult.

FastParser. The implementation of our proposed CSV parser in its Fast Mode. During its initialization it allocates reusable buffers `parsingBuffer`, `tapeBuffer`, `resultBuffer`, and `thrustMemoryPool` on the GPU to store intermediate and final results. To avoid results from being changed by the next batch's parsing while the current batch is still processing its results in another thread, `tapeBuffer` and `resultBuffer` are ExclusiveMemoryPools.

QuotedParser. A derived parser for the Quoted Mode that adds the additional steps needed to enable our parser to consider the quotation scope when indexing valid field delimiters.

4.1.5 RDMA

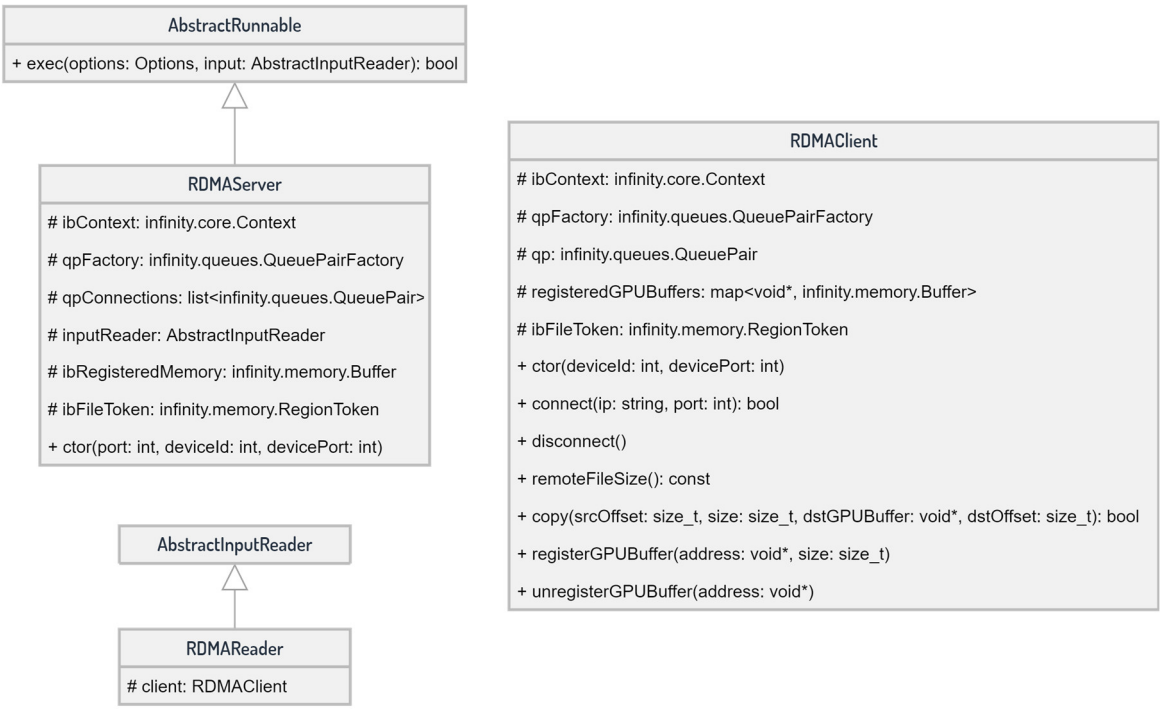

Figure 29: UML class diagram of RDMA specific components

For our RDMA and InfiniBand implementation we are using the *Infinity* library by Claude Barthels[24]. It is a lightweight C++-wrapper around the *ib_verbs* C-API that simplifies working with RDMA and InfiniBand.

RDMAServer. Starts a file server-like CUDAFastCSV instance that allows remote CUDAFastCSV clients to connect to this machine and directly read input data (property `inputReader`) using RDMA and InfiniBand. It automatically pins the memory, registers it with the RDMA device, and creates a token for communication with remote clients.

RDMAClient. A client wrapper that connects to a remote server and copies input data directly into a GPU buffer. Memory is automatically registered and managed when used with the *RDMAReader*-wrapper for reading input data.

[24] Claude Barthels. Infinity. https://github.com/claudebarthels/infinity/

4.1.6 Main

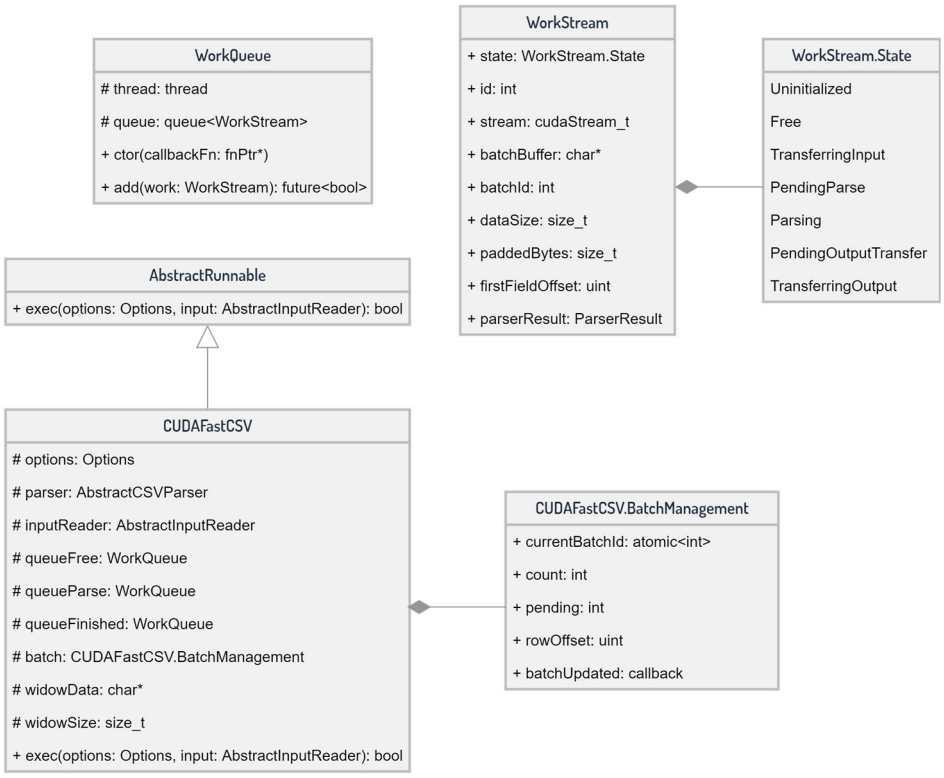

Figure 30: UML class diagram of components that act as a facade for CUDAFastCSV

WorkStream. Represents a CUDA stream and binds a single batch to it for full processing with additional metadata. In addition to its default `Uninitialized` state, a single WorkStream, and consequently the batch and CUDA stream it represents, can be in one of six states: `Free`, `TransferringInput`, `PendingParse`, `Parsing`, `PendingOutputTransfer`, or `TransferringOutput`.

WorkQueue. A queue data structure holding WorkStream items that wait for processing. Once their preceeding batch is finished in the queue, the queue's callback function is called with the next WorkStream item for processing.

CUDAFastCSV. The main facade of our work, putting all the components together. It uses the provided `options`, `parser`, and `inputReader` to process CSV input data with the help of three WorkQueues. The `queueFree` holds WorkStreams that are free to be assigned a new batch to. When a WorkStream is finished with transferring its batch's input data to the GPU, it waits in `queueParse` for parsing. Once finished parsing, it gets put into `queueFinished` for custom result processing or transfer of the batch's result output. When streaming is disabled, simply one WorkStream is used to process the entire input as one batch.

4.2 Implementation Details

In this section, we go more in-depth on some of our implementation particulars. We start with an overview as well as some of the specific optimizations we underwent. Next, a detailed explanation of the streaming process using WorkStreams and WorkQueues is given. A short time breakdown of CUDAFastCSV's processing stages follows. Finally, we provide a list of our implementation limitations with their reasoning.

Our implementation provides system-specific implementations to support all features under Linux and, with the exception of RDMA, all features under Windows.

4.2.1 Optimizations

To further improve the performance of our parser, we optimized multiple areas of our implementation.

In general, our parser makes use of the `__forceinline__` CUDA compiler directive, whenever possible, to guarantee the compiler inlines the function's code, favoring speed over space. Similarly, we annotate static `for`-loops with `#pragma unroll` to hint to the compiler it should fully unroll the loop. To improve throughput, we also try to avoid immediate read-after-write register dependencies in the kernels.

Kernel Fusion Optimization

For Fast Mode our intention was to skip writing out the FieldsIndex to GPU memory and instead write it to shared memory temporarily and essentially combine field indexing and tape copying into one step. We realize this by storing the chunk's own local FieldsIndex in shared memory.

Data Type Optimizations

CSV files that are larger than 4 GB are parsed using streaming. Except for the AbstractInputReader and its subclasses, CUDAFastCSV exclusively uses the `uint32_t` data type for handling input data sizes or index positions of fields,

limiting it to 4 GB of input data in these situations. This limitation was a deliberate design choice. Considering the size of the FieldsIndex and some of the data that is stored in the parser's buffers, using `uint64_t` or `size_t` over `uint32_t` would require twice the amount of data to be read and written several times, causing not only much longer transfer times but also reducing parallelism when stored in shared memory.

The FieldsIndex, when stored in shared memory, is an array of type `uint8_t`. As described in the Thesis Approach, when creating the FieldsIndex we first count the delimiters in every lane in the warp and then create a prefix-sum for the warp to be able to determine the number of preceding fields in the warp. The lane's prefix-sum is stored in shared memory for every warp. The prefix-sum calculation in the warp is then done as a tree-reduction using `__shfl_up_sync()`. Since a warp has 32 lanes and each lane can only contain four delimiters or four quotation marks at most during a 128 byte read, the array's data type `uint8_t` is sufficient, as no warp will exceed 255 total delimiters or quotation marks.

Tunable Buffer Sizes

When the FieldsIndex is stored in shared memory, its exact size needs to be allocated before kernel launch, as shared memory space is limited on the SM. Accounting for the worst case scenario of having only empty fields in a chunk of size, e.g., 4096, we would need to reserve 16 KB of shared memory resources for every warp. This severely limits parallelism for an edge case that might never happen. Instead, we provide the *warpIndexBufferSize* parameter, which limits the maximum number of found fields in a chunk within a warp and is used to reserve the kernel's shared memory space in Fast Mode or, in Quoted Mode, the required space in global memory for the FieldsIndex. Since the default *chunkSize* is 2048 bytes, *warpIndexBufferSize* defaults to 512 field positions, i.e. 2048 bytes. Both parameters can be tuned in accordance, improving performance based on the underlying data characteristics of the CSV input data. Should a chunk's field count exceed this limitation, i.e. not all field positions can be stored in shared memory, the application will not cause a memory violation and crash but instead gracefully stop its current parser and emit an appropriate error message that includes a suggestion for a new parameter value.

The *tapeBuffer* and *resultBuffer* of our parser implementation have a similar field count limitation and tuning capability. Because we reset these buffers for every batch by filling them with NULLs, their size should be small to reduce the runtime of the `cudaMemset()` operation but large enough to fit all of the rows and columns:

$$tapeBufferSize := rows \times \sum_{col=0}^{columns-1} tapeWidth[col]$$

$$resultBufferSize := rows \times \sum_{col=0}^{columns-1} columnTypes[col].dataSize$$

As such, a good estimation of the number of rows in the input data or its batch can further increase performance. Considering our tape design and that resulting string values occupy the same space, or in the case of short deserialized numbers potentially occupying even more space than their raw input values, both of these buffers are by default set to twice the input size or twice the *streamingBatchSize* when streaming.

For the Fast Mode, the parser uses the *parserBuffer* to store the delimiter counts for every chunk and for storing the positions of the first field of every chunk:

$$chunksPerBlock := ceil\left(\frac{blockSize}{warpSize}\right)$$

$$parsingBufferSize_{FM} := 2 \times (gridSize \times chunksPerBlock \times sizeof(uint32\_t))$$

For the Quoted Mode, it uses the *parserBuffer* to store the delimiter counts and quotation mark counts in every chunk, as well as the actual FieldsIndex:

$$parsingBufferSize_{QM} := 2 \times (gridSize \times chunksPerBlock \times sizeof(uint32\_t)) + gridSize \times chunksPerBlock \times warpIndexBufferSize$$

The parser's *thrustMemoryPool* size is fixed to 32 KB. During our tests, Thrust's buffer requirements never exceeded 8 KB in our implementation. To accommodate for any potential internal code changes of Thrust in the future, we opted for 32 KB. Given the dimensions of our input files in comparison, tuning this parameter is irrelevant and only yields improvements in the margin of error. For the Quoted Mode, this buffer is fixed to 32 MB due to the additional stream compaction step.

4.2.2 Data Streaming

As shortly introduced in Chapter 4.1, we implement streaming using WorkStream items, representing a CUDA stream and a batch for processing, with the help of three WorkQueues to queue these WorkStream items for individual processing. Every WorkStream item has a dedicated *batchBuffer* in the GPU's device memory that is used to copy its batch's chunks into. The size the input data is split into for streaming is controlled by the *streamingBatchSize* parameter. To accommodate for leading widow data and additional data padding, the WorkStream's actual size of the *batchBuffer* is larger than *streamingBatchSize*:

$$padding := parser.padDataToMultipleOf$$
$$batchBufferSize := streamingWidowBufferSize + streamingBatchSize + padding$$

For RDMA input data, this *batchBuffer* is also automatically registered for GPUDirect transfers via RDMA.

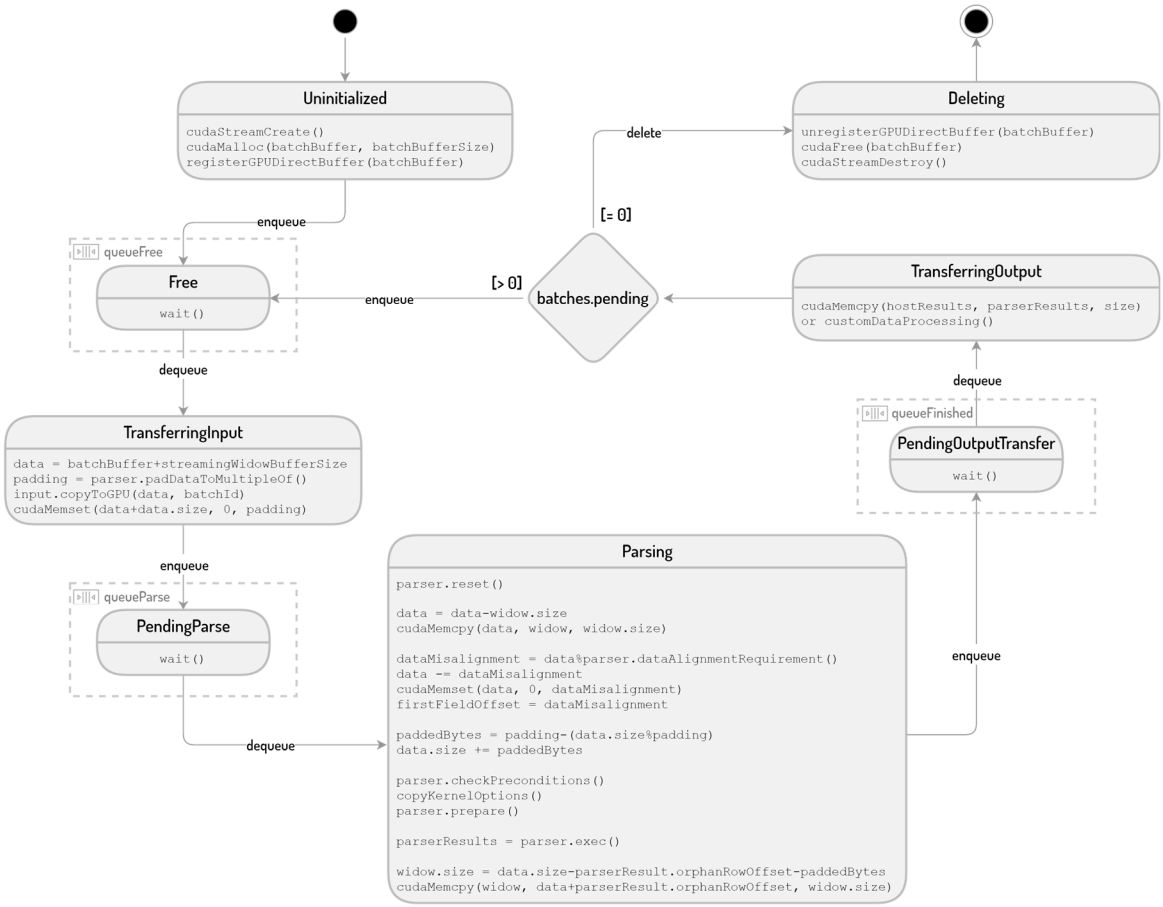

Figure 31: State diagram of a WorkStream item and the three WorkQueues

The states a WorkStream goes through in its lifecycle and the tasks it performs to realize our streaming approach are visualized in a state diagram in Figure 31. At initialization, multiple WorkStream items are created to fill the processing pipeline. The number of created items is controlled by *streamingParallelism* and defaults to four. Besides the one item being currently parsed, one item being fully transferred and waiting for parsing, and one item's results being transferred back to the host, this leaves one more item to be transferred to GPU memory.

It then goes through multiple stages for every batch it gets assigned to until there are no more batches left in the input data. To manage multiple items waiting in the same state we use WorkQueues (first-in-first-out) in a separate host thread that automatically call the next processing stage for the top item or, if the queue is empty, wait for a new item to be queued for processing.

In the following, we describe each stage of the processing pipeline.

Free State. Items in the *Free* state are queued and waiting to be assigned a new batch of chunks to.

Transferring Input State. In *TransferringInput* the appropriate input data is copied to GPU memory. The data is then padded with NULLs to avoid additional branches in the kernel for checking data validity when reading 128 bytes of data in a warp, as described in the Thesis Approach chapter.

Pending Parse State. Once the item is fully transferred, it waits in *PendingParse* for the previous WorkStream to finish parsing and deserializing.

Parsing State. In *Parsing*, the parser's internal state is first reset. The orphan data from the previous batch, now referred to as the widow, is prepended to the data in the batch buffer. To avoid a memory alignment error, the data pointer is then realigned to a four byte address with up to three additional leading bytes, which the kernel will skip. Once the parser's pre-conditions are checked for the new data and the new KernelOptions copied to constant memory, the parser is executed and the results saved to *parserResults*. Lastly, the orphan part of the batch buffer is copied to the dedicated widow buffer on the GPU.

Pending Output Transfer State. The item now waits as *PendingOutputTransfer* in the last queue for its results to be processed.

Transferring Output State. Results are either copied to the host's main memory or processed further on the GPU in *TransferringOutput*.

The batch is then considered fully processed and the WorkStream item is either put back into the *Free* queue or deleted if there are no more batches left to be assigned to WorkStream items.

4.2.3 Time Breakdown of Processing Stages

In Figure 32 we give an overview of the relative performance costs of every step of our Fast Mode approach that we outlined the steps for in our Thesis Approach when parsing and deserializing. We use a synthetic 1 GB CSV test data set with a chunk size of 2048 bytes, consisting of three `uint4` specified columns, each comprised of four digits:

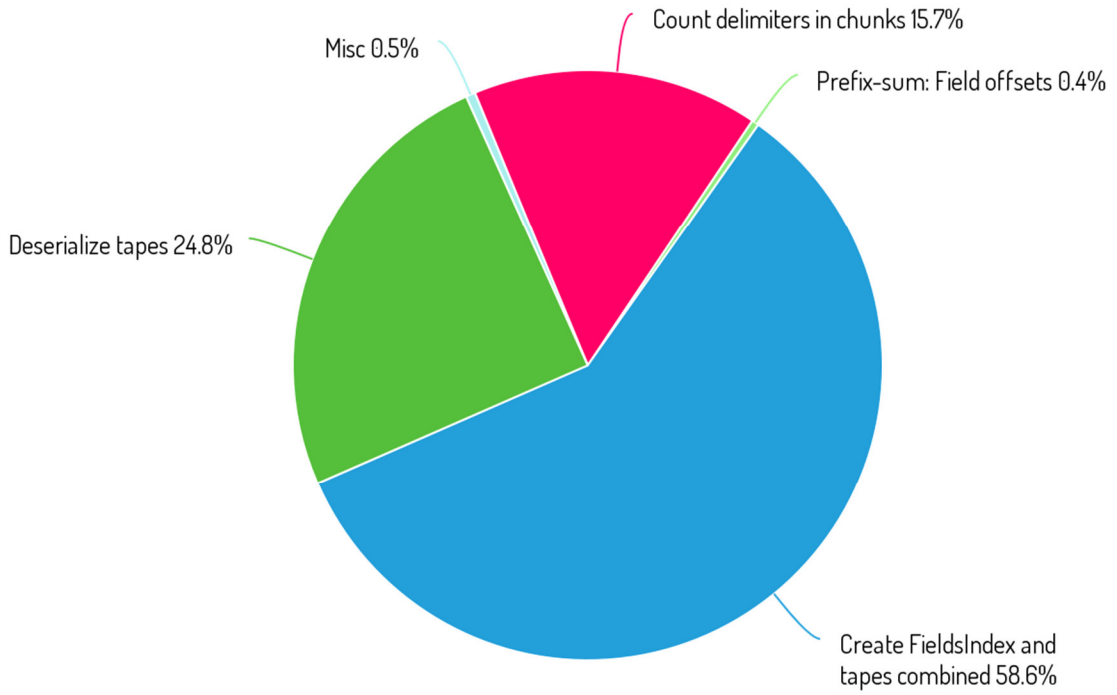

Figure 32: Relative performance costs of Fast Mode's steps on V100

Reading the entire input data, while counting each chunk's delimiters using a tree-reduction and `__shfl_down_sync()` within a warp, and writing out its results to global memory, only accounts for 15.7% of the parsing time. In comparison, time spent on deserializing the tapes is only 24.8% but not only requires reading in the tapes, which are collectively almost the same size as the input data read during

delimiter counting, but also deserialization of three `uint4` fields and writing them to global memory, which are collectively about one fourth of the input's data size. As expected, due to the complexity of the kernel and the amount of data involved, the combined steps for creating the FieldsIndex and tapes accounts for the majority of the parsing time with 58.6%. The device-wide prefix-sum calculation of the counted delimiters per chunk is just 0.4% and the remaining 0.5% are spent on miscellaneous data management operations, e.g. copying individual tape addresses to global memory.

4.2.4 Limitations

In addition to the above discussed limitations, our prototype contains a number of further limitations that we introduced due to time constraints. We document them in the following, so that they can be addressed in a production-ready implementation.

Escaping. For the Quoted Mode, we exploit the fact that a character is considered quoted whenever the number of preceding quotation marks is uneven. However, we do not take non-standard escape characters, i.e. a backslash for escaping a quotation mark, into account.

Missing Fields. For optimization purposes, our algorithm treats record delimiters as field delimiters. Although not standard-compliant, this relies on the assumption that all rows contain an equal amount of columns, i.e. trailing columns that are empty were not trimmed from the output data.

RDMA_COPY_SPLIT_SIZE. During testing we found some Mellanox kernel drivers to throw errors when trying to copy large blocks of data with one operation. This would happen when setting *streamingBatchSize* above approximately 528 MB and then trying to read a batch from the remote RDMA server. To circumvent this behavior, we added `RDMA_COPY_SPLIT_SIZE` to RDMAClient and hardcoded its value to 500 MB. All copy requests with a size larger than that will be automatically split into several smaller RDMA operations.

MAX_COLUMNS. To quickly look up a tape's width inside the kernel, we use the *tapeWidth* property inside the KernelOptions in constant memory. Due to time constraints, that property is a basic static array of size `MAX_COLUMNS`. Dynamic

arrays need pointers but pointers become invalid when copied from host memory to GPU memory, require additional management, and are not available in constant memory. We therefore opted for this macro, defined in a global header file. By default, its value is 32, limiting the maximum amount of columns in the input data to 32 columns. For input data with more columns, this macro needs to be increased accordingly. Additionally, *tapeWidth*'s underlying data type is `uint8_t`, limiting a column's length of type string to 255 characters.

Malformed Numbers. For optimization purposes, the deserialization of numbers only contains a small number of validity checks. Aside from the dot character for decimals, non-digit characters are skipped over during deserialization and treated as zeroes that might influence the converted number if they are between valid digits.

Negative Numbers. Deserialization to signed integer data types was skipped due to time constraints. Similarly, the FloatColumnType ignores leading dash characters.

Limited CUDA Copy Engines on PCIe. CUDA on desktop-grade GPUs only has two copy engines for PCIe, one for host-to-device transfers and one for device-to-host transfers, i.e. no two operations in the same direction can be performed in parallel and need to be queued up. Since operations like `cudaMalloc()` are synchronous by nature and `cudaMemset()` is internally implemented as a copy kernel, operations like these cannot overlap with host-to-device transfers either and need to be queued up as well. This introduced limitations of what the parser can do on the host side while another WorkStream copies a large chunk of input data to the device over PCIe, e.g. the above addressed ThrustMemoryPool to avoid a thread stall during parsing. Additionally, this also limits operations non-*TransferringInput* WorkStream items can do, particularly data padding and data alignment. Any such operations are therefore moved to the *TransferringInput* stage. However, NULLing the leading bytes for parsing after realigning the data pointer requires knowing the size of the widow, which is not yet available during the *TransferringInput* stage but would cause a similar thread stall if done in the *Parsing* stage. As a workaround, we therefore deferred this `memset()` to the kernel itself to the very first lane of the first chunk of a batch.

Windows-Style Line Endings. While a Unix-style line ending simply consists of a single byte, i.e. the newline character `\n`, Windows-style line endings consist of an additional preceding byte, i.e. the carriage return character `\r`. Since our parser treats record delimiters equal to field delimiters using a single byte, CSV data that uses these Windows-style line endings will cause the last field's value to have a trailing carriage return character for string column types. Enabling the more expensive `parseStrings` option could be used to remove this trailing `\r` in such situations. Number type fields will simply ignore this character during deserialization.

Tape Width Lookup in Quoted Mode. Since our focus was primarily on optimizing the Fast Mode, one particular optimization was left to be done when creating the tapes from the FieldsIndex in Quoted Mode. Fields are copied by column in this mode, i.e. lanes in a warp access columns in sequence. When looking up the tape width for their column to copy the field to the correct offset on the tape, each lane accesses the KernelOption's *tapeWidth* property, an array of column's tape widths. Since the KernelOptions are located in constant memory and every lane accesses a different address simultaneously, this causes 32 serialized constant memory accesses instead of one. While the subsequent global memory copy to the tape is slow in comparison to the constant memory read latency, it still is significant enough to optimize.

Infinity's Maximum Resource Usage. The RDMA wrapper library uses hardcoded values for the maximum length of the two completion queues during their initialization. Unfortunately, they cannot be changed during runtime and are both set to 16531, which is the global maximum for completion queue lengths the InfiniBand kernel driver allows on our testing machines. This hindered CUDAFastCSV from working with RDMA if any other RDMA application was already running on either machine. The fields `SEND_COMPLETION_QUEUE_LENGTH` and `RECV_COMPLETION_QUEUE_LENGTH` are available in the `infinity::core::Configuration` namespace and need to be recompiled with a value of, e.g., 2 or be made non-const to allow CUDAFastCSV to change their values during startup.

5. Evaluation

In this chapter we evaluate the performance of parsing CSV files on GPUs. First, we describe our experiment setup. Then, we give an overview of our CPU and GPU baselines. After that, we present and assess our measurement results. We divide our measurements into six categories: Implementation Strategies, Tuning Parameters, Databases and Parsers, I/O, Quoted Mode, and Hardware Scalability. Finally, we discuss our lessons learned.

5.1 Experiment Setup

The following sections give an overview of our testing conditions and their configuration particulars.

Hardware

We used two identical nodes for the majority of our testing, referred to as *Node1* and *Node2*. A third node was used for NVLink related evaluations, referred to as *NodeNVLink*.

Node1/2. Each system has a x86-64 based Intel Xeon Gold 5115 CPU (10 cores with hyper-threading, 2.4 GHz base clock, 3.2 GHz turbo clock) that supports SSE 4.2 string and text instructions and is running Ubuntu 16.04 with a total of 94 GB of DDR4-2400 memory, installed in a six-channel configuration. Each node has a single Nvidia Tesla V100-PCIe GPU with 16 GB of HBM2 memory, that is connected via a PCIe 3.0 bus with 16 lanes. The installed Mellanox ConnectX-4 MT27700 InfiniBand EDR network adapter is RDMA-capable with two ports and supports 100 Gbit/s of theoretical bandwidth per port. Both nodes' InfiniBand adapters are interconnected via a Mellanox SB7700 switch with 100 Gbit/s EDR.

NodeNVLink. This system is an IBM AC922 (8335-GTH) with 2x IBM Power9 CPUs (each 16 cores with SMT, 2.3 GHz base clock, 3.8 GHz turbo block), running Ubuntu 18.04 with a total of 256 GB of DDR4-2666 memory, installed in an eight-channel configuration. The system uses an NVLink 2.0 interconnect to its 2x

Nvidia Tesla V100-SXM2 GPUs with 16 GB of HBM2 memory. For our tests, we use only one NUMA node, i.e. a single GPU and CPU with 128 GB host memory.

Methodology

We measure and average all benchmarks over ten runs with the help of high-resolution timers. For GPU-related measurements, we adhered to Nvidia's recommendations when benchmarking CUDA applications [30]. The time for initialization of processes, CUDA, or memory, is not included in these measurements. All input files are read from the Linux in-memory file system *tmpfs*.

With the exception of NVLink-related measurements, we note that our measurements are stable with a standard error of less than 5% from the mean.

Datasets

For our evaluations we use a real-world, a standardized, and a synthetic dataset.

NYC Yellow Taxi Trips. This dataset contains records of taxi trips in New York City for the first quarter of 2019 [31]. It is provided by the City of New York. The dataset is split into CSV files for each month. We combined the CSV files for January, February, and March into a single CSV file that is 1.9 GB in size with 22.5 million records. Each record is made up of 18 fields, of which 14 are numerical types, with short and consistent record lengths. Because of CUDAFastCSV's limitation to Unix-style line endings (`\n`), we replaced the Windows-style line endings (`\r\n`) with Unix-style line endings. The last column, *congestion_surcharge*, is empty for most records in the original dataset. Since C/C++ does not natively support nullable primitive data types, we replaced these empty fields with `0` in our CSV file. Additionally, since C/C++ does not have a native data type for date-times, we deserialize the two date-time fields to ISO 8601 formatted date-time strings, e.g. *"2019-01-29 16:25:38"*.

TPC-H Lineitem. The H-variant of the TPC benchmark consists of a suite of business oriented ad-hoc queries and concurrent data modifications, specified by the Transaction Processing Performance Council [32]. The large volumes of data populating the test database have been curated to have broad industry-wide

relevance. We use the suite's generated CSV file for its *LINEITEM* table from the TPC Benchmark H revision 2.18.0 with 1x scaling. The generated CSV file uses the pipe-character instead of a comma to separate the fields. It is 719 MB in size with over six million records. Each record is made up of 16 fields of various data types and string fields of varying lengths. Similarly to the NYC Yellow Taxi dataset, we deserialize the three date fields to ISO 8601 formatted date strings.

int_444. For a more controlled test environment for evaluating individual parameters and scaling, we created a synthetic CSV test file. It consists of three numeric fields per record, each comprised of four random digits (i.e. 0000-9999) that need to be parsed and deserialized to a two byte unsigned integer data type (`unsigned short`). Since our measurements are evaluated in GB/s, this represents a balanced middle ground between input size and the total number of fields that need to parsed and deserialized. The chosen length is long enough to not fit into a compact one byte integer data type but short enough to not make optimal use of the two byte data type, while still requiring a significant amount of deserialization work and representing a significant output size. Unless otherwise noted, the CSV file is 1 GB in size with 70 million records that we deserialize to approximately 400 MB of output data.

Databases and Parsers

We compare CUDAFastCSV to four CPU and two GPU baselines. These consist of three databases, two state-of-the-art parsers for CPU and GPU, and a data interchange format. SQL schemas of the databases can be found in the Appendix.

OmniSciDB (v5.1.2). A GPU-accelerated database that utilizes GPU processing power to return SQL query results [33]. We bulk-load our *TPC-H* and *NYC Yellow Taxi* datasets into temporary tables residing in main memory. Column types of the table schemas are all marked `NOT NULL` and chosen as small as viable (e.g. `TINYINT`). For the import, we set the *quoted* parameter to *false* to improve processing speed. Note, however, while OmniSci is a GPU-accelerated platform, its CSV import is entirely executed on the CPU. It utilizes all available CPU cores on the host system for this import [34]. For measurements, we note the query processing time reported by OmniSciDB. Before benchmarking, we import the data fully once for warm-up. On every benchmark run, the table is truncated first.

PostgreSQL (v12.2). A widely popular relational database management system [35]. We bulk-load our *TPC-H* and *NYC Yellow Taxi* datasets into main memory residing tables. Since PostgreSQL does not support memory tables, we created an additional PostgreSQL table space that is located in the Linux in-memory file system *tmpfs*. Column types of the table schemas are all marked `NOT NULL` and chosen as small as viable and supported (e.g. `SMALLINT`) and set to fixed lengths when possible (e.g. `CHAR(10)` for dates). For measurements, we note the query processing time reported by PostgreSQL. Before benchmarking, we import the data fully once for warm-up. On every benchmark run, the table is truncated first.

HyPer DB (v0.5). Hyper DB is a main-memory-based relational database management system, developed by researchers at TU Munich [36]. We bulk-load our *TPC-H* and *NYC Yellow Taxi* datasets into tables that are in-memory by design. Column types of the table schemas are all marked `NOT NULL` and chosen as small as viable and supported (e.g. `SMALLINT`) and set to fixed lengths when possible (e.g. `CHAR(10)` for dates). For measurements, we note the query processing time reported by HyPer. Before benchmarking, we import the data fully once for warm-up. On every benchmark run, the table is truncated first. Note, however, this version of HyPer does not utilize the improved *Instant Loading* approach by Mühlbauer et al. [7] as presented in the Related Work chapter of this thesis. That implementation has since been integrated into the commercial analytics software *Tableau Server* by *Tableau*[25], which do not provide an academic license for this particular product.

ParPaRaw. A massively parallel algorithm implementation for parsing delimiter-separated data formats on GPUs, presented by Stehle and Jacobsen [37]. We use the binaries custom-tailored to our two datasets that were provided to us by the authors. Measurements include the time for reading the datasets from RAM and copying the parsed data from GPU memory to a pre-allocated and pinned memory buffer on the host. The time taken for initializing CUDA and allocating and pinning the host buffer is not included in these measurements. The binaries were compiled using GCC 8.3.0 and CUDA 10.1 with `-O3` optimization flags for both. We note that the primary author of *ParPaRaw* ran the benchmarks on our test system and validated our measurements.

[25] Tableau. Faster analytics with Hyper. https://www.tableau.com/products/new-features/hyper

RAPIDS cuDF (v0.14.0). A GPU library aimed at data engineers and data scientists to easily accelerate workflows using CUDA and the *Apache Arrow* [38] columnar memory format [39]. We implemented a test program in C++ that, using cuDF, reads the datasets from the Linux in-memory file system *tmpfs*, parses and deserializes them to the Apache Arrow format, and copies results back to the host's main memory. For its configuration, we set the *quoting* parameter to `quote_style::NONE` to improve processing speed. The field types are configured as small as viable and supported (e.g. `int16`). For every column, the memory area of the resulting Apache Arrow data is directly copied from GPU memory to a pre-allocated and pinned memory buffer on the host. We use the GCC implementation of `std::chrono::high_resolution_clock` to time our results. The time taken for initializing CUDA and allocating and pinning the host buffer is not included in these measurements. Before benchmarking, we parse and deserialize the data fully once for warm-up. The test application was compiled using GCC 8.3.0 and CUDA 10.1 with `-O3` optimization flags for both. It is worth noting that this version 0.14.0 of cuDF has a completely new and improved CSV parser for its CUDA implementation[26].

csvmonkey (v0.1). A vectorized, zero-copy CPU-based CSV parser that utilizes SSE 4.2, written in C++ [8]. As of this writing, it leads Ewan Higg's microbenchmark shootout of 24 CSV parsers [40]. csvmonkey uses a single thread for parsing and deserialization. We implemented a test program in C++ that reads the datasets as memory mapped files from the in-memory file system *tmpfs* and then parses and deserializes all fields. The fields are deserialized to data types as small as viable and supported (e.g. `unsigned short`). For numeric conversions, csvmonkey uses the *Qi* numeric parser implementation for `double` from Boost's Spirit library[27]. Accordingly, we extended their implementation to allow deserialization to `float`, `(u)int32`, and `(u)int16`. An implementation for converting numbers to one byte sized numeric data types is not provided by Qi. Every field is directly deserialized into its appropriate address in a pre-allocated memory buffer. We use the GCC implementation of the `std::chrono::high_resolution_clock` to time our results. The time taken for

[26] csv_gpu.cu https://github.com/rapidsai/cudf/commit/6d3ed596ce30135226f0bf8c5576d5b585262268

[27] Boost C++ Libraries. Spirit Qi Numeric Parsers. https://www.boost.org/doc/libs/1_73_0/libs/spirit/doc/html/spirit/qi/reference/numeric.html

memory mapping the input file and allocating the result buffer is not included in these measurements. Before benchmarking, we parse and deserialize the data fully once for warm-up. The test application was compiled using GCC 8.3.0 with `-O3 -msse4.2` optimizations for aggressive inlining and copy elision of return values (*RVO*), which we verified in the assembly code.

I/O

In this section, we stream the input data over various I/O sources to compare performance against the potentially transfer bound end-to-end parsing from the previous section, Databases and Parsers. We stream data with interconnects and with InfiniBand using two datasets. In contrast to end-to-end parsing, results are not copied back to the host's main memory.

On-GPU. The input data already resides in GPU memory. This allows us to compare raw parsing and deserialization throughput of CUDAFastCSV for the two datasets without being transfer bound.

PCIe 3.0. The input data resides on the host's in-memory file system *tmpfs*. CUDAFastCSV streams this input data to the GPU for parsing and deserialization. This serves as an upper bound for I/O devices on the host, including NICs and SSDs, as they would all be limited by the interconnect's bandwidth, even if the devices themselves were capable of faster transfer rates. We stream the *TPC-H* dataset with a *streamingBatchSize* of 50 MB and the *NYC Yellow Taxi* dataset with a *streamingBatchSize* of 100 MB.

NVLink 2.0. Similar to the PCIe 3.0 setup, the input data resides in the host's main memory and CUDAFastCSV streams this input data to the GPU for parsing and deserialization. In comparison to PCIe 3.0 and its practical bandwidth of just 12 GB/s, however, NVLink 2.0 can transfer with up to 63 GB/s on our *NodeNVLink* system and with a latency of almost half [21]. We stream the *TPC-H* dataset with a *streamingBatchSize* of 250 MB and the *NYC Yellow Taxi* dataset with a *streamingBatchSize* of 300 MB.

RDMA with GPUDirect. A CUDAFastCSV instance on *Node1* acts as the RDMA file server with the input file completely loaded into pinned host memory and registered with the InfiniBand device. The CUDAFastCSV instance on *Node2*

streams the input data directly from *Node1* using RDMA and directly into the GPU's memory using GPUDirect. This bypasses the CPU and the host's main memory. We stream the *TPC-H* dataset with a *streamingBatchSize* of 75 MB and the *NYC Yellow Taxi* dataset with a *streamingBatchSize* of 100 MB. We observed slow but rising transmission rates on newly established InfiniBand connections. We speculate this is due to the network connection's initial congestion control. To mitigate this, we run the benchmark without measurements several times first to allow the connection between the two nodes to ramp up its transmission rate to the network's maximum capacity before benchmarking.

5.2 Results

In this section we show performance results divided into six categories: Implementation Strategies, Tuning Parameters, Databases and Parsers, I/O, Quoted Mode, and Hardware Scalability. We assess the results and explain their cause. For better comparison of data from all sections, all measured times were converted to their throughput value in GB/s with respect to the input's size.

5.2.1 Implementation Strategies

In this section, we circle back to the various challenges with their proposed solutions we presented in our Thesis Approach chapter and discuss their evaluation. Unless otherwise noted, the input data already resides in GPU memory and its deserialized output data is written to GPU memory.

Parallelization Strategy: Access Patterns

To analyze the viable access patterns we presented in our thesis approach and identify chunk sizes that allow for the most optimal loading and processing, we implemented several kernels that each load chunks at different sizes for a synthetic 1 GB CSV test dataset, consisting of eight columns, each comprised of three characters, and present their measurements in Figure 33. To avoid compiler optimizations, every kernel counts the number of `\n`-characters in its chunk.

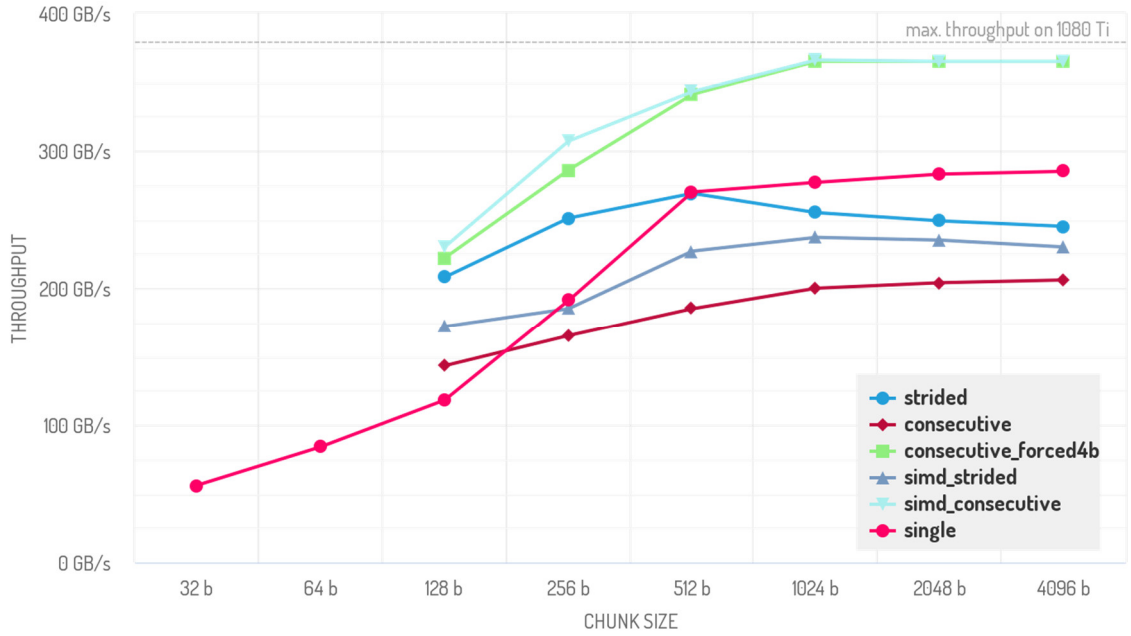

Figure 33: Throughput for chunk size with various access patterns on GTX 1080 Ti

We can see that simply increasing the chunk size above 128 and processing them in a 128 byte sized loop will provide an increase in throughput for every kernel.

single. Reads single bytes within a warp. As discussed in our Thesis Approach, reading single bytes within a warp is a slow strategy that cannot saturate the GPU's memory bandwidth at any chunk size.

consecutive. The *consecutive* kernel accesses 128 bytes per loop within a warp, with each thread accessing four contiguous bytes. Its maximum throughput is comparatively low and is approached quickly. Our analysis showed this is due to an increase in cache misses when processing warps in a thread block, causing additional latencies.

strided. The *strided* kernel accesses 128 bytes per loop within a warp, with each thread accessing four bytes in a stride of 32 bytes. Its throughput begins to decrease early after a chunk size above 512 bytes. Because its access happens in strides, its penalty for an increase in cache misses is lower than of the *consecutive* kernel.

simd_strided. Similar to the regular *strided* kernel, this kernel accesses 128 bytes per loop within a warp, with each thread accessing four bytes in a stride of 32 bytes. In contrast, however, the *simd* kernels use `__vcmpeq4()`, a CUDA function similar to SSE4.2's intrinsic instruction *pcmpestri*, with the four bytes as the input and `0x0a0a0a0a` as a mask (`0x0a` being the ASCII code for `\n`) to compare four

bytes at once. Like the regular *strided* kernel, its throughput begins to decrease early due to an increase in cache misses.

simd_consecutive. Like the *simd_strided* kernel but instead of threads accessing four bytes in a strided pattern, they are accessed contiguously. Unlike the *simd_strided* and the regular *consecutive* kernel, however, it saturates the available bandwidth and offers maximum performance. This is because the four bytes are loaded with one memory transaction, directly by the `__vcmpeq4()` function. This access pattern does not rely on the cache and simplifies the comparison operation.

consecutive_forced4b. To circumvent additional cache dependencies, like *simd_consecutive*, we access the four individual bytes by forcefully loading all of the four bytes as an `int` to a register first and then cast it to a `char` array. With this strategy we are able to saturate the GPU's memory bandwidth at a chunk size of just 1024.

We conclude that *simd_consecutive* and *consecutive_forced4b* achieve the best performance. The *consecutive_forced4b* kernel is our chosen strategy for CUDAFastCSV for loading and accessing chunks. While *simd_consecutive* offers similar performance, it is at the cost of more complexity, especially for a later implementation of succeeding operations.

Deserialization Strategy: Integer Deserialization Kernels

We compare the three different approaches discussed in our Thesis Approach (*Dynamic Parallelism*, *Grouped Warp Lanes*, *Columns with Maximum Lengths*) and identify the fastest deserialization strategy. We implement these approaches as kernels that deserialize a synthetic 1 GB CSV test data set using a pre-calculated FieldsIndex, consisting of only one column of various sized numbers, and present its results in Figure 34.

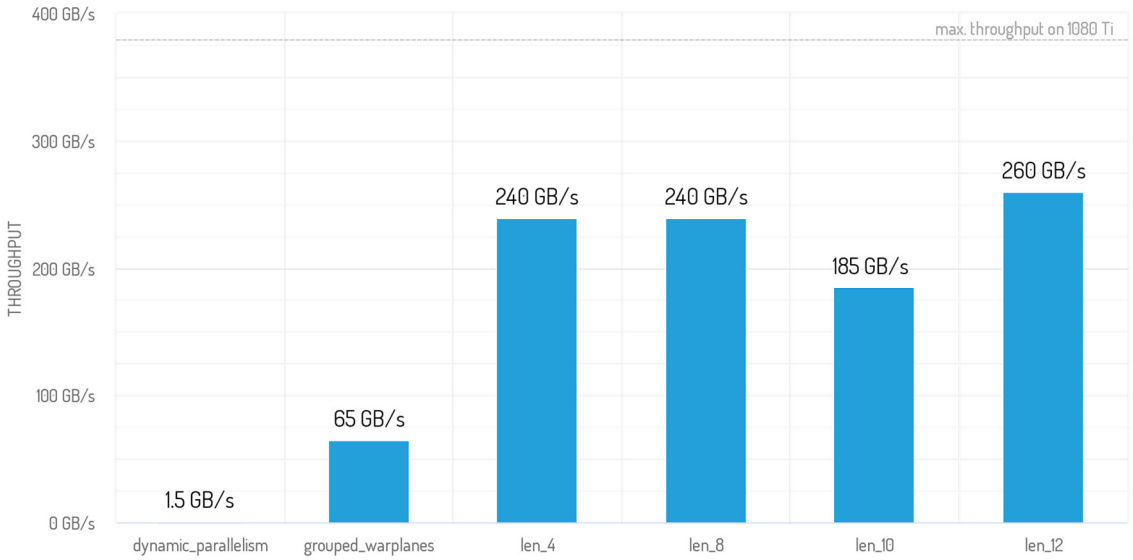

Figure 34: Comparing integer deserialization kernels on GTX 1080 Ti

As discussed in our Thesis Approach, the *Dynamic Parallelism* approach suffers from a lack of resources needed to manage pending kernel launches. The *grouped approach* achieves 65 GB/s but does not access the data in an optimal way and needs either additional synchronization or atomic operations between threads to calculate their sums. Our final approach, using *columns with maximum lengths*, provides a significant improvement and, depending on the input data's structure, a nearly 100% branch efficiency, tested with several specified column lengths. Column lengths that are a multiple of four provide comparable performance. Because memory access is aligned, with a column length of ten the kernel has to continuously account for the misaligned bytes of the field value.

We conclude that the column-based approach with specifying maximum column lengths achieves the best performance. It is our chosen strategy for CUDAFastCSV for deserialization.

Optimizing Deserialization: Transposing to Tapes

To improve deserialization performance for columns with various data types, we introduced deserialization with *tapes* in our Thesis Approach. Using our implemented column-based deserializer, we can now execute a type specific kernel for each tape, i.e. each column, and deserialize its values in parallel up until the first NULL byte or *tapeWidth* without the discussed performance penalty of having to

deserialize various data types in the input data. We illustrate this in Figure 35, which does not take the creation of tapes into account but simply measures the raw deserialization performance.

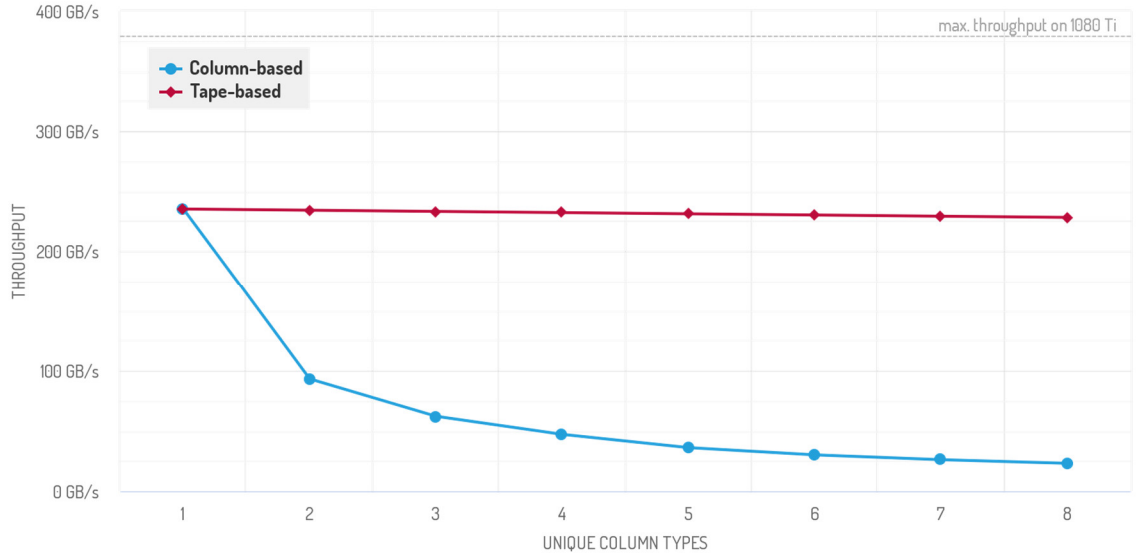

Figure 35: Column-based vs tape-based deserialization performance scaling on GTX 1080 Ti

5.2.2 Tuning Parameters

In this section, we evaluate several parameters for performance tuning and scalability that we presented in our Thesis Approach chapter. Unless otherwise noted, the input data already resides in GPU memory and its deserialized output data is written to GPU memory.

Impact of Block Size

In CUDAFastCSV, kernels require several GPU resources, including registers and shared memory, that are limited on the SM. Developers in CUDA specify a block size when launching kernels that, depending on the kernel's needs and the underlying architecture, can help maximize usage of the device's resources.

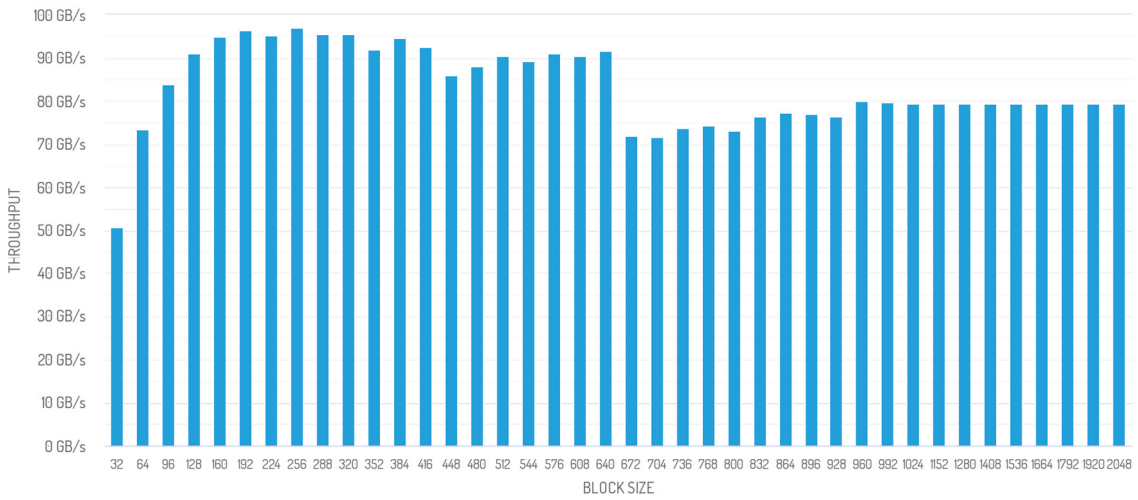

Figure 36: Impact of parameter *blockSize* for *int_444* on V100

In Figure 36 the most optimal block sizes are shown to be between 128 and 640.

The reason for the decreased performance of block sizes smaller than 128 is twofold. For a block size of 32 the SM's occupancy never exceeds 50%. While the architecture allows for up to 2048 threads per SM, the SM itself is limited to 32 thread blocks. A block size of 32 will therefore only ever occupy at most 1024 threads on the SM at all times. For the other block sizes leading up to the more optimal ones, the deserialization kernels are not able to achieve their maximum throughput when reading the tapes from global memory. The maximum number of blocks per SM, 32, simply does not contain enough threads with these small block sizes to effectively hide the memory latency for the deserialization kernels.

The sudden drop in performance when going from a block size of 640 to 672 is due to a lower occupancy caused by a lack of available registers needed to run multiple blocks on an SM simultaneously. In particular, the kernel in question uses 41 registers per thread, which amounts to 1312 registers per warp. CUDA allocates registers on a per-warp basis in multiples of 256, however [41], i.e. the kernel's warp actually occupies 1536 instead of just 1312 registers. Given the architecture's limit of 65,536 registers per SM, 42 warps could theoretically fit into an SM's register budget. However, CUDA schedules warps in groups of four [25], i.e. reducing the maximum number of warps to 40. A block size of 640 equals 20 warps, allowing CUDA to execute a second thread block on that SM simultaneously, since two of such blocks equal exactly the maximum of 40 warps. A block size of 672, however, equals 21 warps, leaving no spare register resources

for a second thread block. The performance of the larger block sizes then steadily rises again, since they make better use of the available resources until the throughput limit is hit at 960.

We conclude that, unless specified unusually low or high, *blockSize* does not have a large impact on performance when changed from its default value of 128.

Impact of Chunk Size

The choice of the *chunkSize* in CUDAFastCSV determines how much of the input data a warp processes. An increasing size requires more SM resources per warp but also reduces the overhead associated with scheduling, launching, and processing new thread blocks or warps.

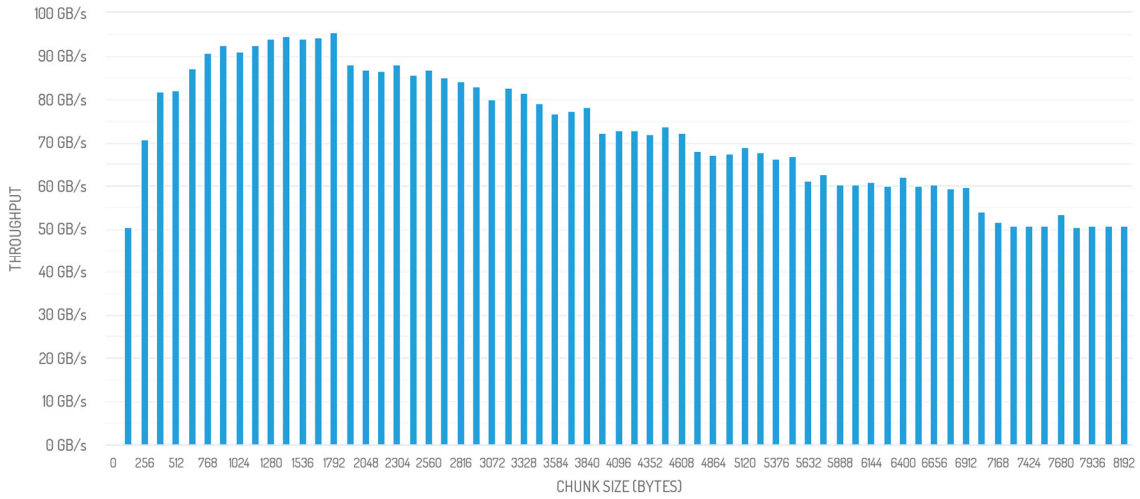

Figure 37: Impact of parameter *chunkSize* for *int_444* on V100

Figure 37 reflects what we previously discussed for access patterns in our parallelization strategy for small chunk sizes. The steep drops, e.g. after 1792, stem from one less concurrent thread block running on the SM due to a lack of available shared memory resources. A slight rise in performance before every drop shows the improved resource utilization of the available resources.

We conclude that the best *chunkSize* is 1024 bytes.

Impact of Input Size

Given a GPU's architecture, it can only fully utilize its resources when given enough workload. For small CSV files, we would need to use a small *chunkSize* to create enough work in the form of threads that can be scheduled to read and process input data to hide memory latencies. As previously shown, however, small *chunkSize* values significantly impact processing performance negatively. So it might not make sense to use a GPU based parser in such instances, especially when the input data needs to be transferred to and its results back from the GPU. Even for small input data that already resides on the GPU, there is still considerable overhead from kernel launches and synchronization to account for.

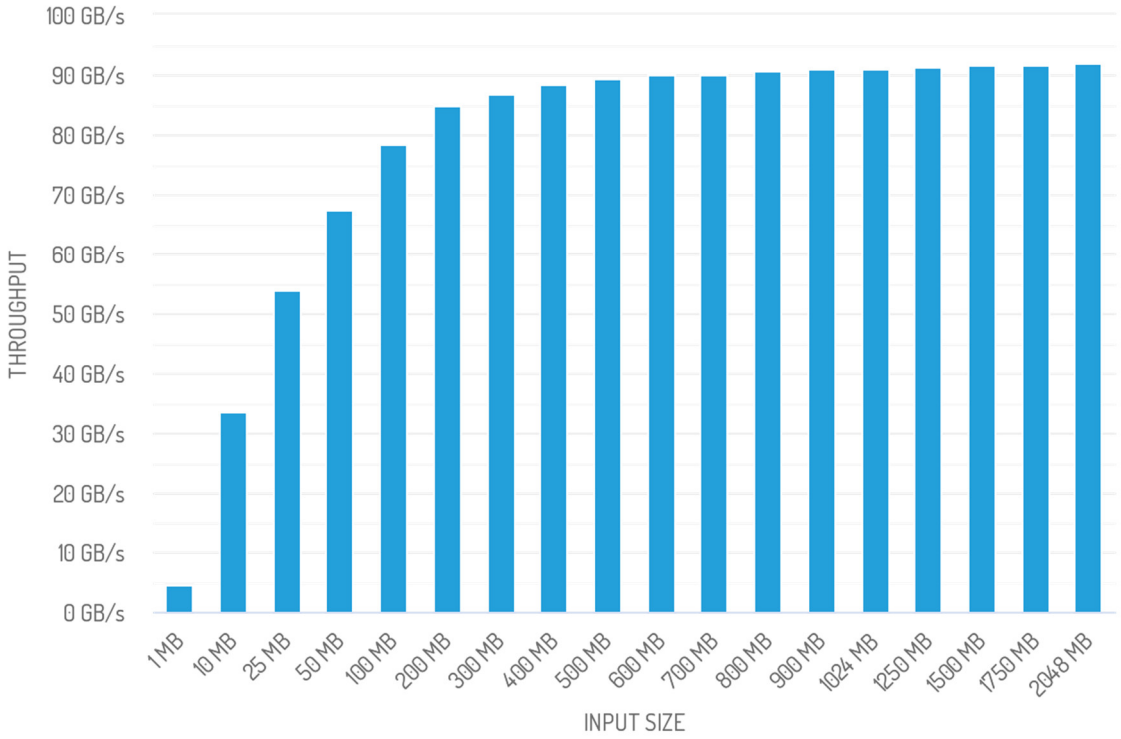

Figure 38: Performance scaling in relation to input size for *int_444* on V100

Figure 38 shows the ramp up of CUDAFastCSV's performance when given an increasingly large input file. While the 1 MB sized file only achieves 4.5 GB/s, the throughput already strongly increases with a 10 MB file to 33.6 GB/s and continues to rise until it approaches its limit of approximately 90 GB/s.

We conclude that even with just a 1 MB sized CSV file a case for loading data using the GPU instead of the CPU could be made in certain cases, while maximum throughput can be approached fairly quickly after just 100 MB.

Impact of Streaming Size

In CUDAFastCSV we can use *streaming* to begin parsing and deserializing of incoming parts of the input data without having to wait for the entire input data to be on the GPU first or for the input data to even fit into GPU memory. The input data is partitioned into smaller sized chunks, controlled by *streamingBatchSize*. Each chunk is transferred to the GPU, parsed and deserialized, and its results transferred back to the host's main memory. Processing and the transfer of chunks in each direction is interleaved. As shown in the previous experiment, *Impact of Input Size*, a small data size cannot utilize the GPU's processing capabilities to its maximum and, for streaming, results in a large amount of synchronizations over the interconnect. However, CUDAFastCSV cannot start parsing until the first chunk is transferred. So a too large *streamingBatchSize* might result in overall worse performance, due to the additional latency from the transfer of the first chunk of the input data and the transfer from the last chunk's result data.

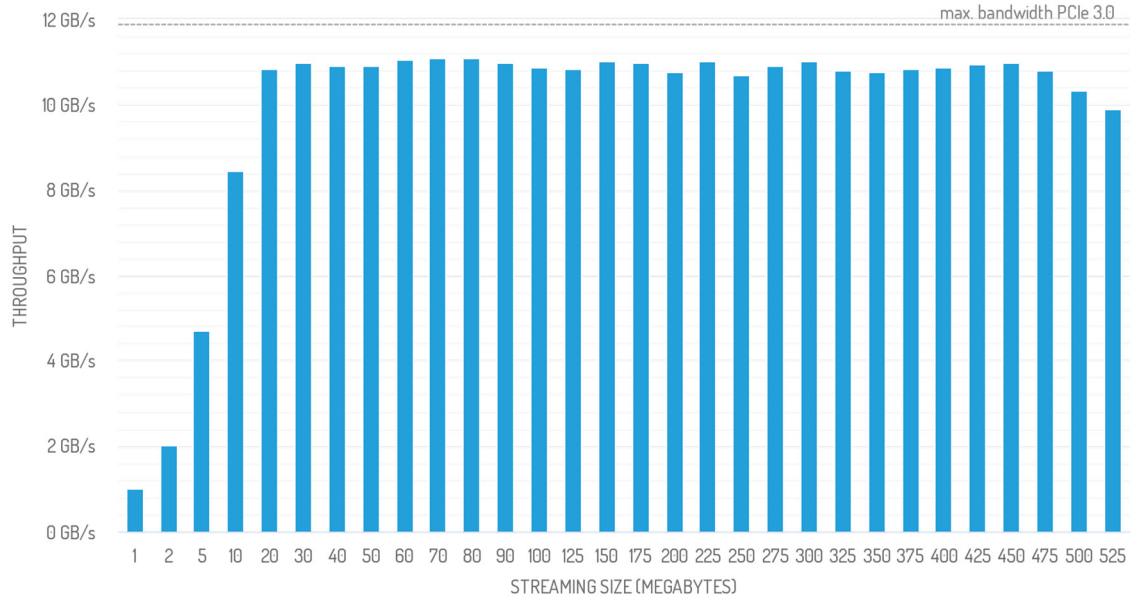

Figure 39: Impact of parameter *streamingBatchSize* for *int_444* on V100 over PCIe 3.0

In Figure 39 we indicate PCIe 3.0 as the baseline as it represents the maximum possible throughput in our experiment on *Node1*, regardless of a GPU parser's performance, due to its latencies and maximal throughput of approximately 12 GB/s. In our results the throughput scales almost linearly with the *streamingBatchSize* up until 10 MB before it hits its maximum of 11 GB/s at 20 MB. At 500 MB, the penalties of having a too large size over the interconnect begin

to show. For comparison, Figure 40 shows the same experiment over NVLink 2.0 on *NodeNVLink*. Ramp-up speed is very similar to PCIe 3.0 but keeps rising when the limitations of PCIe 3.0 would otherwise set in. Due to NVLink 2.0's higher bandwidth and lower latency, the negative impacts of a too large *streamingBatchSize* already begin to show at 400 MB. In comparison to PCIe 3.0's peak throughput of 11.1 GB/s, with streaming over NVLink 2.0 we achieve a peak throughput of 48.3 GB/s. Our implementation is not able to fully utilize the interconnect's available bandwidth due to CUDA's copy engine limitations described in the previous chapter, leading to transfer and processing delays from non-overlappable operations, and due to the overhead from data and buffer management required for streaming.

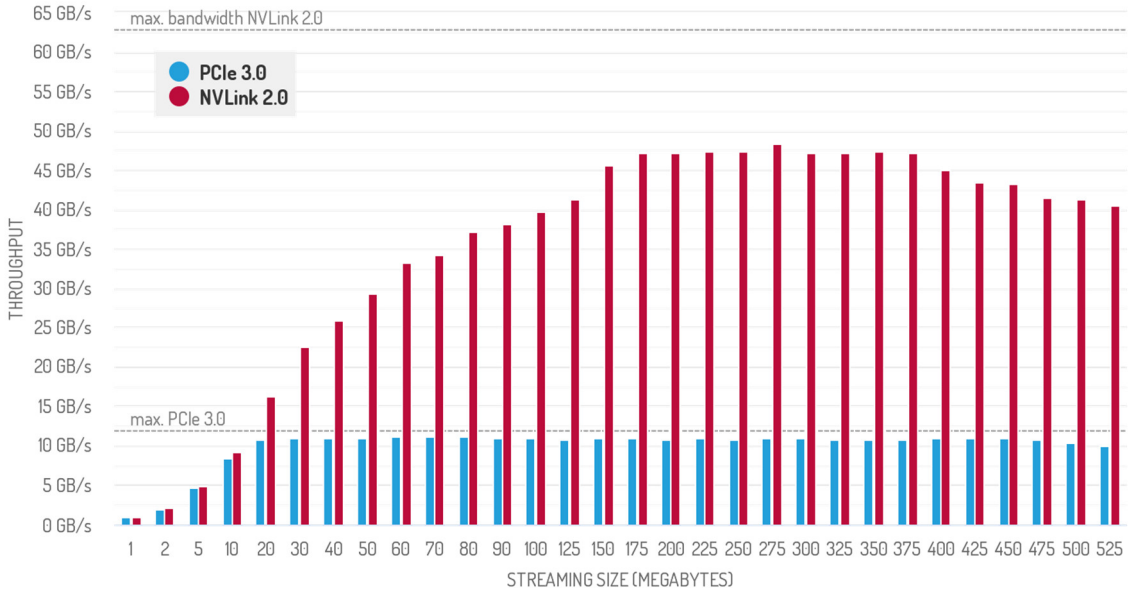

Figure 40: Impact of *streamingBatchSize* for *int_444* on V100 over NVLink 2.0 in comparison

We conclude that PCIe 3.0's bandwidth is saturated quickly and its best *streamingBatchSize* is already achieved at 20 MB. NVLink 2.0 exposes PCIe 3.0 as a bottleneck for end-to-end parsing in comparison.

Impact of Warp Index Buffer Size

The *warpIndexBufferSize* parameter in CUDAFastCSV limits the maximum number of found fields in all chunk segments within a warp and is used to reserve the kernel's shared memory space in Fast Mode or, in Quoted Mode, the required space in global memory for the FieldsIndex. It can be altered from its default, 2048

bytes, to increase parallelism when the underlying data characteristics of the CSV input data allow for it. As such, less shared memory resources are allocated per thread block, allowing for additional thread blocks to run concurrently on the SM.

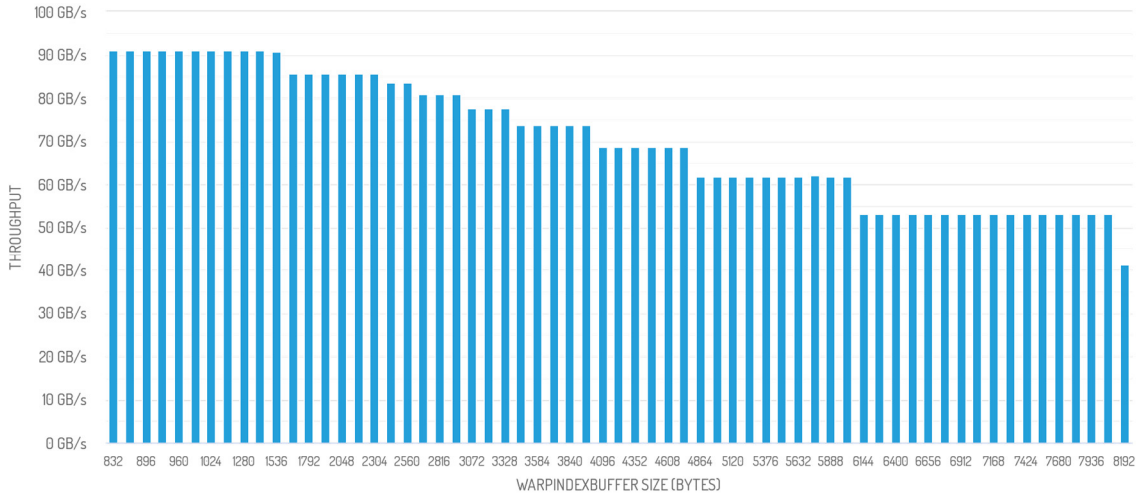

Figure 41: Impact of oversized *warpIndexBufferSize* for *int_444* on V100

Figure 41 illustrates this behavior as the amount of concurrent thread blocks steps down whenever the increasing size allocates too many resources. For a chunk size of 1024, the smallest viable *warpIndexBufferSize* for the *int_444* dataset is 832. Maximum throughput of around 90.9 GB/s is kept up until 1536. The default of 2048 falls into the 85.6 GB/s range. To accommodate for a worst-case scenario of only having empty fields in a 1024 byte chunk in any part of our *int_444* data, we would need a *warpIndexBufferSize* of 4096, which reduces our performance to 68.6 GB/s. Larger sizes reduce performance even further.

We conclude that the *warpIndexBufferSize* shows to have a large impact on performance, as it is dependent on the underlying structure of the input data.

5.2.3 Databases and Parsers

To evaluate end-to-end parsing performance of CUDAFastCSV, we benchmarked our approach against several implementations from different categories as described in our experiment setup. We use the *TPC-H* and *NYC Yellow Taxi* dataset, residing in the host's main memory, and measure the time until all deserialized fields are available in the host's main memory in an accessible and either row- or column-oriented data storage format. Since CUDAFastCSV is a

GPU-based implementation that heavily relies on the performance of the system's interconnect in this scenario, we include measurements not only over PCIe 3.0 (*Node1*) but also over NVLink 2.0 (*NodeNVLink*).

NYC Yellow Taxi

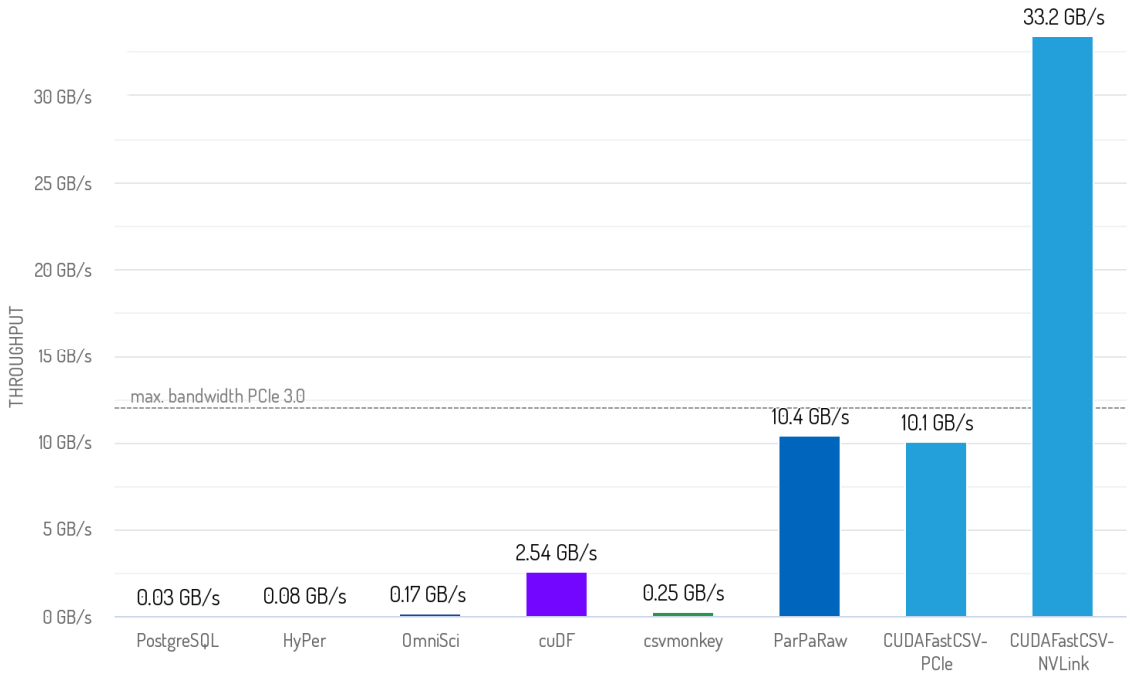

Figure 42: End-to-end performance comparison for NYC Yellow Taxi dataset

The performance numbers reported for parsing and deserializing the 1.9 GB from the *NYC Yellow Taxi* dataset in Figure 42 highlight the strength of CUDAFastCSV, which is only limited by the PCIe 3.0's available bandwidth. This is especially noteworthy, as deserializing includes nine floating point numbers and five integers out of the 18 total fields.

The GPU-based implementation, *cuDF* with its new and updated CSV implementation, still achieves just a quarter of the performance of CUDAFastCSV. All CPU-based approaches, i.e. *PostgreSQL*, *HyPer DB*, *OmniSciDB*, and *csvmonkey*, are slower by up to three orders of magnitude. CUDAFastCSV over NVLink 2.0 more than triples the performance over its PCIe 3.0 variant and is approaching I/O performance of DDR4 main memory [17].

Only *ParPaRaw* provides comparable performance to CUDAFastCSV. To determine if *ParPaRaw* is being limited by the interconnect in this instance, we

additionally measured its on-GPU throughput for this dataset and compared it to our implementation in Figure 43. *ParPaRaw* achieves 16.2 GB/s on-GPU throughput on our system. In comparison, our Quoted Mode measures 25.9 GB/s and our Fast Mode even 60 GB/s. Using our *early context detection* approach, we are able to reduce the overall amount of work, as we do not need to track multiple DFAs, and are less processing-intensive as a result.

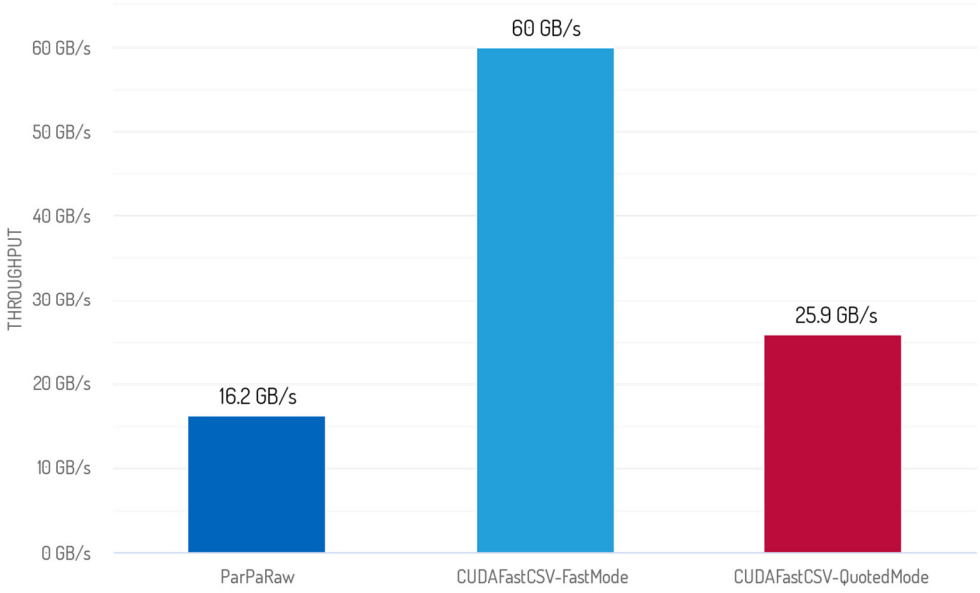

Figure 43: Comparing on-GPU throughput of ParPaRaw to CUDAFastCSV

TPC-H Lineitem

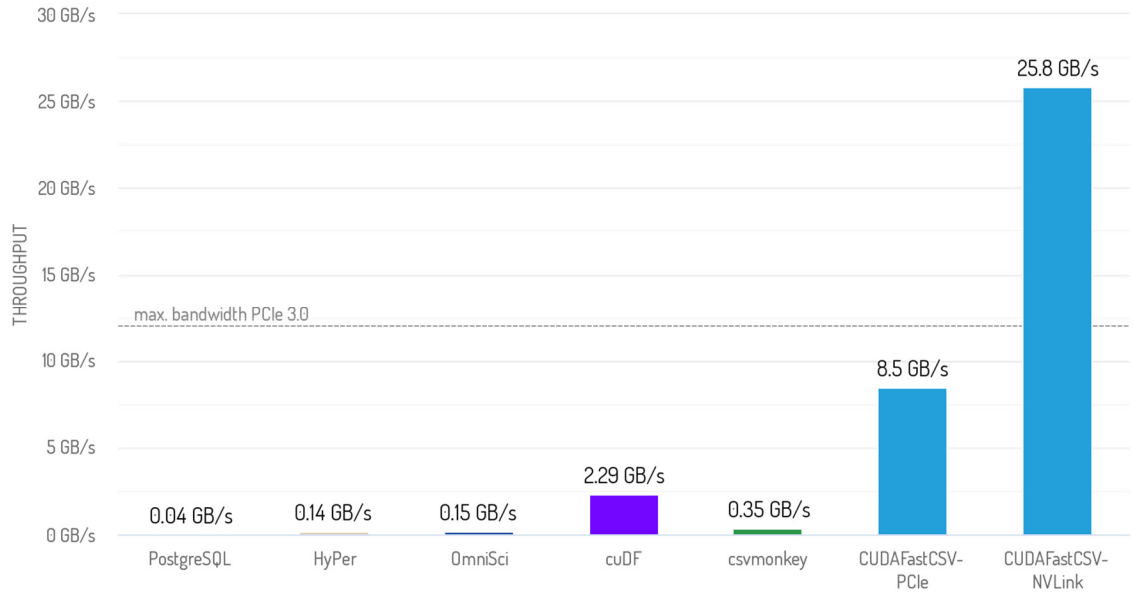

Figure 44: End-to-end performance comparison for TPC-H's lineitem dataset

For the 719 MB *TPC-H* dataset, Figure 44 shows CUDAFastCSV to be slightly slower when compared to the *NYC Yellow Taxi* dataset on both, PCIe 3.0 and NVLink 2.0. The bottlenecking factor for this dataset is found in the transfer of the larger result data back to the host, causing increasingly longer delays between streamed chunks. For every 100 MB chunk of *TPC-H* data transferred to the GPU, approximately 118 MB of result data needs to be transferred back to the host, while the *NYC Yellow Taxi* data only needs 93 MB per 100 MB. This causes delays in input streaming and during processing, as kernel invocations get hindered by data dependencies and synchronization. In the taxi dataset, when the last chunk is fully deserialized it has no pending output from preceding chunks waiting in the device-to-host pipeline. In comparison, the last chunk of the *TPC-H* dataset still has multiple preceding chunks waiting in the pipeline for their transfer.

RAPIDS cuDF, another GPU-based implementation, shows a similar drop in performance of approximately 10%. In contrast, some of the CPU-based implementations were able to significantly improve their performance for the *TPC-H* dataset, namely *HyPer* and *csvmonkey*, due to the smaller number of numeric fields that need to be deserialized.

Again, CUDAFastCSV over NVLink 2.0 can more than triple its performance in comparison to the PCIe 3.0 variant.

Unfortunately, for this dataset we were not able to get a *ParPaRaw* binary in time.

5.2.4 I/O

We present results for CUDAFastCSV with various interconnects and an RDMA with GPUDirect approach with our two datasets. In contrast to the previous section's setup, results are not copied back to the host's main memory but instead stored in GPU memory to avoid unrelated bottlenecks in the host's interconnect. These results are in a column-oriented data storage format and accessible for potential further processing on the GPU.

NYC Yellow Taxi

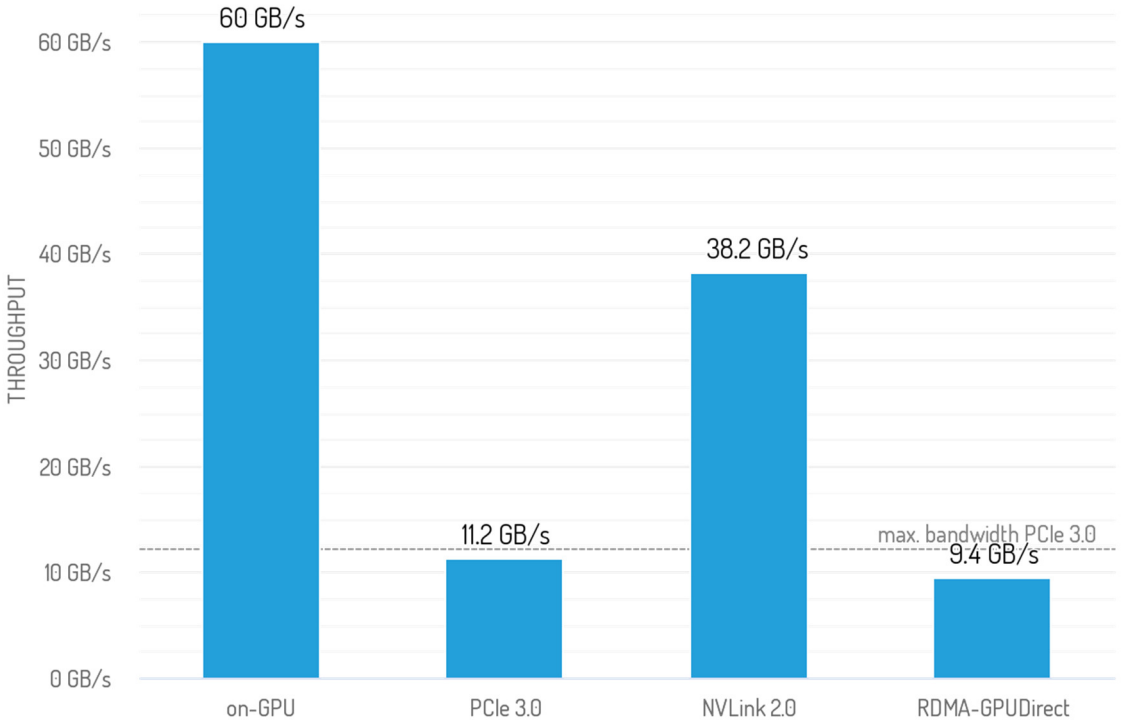

Figure 45: Interconnect streaming performance comparison for NYC Yellow Taxi dataset

The throughput for parsing and deserializing the *NYC Yellow Taxi* dataset when it is already in GPU memory is at 60 GB/s and serves as a baseline, representing the maximum possible performance an interconnect to the GPU could potentially achieve. As seen in the previous section, our implementation over PCIe 3.0 can fully saturate the bus with 11.2 GB/s. Again, throughput over NVLink 2.0 more than triples and shows the limitations of the PCIe 3.0 system in comparison. Our RDMA with GPUDirect approach, streaming the input data from a remote machine directly onto GPU memory over the internal PCIe 3.0 bus, shows 9.4 GB/s. Although InfiniBand's maximum throughput is 12 GB/s on our system, we observe the same overhead from GPUDirect RDMA as previous experiments [42].

TPC-H Lineitem

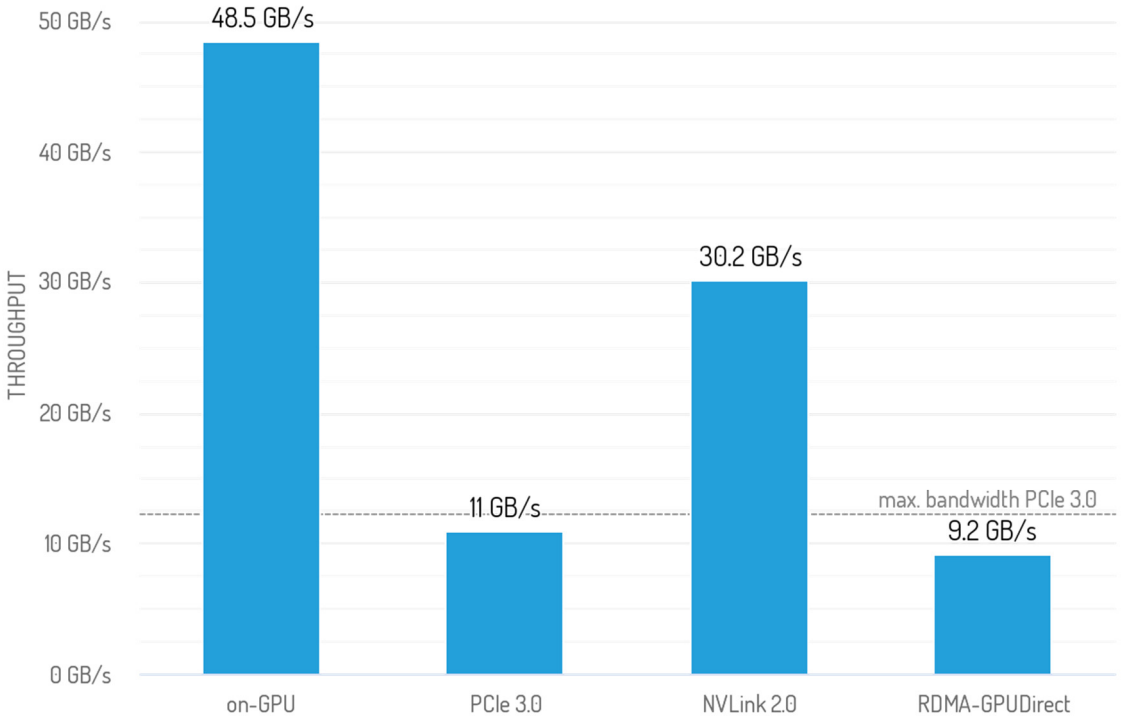

Figure 46: Interconnect streaming performance comparison for TPC-H's lineitem data

As with the *NYC Yellow Taxi* dataset, the baseline for the *TPC-H* dataset is established by measuring the throughput of the data when it is already in GPU memory. The measured 48.5 GB/s represent the maximum possible performance an interconnect to the GPU could potentially achieve. Similarly to the taxi dataset, PCIe 3.0 is saturated at 11.0 GB/s and NVLink 2.0 performance is almost triple in comparison. For the RDMA with GPUDirect approach we achieve similar performance at 9.2 GB/s for the *TPC-H* dataset. Overall, throughput for this dataset is slightly lower for the baseline and for every interconnect, due to the increased size of the result data and its consequences as described in the previous section.

5.2.5 Quoted Mode

In our thesis approach, we introduced the *Quoted Mode* as an alternative parsing mode that keeps track of quotation marks to create a context-aware FieldsIndex, using *early context detection*. As the main focus of our work was the default *Fast*

Mode, however, we want to include an illustration of the additional processing costs involved and show a comparison between the two modes for our three datasets in Figure 47.

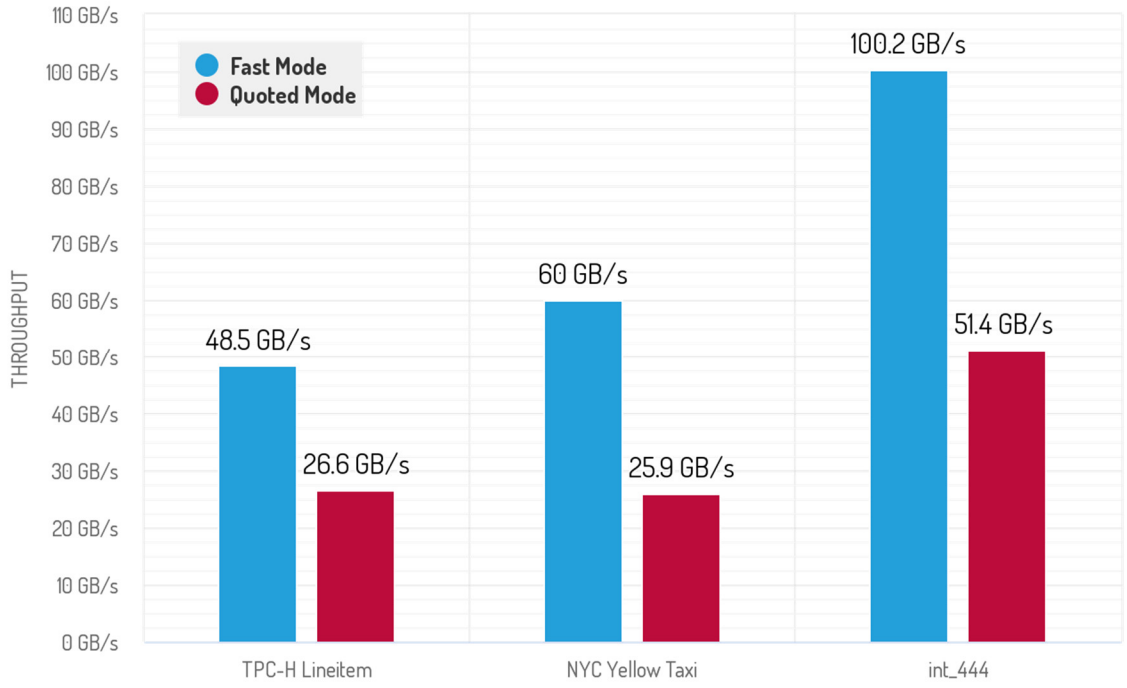

Figure 47: Comparison of Fast Mode and Quoted Mode on V100

For all three datasets, throughput is roughly cut in half. Writing out the large FieldsIndex to GPU memory and subsequent stream compaction is especially punishing in the *NYC Yellow Taxi* dataset with many more potential fields per MB than the *TPC-H* or the *int_444* dataset. Nevertheless, performance numbers are promising and since our main focus was the Fast Mode, there is much more optimization left to be done for the Quoted Mode.

5.2.6 Hardware Scalability

To compare desktop-grade GPUs with server-grade GPUs and the generational leap in hardware advancements and how both could further fare in regards to scalability in the future, we include measurements on a desktop-grade Pascal GPU from the preceding generation. The GPU is a Nvidia GTX 1080 Ti (GP102) with 11 GB of GDDR5X memory and a theoretical bandwidth of 450 GB/s. It has 28 SMs, each with 128 CUDA cores, totaling 3584 cores with a maximum clock of 1999 MHz. For comparison, *Node1*'s GPU is the succeeding Volta generation, a server-

grade Nvidia Tesla V100 with 16 GB of HBM2 memory and a theoretical bandwidth of 835 GB/s. It has 80 SMs, each with 64 CUDA cores, totaling 5120 cores with a maximum clock of 1380 MHz. While not accounting for any performance improvement from IPC or architectural advancements in Volta and its cores, the combined clock of all cores is slightly higher on the Pascal GPU at hand. Volta's memory bandwidth, however, is significantly higher.

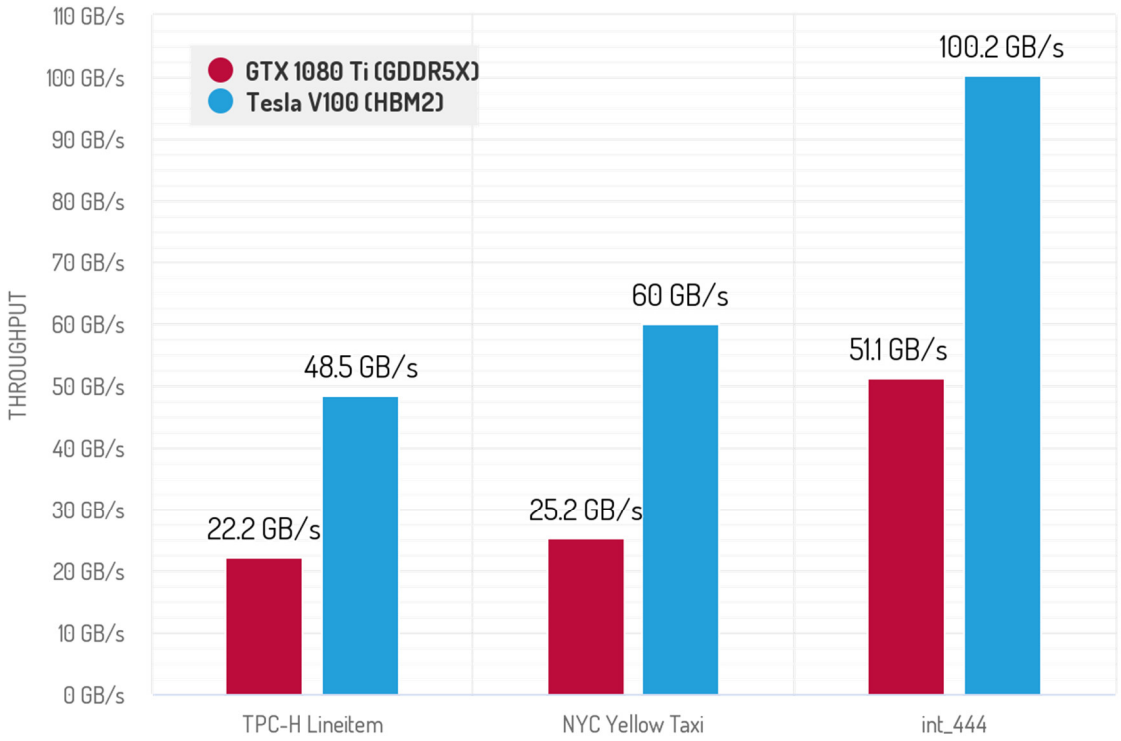

Figure 48: Performance improvement going from desktop-Pascal to server-Volta

The performance improvements shown for Volta in Figure 48 are at least 100% over Pascal for our three datasets. As expected, our analysis showed a considerable chunk of the performance improvement stems from the higher memory bandwidth and its efficiency. Further analysis we conducted showed the enhancements in instruction throughput and latency account for the other significant improvement in performance [22].

5.3 Discussion

In this section, we discuss the lessons we learned in our evaluation.

GPUs improve parsing performance. Our measurements show that parsing on the GPU improves throughput by 134x for the NYC Yellow Taxi dataset and 73x for the TPC-H Lineitem dataset, when compared to the fastest CPU parser. Thus, offloading parsing to the GPU can provide significant value for databases.

Parallelizing context-awareness of quoted data. We show that, using our *early context detection* approach, we are able to parallelize context-awareness in Quoted Mode and with 51 GB/s scale performance to rates necessary for fast interconnects.

Interconnect bandwidth limits performance. In all our measurements, PCIe 3.0 does not provide sufficient bandwidth to achieve peak throughput. Using NVLink 2.0 instead, the throughput increases by 2.8-3.4x. This improvement shifts the bottleneck to our pipelining strategy. Removing this limitation would increase throughput further by 1.6x.

Network streaming is feasible. We show that streaming data from the network to the GPU is possible and provides comparable performance to loading data from the host's main memory over PCIe 3.0. This strategy provides an interesting building block for data streaming frameworks.

GPUs can efficiently handle complex data format features. Features, such as quoting fields, decrease parsing throughput to 43-55% of the non-quoted throughput. However, this reduced throughput is still higher than the bandwidth provided by PCIe 3.0 and InfiniBand. Thus, the overall impact is no loss in performance. Only for faster interconnects would performance tuning have a practical impact.

Desktop-grade GPUs provide good performance per cost. For all our datasets, a desktop-grade GPU is sufficient to saturate the PCIe 3.0 interconnect. In contrast to a server-grade Nvidia Tesla V100 costing 7000 EUR in 2020, a desktop-grade Nvidia GTX 1080 Ti is only 10% of the cost at 700 EUR. Thus, only for NVLink 2.0 and files that are complex to parse does a server-grade GPU make sense.

6. Related Work

Our work is built upon concepts researched in other areas. In this chapter we present an overview of the related research areas that are relevant to the approach of this thesis.

Loading

As modern in-memory DBMSs can process millions of transactions per second [43], the question of how to actually get the data into the system first, to ultimately make use of this performance, becomes increasingly important.

CPU-Limited. Dziedzic et al. [3] evaluated CSV data loading performance of multiple modern and popular DBMSs (*database management system*) along several dimensions to understand various software and hardware limitations for such workloads. With a variety of hardware configurations and datasets, they provide an extensive analysis. They show that modern DBMSs are unable to saturate I/O bandwidth. Their evaluation shows that data loading is mostly CPU bound.

We implement an end-to-end parsing approach to offload these CPU bound workloads to the GPU and thereby saturate I/O bandwidth, leaving the host system's interconnect as the new bottleneck.

RDMA-Enabled. Fent at al. [28] show experimentally, that for modern high-performance systems, networking has become a performance bottleneck. They propose a high-performance communication layer for DBMSs that redesigns how data flows in and out of these systems. Among other things, it is based on RDMA for intra-datacenter communication. Their results show that with the help of InfiniBand RDMA, network bottlenecks for DBMSs can largely be eliminated.

Given their results, we utilize InfiniBand RDMA and GPUDirect as a data loading technique for GPU-based CSV parsing to avoid such network bottlenecks.

Vectorization

Mühlbauer et al. [7] already analyzed the problems of CPU-based CSV parsing and deserialization. Modern CPUs try to predict the outcome of *if-then* conditional branches for their pipelined architecture. When parsing CSV input, based on character-at-a-time comparisons, these branches can hardly be predicted. They observed that, amongst other things, the CPU pipeline needs to be flushed often due to constant branch miss penalties from mispredictions when doing so. This behavior can not only be observed during parsing but also during deserialization. By utilizing SSE 4.2 SIMD instructions for delimiter identification during parsing and deserialization, they reduced the number of control flow branches to avoid these pipeline flushes.

We adapt SSE 4.2 string specific SIMD instructions to GPU warp-level primitives and CUDA's Math API SIMD intrinsics. We analyze their viability for CSV parsing and compare their performance to alternative solutions on the GPU.

Parallelization

An inherent challenge of parsing the CSV data format in parallel is its sequential text stream of data, separated by delimiters to represent rows and columns, and quoting without losing the correct context of found delimiter characters.

Instant Loading. Mühlbauer et al. [7] present an approach to parallelize CSV parsing by splitting the data into equally sized chunks. To identify the ideal chunk size, they concluded a high dependency on the CPU's L3 cache size and number of hardware threads. Their approach for *Instant Loading* allows scalable bulk loading of CSV data at wire speeds on just a single node. We classify their approach as *early context detection*. However, when context-awareness is needed, their approach requires a serialized pass over the input data for context detection, which is inefficient.

We adapt *chunking* as a parallelization strategy for our approach and analyze its most efficient memory access patterns in regards to threads and multi-level caches on the GPU. In contrast to their serialized approach on context-awareness, we also parallelize *early context detection*.

ParPaRaw. In *ParPaRaw* [37], a massively parallel algorithm for parsing delimiter-separated data formats on GPUs is presented. When using aforementioned *chunking* to parallelize parsing on, the parsing thread is not aware of the actual parsing context of its chunk. A record or value delimiter might be an actual delimiter or just plain text because of enclosing characters (e.g. double-quotes) or escape characters that are in a preceding chunk. *ParPaRaw*'s main focus was to create a solution that takes the chunk's parsing context into account without the need for any initial sequential pass over the data or the need to wait for all preceding threads to finish first, allowing for true massive scalability. For this, they exploit the fact that there are only a few possible contexts to consider while parsing. A DFA *(deterministic finite automaton)* keeps track of the thread's current context while reading characters. However, instead of just one DFA, each thread instantiates one DFA for every possible starting state (a similar concept can be observed in work done by Ge et al. [44] and by Döhmen et al. [45]). While reading delimiters, these DFAs will then transition accordingly until the thread reaches the end of the chunk where the DFAs' final state will then be saved to a vector. The composite of these vectors allow the algorithm to deduce the correct starting state of every thread which then can correctly interpret its chunk's symbols in a subsequent step. We classify their approach as *late context detection*.

Inferring the chunks' parsing contexts requires additional memory for each DFA and global synchronization between thread blocks. Although *late context detection* scales to many threads, tracking multiple DFAs incurs more work overall. As we reduce the amount of work, *ParPaRaw* is more processing-intensive than our approach. While their model shows that it can saturate PCIe 3.0's bandwidth on some inputs, due to the DFA simulations our evaluation shows it cannot saturate the bandwidth of faster interconnects, such as NVLink 2.0. The performance of data loaded from I/O devices, such as NICs, is also not considered in *ParPaRaw*. With the rise in bandwidth saturation using GPU parsing, shown for CUDAFastCSV and *ParPaRaw*, loading the data becomes increasingly important to not become the new bottleneck.

Other Data Formats

Current research on how to improve parsing performance is not limited to the CSV data format.

Langdale and Lemire [46] implemented a CPU-based state-of-the-art JSON parser, *simdjson*, in their research that makes heavy use of SIMD instructions.

With *Mison*, Li et al. [47] deviate from the traditional approach of parsing JSON using finite state machines. Instead, projection and filter operators are integrated into the parser itself, which uses previously seen patterns in the dataset to speculatively predict logical locations of queried fields.

Xie et al. introduce *FishStore* [48], a storage layer that combines a generic data parser with a hash-based primary subset index and a user-defined function to dynamically register a subset of the parsed data. They find this subset hashing to be a powerful primitive that supports a broad range of analytical queries on data that becomes immediately available during parsing.

7. Conclusion

In this chapter, we conclude our work with a summary of the methods and results of our thesis. We follow up by giving an overview of potential future work.

7.1 Summary

This thesis has shown the feasibility and potential of loading CSV data using the GPU for either in-memory data processing or for offloading this CPU-bound task using end-to-end streaming to either a GPU in the host system or to a GPU in a remote network host using RDMA and GPUDirect.

We analyzed different implementation strategies for parsing CSV data in parallel on GPUs and introduced *early context detection*. We adapted *chunking* from CPU-based related work. We compared memory access patterns for their efficiency. We implemented several deserialization models and, using *tapes*, presented a solution to efficiently deserialize CSV fields in parallel.

In this thesis we introduced *CUDAFastCSV*, our implementation for a GPU-based CSV parser. With its ability for end-to-end parsing, we demonstrated an approach for CSV loading over the GPU to saturate I/O bandwidth and hide latency from data transfers. CUDAFastCSV's implementation to efficiently load data onto the GPU from network devices using RDMA and GPUDirect was also presented.

We evaluated multiple CSV implementations on the CPU and GPU and compared different ways and interconnects for how to load CSV data onto the GPU. We underlined the need for faster interconnects with performance numbers of on-GPU parsing of up to 100 GB/s. We showed how parameters of CUDAFastCSV can impact performance and demonstrated its scalability in regards to input size, while still being a viable alternative for files as small as 1 MB.

7.2 Future Work

In addition to the limitations shown in chapter 4.2.4 that should be addressed in potential future work, we discuss perspectives to improve our approach.

Multi-GPU. Systems with multiple GPUs and CPUs are becoming more common. In a system with multiple GPUs, each GPU could be used to parse and deserialize streamed batches during end-to-end parsing. This is especially useful on NVLink 2.0 systems with multiple links, as a PCIe 3.0 system is already bottlenecking the process with just a single GPU. While performance would not proportionally scale with additional GPUs because of the dependency on the widow from the preceding batch, we still expect a significant increase in performance.

Out-of-Order Parser Scheduling. A major performance bottleneck in end-to-end parsing are operations that were scheduled in another stream and block the bus. For instance, transferring large results from device to host can block more important copy operations in the parsing stream needed for further processing, ultimately decreasing performance. CUDA-calls are scheduled on a first-in-first-out basis, regardless of the issuing stream. Adding a Heap- or Priority Queue-like data structure that can hold CUDA-operations before they are actually passed on to the CUDA runtime, could allow for parsing related operations to be executed sooner and improve overall performance.

Quoted Mode. Since our focus for this work was mainly on the Fast Mode, most of our optimization efforts went into it as well. The Quoted Mode still has lots of room for performance improvement. Additionally, support for detecting escape characters would allow for parsing even complex text-filled CSV files. A versatile first approach would be to check for an immediate escape character before counting an encountered quotation mark as such.

Filtering. Support for custom filtering during loading would not only decrease the amount of unnecessarily large data transfers but also improve overall performance. Given the architecture in our work, filtering could be added during the deserialization step to determine if the field, and consequently the entire row, should be written to the result buffer.

Hiding Pipeline Latency. One of the most punishing effects in end-to-end parsing is the delay of the initial chunk of data that is transferred to the GPU. The actual parsing does not start until the first chunk is fully transferred, so keeping the *streamingBatchSize* small will reduce this latency. However, small batch sizes do not fully utilize the GPU resources, reducing overall performance. An alternative would be to start with a small batch size to reduce the initial latency and then automatically ramp-up the batch size to *streamingBatchSize* for the subsequent batches.

Automatic Parameter Adjustments. In the current implementation, the user has to specify the *tapeWidth*s in the form of column lengths, indicating a column's maximum field length. Alternatively, CUDAFastCSV could keep track of the currently longest field for each column during the *Indexing Fields* step. Additionally, the importance of an optimal *warpIndexBufferSize* for performance was shown. However, when over-optimistically choosing a too small value for a given dataset, the current implementation cannot execute and will instead gracefully exit with a warning and suggestion for an alternative value. The simple option to automatically restart the parser with the now known smallest possible value for *warpIndexBufferSize*, can improve the user experience tremendously.

Unicode. Due to time constraints, our implementation focused on simply supporting ASCII encoded content. However, by 2009, UTF-8 has already established itself as the main encoding on the web and as of 2020 is used by over 95% of websites, underlining its importance and need for support in CUDAFastCSV [49] [50].

Appendix

Source Code and Software

The source code of the software of this thesis, CUDAFastCSV, is available on GitHub: https://github.com/alxkum/CUDAFastCSV

SQL Schemas

OmniSci:

```
CREATE TEMPORARY TABLE omnisci_taxi(
 VendorID TINYINT NOT NULL,
 tpep_pickup_datetime CHAR(19) NOT NULL,
 tpep_dropoff_datetime CHAR(19) NOT NULL,
 passenger_count TINYINT NOT NULL,
 trip_distance FLOAT NOT NULL,
 RatecodeID TINYINT NOT NULL,
 store_and_fwd_flag CHAR(1) NOT NULL,
 PULocationID SMALLINT NOT NULL,
 DOLocationID SMALLINT NOT NULL,
 payment_type CHAR(1) NOT NULL,
 fare_amount FLOAT NOT NULL,
 extra FLOAT NOT NULL,
 mta_tax FLOAT NOT NULL,
 tip_amount FLOAT NOT NULL,
 tolls_amount FLOAT NOT NULL,
 improvement_surcharge FLOAT NOT NULL,
 total_amount FLOAT NOT NULL,
 congestion_surcharge FLOAT NOT NULL
);

CREATE TEMPORARY TABLE omnisci_tpch_lineitem(
 orderkey INT NOT NULL,
 partkey INT NOT NULL,
 suppkey INT NOT NULL,
 linenumber TINYINT NOT NULL,
 quantity TINYINT NOT NULL,
 extendedprice FLOAT NOT NULL,
 discount FLOAT NOT NULL,
 tax FLOAT NOT NULL,
 returnflag TEXT NOT NULL ENCODING DICT(8),
 linestatus TEXT NOT NULL ENCODING DICT(8),
```

```
 shipdate TEXT NOT NULL,
 commitdate TEXT NOT NULL,
 receiptdate TEXT NOT NULL,
 shipinstruct TEXT NOT NULL,
 shipmode TEXT NOT NULL,
 comment TEXT NOT NULL
);
```

PostgreSQL:

```
CREATE TABLE postgresql_taxi(
 VendorID SMALLINT NOT NULL,
 tpep_pickup_datetime CHAR(19) NOT NULL,
 tpep_dropoff_datetime CHAR(19) NOT NULL,
 passenger_count SMALLINT NOT NULL,
 trip_distance FLOAT NOT NULL,
 RatecodeID SMALLINT NOT NULL,
 store_and_fwd_flag CHAR(1) NOT NULL,
 PULocationID SMALLINT NOT NULL,
 DOLocationID SMALLINT NOT NULL,
 payment_type CHAR(1) NOT NULL,
 fare_amount FLOAT NOT NULL,
 extra FLOAT NOT NULL,
 mta_tax FLOAT NOT NULL,
 tip_amount FLOAT NOT NULL,
 tolls_amount FLOAT NOT NULL,
 improvement_surcharge FLOAT NOT NULL,
 total_amount FLOAT NOT NULL,
 congestion_surcharge FLOAT NOT NULL
) TABLESPACE alxkumbenchmarkspace;

CREATE TABLE postgresql_tpch_lineitem(
 orderkey INT NOT NULL,
 partkey INT NOT NULL,
 suppkey INT NOT NULL,
 linenumber SMALLINT NOT NULL,
 quantity SMALLINT NOT NULL,
 extendedprice FLOAT NOT NULL,
 discount FLOAT NOT NULL,
 tax FLOAT NOT NULL,
 returnflag CHAR(1) NOT NULL,
 linestatus CHAR(1) NOT NULL,
 shipdate CHAR(10) NOT NULL,
 commitdate CHAR(10) NOT NULL,
 receiptdate CHAR(10) NOT NULL,
 shipinstruct TEXT NOT NULL,
```

```
 shipmode TEXT NOT NULL,
 comment TEXT NOT NULL
) TABLESPACE alxkumbenchmarkspace;
```

HyPer DB:

```
CREATE TABLE hyper_taxi(
 VendorID SMALLINT NOT NULL,
 tpep_pickup_datetime CHAR(19) NOT NULL,
 tpep_dropoff_datetime CHAR(19) NOT NULL,
 passenger_count SMALLINT NOT NULL,
 trip_distance FLOAT NOT NULL,
 RatecodeID SMALLINT NOT NULL,
 store_and_fwd_flag CHAR(1) NOT NULL,
 PULocationID SMALLINT NOT NULL,
 DOLocationID SMALLINT NOT NULL,
 payment_type CHAR(1) NOT NULL,
 fare_amount FLOAT NOT NULL,
 extra FLOAT NOT NULL,
 mta_tax FLOAT NOT NULL,
 tip_amount FLOAT NOT NULL,
 tolls_amount FLOAT NOT NULL,
 improvement_surcharge FLOAT NOT NULL,
 total_amount FLOAT NOT NULL,
 congestion_surcharge FLOAT NOT NULL
);

CREATE TABLE hyper_tpch_lineitem(
 orderkey INT NOT NULL,
 partkey INT NOT NULL,
 suppkey INT NOT NULL,
 linenumber SMALLINT NOT NULL,
 quantity SMALLINT NOT NULL,
 extendedprice FLOAT NOT NULL,
 discount FLOAT NOT NULL,
 tax FLOAT NOT NULL,
 returnflag CHAR(1) NOT NULL,
 linestatus CHAR(1) NOT NULL,
 shipdate CHAR(10) NOT NULL,
 commitdate CHAR(10) NOT NULL,
 receiptdate CHAR(10) NOT NULL,
 shipinstruct TEXT NOT NULL,
 shipmode TEXT NOT NULL,
 comment TEXT NOT NULL
);
```

List of Figures

Figures 3, 4, and 15 derived from illustrations done by Cheng et al. [51].

Bibliography

[1] W3C, "CSV on the Web: A Primer," 25 February 2016. [Online]. Available: https://www.w3.org/TR/2016/NOTE-tabular-data-primer-20160225/. [Accessed 11 May 2019].

[2] S. Idreos, I. Alagiannis, R. Johnson and A. Ailamaki, "Here are my Data Files. Here are my Queries. Where are my Results?," *CIDR,* pp. 57-68, 2011.

[3] A. Dziedzic, M. Karpathiotakis and A. Ailamaki, "DBMS Data Loading: An Analysis on Modern Hardware," in *ADMS/IMDM@VLDB*, New Delhi, India, 2016.

[4] "Apache Parquet," [Online]. Available: https://parquet.apache.org. [Accessed 16 June 2019].

[5] R. Kitchin and G. McArdle, "What makes Big Data, Big Data?," *Big Data & Society,* pp. 1-10, 2016.

[6] M. Hilbert, "The World's Technological Capacity to Store, Communicate, and Compute Information," *Science (AAAS),* vol. 332, pp. 60-67, 2011.

[7] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper and T. Neumann, "Instant Loading for Main Memory Databases," *Proceedings of the VLDB Endowment,* vol. 6, no. 14, pp. 1702-1713, 2013.

[8] D. Wilson, "csvmonkey: Header-only vectorized, lazy-decoding, zero-copy CSV file parser," [Online]. Available: https://github.com/dw/csvmonkey. [Accessed 1 June 2020].

[9] "Nvidia Developer Zone: CUDA C Programming Guide," 26 March 2019. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. [Accessed 29 April 2019].

[10] "Brytlyt: GPU based PostgreSQL Database," [Online]. Available: https://www.brytlyt.com. [Accessed 16 May 2019].

[11] "SQream DB: GPU-accelerated data warehouse," [Online]. Available: https://sqream.com/product/. [Accessed 16 May 2019].

[12] Y. Shafranovich, "RFC 4180," October 2005. [Online]. Available: https://tools.ietf.org/pdf/rfc4180.pdf. [Accessed 29 March 2019].

[13] I. Corporation, IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information, New York, NY, USA, 1972.

[14] R. Jadrnicek, "Software Reviews - SuperCalc²," *InfoWorld,* vol. 5, no. 37, pp. 38-40, 1983.

[15] G. J. van den Burg, A. Nazabal and C. Sutton, "Wrangling Messy CSV Files by Detecting Row and Type Patterns," 2018.

[16] J. Mitlöhner, S. Neumaier, J. Umbrich and A. Polleres, "Characteristics of Open Data CSV Files," in *International Conference on Open and Big Data (OBD)*, 2016.

[17] U. Drepper, "What Every Programmer Should Know About Memory," 2007.

[18] S. Manegold and P. Boncz, "Optimizing Main-Memory Join on Modern Hardware," *IEEE Transactions on Knowledge and Data Engineering,* vol. 14, no. 4, pp. 709-731, 2002.

[19] A. Trivedi, P. Stuedi and J. Pfefferle, "Albis: High-Performance File Format for Big Data Systems," in *USENIX Annual Technical Conference*, Boston, MA, USA, 2018.

[20] P. Boncz and S. Manegold, "Database Architecture Optimized for the New Bottleneck: Memory Access," *Proceedings of the VLDB Endowment,* vol. 99, pp. 54-65, 1999.

[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*, Portland, OR, USA, 2020.

[22] N. Corp., "Nvidia Tesla V100 GPU Architecture (Whitepaper)," 2017.

[23] N. Corp., "Nvidia Tesla P100 GPU Architecture (Whitepaper)," 2016.

[24] E. Z. Zhang, Y. Jiang, Z. Guo and X. Shen, "Streamlining GPU Applications on the Fly - Thread Divergence Elimination through Runtime Thread-Data Remapping," in *ICS*, Tsukuba, Ibaraki, Japan, 2010.

[25] Z. Jia, M. Maggioni, B. Staiger and D. P. Scarpezza, "Dissecting the Nvidia Volta GPU Architecture via Microbenchmarking," in *GPU Technology Conference*, San Jose, CA, USA, 2018.

[26] A. Dragojevic, D. Narayanan, O. Hodson and M. Castro, "FaRM: Fast Remote Memory," in *USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014.

[27] A. Kalia, M. Kaminsky and D. Andersen, "Design Guidelines for High Performance RDMA Systems," in *USENIX Annual Technical Conference*, Denver, CO, USA, 2016.

[28] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann and A. Kemper, "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory," in *IEEE International Conference on Data Engineering*, Dallax, TX, USA, 2020.

[29] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Cambridge, MA, USA: Morgan Kaufmann Publications, Elsevier, 2013.

[30] N. Corp., B. Fiser and S. Jodlowski, "Best Practices When Benchmarking CUDA Applications," in *GTC - GPU Tech Conference*, San Jose, CA, USA, 2019.

[31] N. T. &. L. Commision, "TLC Trip Record Data," 2019. [Online]. Available: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. [Accessed 26 March 2020].

[32] T. P. P. Council, "TPC-H is a Decision Support Benchmark," [Online]. Available: http://www.tpc.org/tpch/. [Accessed 26 March 2020].

[33] O. Inc., "OmniSci - Accelerated Analytics Platform," [Online]. Available: https://www.omnisci.com/. [Accessed 1 June 2020].

[34] O. Inc., "OmniSci Docs: Loading Data with SQL - CSV/TSV Import," [Online]. Available: https://docs.omnisci.com/loading-and-exporting-data/command-line/load-data#csv-tsv-import. [Accessed 1 June 2020].

[35] T. P. G. D. Group, "PostgreSQL: The world's most advanced open source database," [Online]. Available: https://www.postgresql.org/.

[36] L. D. Technische Universität München, "HyPer," [Online]. Available: https://hyper-db.de. [Accessed 16 May 2019].

[37] E. Stehle and H.-A. Jacobsen, "ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data," *CoRR,* vol. abs/1905.13415, 2019.

[38] "Apache Arrow," [Online]. Available: https://arrow.apache.org/. [Accessed 1 June 2020].

[39] RAPIDS, "cuDF," [Online]. Available: https://rapids.ai/. [Accessed 1 June 2020].

[40] E. Higgs, "CSV Game - Benchmark game for reading CSV files," [Online]. Available: https://bitbucket.org/ewanhiggs/csv-game/src/master/. [Accessed 1 June 2020].

[41] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky and A. W. Toga, "CUDA Optimization Strategies for Compute- and Memory-Bound Neuroimaging Algorithms," *Computer Methods and Programs in Biomedicine,* vol. 106, no. 3, pp. 175-187, 2012.

[42] L. J. Toso, "Efficient Join Operators on Heterogeneous Systems Using RDMA and Coprocessors," Technische Universität Berlin, 2019.

[43] S. Tu, W. Zheng, E. Kohler, B. Liskov and S. Madden, "Speedy Transactions in Multicore In-Memory Databases," in *SOSP*, Farmington, PA, USA, 2013.

[44] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli and D. Kossmann, "Speculative Distributed CSV Data Parsing for Big Data Analytics," in *SIGMOD*, Amsterdam, Netherlands, 2019.

[45] T. Döhmen, H. Mühleisen and P. Boncz, "Multi-Hypothesis CSV Parsing," in *SSDBM*, Chicago, IL, USA, 2017.

[46] G. Langdale and D. Lemire, "Parsing Gigabytes of JSON per Second," 2019.

[47] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein and D. Kossmann, "Mison: A Fast JSON Parser for Data Analytics," *Proceedings of the VLDB Endowment,* vol. 10, no. 10, pp. 1118-1129, 2017.

[48] B. Chandramouli, D. Xie, Y. Li and D. Kossmann, "FishStore: Fast Ingestion and Indexing of Raw Data," *Proceedings of the VLDB Endowment,* vol. 12, no. 12, pp. 1922-1925, 2019.

[49] M. Davis, "Official Google Blog," [Online]. Available: https://googleblog.blogspot.com/2012/02/unicode-over-60-percent-of-web.html. [Accessed 23 July 2020].

[50] W. -. W. T. Surveys, "Usage of character encodings broken down by ranking," [Online]. Available: https://w3techs.com/technologies/cross/character_encoding/ranking. [Accessed 23 July 2020].

[51] J. Cheng, M. Grossman and T. McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014.

[52] "Nvidia Developer Zone: CUDA C Best Practices Guide," 26 March 2019. [Online]. Available:

https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf. [Accessed 29 April 2019].

[53] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann and A. Kemper, "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory," in *ICDE*, 2020.