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

Alexander Kumaigorodski[1] Clemens Lutz[2] Volker Markl[3]

Abstract: Comma-separated values (CSV) is a widely-used format for data exchange. Due to the format's prevalence, virtually all industrial-strength database systems and stream processing frameworks support importing CSV input.

However, loading CSV input close to the speed of I/O hardware is challenging. Modern I/O devices such as InfiniBand NICs and NVMe SSDs are capable of sustaining high transfer rates of 100 Gbit/s and higher. At the same time, CSV parsing performance is limited by the complex control flows that its semi-structured and text-based layout incurs.

In this paper, we propose to speed-up loading CSV input using GPUs. We devise a new parsing approach that streamlines the control flow while correctly handling context-sensitive CSV features such as quotes. By offloading I/O and parsing to the GPU, our approach enables databases to load CSVs at high throughput from main memory with NVLink 2.0, as well as directly from the network with RDMA. In our evaluation, we show that GPUs parse real-world datasets at up to 60 GB/s, thereby saturating high-bandwidth I/O devices.

Keywords: CSV; Parsing; GPU; CUDA; RDMA; InfiniBand

1 Introduction

Sharing data requires the data provider and data user to agree on a common file format. *Comma-separated values (CSV)* is currently the most widely-used format for sharing tabular data [DMB17, Ne17, Me16]. Although alternative formats such as Apache Parquet [Ap17] and Albis [Te18] exist, in the future the CSV format will likely remain popular due to continued advocacy by open data portals [KH15, Eu20]. As a result, database customers request support for loading terabytes of CSV data [Oz18]. Fast data loading is necessary to reduce the delay before the data are ready for analysis.

Fresh data are typically sourced either from disk or streamed in via the network, thus *loading* the data consists of device I/O and parsing the file format [Me13]. However, recent advances in I/O technologies have lead to a *data loading bottleneck*. RDMA network interfaces and NVMe storage arrays can transfer data at 12.5 GB/s and beyond [Be16, Ze19]. Research suggests that CPU-based parsers cannot ingest CSV data at these data rates [Ge19, SJ20].

[1] TU Berlin, Germany, alxkum@gmail.com
[2] DFKI GmbH, Berlin, Germany, clemens.lutz@dfki.de
[3] TU Berlin & DFKI GmbH, Germany, volker.markl@tu-berlin.de

In this paper, we investigate how I/O-connected GPUs enable fast CSV loading. We stream data directly to the GPU from either main memory or the network using fast GPU interconnects and GPUDirect. Fast GPU interconnects, such as AMD Infinity Fabric [AM19], Intel CXL [CX19], and Nvidia NVLink [Nv17], provide GPUs with high bandwidth access to main memory [Le20b], and GPUDirect provides GPUs with direct access to RDMA and NVMe I/O devices [Nv20]. To parse at high bandwidth, we propose a new, GPU-optimized CSV parsing approach. The key insight of our approach is that multiple data passes in GPU memory simplify complex control flows, and increase computational efficiency.

In summary, our contributions are as follows:

(1) We propose a new approach for fast, parallel CSV parsing on GPUs (Section 3).
(2) We provide a new, streamed loading strategy that uses GPUDirect RDMA [Nv20] to transfer data directly from the network onto the GPU (Section 4).
(3) We evaluate the impact of a fast GPU interconnect for end-to-end streamed loading from main memory and back again (Section 5). We use NVLink 2.0 to represent the class of fast GPU interconnects.

The remainder of this paper is structured as follows. In Section 2, we give a brief overview of performing I/O on GPUs, and of related work on CSV parsing. Then, we describe our contributions to CSV parsing and streaming I/O in Sections 3 and 4. Next, we evaluate our work in Section 5, and discuss our findings on loading data using GPUs in Section 6. Finally, we give our concluding remarks in Section 7.

2 Background and Related Work

In this section, we describe how GPUDirect RDMA and fast GPU interconnects enable high-speed I/O on GPUs. We then give an overview of CSV parsing, and differentiate our approach from related work on CSV loading.

2.1 I/O on GPUs

GPUs are massively parallel processors that run thousands of threads at a time. The threads are executed by up to 80 streaming multiprocessors (*SM*s) on the Nvidia "Volta" architecture [Nv17]. Each SM runs threads in *warps* of 32 threads, that execute the same instruction on multiple data items. Branches that cause control flow to diverge thus slow down execution (*warp divergence*). Within a warp, threads can exchange data in registers using *warp shuffle* instructions. Up to 32 warps are grouped as a *thread block*, that can exchange data in a *shared memory* cache. The GPU also has up to 32 GB of fast on-board *GPU memory*.

I/O on the GPU is typically conducted via a PCIe 3.0 interconnect that connects the GPU to the system at 16 GB/s. Recently, fast interconnects have emerged that provide system-wide cache-coherence and, in the case of NVLink 2.0, up to 150 GB/s. Databases usually use these interconnects to transfer data between main memory and GPU memory, thus linking the GPU to the CPU. However, *GPUDirect RDMA* and *GPUDirect Storage* connect the GPU directly to an *I/O device*, such as an RDMA network interface (e.g., InfiniBand) or an NVMe storage device (e.g., a flash disk). This connection bypasses the CPU and main memory by giving the I/O device direct memory access to the GPU's memory. Although the data bypasses the CPU, the CPU orchestrates transfers and GPU execution.

GPUDirect Storage has been used in a GPU-enabled database to manage data on flash disks [Le16]. In contrast, we propose to load data from external sources into the database.

In summary, fast interconnects and GPUDirect enable the GPU to efficiently perform I/O. In principle, these technologies can be combined. However, due to our hardware setup, in our experimental evaluation we distinguish between NVLink 2.0 to main memory, and GPUDirect RDMA with InfiniBand via PCIe 3.0.

2.2 CSV Parsing

CSV is a tabular format. The data are logically structured as records and fields. Thus, parsers split the CSV data at record or field boundaries to facilitate later deserialization of each field. The parser determines the structure by parsing field (’,’) and record (’\n’) delimiters, as CSV files provide no metadata mapping its logical structure to physical bytes. Quotes (’"’) make parsing more difficult, as delimiters within quotes are literal characters and do not represent boundaries. The CSV format is formalized in RFC4180 [Sh05], although variations exist [DMB17].

Parsing CSV input in parallel involves splitting the data into *chunks* that can be parsed by independent threads. Initially, the parser splits the file at arbitrary bytes, and then adjusts the split offsets to the next delimiter [Me13]. Detecting the correct quotation context requires a separate data pass, most easily performed by a single thread [Me13]. As quotes come in pairs, parallel *context detection* first counts all quotes in each chunk, and then performs a prefix sum to determine if a quote opens (odd) or closes (even) a quotation [Ea16, Ge19].

CPU parsers have been optimized by eliminating data passes through speculative context detection [Ge19, Le17], and replacing complex control flows with SIMD data flows [Ge19, LL19, Le17, Me13]. In contrast to these works, we optimize parsing for the GPU by applying data-parallel primitives (i.e., prefix sum) and by transposing the data into a columnar format. These optimizations reduce warp divergence on GPUs, but add two data passes for a total of 4 passes (without context detection) or 5 passes (with context detection). We reduce the overhead of these additional passes by caching intermediate data in GPU memory.

Stehle and Jacobsen have presented a GPU-enabled CSV parser [SJ20]. Their parser tracks multiple finite state machines to enable a generalization to other data formats, e.g., JSON or XML. Our evaluation shows that this generality is computation-intensive and limits throughput. In contrast, we explore loading data directly from an I/O device and a fast GPU interconnect. These technologies require a fast CSV parsing approach. We thus minimize computation by specializing our approach to RFC4180-compliant CSV data.

3 Approach

In this section we introduce a new algorithm for parsing CSV data that is optimized for GPUs. 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 exploit the GPU's high memory bandwidth to cache the data during these passes. This insight forms the basis on which we adapt CSV parsing to the GPU architecture.

We first give a conceptual overview of our approach in Figure 1. Next, we describe and discuss each step in more detail with its challenges and solutions in the following subsections.

Conceptually, the CSV input is first transferred to GPU memory from a data source, e. g., main memory or an I/O device. The input 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 these counted delimiters. Using the prefix sums, the chunks are processed again to create the *FieldsIndex*. This index 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,

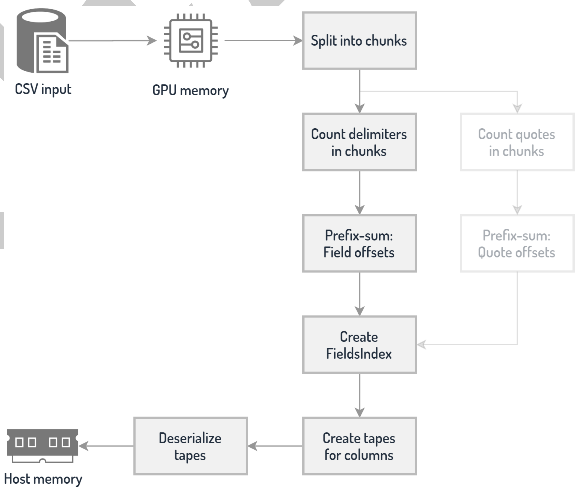


Fig. 1: Conceptual overview of our approach

each tape is deserialized in parallel. The resulting data are 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 is unaware of the context and correct quotation scope when fields are enclosed in quotation marks. To create a context-aware FieldsIndex, we introduce the *Quoted Mode*, as fields may themselves contain field delimiters. Quoted Mode is an alternative parsing mode that additionally keeps track of quotation marks. Quoted Mode allows us to parallelize parsing of quoted CSV data, but it is more processing intensive than the default parsing mode. However, as well-known public data sources indicate that quotes are rarely used in practice[4][5], the main focus of our work is on the Fast Mode.

In this section, we assume the CSV input already resides in GPU memory. In Section 4, we present *Streaming I/O*, which allows incoming chunks of data to be parsed without the need for the entire input data to be in GPU memory.

Overall, our data-parallel CSV parser solves three main challenges: (1) splitting the data into chunks for parallel processing, (2) determining each chunk's context, and (3) vectorized deserialization of fields with their correct row and column numbers.

3.1 Parallelization Strategy

Parsing data on GPUs requires massive parallelism to achieve high throughput. In the following, we explain how we parallelize CSV parsing in our approach.

Simply parallelizing by rows requires iterating over the entire data first. It also results in unevenly sized row lengths. This causes subsequent parsing or deserialization threads in a warp to stall during individual processing and, thus, not make maximum use of the hardware. Instead, Figure 2 shows how we split the input data at fixed offsets to get equally sized chunks. These chunks are independent of each other and individually processed by a warp. This avoids any thread from becoming idle as they transfer and process the same amount of data.

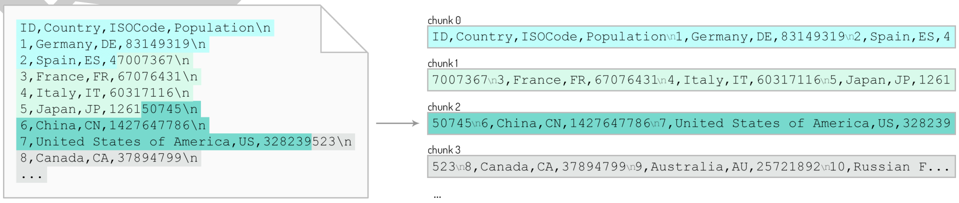

ID,Country,ISOCode,Population\n
1,Germany,DE,83149319\n
2,Spain,ES,47007367\n
3,France,FR,67076431\n
4,Italy,IT,60317116\n
5,Japan,JP,12615O745\n
6,China,CN,1427647786\n
7,United States of America,US,328239523\n
8,Canada,CA,37894799\n
...
chunk 0
ID,Country,ISOCode,Population\n1,Germany,DE,83149319\n2,Spain,ES,4
chunk 1
7007367\n3,France,FR,67076431\n4,Italy,IT,60317116\n5,Japan,JP,1261
chunk 2
50745\n6,China,CN,1427647786\n7,United States of America,US,328239
chunk 3
523\n8,Canada,CA,37894799\n9,Australia,AU,25721892\n10,Russian F...
...

Fig. 2: Splitting input into equally sized, independent, chunks

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

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

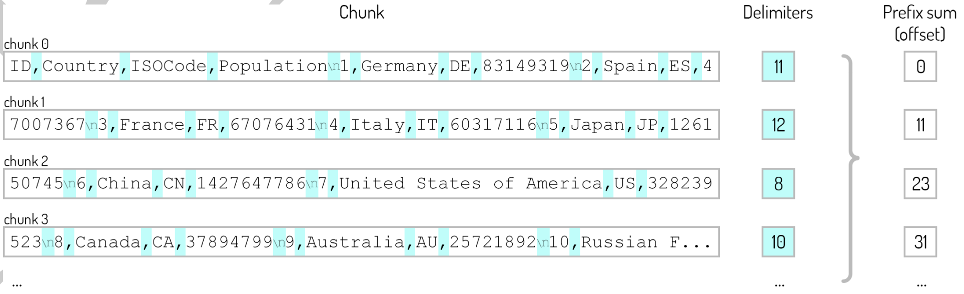
The choice of *chunk size* and how a warp loads and reads its chunks impacts the parallelizability of the delimiter-counting process and, ultimately, the entire parsing process. Because coalesced byte-wise access is not enough to saturate the available bandwidth, we read four bytes per thread, which correspond to the 128 bytes of a GPU memory transaction per warp. However, we experimentally find that looping over multiple consecutive 128-byte chunks per warp increases bandwidth even more, compared to increasing the number of thread blocks. This reduces the pressure on the GPU's warp scheduler as well as the CUDA runtime.

To identify chunk sizes that allow for optimal loading and processing, we evaluate several kernels that each load chunks of different sizes. We discover that casting four consecutive bytes to an `int` and loading the `int` into a register is more efficient than loading the bytes one at a time. For input that is not a multiple of 128 bytes, a challenge here is to efficiently avoid a memory access violation in the last chunk. Using a branch condition for bounds-checking takes several cycles to evaluate. We avoid a branch altogether by padding the input data with NULLs to a multiple of 128 bytes during input preparation. Conventionally, strings are NULL-terminated, thus any such occurrence simply causes these padded bytes to be ignored during loading and later parsing.

3.2 Indexing Fields

We can now start processing the chunks. Our goal in this phase is to index all of the field positions of the input data in the *FieldsIndex*. This index is an integer array of yet unknown size `rows*columns` with a sequence of continuous field positions. We construct the FieldsIndex in three steps.

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.



	Chunk	Delimiters	Prefix sum (offset)
chunk 0	`ID,Country,ISOCode,Population\n1,Germany,DE,83149319\n2,Spain,ES,4`	11	0
chunk 1	`7007367\n3,France,FR,67076431\n4,Italy,IT,60317116\n5,Japan,JP,1261`	12	11
chunk 2	`50745\n6,China,CN,1427647786\n7,United States of America,US,328239`	8	23
chunk 3	`523\n8,Canada,CA,37894799\n9,Australia,AU,25721892\n10,Russian F...`	10	31
...		...	...

Fig. 3: Computing the field offset for every chunk using a prefix sum

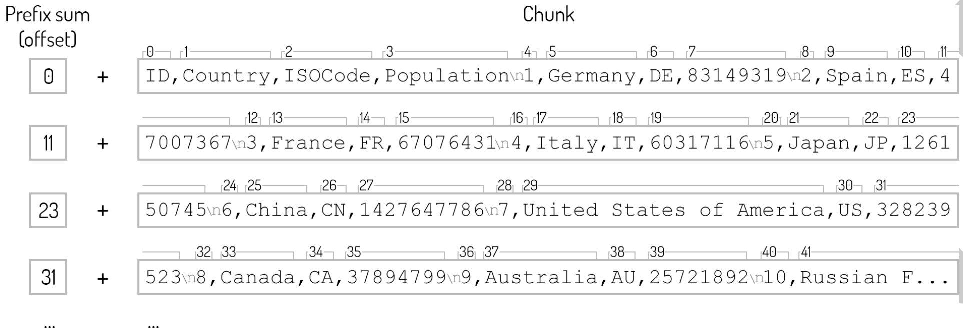

Prefix sum (offset)
Chunk
0 1 2 3 4 5 6 7 8 9 10 11
0 + ID,Country,ISOCode,Population\n1,Germany,DE,83149319\n2,Spain,ES,4
12 13 14 15 16 17 18 19 20 21 22 23
11 + 7007367\n3,France,FR,67076431\n4,Italy,IT,60317116\n5,Japan,JP,1261
24 25 26 27 28 29 30 31
23 + 50745\n6,China,CN,1427647786\n7,United States of America,US,328239
32 33 34 35 36 37 38 39 40 41
31 + 523\n8,Canada,CA,37894799\n9,Australia,AU,25721892\n10,Russian F...
...
...

Fig. 4: Using the chunk's prefix sum to infer field positions

In the second phase, we compute the chunks' field offsets with an exclusive prefix sum, as illustrated in Figure 3. At the end of the prefix sum calculation, the total number of fields is automatically available. From the number of fields we infer the number of rows and allocate the necessary space for the FieldsIndex array in GPU memory.

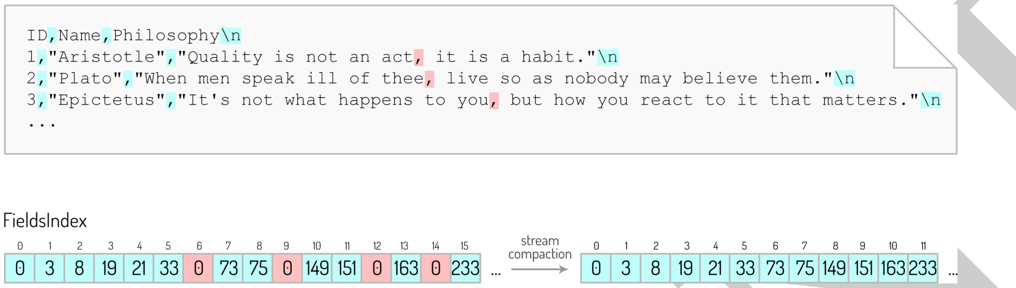
In the third and final phase, the FieldsIndex can now be filled in parallel. We perform a second pass over all the chunks and scan for field delimiters again. As shown in Figure 4, the total number of preceding fields in the input data can instantly be inferred using the prefix sum at a chunk's position.

For a thread to correctly determine a field's index when encountering its delimiter, however, it also needs to know the total number of delimiters in the warp's preceding threads. Thus, 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. These prefix sums provide the complete information needed to determine a field's exact position and index to store it in the FieldsIndex array. The length of a field can also be inferred from the FieldsIndex.

3.2.1 Quoted Mode

For the *Quoted Mode*, additional steps need to be taken throughout to create a correct FieldsIndex.

When counting delimiters in the first phase, quotation marks are simultaneously counted in a similar manner. 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 when counting delimiters again, quotation marks are also counted again, and prefix sums are created for both within the warp.



```
ID,Name,Philosophy\n
1,"Aristotle","Quality is not an act, it is a habit."\n
2,"Plato","When men speak ill of thee, live so as nobody may believe them."\n
3,"Epictetus","It's not what happens to you, but how you react to it that matters."\n
...
```

FieldsIndex

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	3	8	19	21	33	0	73	75	0	149	151	0	163	0	233	...

0	1	2	3	4	5	6	7	8	9	10	11	
0	3	8	19	21	33	73	75	149	151	163	233	...

Fig. 5: Additional pass in Quoted Mode to remove invalid delimiters

We then exploit the fact that a character is considered quoted whenever the number of preceding quotation marks is uneven. 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 uneven, a sentinel value of 0 is written to the FieldsIndex at the index that the field's position would otherwise have been written to. The sentinel value represents 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 5. Separating this additional step from the actual FieldsIndex creation simplifies control flow and helps to coalesce writes to memory.

3.3 Deserialization

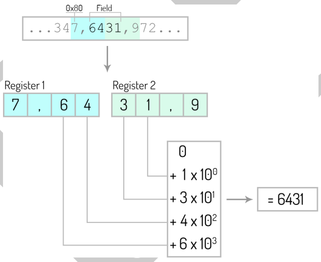


Fig. 6: Thread reading aligned bytes to registers for looped deserialization

Efficient deserialization on the GPU is a many-sided problem. Not only is the question of how to vectorize deserialization challenging, but also how to keep the entire warp occupied. A simple approach is to have every thread deserialize a field. However, we must assume that neighboring columns have different data types. Constructing a generic kernel that can handle all data types involves many branches, causing warp divergence. Additionally, using any row-based approach 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 vectorize. 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 specify a column's maximum length along its type for deserialization purposes. To keep the warp's memory access pattern coalesced, 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. If all column fields are contiguous in memory, the warp is likely able to coalesce memory accesses. 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 6.

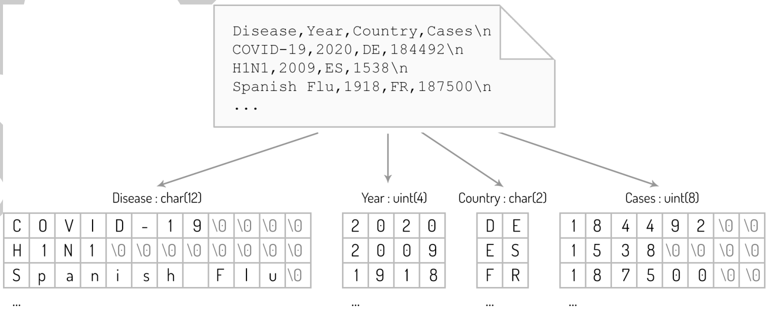
While this approach can lead to workload imbalances within a warp, i. e., when neighboring fields in a warp have unequal lengths, this approach causes no warp divergence and only uses one branch in the entire kernel.

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 only come into effect when there is just one column or the field's data types are identical along a multiple of 32 wide field count. To improve deserialization performance for columns with various data types we introduce deserialization with tapes. *Tapes* are buffers in which the parser temporarily stores fields in a column-oriented layout. The column-oriented layout enables vectorized deserialization of fields.

We illustrate our approach with an example in Figure 7. 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. For every field in the FieldsIndex, the input's field value is copied to its column's tape at an offset equal to the field's row number:

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



```
Disease,Year,Country,Cases\n
COVID-19,2020,DE,184492\n
H1N1,2009,ES,1538\n
Spanish Flu,1918,FR,187500\n
...
```

C	O	V	I	D	-	1	9	\0	\0	\0	\0
H	1	N	1	\0	\0	\0	\0	\0	\0	\0	\0
S	p	a	n	i	s	h		F	l	u	\0

...

2	0	2	0
2	0	0	9
1	9	1	8

...

D	E
E	S
F	R

...

1	8	4	4	9	2	\0	\0
1	5	3	8	\0	\0	\0	\0
1	8	7	5	0	0	\0	\0

...

Fig. 7: Visual representation of deserialization tapes

Field values that do not fully utilize their tapeWidth are right-padded with NULLs on the tape.

For our Fast Mode, materializing the entire FieldsIndex in GPU memory can be skipped and instead the chunk's FieldsIndex is temporarily written to shared memory. When a chunk's FieldsIndex is complete, the field values can be directly copied onto the tapes. However, the length of the chunk's last field cannot be calculated from the chunk's FieldsIndex alone. Instead, 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 materializing the FieldsIndex and from having to do a third pass over the input data for creating the tapes. However, we cannot perform this optimization in the Quoted Mode, as the complete FieldsIndex is required to detect the quotation context.

4 Streaming I/O

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

The input is split into *partitions* 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. The partitions are equal in length and of size *streamingPartitionSize*.

4.1 Context Handover

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. Partitioning our input data creates a similar effect that we need to account for, as illustrated in Figure 8. In a partition, we consider the last row an orphan, which will not be parsed. Instead we copy the orphan's bytes to a widow buffer. The next partition prepends available data from the widow buffer to its partition data before starting the parsing process. The widow buffer's size is defined

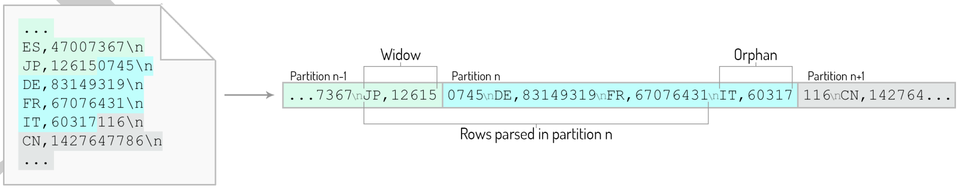


Fig. 8: Widows are taken from the previous partition, while orphans are left for the next partition

by a configuration variable. We set its default size to 10 KB, which is sufficient to handle rows spanning over 10,000 characters.

4.2 End-to-End Loading

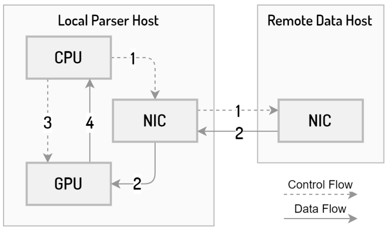


Fig. 9: A WorkStream's control and data flow for its partition

We realize streaming as *WorkStream* items in our implementation, representing a CUDA stream and a partition for processing. Every WorkStream item has a dedicated *partitionBuffer* in the GPU's device memory that is used to copy its partition's chunks into. For RDMA input data, this *partitionBuffer* is also automatically registered for GPUDirect transfers via RDMA.

In Figure 9, we show the four states of a WorkStream. (1) First, an available WorkStream requests a remote memory read, which (2) the remote host responds to by copying the requested data directly into GPU memory. Afterwards, (3) the local CPU schedules a kernel for the WorkStream for parsing. Finally, (4) the kernel is synchronized after finishing and the partition's result data can be optionally transferred to main memory.

5 Evaluation

In this section we evaluate the performance of parsing CSV data on GPUs.

5.1 Experiment Setup

In the following, we give an overview of our experimental evaluation environment.

Hardware. We use two identical machines for the majority of our testing (Intel Xeon Gold 5115, 94 GB DDR4-2400, Nvidia Tesla V100-PCIe with 16 GB HBM2, Mellanox MT27700 InfiniBand EDR, Ubuntu 16.04). A third machine was used for NVLink related evaluations (IBM AC922 8335-GTH, 256 GB DDR4-2666, Nvidia Tesla V100-SXM2 with 16 GB HBM2, Ubuntu 18.04). For all our tests, we use only one NUMA node, i.e., a single GPU and CPU with their respective memories.

Methodology. We measure the mean and standard error over ten runs with the help of high-resolution timers. For GPU-related measurements, we adhere to Nvidia's recommendations when benchmarking CUDA applications [FJ19]. The time for the 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. We measure the throughput in GB/s.

Datasets. For our evaluations we use a real-world dataset (*NYC Yellow Taxi Trips* Jan-Mar 2019, 1.9 GB, 22.5M records, 14 numerical fields out of 18, short and consistent record lengths), a standardized dataset (*TPC-H Lineitem* 2.18.0, 719 MB, 6M records, 16 fields of various data types and string fields with varying lengths), and a synthetic dataset (*int_444*, 1 GB, 70M records, three fields of four random digits).

Databases and Parsers. In Section 5.2.2, we compare CUDAFastCSV to CPU and GPU baselines. *OmniSciDB* (v5.1.2), *PostgreSQL* (v12.2), *HyPer DB* (v0.5), *ParPaRaw* [SJ20], *RAPIDS cuDF* (v0.14.0), and *csvmonkey* (v0.1).

I/O. In Section 5.2.3, we stream the input data from four I/O sources to compare performance against the potentially transfer bound end-to-end parsing from the previous Section 5.2.2. We stream data over interconnects and InfiniBand using two datasets. In contrast to end-to-end parsing, results are not copied back to the host's main memory. *On-GPU* serves as a baseline with the input data already residing in GPU memory. *PCIe 3.0* serves as an upper bound for I/O devices on the host. *NVLink 2.0* is, in comparison to *PCIe 3.0*, a higher-bandwidth and lower-latency alternative [Le20b]. In *RDMA with GPUDirect* one machine acts as the file server, while another machine with CUDAFastCSV streams the input data using RDMA directly into the GPU's memory using GPUDirect, bypassing the CPU and main memory.

5.2 Results

In this section, we present our performance results and comment on our observations.

5.2.1 Tuning Parameters

In this section, we evaluate the parameters for performance tuning and scalability that we introduce in Section 3.

Chunk Size. The choice of the *chunkSize* in CUDAFastCSV determines how much of the input data a warp processes. An increasing size requires more hardware resources per warp but also reduces the overhead associated with scheduling, launching, and processing new thread blocks or warps. Figure 10 shows the throughput as a function of the chunk size. The observed performance reflects our design discussion in Section 3.1. The memory bandwidth for small chunks initially increases when processing 128-byte multiples, but then drops, e. g., before 2048 bytes. The drop stems 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 chunk size is 1024 bytes.

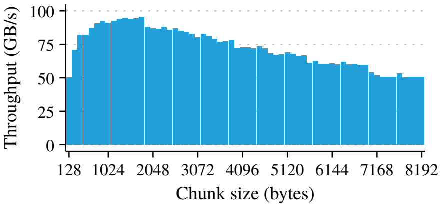


Fig. 10: Impact of *chunkSize* on *int_444*

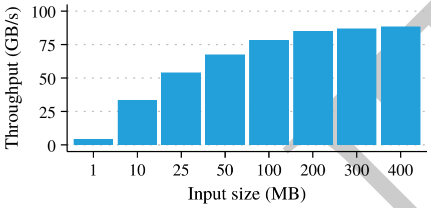


Fig. 11: Performance of input size for *int_444*

Input Size. Figure 11 shows the ramp up of CUDAFastCSV's performance when given an increasingly large input file. While the 1 MB 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.

Streaming Size. In Figure 12, we define a baseline of approximately 12 GB/s for PCIe 3.0 as it represents the maximum possible throughput for that machine. In our results, the throughput scales almost linearly with the *streamingPartitionSize* up until 10 MB before it hits its maximum of 11 GB/s at 20 MB. For comparison, we include results from the same experiment over NVLink 2.0. Ramp-up speed is very similar to PCIe 3.0 but keeps rising when the limitations of PCIe 3.0 would otherwise set in. In contrast, with NVLink 2.0 we achieve a peak throughput of 48.3 GB/s. Thus, NVLink 2.0 is 4.4× faster than PCIe 3.0. However, our implementation is not able to achieve NVLink's peak bandwidth due to the limited amount of DMA copy engines, and due to the overhead from data and buffer management required for streaming. This leads to delays, as transfers and compute are not fully overlapped. We conclude that PCIe 3.0's bandwidth is saturated quickly and its best *streamingPartitionSize* is already achieved at 20 MB. NVLink 2.0 exposes PCIe 3.0 as a bottleneck for end-to-end parsing in comparison.

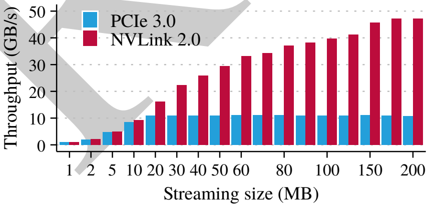


Fig. 12: Impact of parameter *streamingPartitionSize* on *int_444*

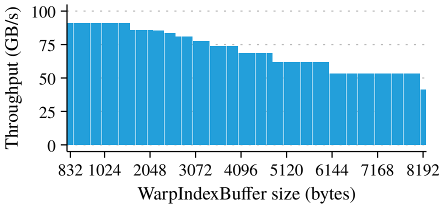


Fig. 13: Impact of oversized *warpIndexBufferSize* on *int_444*

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 13 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, 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* has a large impact on performance, as it is dependent on the underlying structure of the input data.

5.2.2 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 *NYC Yellow Taxi* and *TPC-H* 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 either a row- or a column-oriented data storage format.

NYC Yellow Taxi. The performance numbers reported for parsing and deserializing the 1.9 GB dataset in Figure 14 highlight the strength of CUDAFastCSV, which is only limited by the PCIe 3.0's available bandwidth. This is especially noteworthy, as deserialization 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, achieves only a quarter of the performance of CUDAFastCSV. Our approach is at least 4x times faster than all CPU-based approaches, i. e., *PostgreSQL*, *HyPer DB*, *OmniSciDB*, *csvmonkey*,

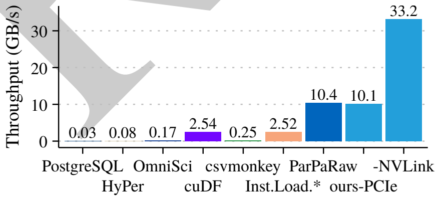


Fig. 14: *Taxi* end-to-end performance

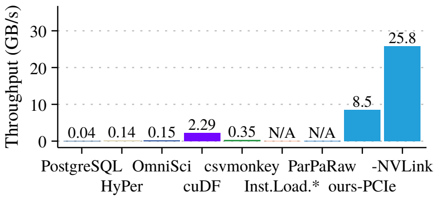


Fig. 15: *TPC-H* end-to-end performance

and **Instant Loading* (taken from Stehle and Jacobsen [SJ20] using 32 CPU cores). CUDAFastCSV over NVLink 2.0 more than triples the performance compared to PCIe 3.0. 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 16. In comparison to *ParPaRaw*, our Quoted Mode is 1.6x faster and our Fast Mode is even 3.7x faster. The reason is that we are able to reduce the overall amount of work, as we do not need to track multiple state machines, and our approach is less processing-intensive as a result.

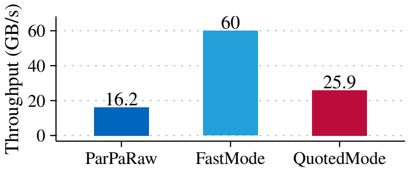


Fig. 16: On-GPU throughput of *ParPaRaw* and CUDAFastCSV

TPC-H Lineitem. Figure 15 shows CUDAFastCSV to have a slightly lower throughput when compared to the previous dataset on both PCIe 3.0 and NVLink 2.0. The bottleneck for this data set is the transfer of the larger result data back to the host, causing increasingly longer delays between streamed partitions. For every 100 MB partition of *TPC-H* data transferred to the GPU, approximately 118 MB of result data need to be transferred back to the host, while the *NYC Yellow Taxi* data only need 93 MB per 100 MB. This causes delays in input streaming and during processing, as kernel invocations are hindered by data dependencies and synchronization. *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 DB* and *csvmonkey*, due to the smaller number of numeric fields that need to be deserialized. NVLink 2.0 again more than triples the performance of CUDAFastCSV in comparison to PCIe 3.0.

5.2.3 I/O

We present results for CUDAFastCSV when streamed over two interconnects and InfiniBand. We evaluate performance using the NYC Yellow Taxi and TPC-H Lineitem datasets to compare against the potentially transfer bound end-to-end parsing.

NYC Yellow Taxi. The baseline for Figure 17 is 60 GB/s, representing CUDAFastCSV's maximum possible performance over an interconnect to the GPU. As seen in the previous section, while our implementation over PCIe 3.0 can fully saturate the bus, it is still less than a fifth of the on-GPU performance. 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, is at an expected sixth of the on-GPU performance. It is unclear why the GPUDirect connection is slower than the local copy, as the network

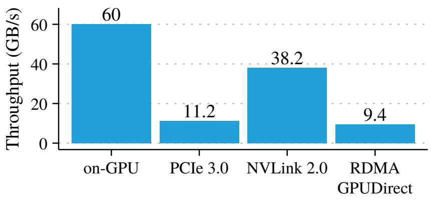


Fig. 17: *Taxi* I/O streaming performance

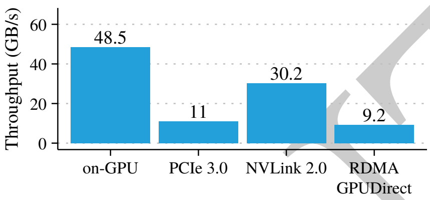


Fig. 18: *TPC-H* I/O streaming performance

is not the bottleneck. Li et al. [Le20a] present similar results, and suggest that PCIe P2P access might be limited by the chipset.

TPC-H Lineitem. The baseline is established in Figure 18 with 48.5 GB/s. Similarly to the taxi dataset, PCIe 3.0 is saturated but only at a sixth of the on-GPU performance, while NVLink 2.0 performance is almost triple in comparison. For the RDMA with GPUDirect approach we achieve similar performance 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.

5.2.4 Quoted Mode

The Quoted Mode is an alternative parsing mode that keeps track of quotation marks to create a context-aware FieldsIndex. In contrast to the Fast Mode, the Quoted Mode involves additional processing steps. We show a comparison between the two modes for three datasets in Figure 19. For all three datasets, the Quoted Mode has roughly half the throughput of the Fast Mode. The cause of this performance drop is the materialization of the FieldsIndex in GPU memory combined with a subsequent stream compaction pass, which are avoided in Fast Mode. We observe that the performance drops more for the *NYC Yellow Taxi* dataset than for the *TPC-H* and *int_444* datasets. The reason is that fields have less content on average in *NYC Yellow Taxi*, thus the FieldsIndex is larger in proportion to the data size (i.e.,

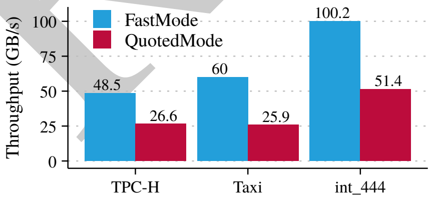


Fig. 19: Fast Mode vs. Quoted Mode

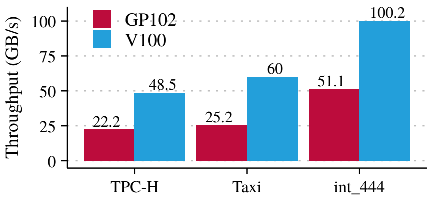


Fig. 20: Desktop-Pascal vs. Server-Volta

more delimiters per MB of data). Nevertheless, throughput is still higher than that of other implementations.

5.2.5 Hardware Scalability

To compare desktop-grade GPUs with server-grade GPUs and the generational leap in hardware advancements, we include measurements on a desktop-grade Pascal GPU from the preceding generation (Nvidia GTX 1080 Ti, 11 GB GDDR5X). This also shows how desktop- and server-grade GPUs could further fare in the future in regards to scalability. The performance improvements shown for Volta in Figure 20 are at least 100% over Pascal for our three datasets. Profiling reveals that the performance improvement mainly stems from the higher memory bandwidth. However, we must also take into account that generational enhancements in instruction throughput and latency are necessary to effectively utilize the faster memory bandwidth [Nv17].

6 Discussion

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

GPUs improve parsing performance. In comparison to a strong CPU baseline, our measurements show that parsing on the GPU still improves throughput by 13x for the *NYC Yellow Taxi* dataset. Compared to a weak baseline, throughput can even be improved by 73x for the *TPC-H Lineitem* dataset. Thus, offloading parsing to the GPU can provide significant value for databases.

Fast parsing of quoted data. We show that our approach is able to parallelize context detection in Quoted Mode, and scale performance up to 51 GB/s. At this throughput, we are near the peak bandwidth of NVLink 2.0.

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 quoted 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 I/O devices, e.g., 400 Gbit/s InfiniBand, would Quoted Mode become a bottleneck.

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 data that are complex to parse does a server-grade GPU make sense.

7 Conclusion

In this work, we explore the feasibility of loading CSV data close to the transfer rates of modern I/O devices. Current InfiniBand NICs transfer data at up to 100 Gbit/s, and multiple devices can be combined to scale the bandwidth even higher. Our analysis shows that CPU-based parsers cannot process data fast enough to saturate such I/O devices, which leads to a data loading bottleneck.

To achieve the required parsing throughput, we leverage GPUs by using a new parsing approach and by connecting the GPU directly to the I/O device. Our implementation demonstrates that GPUs reach a parsing throughput of up to 100 GB/s for data stored in GPU memory. In our evaluation, we show that this is sufficient to saturate current InfiniBand NICs. Furthermore, our NVLink 2.0 measurements underline that GPUs are capable of scaling up to emerging 200 and 400 Gbit/s I/O devices. We envision that in the future, loading data directly onto the GPU will free up computational resources on the CPU, and will thus enable new opportunities to speed-up query processing in databases and stream processing frameworks.

In conclusion, I/O-connected GPUs are able to solve the data loading bottleneck, and represent a new way with which database architects can integrate GPUs into databases.

Acknowledgments
We thank Elias Stehle for sharing and helping us to measure ParPaRaw. This work was funded by the EU Horizon 2020 programme as E2Data (780245), the DFG priority programme "Scalable Data Management for Future Hardware" (MA4662-5), the German Ministry for Education and Research as BBDC (01IS14013A) and BIFOLD — "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A), and the German Federal Ministry for Economic Affairs and Energy as Project ExDra (01MD19002B).

Bibliography

[AM19] AMD: AMD EPYC CPUs, AMD Radeon Instinct GPUs and ROCm Open Source Software to Power World's Fastest Supercomputer at Oak Ridge National Laboratory. https://www.amd.com/en/press-releases/2019-05-07-amd-epyc-cpus-radeon-instinct-gpus-and-rocm-open-source-software-to-power, Accessed: 2019-07-05, May 2019.

[Ap17] Apache Software Foundation: Apache Parquet. https://parquet.apache.org/, Accessed: 2020-10-08, October 2017.

[Be16] Binnig, Carsten; et al.: The End of Slow Networks: It's Time for a Redesign. PVLDB, 2016.

[CX19] CXL: Compute Express Link Specification Revision 1.1. https://www.computeexpresslink.org, June 2019.

[DMB17] Döhmen, Till; Mühleisen, Hannes; Boncz, Peter A.: Multi-Hypothesis CSV Parsing. In: SSDBM. 2017.

[Ea16] Eads, Damian: ParaText: A library for reading text files over multiple cores. https://github.com/wiseio/paratext, Accessed: 2020-09-09, 2016.

[Eu20] European Commission & Open Data Institute: European Data Portal e-Learning Programme. https://www.europeandataportal.eu/elearning/en/module9/#/id/co-01, Accessed: 2020-08-31, March 2020.

[FJ19] Fiser, Bill; Jodłowski, Sebastian: Best Practices When Benchmarking CUDA Applications. In: GTC - GPU Tech Conference. 2019.

[Ge19] Ge, Chang; et al.: Speculative Distributed CSV Data Parsing for Big Data Analytics. PVLDB, 2019.

[KH15] Kim, James G; Hausenblas, Michael: 5-Star Open Data. https://5stardata.info, Accessed: 2020-08-31, 2015.

[Le16] Li, Jing; et al.: HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. PVLDB, 2016.

[Le17] Li, Yinan; et al.: Mison: A Fast JSON Parser for Data Analytics. PVLDB, 2017.

[Le20a] Li, Ang; et al.: Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Trans. Parallel Distrib. Syst., 2020.

[Le20b] Lutz, Clemens; et al.: Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. SIGMOD, 2020.

[LL19] Langdale, Geoff; Lemire, Daniel: Parsing gigabytes of JSON per second. VLDB J., 2019.

[Me13] Mühlbauer, Tobias; et al.: Instant Loading for Main Memory Databases. PVLDB, 2013.

[Me16] Mitlöhner, Johann; et al.: Characteristics of Open Data CSV Files. OBD, 2016.

[Ne17] Neumaier, Sebastian; et al.: Data Integration for Open Data on the Web. In: Reasoning Web. Lecture Notes in Computer Science, 2017.

[Nv17] Nvidia: Nvidia Tesla V100 GPU Architecture (Whitepaper). https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed: 2019-03-26, 2017.

[Nv20] Nvidia: GPUDirect. https://developer.nvidia.com/gpudirect, Accessed: 2020-09-08, 2020.

[Oz18] Ozer, Stuart: How to Load Terabytes into Snowflake — Speeds, Feeds and Techniques. https://www.snowflake.com/blog/how-to-load-terabytes-into-snowflake-speeds-feeds-and-techniques, Accessed: 2020-08-31, April 2018.

[Sh05] Shafranovich, Yakov: RFC 4180. https://tools.ietf.org/pdf/rfc4180.pdf, Accessed: 2019-03-29, 2005.

[SJ20] Stehle, Elias; Jacobsen, Hans-Arno: ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. PVLDB, 2020.

[Te18] Trivedi, Animesh; et al.: Albis: High-Performance File Format for Big Data Systems. In (Gunawi, Haryadi S.; Reed, Benjamin, eds): USENIX ATC. 2018.

[Ze19] Zeuch, Steffen; et al.: Analyzing Efficient Stream Processing on Modern Hardware. PVLDB, 2019.