Report for 15-618 Project: Efficient CUDA / C++ Implementation of GpJSON

Rui Zhang
`AndrewID: ruiz3`

Xinyu Li
`AndrewID: xinyuli4`

May 1, 2026

1 Summary

This project studies the CUDA implementation of GPJSON, a GPU based system for JSONPath query processing over LDJSON data. Through Nsight Compute profiling, we identify that the main bottlenecks in the original implementation come from inefficient global memory accesses in the index builder and irregular data dependent memory accesses in the query executor. To improve index construction, we replace the original custom scan kernels with Thrust exclusive scan and introduce 16 byte vectorized global memory reads for file scanning kernels. These optimizations significantly reduce global load instruction pressure and memory throttle stalls. We also evaluate shared memory staging, constant memory for query IR, and CUDA bit manipulation intrinsics for structural character search, but these alternatives do not improve end to end performance due to extra overheads or limited impact on the dominant bottlenecks. On the 800 MB Twitter dataset, our optimized implementation achieves a 1.82x end to end speedup, with string indexing improved by about 5.09x and newline indexing by about 3.34x. The results show that targeted CUDA kernel optimization can substantially accelerate GPJSON index construction, while query execution remains mainly limited by irregular memory access and result materialization overhead.

2 Background

2.1 JSON and the JSONPath Query Language

JavaScript Object Notation (JSON) [1] is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data. Line Delimited JSON (LDJSON) [2] is a format for delimiting multiple JSON texts in a stream.

Each JSON text must be followed by a newline character and it is vastly used in many modern systems. After the structural layout of a JSON document has been identified, many applications do not need to materialise or inspect the entire document. Instead, they usually need to extract a limited number of fields from each JSON object, such as identifiers, timestamps, nested attributes, or values inside arrays. JSONPath was proposed as an XPath-like query language for JSON structures [3]. It provides a compact notation for navigating JSON objects and arrays from the root object to the target values, using object keys, array indexes, wildcards, and recursive descent operators. Although it is flexible and human readable, JSON is expensive to process. This cost becomes especially important for large scale analytics workloads, where parsing and query execution can dominate the total processing time [4].

2.2 Common Approches

To reduce this cost, modern JSON processors often avoid constructing a full object tree [5] and instead build structural indexes over the raw JSON text. A structural index records the positions of important characters, including braces, brackets, colons, commas, quotes, and newlines. These positions capture the logical layout of the document in a compact form. When evaluating a JSONPath query, the system can use the index to locate object boundaries, match field names, follow nesting levels, and skip regions that cannot contain the target value. This makes structural indexing particularly useful for LDJSON analytics, where the same query is repeatedly applied to many independent records.

Structural indexes also separate the expensive task of identifying JSON syntax from the later task of extracting values. Once the positions of strings, structural characters, and nesting levels are known, the query engine can reuse this information instead of reparsing the raw text for every query. Existing high performance JSON parsers, such as SIMD based CPU systems, exploit this idea to improve throughput. However, constructing a structural index is not simply a matter of scanning for structural characters. In JSON, characters such as {, }, `[`, `]`, `:`, and `,` only have syntactic meaning when they appear outside string literals. The same characters may also appear as ordinary text inside a string value. Therefore, a parser must first distinguish string regions from non-string regions before it can decide which structural characters are valid. This requires additional preprocessing beyond structural character detection. In particular, quotation marks define the boundaries of string literals, but quotation marks can themselves be escaped using backslashes. As a result, the parser also needs to identify escaped characters and determine which quotes are real string delimiters. Only after escape handling, quote detection, and string-region identification can the system safely construct a valid structural index for later query processing. For this reason, high performance JSON parsers(Mison[6], SIMDJson[7], Pison[8]) usually construct several intermediate indexes, such as escape indexes, quote indexes, string

indexes, and finally structural indexes. These indexes together provide the information needed to support efficient JSONPath evaluation.

2.3 GPJSON

GPJSON[9] is a GPU based JSON processing system designed for LDJSON input and JSONPath queries. Instead of constructing a full DOM tree for each JSON object, GPJSON first builds a set of structural indexes over the raw input text and then evaluates JSONPath queries on top of these indexes. At a high level, the system contains two main components: an index builder and a query executor. The index builder extracts structural information from the input file, while the query executor uses this information to locate and return the values requested by a JSONPath expression.

The index builder performs most of its work on the GPU. It first constructs a Newline index, which records the positions of newline characters in the LDJSON file. Since each line represents an independent JSON record, this index provides the record boundaries used later during query execution. GPJSON then builds several intermediate indexes to correctly identify string regions. This step is necessary because JSON structural characters such as braces, brackets, colons, and commas are only meaningful outside string literals. If such characters appear inside a string value, they must be ignored by the structural parser. Therefore, GPJSON constructs escape related indexes, quote related indexes, and carry indexes to handle dependencies across parallel GPU processing units. These intermediate results are used to determine which characters are inside strings and to produce the String index.

After the String index has been constructed, GPJSON builds the final Leveled bitmap index. This index records valid structural characters outside string literals, including object and array delimiters, colons, and commas, together with their nesting levels. The nesting level information is important because JSONPath evaluation depends not only on the position of a character but also on its structural context in the hierarchy of objects and arrays. For example, a field name at one nesting level should not be confused with a field of the same name inside a deeper nested object. The Leveled bitmap index therefore provides a compact representation of the JSON structure that can be traversed efficiently by GPU kernels.

For query processing, GPJSON separates compilation and execution. On the CPU side, a JSONPath query is first processed by a compiler. The compiler tokenizes the query, builds an abstract syntax tree, and translates the expression into a compact bytecode representation. This bytecode encodes the operations needed to navigate the indexed JSON structure, such as selecting an object key, accessing an array element, applying a wildcard, or evaluating a simple condition. Since this compilation step is performed once for a query, its cost can be amortized when the same query is applied to many LDJSON records. The generated bytecode is then executed on the GPU

by a query executor. During execution, GPU threads use the Newline index to determine the boundaries of the JSON records they are responsible for, and then use the Leveled bitmap index to navigate within each record according to the bytecode instructions. In this way, GPJSON avoids repeatedly parsing the raw JSON text during query evaluation and avoids materialising a full in memory object tree. The CPU is mainly responsible for query compilation, while the GPU handles the data intensive stages of index construction and JSONPath execution.

Although GPJSON provides an effective GPU based architecture for JSONPath query processing, we observed that there is still significant room for their cuda kernel optimization. Since the most performance critical stages are implemented as CUDA kernels, we therefore focus our work on improving the GPU kernel components rather than redesigning the overall system architecture.

3 Approach

We have tried out different optimization technique on various components of the GpJSON system. In specific, the two components that are related to CUDA kernel design are Index Building and Query Execution.

3.1 Index Building

In this section, we will discuss how we optimize the index builder of GpJSON. First, we will start by profiling the original implementation and identify bottlenecks of different kernels. Next, we will discuss two key optimization techniques that significantly improves execution time. Then, we will discuss several failed attempts to further optimize the index builder. These attempts either brings too much overhead and damage performance, or didn't impact performance for various reason. Either way, it's still valuable to understand why they didn't work. Finally, we will present the final result of the optimized index builder.

3.1.1 Profiling the Original Index Builder

We start by profiling the original index builder and identifying its bottlenecks. The index building process consists of four types of kernels: newline-related kernels, string-related kernels, leveled-bitmap-related kernels, and scan kernels. The newline-related kernels include `newline_count_index` and `newline_index`. The string-related kernels include `escape_carry_index`, `escape_index`, `quote_index`, and `string_index`. The leveled-bitmap-related kernels include `leveled_bitmap_carry_index` and `leveled_bitmap_index`. The scan kernels include `int_sum_scan`, `xor_scan`, and `char_sum_scan`.

We are mostly interested in the first three types of kernels. The scan-related kernels are also

Kernel	Time (ms)	Inst. tput. (%)	LG throttle (warp/issue)	Sectors / req.	Inst. executed (M)
`newline_count_index`	6.11	11.1	61.6	31.99	184.8
`newline_index`	6.83	17.9	19.7	31.45	343.2
`escape_carry_index`	12.08	5.0	175.6	32.00	143.3
`escape_index`	12.85	25.4	35.5	31.41	664.5
`quote_index`	12.86	28.6	20.2	31.63	973.1
`leveled_bitmap_carry_index`	5.89	57.4	0.0	6.24	830.7
`leveled_bitmap_index`	10.29	40.3	0.0	6.13	1077.1

Table 1: Selected Nsight Compute metrics for time-consuming kernels in the original index builder. `string_index` is omitted because it only takes around 0.6 ms.

expensive, but their bottleneck is less subtle. The original implementation uses custom scan kernels, which are less efficient than highly optimized scan implementations in Thrust. Therefore, we focus this subsection on profiling the newline, string, and leveled-bitmap kernels.

We run our C++ implementation of the original index builder on GHC machine (GHC 27) with an RTX 2080 Ti, and profile the result with Nsight Compute. Table 1 shows a selected list of metrics. The kernels can be roughly grouped into two types based on their bottlenecks.

The first group is the newline-related and string-related kernels. Their main bottleneck is inefficient global memory access when reading the input file. In these kernels, each thread owns a consecutive 64-byte chunk, or a multiple of 64 bytes, and scans its assigned bytes one byte at a time. Therefore, consecutive threads access bytes that are 64 bytes apart in global memory, causing severely uncoalesced global loads. This is reflected by the 31–32 memory sectors per global load request in Table 1, which is the worst case for uncoalesced global loads.

The stall metrics support this interpretation. For example, `newline_count_index` has only 11.1% instruction throughput, but its `lg_throttle` stall metric is 61.6. The `escape_carry_index` kernel is even more extreme: it has only 5.0% instruction throughput, while its `lg_throttle` stall metric reaches 175.6. This suggests that these kernels are not limited by arithmetic throughput, but by load/store pressure and latency from uncoalesced global loads.

The second group is the leveled-bitmap kernels. They use the same 64-byte-per-thread work assignment, so their raw file-byte accesses would still be uncoalesced. However, they do not need to read the raw file byte for every position. They take both the file and the string index as input, and only read the file byte when the byte is outside a string. The string index is stored as an array of 64-bit `long` values, so each thread only needs one coalesced string-index load for its 64-byte chunk.

This explains why the leveled-bitmap kernels have a different profile. Their `lg_throttle` stalls are almost zero, and they generate only about 6 sectors per global load request instead of 31–

32. On the Twitter dataset, 678,676,823 out of 842,479,395 bytes are inside strings. Therefore, 80.56% of the input does not require a raw file-byte check, and only 19.44% of bytes require such checks. If each lane in a warp performs the file-byte load with probability 0.1944, then a warp has about $32 \times 0.1944 = 6.22$ active lanes for that load, which is close to the measured 6.13–6.24 sectors per request.

Instead, the leveled-bitmap kernels are more affected by computation workload. For example, `leveled_bitmap_index` executes 1077.1 million instructions, compared with 184.8 million for `newline_count_index` and 143.3 million for `escape_carry_index`. This extra work comes from checking string membership, classifying structural tokens, and updating nesting-level state. Thus, leveled-bitmap construction is less limited by global load issue and more limited by local computation and control flow.

Implication on optimization directions Overall, the profiling results suggest two main optimization directions. First, the custom scan kernels should be replaced with a more optimized scan implementation. Second, the inefficient global loads in the newline-related, string-related, and leveled-bitmap-related kernels should be addressed. The least intrusive approach is to vectorize global loads, which reduces the number of memory requests without changing the original work assignment. A more radical approach which we did not attempt is to change the work assignment to make consecutive threads read consecutive bytes, but this is difficult because JSON index construction has long dependencies. For example, determining whether a byte is inside a string may depend on all previous bytes. The original implementation reduces this dependency length using carry indices, allowing each thread to process a small chunk sequentially. With a coalesced work assignment, each thread would instead read the file in a strided way, and there is no obvious way to preserve sequential processing within each thread. One middle ground would be to stage the files from global memory to shared memory in a coalesced way, and read files from shared memory. However, it's possible that the overheads may outweight the benefit.

3.1.2 Optimization: Replacing Custom Scans with Thrust Exclusive Scan

The first optimization is to replace the custom scan kernels in the original index builder with Thrust's `exclusive_scan`. The original implementation uses three scan kernels: `int_sum_scan`, `xor_scan`, and `char_sum_scan`. These scans are used to propagate count and carry information between chunks. Since scan is a well-studied parallel primitive, we expect a highly optimized library implementation to outperform the simple custom implementation.

Table 2 shows the result. Replacing the custom scans with Thrust significantly reduces the runtime of all three scan stages. The `int_sum_scan` and `char_sum_scan` kernels are improved by about 7.5× and 7.3×, respectively. The `xor_scan` kernel improves by 12.4×.

Scan kernel	Original time (ms)	Thrust time (ms)	Speedup
`int_sum_scan`	4.329	0.575	7.529×
`xor_scan`	4.442	0.359	12.373×
`char_sum_scan`	4.366	0.598	7.301×

Table 2: Speedup from replacing the original custom scan kernels with Thrust's `exclusive_scan`.

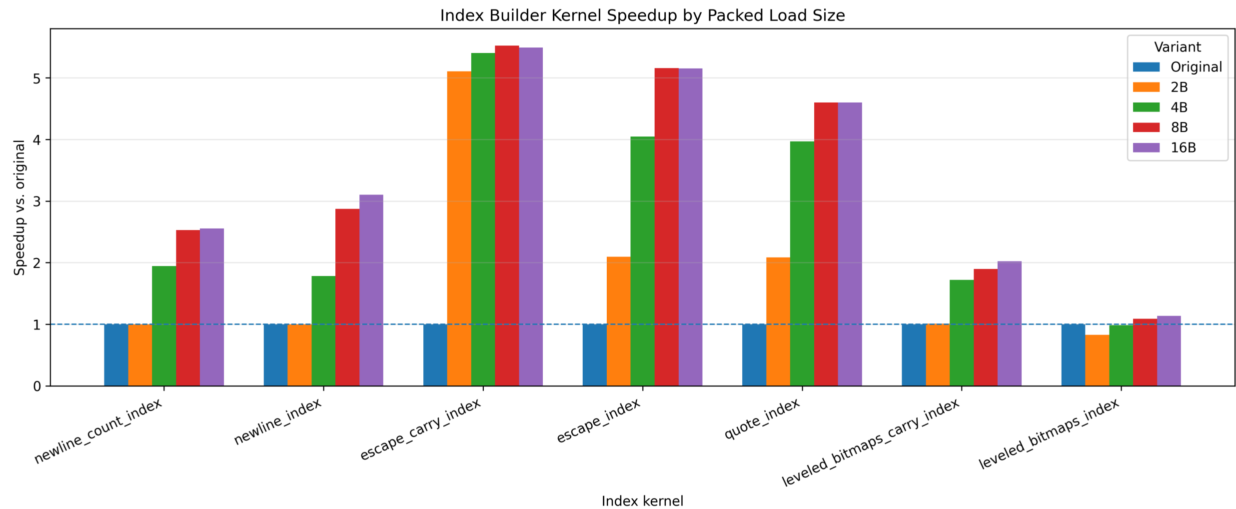


Figure 1: Speedup of vectorized global reads with pack sizes of 2, 4, 8, and 16 bytes, relative to original index builder.

3.1.3 Optimization: Vectorized Global Memory Reads

The second optimization is to vectorize global memory reads. As discussed in the profiling subsection, many kernels in the original index builder scan the input file one byte at a time. Each thread owns a consecutive 64-byte chunk, but consecutive threads access bytes that are 64 bytes apart. This causes severely uncoalesced global loads.

Motivation. To alleviate this problem, we can keep the same work assignment but reduce the number of global load instructions. This is done by making each thread loads a multi-byte, packed word from its chunk, and then scans the bytes inside the packed word locally.

We experiment with pack sizes of 2, 4, 8, and 16 bytes. We stop at 16 bytes because PTX vector load/store instructions are limited to 128-bit vectors [10] Figure 1 shows the speedup of different pack sizes over the original implementation.

Overall speedup. The speedup is significant for most newline-related and string-related kernels. For `escape_carry_index`, even 2-byte packing gives a speedup of 5.108×, and larger pack sizes only improve slightly. For `escape_index` and `quote_index`, the speedup increases until 8

Kernel	Orig. req. (M)	16B req. (M)	Orig. LG throttle (warp/issue)	16B LG throttle (warp/issue)
`newline_count_index`	26.33	1.65	61.6	0.15
`newline_index`	26.85	2.06	19.7	2.43
`escape_carry_index`	26.33	0.41	175.6	0.00
`escape_index`	26.86	2.06	35.5	0.22
`quote_index`	26.74	2.06	20.2	0.39
`leveled_bitmap_carry_index`	26.69	2.06	0.0	0.00
`leveled_bitmap_index`	27.10	2.47	0.0	0.33

Table 3: Selected Nsight Compute metrics comparing the original implementation with 16-byte vectorized global reads.

bytes, reaching 5.158× and 4.601× respectively. Using 16-byte packing does not further improve these two kernels. For the newline kernels, 2-byte packing provides almost no speedup, but larger pack sizes are effective. At 16 bytes, `newline_count_index` reaches 2.556× speedup, and `newline_index` reaches 3.104× speedup.

The leveled-bitmap kernels benefit less. The `leveled_bitmap_carry_index` kernel still reaches about 2.024× speedup at 16 bytes. However, `leveled_bitmap_index` only reaches 1.136× speedup, and the 2-byte version is slower than the original implementation. This suggests that vectorized reads are highly effective for some kernels, but the benefit depends on the original bottleneck and the extra work introduced by packing.

Effect on global load pressure. Table 3 compares the original implementation with 16-byte vectorized global reads. The number of global load requests drops by around an order of magnitude for most kernels. For example, `newline_count_index` reduces global load requests from 26.33 million to 1.65 million. Similarly, `escape_index` and `quote_index` both reduce global load requests to about 2.06 million.

This reduction also removes most of the `lg_throttle` stalls. For `escape_carry_index`, the `lg_throttle` stall metric drops from 175.6 to almost zero. For `newline_count_index`, it drops from 61.6 to 0.15. This confirms that vectorized reads directly address the load issue bottleneck identified in the original profile.

Vectorization does not fix coalescing. It is important to note that this optimization does not make the memory accesses any more coalesced. Adjacent threads still process different 64-byte chunks, so their packed loads are still far apart in global memory. For example, `newline_count_index` still has around 32 sectors per global load request after packing. The benefit is that the kernel issues far fewer global load requests. In other words, vectorized reads

Kernel	Original (M)	2B (M)	4B (M)	8B (M)	16B (M)
newline_count_index	184.8	340.0	234.6	165.5	142.5
newline_index	343.2	401.1	315.5	256.3	239.0
escape_carry_index	143.3	70.0	52.4	39.9	34.5
escape_index	664.5	445.3	366.8	314.5	289.8
quote_index	973.1	451.1	373.8	323.1	302.7
leveled_bitmap_carry_index	830.7	705.6	626.6	596.5	575.2
leveled_bitmap_index	1077.1	1143.6	1048.3	1014.0	993.3

Table 4: Instruction count across different global-read pack sizes.

do not fix the access pattern, but they make the same uncoalesced pattern happen fewer times.

Packing overhead. Packing is not always free. This is most visible when the pack size is small. A packed load reduces the number of global load instructions, but the kernel must still extract individual bytes from the packed word and compute their local offsets. This introduces extra integer instructions and indexing logic. When the pack size is only 2 bytes, this overhead is not always amortized. For example, `newline_count_index` has almost no speedup at 2 bytes, and `newline_index` also remains at around 1× speedup. Table 4 shows that this is not only a memory effect: the 2-byte version increases the instruction count of `newline_count_index` from 184.8 million to 340.0 million, and increases the instruction count of `newline_index` from 343.2 million to 401.1 million.

Larger pack sizes amortize this overhead better. For a 64-byte chunk, 2-byte packing still requires 32 packed groups per thread, while 16-byte packing only requires 4 packed groups per thread. Therefore, larger pack sizes reduce not only the number of global load instructions, but also the number of outer loop iterations and repeated address calculations. This explains why the newline kernels start to improve at 4 bytes and become much faster at 8 or 16 bytes.

Why some kernels saturate. The benefit of larger pack sizes eventually saturates. Once the `lg_throttle` bottleneck is mostly removed, reducing global load requests further has less impact. At that point, the remaining runtime comes from byte extraction, local index computation, branch/control-flow logic, and dependency stalls from the remaining loads.

This explains why `escape_carry_index` saturates especially early. In the original implementation, `escape_carry_index` is the clearest load-issue bottleneck: it has low instruction throughput, relatively small instruction count, and the largest `lg_throttle` stall metric among the profiled kernels. Even 2-byte packing is enough to remove most of this load issue pressure. After that, larger pack sizes continue to reduce load requests, but the dominant bottleneck has already

Kernel	No skip string (ms)	Skip string (ms)	Relative speedup
`leveled_bitmap_carry_index`	3.097	3.223	0.961×
`leveled_bitmap_index`	9.515	9.911	0.960×

Table 5: Effect of skipping packed file loads for bytes inside strings in leveled-bitmap kernels.

shifted away from `lg_throttle`. Therefore, 4-byte, 8-byte, and 16-byte packing only provide small additional improvements.

For `escape_index` and `quote_index`, the speedup saturates later, around 8 bytes. These kernels still benefit from reducing repeated file loads, but they also perform more local index computation and control-flow work than `escape_carry_index`. Once 8-byte packing removes most of the load issue pressure, the remaining work limits further speedup at 16-byte packing.

Leveled-bitmap kernels. The leveled-bitmap kernels benefit much less from this optimization. This matches the profiling result from the original index builder. Leveled-bitmap construction is not mainly limited by global load issue. It also performs many computation and branching to check string membership, classify structural tokens, and update nesting-level state. Table 4 also shows this effect: even at 16 bytes, `leveled_bitmap_index` still executes 993.3 million instructions, only slightly lower than the 1077.1 million instructions in the original implementation. This explains why its final speedup is only 1.136×.

For leveled-bitmap construction, packed reads also interact with the original string-skipping behavior. In the original implementation, the kernel can check the string index before reading the raw file byte, so it only needs to read the file byte when the corresponding position is outside a string. In the packed implementation, the no-skip version instead loads the packed file word first, and then checks the string mask for each byte inside the word. Thus, it unnecessarily reads some file bytes that are inside strings, even though those bytes are not semantically needed for structural-token classification. We tested a skip-string variant that skips the packed read if string index indicates all bytes are in-string.

Table 5 shows that the skip-string variant does not improve performance, and even makes it slightly worse. The Nsight metrics confirm that the variant does reduce memory traffic: for both leveled-bitmap kernels, the number of global load sectors drops by about 32%. However, it does not reduce the number of global load requests, which remains almost unchanged. Therefore, the skip-string variant does not reduce load-instruction pressure, and the `lg_throttle` stall metric is already close to zero for both versions.

Instead, the skip-string variant makes the local control flow more expensive. For `leveled_bitmap_index`, the instruction count increases from 993.3 million to 1096.5 million, and the number of branch

instructions increases from 277.3 million to 323.0 million. The branch uniformity also decreases, indicating more divergent control flow. As a result, the saved memory sectors are outweighed by the extra instruction and branch overhead. Therefore, the final packed implementation uses the simpler no-skip path for leveled-bitmap construction, even though it reads more file bytes than necessary.

Takeaway. Overall, vectorized global reads are effective because they reduce global load instruction pressure while preserving the original dependency-friendly work assignment. They provide the largest speedups for kernels whose original profiles were dominated by `lg_throttle` stalls. They provide smaller speedups for leveled-bitmap construction, where local computation and control flow dominate. In the final implementation, we use 16-byte packing because it gives the best or near-best performance across the index-builder kernels.

3.1.4 Failed Attempt: Staging File Chunks in Shared Memory

Motivation. The vectorized global-read optimization reduces the number of uncoalesced global loads, but it does not make the global load any less uncoalesced. Consecutive threads still process different 64-byte chunks, so their direct global reads are still far apart. When we want to avoid modifying work assignment while still having coalesced global read, a natural next attempt is to stage into share memory a contiguous file region that covers all threads' bytes in a thread block. In this design, threads in a block first cooperatively load a contiguous region of the file into shared memory, and then each thread reads its assigned chunk from shared memory. The hope is that the staging phase can use coalesced global reads, while the later per-thread processing reads from faster shared memory.

Layouts and variants. We evaluate this idea on `escape_carry_index`, which is a favorable case because the original kernel is strongly affected by global load issue pressure. All shared-memory variants use 16-byte packed global-memory loads for the staging phase. We vary the shared-memory pack size and layout.

In the non-transposed layout, each thread's chunk is stored contiguously in shared memory:

$$\text{smem[thread id][group id]}.$$

This is simple, but when a warp reads the same group from different threads, consecutive lanes access shared memory with a stride of the number of groups per thread. This can cause many lanes to hit the same shared-memory bank. In the transposed layout, the tile is stored as

Variant	Gmem pack	Smem pack	Time (ms)	Speedup
Global only	1B	–	12.161	1.000×
Global only	16B	–	2.127	5.718×
Smem	16B	1B	16.042	0.758×
Smem transposed	16B	1B	4.861	2.502×
Smem	16B	16B	2.198	5.533×
Smem transposed	16B	16B	2.345	5.186×

Table 6: Performance of shared-memory staging variants for `escape_carry_index`. All shared-memory variants use 16-byte packed global-memory staging.

$$\text{smem[group id][thread id]}.$$

Now consecutive lanes reading the same group access adjacent shared-memory locations. For 1-byte shared-memory reads, this layout is much more bank-friendly during the computation phase.

Result. Table 6 shows that even the best variant shared-memory staging is still slightly worse than direct packed global reads. The best shared-memory variant takes 2.198 ms, while the direct 16-byte global read version takes 2.127 ms. However, the intermediate results are also very interesting. The non-transposed 1-byte shared-memory version is much slower than the original, taking 16.042 ms. Transposition reduces this to 4.861 ms, showing that layout matters significantly. With 16-byte shared-memory packing, the staged version becomes competitive, but it still does not outperform direct 16-byte global reads.

Analysis. The staging phase does improve global-memory coalescing. The direct 16-byte global-read version generates 31.20 memory sectors per global load request and 51.344 million global sectors. The shared-memory variants reduce this to 16.00 sectors per request and 26.332 million global sectors. Thus, shared-memory staging achieves its intended global-memory effect.

The problem is the cost of consuming the staged data from shared memory. For 1-byte shared-memory reads, the non-transposed layout generates 496.773 million total shared-memory bank conflicts. The transposed layout reduces this to 101.902 million, a 79.5% reduction. This happens because transposition almost eliminates read-side conflicts: shared-memory load bank conflicts drop from 395.001 million to 0.002 million. The store-side conflicts remain roughly the same, changing from 101.772 million to 101.901 million. Therefore, for 1-byte shared-memory reads, transposition reduces total bank conflicts rather than merely moving conflicts from reads to writes.

For 16-byte shared-memory reads, the behavior is different. The non-transposed layout already has much fewer shared-memory operations, so there is less read-side conflict to remove. Transposition reduces load bank conflicts from 19.754 million to zero, but it increases store bank conflicts from 0.644 million to 25.996 million. As a result, total bank conflicts increase from 20.398 million to 25.996 million, and runtime worsens from 2.198 ms to 2.345 ms. Thus, transposition is very effective for byte-sized shared-memory reads, but shared-memory packing changes the tradeoff.

Overall, shared-memory staging improves global coalescing, and the transposed layout is effective when shared-memory reads are byte-sized. However, after global reads are already vectorized, the remaining global-memory benefit is too small to pay for the shared-memory stores, shared-memory loads, extra indexing, bank conflicts, and synchronization. Therefore, we do not use shared-memory staging in the final implementation.

3.1.5 Final Optimized Index Builder

The final optimized index builder includes the two optimizations that significantly improves performance in our experiments. First, we replace the custom scan kernels with Thrust's `exclusive_scan`. Second, we use 16-byte packed global reads for the file-scanning index kernels. This pack size gives the best or near-best speedup across the kernels we tested, and it is the largest single vector load size supported by PTX.

We do not include shared-memory staging in the final implementation because it improves global-memory coalescing but introduces extra shared-memory traffic, bank conflicts, indexing overhead, and synchronization.

3.2 Query Execution

The query part starts with query compilation. A JSONPath query, such as $.user.lang, is first parsed and compiled into a compact intermediate representation (IR). This IR is a bytecode-like sequence of operations. Each opcode describes one step of the query, such as moving to a key, moving down into a nested object, checking an array index, evaluating a string predicate, or storing a matched result. For example, $.user.lang is compiled into a sequence similar to: move to key user, move down one level, move to key lang, store the result, and end the query. The query kernel evaluates a compiled JSONPath query over JSON records in parallel. During query execution, each GPU thread is responsible for one or more JSON records. For each assigned record, the thread interprets the query IR bytecode from the beginning, checking an array index, and storing a result. While interpreting the IR, the thread repeatedly uses the indices to locate the next relevant structural character and then reads the original file buffer to verify keys or values.

3.2.1 Profiling of original query executor

We also start by profiling the original query kernel. The profiling results showed that the query kernel was strongly limited by memory behavior. The achieved memory throughput was only about 35%, and each thread accessed only around 5.5 bytes on average per memory request. This indicated poor memory coalescing: although the kernel performs many global memory accesses, the warp-level memory requests do not efficiently use the available memory bandwidth. This behavior comes from the query access pattern. Different threads in a warp process several consecutive JSON records, and each record has different field positions and lengths. Therefore, accesses to `file`, `string_index`, and `leveled_bitmaps_index` are highly data-dependent and often uncoalesced. Besides, each thread only access its own region of index and file, which made shared memory a poor fit for the query kernel.

Based on these profiling results, the natural optimization target was to reduce unnecessary global memory traffic and shorten repeated index-search operations. Since improving coalescing directly would require changing how records are assigned to threads or redesigning the data layout, I focused on smaller, local changes first. In particular, I investigated whether read-only query data could be placed in a faster memory space, and whether structural-character search could be made more efficient by reducing redundant bitmap loads.

3.2.2 Attempt 1: Moving The Query IR To Constant Memory

The first optimization attempt is to put query IR into constant memory. I observed that both the query IR and those indexes are read-only during query execution. In principle, read-only data may benefit from constant memory. As a result, if we can put them into constant memory, it should run faster. However, profiling and size analysis showed that the indices are not suitable for constant memory. The string index and leveled bitmap index scale with the input size, and for large datasets they are far larger than the constant memory capacity.

Therefore, I only tried moving the query IR into constant memory. This seemed promising because all threads execute the same query and often read the same IR locations at similar times, which is the access pattern constant memory is designed to support.

Table7 shows the result of running two kernels on different datasets. The result suggests that IR access is not a bottleneck. The IR for a query such as $.user.lang is very small, usually only a few bytes or tens of bytes. Each thread reads only a small number of opcodes and operands, and some threads may terminate early if the record does not match the query path. Even when the IR is stored in global memory, it is small enough to be cached effectively. The result therefore indicated that the dominant memory cost came from the file buffer and index structures, not from the IR.

Kernel	100KB (ms)	100MB (ms)	800MB (ms)	1600MB (ms)
original	0.220	1.092	8.117	15.979
const-opt	0.221	1.090	8.126	15.948

Table 7: Effect of using constant memory

3.2.3 Attempt 2: Optimizing Structural Character Search

The second optimization targeted the helper functions used to find structural characters, such as `find_next_structural_char` and `find_previous_structural_char`.

In the original implementation, the search logic scans through bitmap positions step by step. It first loads the bitmap word corresponding to the current position, then checks whether the current bit is set. If not, it increments or decrements the position and checks again. This can introduce redundant memory operations and repeated bit checks, especially when many nearby bits are not structural characters. In the worst case, if the last bit of the word is set, this scan logic need to go through all 64bits which results in 64 global memory access.

Based on this observation, I tried using CUDA bit-manipulation intrinsics[11]. `__ffsll` can find the first set bit in a 64-bit word, and `__clzll` can be used to locate the last set bit. With these intrinsics, the kernel can load one bitmap word, mask out irrelevant bits, and directly jump to the next or previous structural bit. The expected benefit was fewer bitmap checks and thus fewer global memory loads.

Table8 shows the execution time of each kernel running on different size of twitter dataset. As we can see from the figure, we do have a little bit improvement on 100KB and 100MB input. However, for 800 MB and 1.6 GB inputs, the optimized kernel was slightly slower.

I then profiled the optimized kernel again to see if our optimization real worked. The profiling results confirmed part of this hypothesis: for 800MB dataset, the optimized version did reduce the number of global loads by 81% and thus reduced the number of long-scoreboard-stall. It also reduced divergent branches by 61%. However, compared to original design the average cycles a warp spend on long-scoreboard-stall increased from 55.0 cycles to 214.5 cycles. As a result, even though we have less number of stalls, the overall stall cycles increased. This is especially obvious on larger datasets. On larger datasets, the optimized kernel became slightly slower. In other words, the kernel performed fewer memory operations, but the remaining memory operations were more expensive.

The likely reason is that the optimization changed the latency profile of the kernel. The original scan performs more local, incremental accesses. Even though it does more work, many of these accesses are close to each other and can benefit from cache locality. The optimized version loads

Kernel	100KB (ms)	100MB (ms)	800MB (ms)	1600MB (ms)
original	0.220	1.092	8.117	15.979
intrinsic-opt	0.105	1.073	8.626	17.948

Table 8: Effect of using CUDA intrinsics

a bitmap word, uses `__ffsll` or `__clzll` to jump directly to the next structural position, and then immediately uses that result to access file or another index location. This creates a tighter dependency chain: the next memory address depends directly on the result of the previous bitmap load.

For large datasets, the working set is much larger and cache locality is weaker. The profiling showed that for 800MB dataset, the L1 and L2 cache hate rate decreased by 74.90% and 57.79% respectively in the optimized version. As a result, the remaining global loads were more likely to miss in cache, and because they were on a dependent path, their latency was harder to hide. This explains the observed behavior: fewer global loads, but longer long scoreboard stalls.

Therefore, the performance-debugging conclusion is that this optimization improved a local operation but did not improve the dominant bottleneck. The query kernel is mainly limited by irregular, data-dependent global memory accesses to the file and index structures. Reducing the number of bitmap checks is helpful only when those accesses are cache-friendly, which is more likely on small datasets. On large datasets, the cost of cache-missing dependent loads dominates, so the bit-scan optimization does not translate into end-to-end speedup.

4 Results

All of our experiments are run on ghc machines.

4.1 Testing on different benchmark size

The datasets used in this experiment were derived from a common base dataset of approximately 800 MB. The input data is Twitter JSON data, and all experiments use the same JSONPath query, ["$.user.lang"], which extracts the language field from the user object of each tweet record. Smaller datasets were constructed by taking prefixes of the base file using *head*, so they preserve the original record order and structure while reducing input size. The 1.6 GB dataset was constructed by concatenating the 800 MB dataset twice using cat, creating a larger input with the same data distribution but roughly double the size.

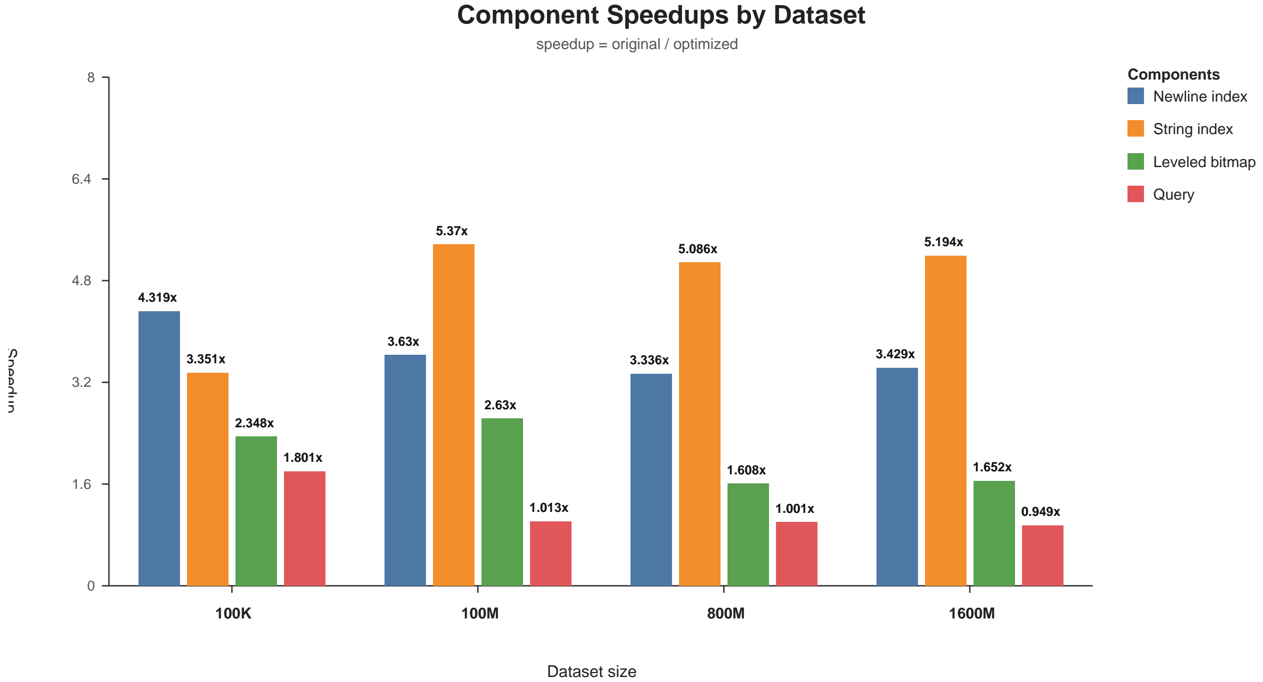


Figure 2: component-speedup

4.1.1 Analysis

Figure2 shows the component-wise speedup. The optimized implementation does not improve all parts of execution uniformly. Instead, the performance gain is mainly concentrated in the index construction components, especially string indexing and newline indexing. Across the larger datasets, the optimized version achieves strong speedups for these indexing stages, while the query component remains almost unchanged.

The largest improvement comes from the string index component. On the 800MB dataset, string indexing improves from 43.205 ms to 8.495 ms, which is about 5.09x speedup. On the 1600MB dataset, it improves from 86.095 ms to 16.577 ms, about 5.19x speedup. Newline indexing also benefits significantly on large inputs. Its speedup is about 3.34x on the 800MB dataset and about 3.43x on the 1600MB dataset. This suggests that the optimization is not limited to string-related scans, but also improves the newline indexing once the dataset is large enough for memory traffic to dominate.

Leveled bitmap indexing improves more moderately. It reaches about 1.61x speedup on the 800MB dataset and about 1.65x on the 1600MB dataset. This indicates that the optimized implementation still helps this stage, but the benefit is smaller than for string indexing and newline indexing.

Query execution shows almost no improvement. Its speedup is close to 1x across most dataset sizes and is slightly below 1x on the 1600MB dataset. This is because the query-kernel opti-

mization does not scale well on the large datasets. Although the optimized query kernel reduces some redundant bitmap searches, profiling shows that it also suffers from longer memory stalls on large inputs, where cache misses and irregular global memory accesses become more significant. Therefore, the optimized query kernel itself provides little benefit, and can even be slightly slower.

Moreover, the query kernel is not the dominant part of the overall query stage. The CUDA query kernel accounts for only about 20% of the total query time. Most of the query-stage cost comes from result handling, especially `append_query_line_results`. As a result, even if the query kernel is improved, the overall query execution time changes only slightly. This explains why the query component remains close to 1x speedup in the component-wise results.

Figure3 shows the execution time decomposition for each dataset. For the large datasets, this bottleneck becomes especially visible. After index construction is optimized, string indexing and newline indexing become much faster, but query result handling remains mostly unchanged. As a result, the remaining bottleneck shifts toward the query stage, especially append and materialization work.

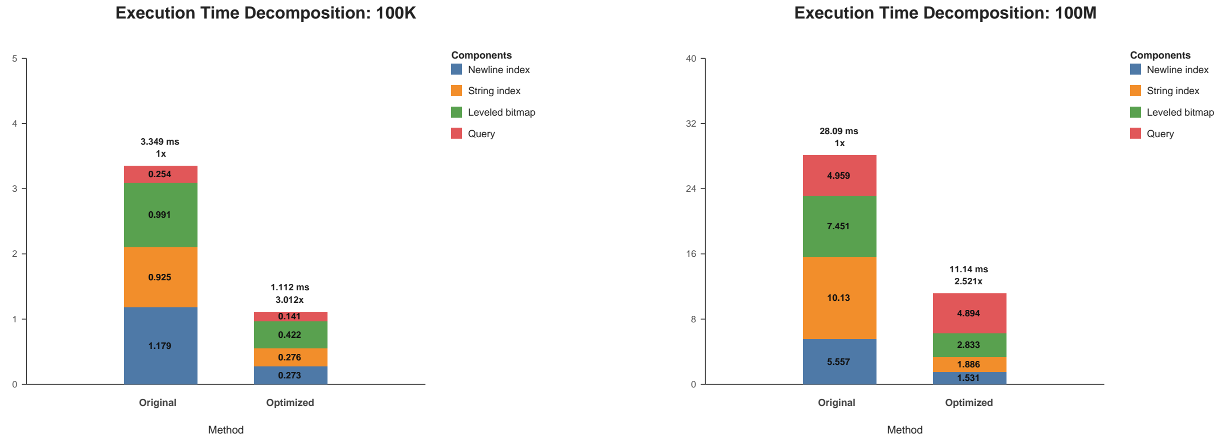
Overall, the optimized implementation is effective because it substantially accelerates index construction, especially string indexing and newline indexing. For large datasets, this reduces the indexing cost significantly, but the end-to-end speedup is limited by query result handling, which is mostly unchanged by the current kernel optimizations.

4.2 Testing on different partition size

GpJSON can process the input file either as one large partition or as several smaller partitions. Previously, we have been using the entire file as a single partition. In real usage, large JSON files may need to be split into partitions because the full file and its auxiliary indices may not fit in GPU memory. Therefore, file partition size is an important practical parameter. We evaluate the impact of partition size on end-to-end performance using five partition sizes: full file (800 MB), 50 MB, 100 MB, 200 MB, and 500 MB. The full-file setting is denoted as partition size 0 and is used as the baseline.

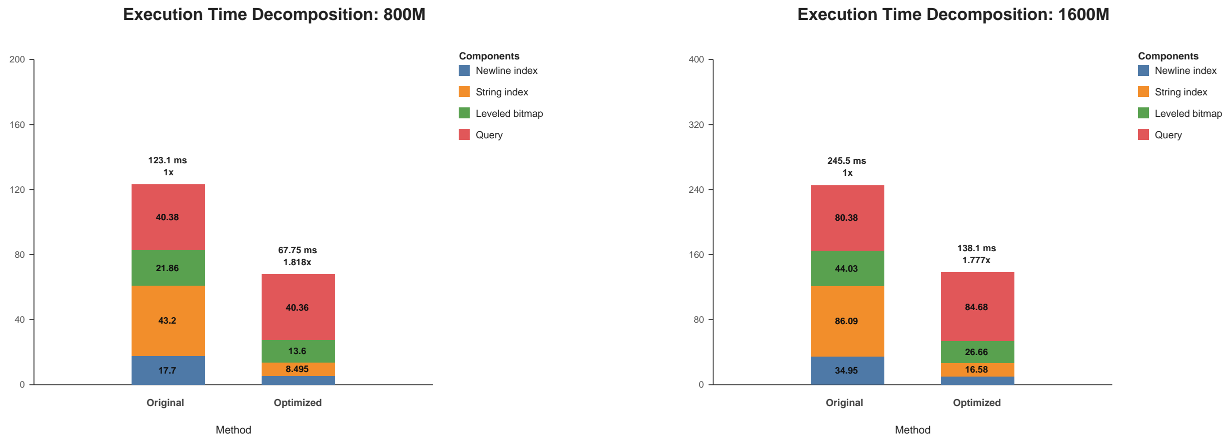
Table 9 shows the speedup of different partition sizes relative to the full-file setting. Smaller partitions improve the query phase. For example, 50 MB partitions improve `execute_query` by 1.475× compared to the case of single-file partition. However, smaller partitions hurt end-to-end performance once index construction is included. The same 50 MB setting only reaches 0.629× end-to-end speedup. The 500 MB setting is close to the full-file baseline, but is still slightly slower at 0.979×.

The main tradeoff is between query locality and index-building overhead. At 50 MB, `execute_query`



(a) 100KB dataset

(b) 100MB dataset



(c) 800MB dataset

(d) 1600MB dataset

Figure 3: Execution Time Decomposition for different dataset

Partition size	Full	50 MB	100 MB	200 MB	500 MB
`Engine::query` speedup	1.000×	0.629×	0.788×	0.893×	0.979×
`execute_query` speedup	1.000×	1.475×	1.310×	1.238×	1.120×
`build_index` speedup	1.000×	0.244×	0.397×	0.576×	0.834×

Table 9: Speedup of different partition sizes relative to the full-file setting. `Engine::query` is the end-to-end execution; `build_index` is the index building portion; `execute_query` is the query execution portion.

Metric	Full	200 MB	50 MB
Query kernel time (ms)	8.539	7.738	3.719
Instructions executed (M)	87.312	87.759	89.193
L2 sector hit rate (%)	34.846	43.267	81.048
Long scoreboard stall	205.114	163.202	27.892
Avg. warp latency (cycles)	210.681	171.604	34.655

Table 10: Selected Nsight Compute metrics for the query kernel across partition sizes.

decreases from 82.341 ms to 55.843 ms. However, `build_index` increases from 117.025 ms to 480.534 ms, and `Engine::query` increases from 570.428 ms to 907.594 ms. Thus, the query-side improvement is much smaller than the additional index-building cost.

To understand why smaller partitions improve query execution, we profile the query kernel with Nsight Compute. Table 10 compares the full-file, 200 MB, and 50 MB settings.

The query kernel does not become faster because it performs less work. The instruction count is almost unchanged across partition sizes: 87.312 million for the full-file setting, 87.759 million for 200 MB partitions, and 89.193 million for 50 MB partitions. Instead, the improvement comes from better memory locality. The L2 sector hit rate increases from 34.846% to 81.048% when moving from the full-file setting to 50 MB partitions. At the same time, long scoreboard stalls drop from 205.114 to 27.892, and the average warp latency drops from 210.681 cycles to 34.655 cycles. This suggests that smaller partitions speed up the query kernel by improving cache locality and reducing memory-latency stalls.

The end-to-end runtime still worsens because index construction is repeated for more partitions. This overhead is especially visible in the scan stages. At 50 MB, `int_sum_scan` increases from 4.853 ms to 80.423 ms, `xor_scan` increases from 4.687 ms to 75.721 ms, and `char_sum_scan` increases from 4.999 ms to 81.457 ms in partition-summed time. Although each scan is smaller, the scan setup, kernel launches, and coordination overhead are paid once per partition.

Overall, partitioning is useful for handling files that cannot fit in GPU memory as a single partition, and it can improve query-kernel locality. However, when index construction is included in the end-to-end runtime, smaller partitions hurt performance because they repeat scan-heavy preprocessing work. For the evaluated dataset, the full-file setting gives the best end-to-end runtime.

5 List of work by each student

Both team members contributed equally to the project, with a 50%–50% credit distribution.

For the C++ implementation of GpJSON, Xinyu designed the skeleton system and implemented

the CUDA device array and index builder subsystem. Rui implemented the query compilation, file reader, and query executor subsystems.

For optimization, Xinyu focused on optimizing the index builder, while Rui focused on optimizing query execution.

For experiments and report writing, Xinyu wrote the index-builder optimization journey and evaluated the optimized GpJSON implementation under different partition sizes. Rui wrote the query-executor optimization journey and evaluated the optimized GpJSON implementation on benchmarks of different sizes.

References

[1] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014. URL `https://www.rfc-editor.org/info/rfc7159`.

[2] Hoeger, Thorsten and Dew, Chris and Pauls, Finn and Wilson, Jim. NDJSON – Newline Delimited JSON. `https://github.com/ndjson/ndjson-spec`, 2014. Version 1.0.0; created 2013-07-05; last updated 2014-10-19; accessed 2026-04-27.

[3] Stefan Gössner. JSONPath – XPath for JSON. `https://goessner.net/articles/JsonPath/`, 2007. Accessed: 2026-04-27.

[4] W3Techs. Usage statistics of json-ld for websites. `https://w3techs.com/technologies/details/da-jsonld`, 2026. Accessed: 2026-04-27.

[5] RapidJSON. Document Object Model (DOM). `https://rapidjson.org/md_doc_dom.html`, 2024. Accessed: 2026-04-27.

[6] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: a fast json parser for data analytics. *Proc. VLDB Endow.*, 10(10): 1118–1129, June 2017. ISSN 2150-8097. doi: 10.14778/3115404.3115416. URL `https://doi.org/10.14778/3115404.3115416`.

[7] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28:941–960, 2019. doi: 10.1007/s00778-019-00578-5. URL `https://doi.org/10.1007/s00778-019-00578-5`.

[8] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable structural index construction for json analytics. *Proc. VLDB Endow.*, 14(4):694–707, December 2020. ISSN 2150-8097. doi: 10.14778/3436905.3436926. URL `https://doi.org/10.14778/3436905.3436926`.

[9] AtLarge Research. GPJSON: A GPU-Accelerated JSON Processing System. https://atlarge-research.com/pdfs/2025-vldb-gpjson.pdf, 2025. Accessed: 2026-04-27.

[10] *PTX ISA*. NVIDIA Corporation, 2022. URL https://docs.nvidia.com/cuda/archive/11.7.1/parallel-thread-execution/index.html#vectors. CUDA Toolkit v11.7.1, PTX ISA Version 7.7, Section 5.4.2: Vectors.

[11] NVIDIA Corporation. *CUDA Math API Reference Manual: Integer Intrinsics*. NVIDIA Corporation, 2026. URL https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__INTRINSIC__INT.html. Version 13.2. Accessed: 2026-05-01.