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

Seminar: Modern Database Systems for Machine Learning and Knowledge Discovery 2023

ANDREAS HADERLEIN

Since the goal of Kumaigorodski et al. is to saturate the bandwidth of modern interconnects in terms of CSV parsing, this paper represents a summary of their work as they build a new parser for GPUs. First showing why their main concept is to "simplify control flow at the expense of additional data passes"[2]. And then explaining their parsing concept, which allows parallel processing on GPUs by introducing independence between the processing units. This independence is achieved by setting some fixed parameters in a convection-like manner, on which each unit then can rely in the ongoing process. A further clue of their algorithm is that they used early context detection to convert the input data from a row-based to a columnar format [2]. This allows them to take advantage of the fact that a column normally contains only a single data type, which excels their parser during deserialization due to the application of vectorization. After explaining the concept of the parser, their results on the implementation of their new algorithm are shown and compared to others, where the key takeaway is that they indeed achieve their main goal of saturating modern interconnects with a CSV parser. This finally leads Kumaigorodski et al. to the conclusion, that this will"[...] enable new opportunities to speed-up query processing in databases and stream processing frameworks".

1 INTRODUCTION

Although more efficiently parsable data formats exist, *Comma-seperated values (CSV)* format continues to be promoted by open data portals. And thus making it one of the most widely used data formats for data exchange. This data exchange is done via disk or most likely via the network. Therefore, loading the data into the memory of a system consists of two steps. Namely Device I/O and file format parsing [2].

Due to the technical progress in I/O devices, the former has become so much faster than the latter, that it created the so-called *data loading bottleneck*, as described by Kumaigorodski et al.. As they point out in their paper, "Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing", novel InfiniBand NICs are capable of transferring data at up to 100 GB/s. Transferring data from main memory into GPU memory is currently up to 63 GB/s using Nvidia’s NVLink 2.0 [2].

The fact that Zeuch et al.’s work showed that CPU-based parsers fail to saturate such high data transfer rates by a huge margin, makes it relevant to think about alternative approaches. Since there already exists a GPU-based parser by Stehle and Jacobsen, called *ParPaRaw*, that is able to saturate PCIe 3.0 with a bandwidth of 12.3 GB/s, as they accomplished a maximum of 14.2 GB/s. What now remains to be done is to saturate the above-stated faster interconnects. So the authors’ primary goal is to do so by really taking advantage of the GPU’s ability to do processing in parallel [2].

With this in mind, the following is a round-up of the work of Kumaigorodski et al., first describing the main challenges of designing a fast GPU-based parser. From there, explaining the concept of their new algorithm. After that, an evaluation is given and thereby their results are shown and discussed. Finally, a conclusion of their work is drawn.

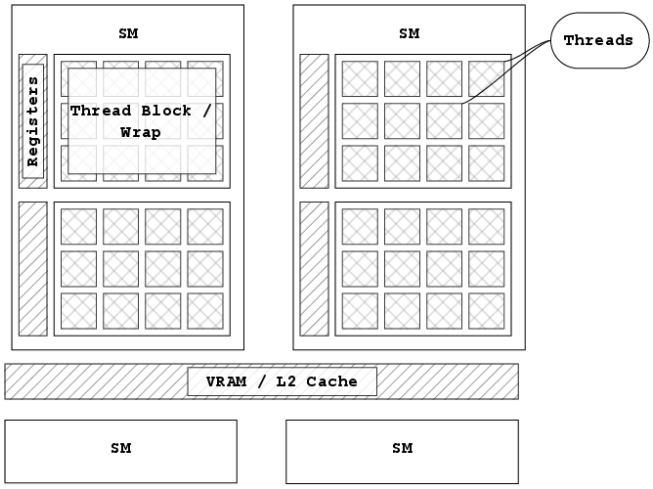

Fig. 1. Schematicall structure of an Nvidia-"Tesla"-like GPU (Own illustration based on [4])

2 CHALLANGES OF A GPU-BASED PARSER

In order to understand the challenges of building a CSV parser for GPUs, it is necessary to know the structure of a GPU, which is schematically shown in Figure 1. A GPU consists of a GPU memory, also called *VRAM*, and, in the case of the *Nvidia Tesla* used by the authors for the evaluation later on, of 80 streaming multiprocessors *(SMs)* [4]. As Figure 1 shows each thread block, also called warp, has its own registers and consists of up to 32 threads. Each of these threads can access these registers making it easy to exchange data within a warp. Exchanging data from warp to warp, on the other hand, is slow because it has to be done through the VRAM, which takes more cycles for storing and accessing.

So parallelism on a GPU means the processing of thread blocks by each SM at the same time. And in doing so, each warp executes the same instruction on the data within the threads it contains. This is the so-called *Single Instruction Multiple Threads (SIMT)* principle. And exactly here lies one of the main challenges of the parser under development. Usually, parsers have a rather complex control flow, which is expressed in many branches that, for example, query the given data type in order to process it accordingly. These branches would then force a warp to execute different instructions on its contained threads. As a result of that, some threads would stall, causing what is known as *warp divergence*, slowing down the process as a whole [1]. Things get even worse when a warp is dependent on one that will be computed a few cycles from now. What follows is that these situations should be avoided. Therefore, each warp

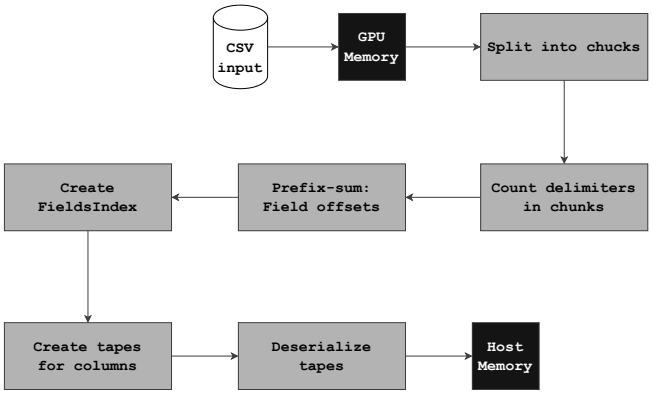

Fig. 2. Conceptual overview of the CSV parsing algorithm (Own illustration based on [2])

must run independently and according to the SIMT principle in order to achieve a fast parser. This causes the authors to formulate the simplification of "control flow at the expense of additional data passes" [2] as their main idea for the parsers concept.

3 CONCEPT OF THE NEW CSV PARSER ALGORITHM

Figure 2 outlines the steps of their designed algorithm to realize this. For the following explanation of the parsers concept, it is assumed that the CSV file is already in GPU memory. To give a more concise approach to the algorithm, we will start by working backward from the end to the beginning, and then, having already seen some fundamental ideas, continue the other way around to reveal some more details.

So the last step in Figure 2 "deserialize tapes" means converting the input data into concrete data structures for further processing. To do so efficiently they, beforehand, introduced these tapes. Tapes are simply buffers holding the row-based CSV input data in columnar format. And thereby taking advantage of the fact that normally columns only contain a single type of data, which then can be processed by the warps using the already mentioned SIMT principle. To set these tapes up, however, they first had to find out which field of the input data belongs to which column. For that purpose, they "create[d] [the] FieldsIndex" array, which is also stored in GPU memory and thus is accessible by every warp at every time. By using the modulo function on the index of the current field each thread is then able to independently compute the corresponding column.

So starting from the very beginning now, what Kumaigorodski et al. did at first to get this FieldsIndex array is to "split [the CSV input file] into [self-contained, equal-sized] chunks", compare Figure 2. This also represents the first step towards parallelization, as each warp gets exactly one of these chunks. By equalizing the size of the chunks, they accomplished at least the following three things at once. First, they achieved that each warp gets the same amount of data and thus the same workload, so they all finish at the same time. Second, they achieved independence between the warps. Because when a warp knows that it is processing chunk x, it also knows exactly which characters of the entire input data it contains by simply calculating $x \cdot chunkSize$. And third, they avoided doing a

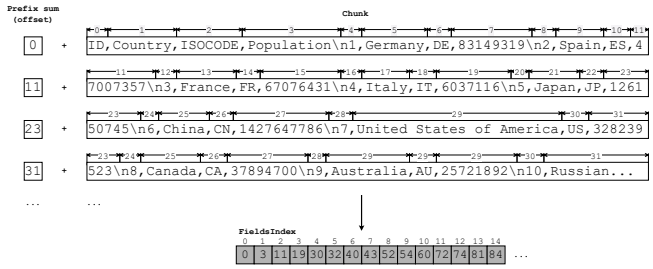

Fig. 3. Fill up the FieldsIndex array (Own illustration based on [2])

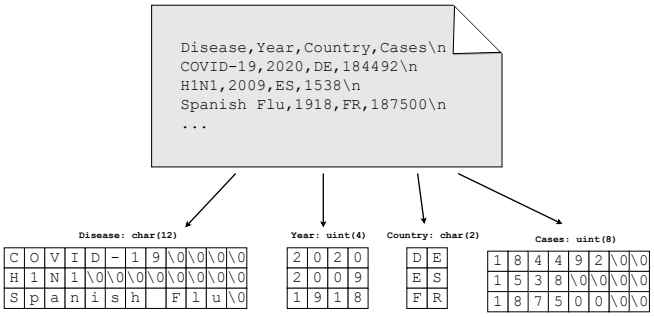

Fig. 4. Building the column-based tapes (Own illustration based on [2])

data pass at least for now, by splitting the input file at these fixed offsets.

But by splitting the input data in this way, they were still nowhere near knowing which field belongs to which column. So what they then did is "count[ing] the delimiters in [the] chunks" in a first pass over the data. And right after that they accumulated these delimiter numbers to form the *prefix sum* for each chunk, which is shown in Figure 3. This costs very little as it is again not a data pass but just an operation on an array. By obtaining these prefix sums, it is now possible to allocate the necessary memory for the FieldsIndex array in GPU memory, since the prefix sum directly reveals the total number of fields in the input file. And what it also does is that each warp now knows exactly how many fields there were in the chunks before it, and therefore also knows where to write into the FieldsIndex array that is to be filled in next. Not only does the warp know where to write to the array, but it also knows what to write by relying on the fixed and equal *chunkSize*.

As can be seen in Figure 2 the next step is to fill the FieldsIndex array by writing the character-offset of the first character of each field into the FieldsIndex array at the corresponding position. So looking at Figure 3 now, writing the number 74 at index 12 means that field 12 starts at character 74 of the entire input file. The length of each field can then be inferred by looking at the starting character within the successor in the array and subtracting one from that number because of the delimiter in between.

What can be done sequentially, but still in the same data pass as filling the FieldsIndex array, is the creation of the aforementioned tapes. This is because now, immediately after the array for a field has been filled in, it is finally known to which column this field belongs and it can be directly assigned to its tape accordingly. The creation of the, in terms of data type, homogenous tapes is shown in Figure 4. What Figure 4 also reveals is that fields that do not fully fill

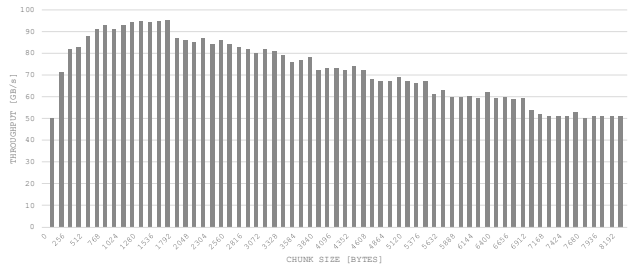

Fig. 5. Impact of *chunkSize* on *int_444* (Own illustration based on [2])

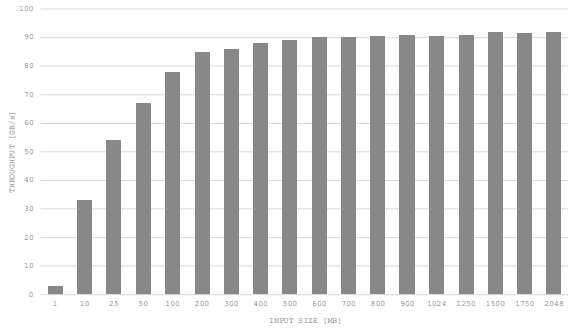

Fig. 6. Performance of input size on *int_444* (Own illustration based on [2])

their corresponding *tapeWidth* are right-padded with NULL-Bytes. This is not a problem in the following, since strings terminate with a NULL-Byte and thus get recognized correctly. After this is done, we are back at the last step of the algorithm namely "deserialize tapes", which then takes another data pass on its own. So overall three passes over the data are needed to parse the file.

But as already can be inferred from the title of the paper the goal of the authors was not to build a parser for files that are already in GPU memory, but for files that get streamed from either main memory or a remote data host. With that in mind, they extended their above-described algorithm with a concept for streaming. And in doing so not only did they get the benefit of simply streaming the files, but they were also able to parse files that would otherwise not fit into the GPU's memory. What this furthermore enables is an earlier start of the parsing process. As it now is possible to start before the whole file is loaded into GPU's memory and thus "reducing overall latency" [2]. Streaming is done by simply dividing the input data into *partions* before copying it to GPU memory. These *partitions* have a fixed and equal *streamingPartitionSize* and can therefore be processed independently, similar to the chunks before, but on a higher level of abstraction.

4 EVALUATION AND DISCUSSION

For the evaluation of the implementation of their parsing algorithm, the authors used three machines. Two identical ones equipped with Nvidia Tesla V100-PCIe GPUs capable of doing the RDMA-GPUDirect part. And one with an Nvidia Tesla-SXM2 for the NVLink-related evaluations. In terms of datasets, they used a real-world dataset (*NYC Yellow Taxi Trips*) and a standardized dataset (*TPC-H Lineitem*) to obtain realistic, reproducible results and to compare their parser with others. They also used a synthetic dataset (*int_444*) for optimizing the tuning parameters before doing so [2].

4.1 Tuning Parameter

As stated in section 3, each chunk is processed by a single warp. So the chosen *chunkSize* dictates how much workload a warp has to cope with. As this means "an increasing size requires more hardware resources per warp" [2], one can see in Figure 5 that at a *chunkSize* above 2048 bytes, the throughput drops due to the overloading of the warps. Initially, a low throughput is also present at very small *chunkSizes* of about 128 bytes, because of "the overhead associated with scheduling, launching, and processing new [...] warps" [2]. As a result of that, the processing at these small *chunkSizes* happens to be more sequential than parallel. Thus, Kumaigorodski et al. concluded that the sweet spot between the two above-stated effects lies at a *chunkSize* of 1024 bytes, as the throughput dose peak around this value, as evident in Figure 5. However, what is most striking about Figure 5, is the very high throughput of up to 90 GB/s achieved on this homogeneous, easy-to-parse *int_444* dataset of unsigned short values residing in GPU memory.

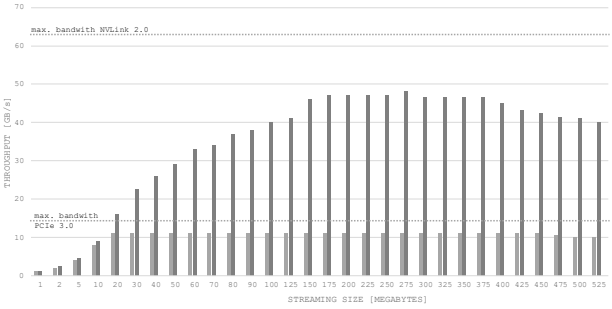

Fig. 7. Impact of *streamingPartionSize* on *int_444* (Own illustration based on [2])

Figure 6 shows how the performance goes up with an increasingly larger input file, due to a better utilization of parallelism. Parallelism is poorly exploited when the *inputSize* is not much larger than the *chunkSize*, because some warps are simply unloaded in this case. These situations could be avoided by using a very small *chunkSize* for *inputSizes* this small [1]. What Figure 6 moreover shows is that the throughput actually increases very quickly with respect to the size of the input, and is already very high at 100 MB.

Now having seen with Figure 6 how poor the performance is on small *inputSizes*, it becomes clearer why the authors earlier decided on the smaller *chunkSize* of 1024 bytes rather than 2048 bytes, as the throughput peaks between this two values in Figure 5.

Figure 7 shows how performance ramps up as *streamingPartionSize* increases, due to similar effects as it previously did on increasing *inputSizes*. While one machine's throughput continues to increase until it reaches a maximum of 48.3 GB/s, the other machine's maximum throughput is limited to 11 GB/s by PCIe 3.0. So what can be concluded from Figure 7 is that PCIe 3.0 is the bottleneck for end-to-end parsing, as it is unavoidable in the author's hardware setup for

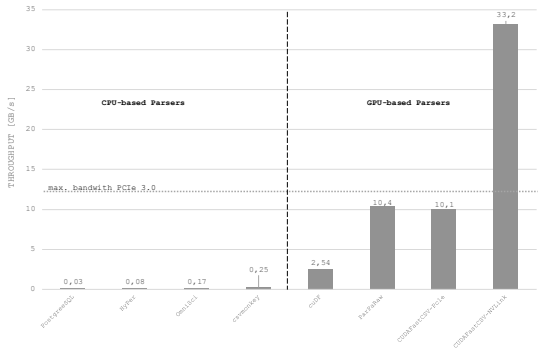

Fig. 8. End-to-end performance on *NYC Yellow Taxi Trips* (Own illustration based on [2])

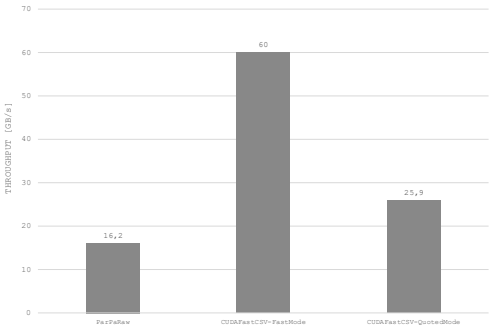

Fig. 9. On-GPU performance on *NYC Yellow Taxi Trips* (Own illustration based on [2])

streaming via the network. As previously stated, the parser is able to reach a throughput of up to 90 GB/s on this *int_444* dataset residing in GPU memory. And thus it is expected to saturate NVLink 2.0, which it fails to do "due to the limited amount of DMA copy engines, and due to the overhead from data and buffer management required for streaming. This", so Kumaigorodski et al. further explain, "leads to delays, as transfers and compute are not fully overlapped".

4.2 Results

Figure 8 shows the comparison of the performance in parsing a real-world dataset between this parser, called *CUDAFastCSV*, and other parsers, both GPU-based and CPU-based. Note that in this case, the dataset is initially in main memory of the host and therefore needs to be streamed to the respective parsing device. As Figure 8 demonstrates, and as already stated in section 1, the performance of CPU-based parsers on such a large dataset (1.9 GB) is far from that of GPU-based parsers. Especially since the results of the CPU-based parsers are more comparable to the NVLink version of *CUDAFastCSV* as they are hardware-wise not limited by the PCIe bus. Moreover, and since this is a real-world dataset containing some complex to-parse fields, it is worth noting that both *ParPaRaw* and *CUDAFastCSV* were able to saturate PCIe 3.0. But since it is not apparent from this graph whether *ParPaRaw* was limited by PCIe 3.0, Kumaigorodski et al. did another evaluation with the dataset initially residing within GPU memory to see if this was the case.

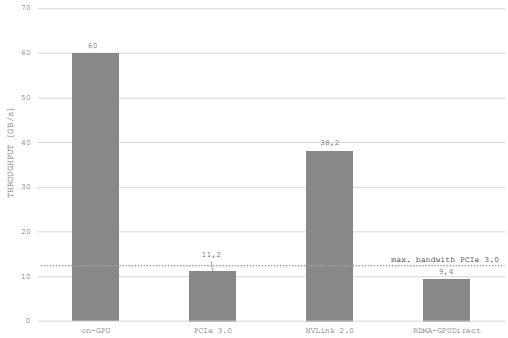

Fig. 10. Streaming performance on *NYC Yellow Taxi Trips* (Own illustration based on [2])

The results of that are shown in Figure 9 and it is now apparent that *ParPaRaw* actually was limited by PCIe 3.0 before as it is now able to parse at a throughput of 16.2 GB/s. But what Figure 9 also shows is that *CUDAFastCSV* is actually 3.7x faster than *ParPaRaw* using the same parsing device. "The reason [for that] is that we", as Kumaigorodski et al. try to explain this big margin, "are able to reduce the overall amount of work, as we do not need to track multiple state machines, and our approach is [thus] less processing-intensive".

What remains to be done is to measure the end-to-end streaming performance of *CUDAFastCSV* from a remote streaming host via *RDMA* and *GPUDirect*, which also internally uses the PCIe bus for connecting the *network-interface-card (NIC)* with the GPU's memory. The results of that, streaming the *NYC Yellow Taxi Trips* dataset used above, are shown in Figure 10, next to some baselines copied from these previous evaluations. What can be seen from that is that *CUDAFastCSV* also saturates the PCIe bus on remote streaming. However, the performance is slightly worse than with input data residing in the host's main memory, for reasons that are unclear and have already been reported by other researchers in the past [3]. Note that the presentation of the evaluations of the *TPC-H Lineitem* dataset has been omitted in this summary, as it merely underpins the results shown so far.

5 CONCLUSION

As pointed out in section 1, the author's goal was to build a CSV parser that could saturate modern interconnects such as InfiniBand NICs with bandwidths up to 100 GB/s. As seen in section 4, this goal was achieved to that extent as their parser *CUDAFastCSV* achieved a throughput of 90 GB/s on a homogenous, ease-to-parse dataset on their used hardware setting. But as Kumaigorodski et al. go on to explain, NVLink 2.0 allows two GPUs to compute in parallel, so scaling up too much higher throughputs is thus possible. And so, with the author's enthusiastic proclamation "[...] 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" [2], we conclude our roundup.

REFERENCES

[1] Alexander Kumaigorodski. 2020. Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing. (2020), 1–106. https://www.clemenslutz.com/pdfs/msc_thesis_alexander_kumaigorodski.pdf

[2] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing. In *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings (LNI)*, Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner (Eds.), Vol. P-311. Gesellschaft für Informatik, Bonn, 19–38. https://doi.org/10.18420/btw2021-01

[3] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2019. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *CoRR* abs/1903.04611 (2019). arXiv:1903.04611 http://arxiv.org/abs/1903.04611

[4] N.N. 2023. CUDA C++ Programming Guide. 12.1 (2023), 1–496. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[5] Elias Stehle and Hans-Arno Jacobsen. 2019. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *CoRR* abs/1905.13415 (2019). arXiv:1905.13415 http://arxiv.org/abs/1905.13415

[6] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (2019), 516–530. https://doi.org/10.14778/3303753.3303758