GPU-based JSON data processing using structural indexes

Vlaswinkel, Koen R.

Award date:
2021

MSc Thesis

GPU-based JSON data processing using structural indexes

K.R. (Koen) Vlaswinkel

Supervisor:
D. Bonetta

Assessment committee members:
D. Bonetta
J.J. Vinju
N. Yakovets

Department of Mathematics and Computer Science

Database Group

Eindhoven, July 2021

Abstract

In recent years, large amounts of data are being increasingly generated and stored every day. Big data is often processed by different software systems, which require a common data interchange format. JavaScript Object Notation, or JSON, is one of the most popular data exchange formats and is widely used in web and data-intensive applications. Unfortunately, parsing and processing JSON data is often a bottleneck in data processing pipelines due to the lacking performance of JSON.

To improve JSON data processing performance, recent work has proposed the usage of auxiliary data structures such as structural indexes and speculative data access. All such existing techniques use CPU-based sequential processing, although some recent systems have proposed using Single Instruction Multiple Data (SIMD) instructions to accelerate and partly parallelize JSON data access.

While SIMD-based parallelization could bring significant speedups over sequential approaches, modern hardware architectures often provide even more parallel data processing opportunities thanks to massively parallel Graphical Processing Units (GPUs).

In recent years, GPUs have been employed in multiple data processing domains and have demonstrated that they can speed up various data processing tasks. However, none have explored using GPUs to massively parallelize JSON data processing.

This thesis explores the usage of GPUs for parsing and querying large JSON documents from high-level dynamic languages. To our knowledge, we present the first GPU-based JSON query evaluation engine, which is able to speed up query execution compared to state-of-the-art JSON parsers and query engines. We show how existing sequential techniques can be adapted to run on the GPU and how novel techniques can be used to evaluate queries in parallel on a GPU. Our implementation is compared to existing state-of-the-art JSON parsers and query engines. We also compare our implementation to JSON query engines available on Node.js since our implementation is usable from high-level scripting languages due to the use of GraalVM for exposing the GPU to languages such as JavaScript and Python.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the last years, big data and machine learning are increasing in popularity. While these technologies are promising, some steps need to be taken before data can be processed. Among others, this includes pre-processing and extracting data. Popular human-readable formats for supplying data are CSV, XML, and JSON. Often, these are parsed sequentially and can be the bottleneck when doing query processing on data analytics [4]. Therefore, faster methods are necessary.

One solution to speed up the parsing and querying performance of JSON data is parallel data processing. In this context, the excellent availability of highly parallel GPU architectures on common commodity hardware corresponds to an opportunity to expose high-performance data analytics even to high-level dynamic languages such as JavaScript or Python. GPUs have already been employed successfully in the context of large data processing systems (e.g., GPU-based databases) as well as to speed up the processing of CSV [3] and XML [5] datasets. However, there is currently no published algorithm or software system for JSON parsing and query processing on the GPU.

Moreover, existing fast JSON parsers are only available for relatively low-level languages such as C++, making it hard for everyone to have access to fast JSON parsers. Higher-level scripting languages, such as Python and JavaScript, are being used for data visualization and processing, making it interesting to explore whether a fast JSON parser can be developed for high-level scripting languages.

Due to these developments and techniques, it is interesting to explore whether JSON parsing and query processing on the GPU is possible and what benefits it brings when exposed to popular dynamic languages.

1.2 Problem definition

Various general-purpose GPU-based algorithms have shown that algorithms run on GPUs can outperform their CPU-based counterparts. This has been shown for various data formats, such as CSV [3] and XML [5]. However, to our knowledge, no GPU-based system exists for parsing and querying JSON documents.

While multi-threading or SIMD-based parallel processing techniques for JSON data have been proposed in recent years, no GPU-based implementation exists. Despite the excellent availability of massively parallel processing threads, GPU hardware operates on a separate memory space than that of a running application. This memory separation, as well as the complex programming model of GPUs, make it challenging to develop fully-parallel processing techniques for JSON data. Moreover, JSON is not a data format designed for parallel parsing. This makes it hard to fully parallelize all parsing steps without synchronization or extensive communication between parallel threads.

Given the above challenges and the great potential of GPUs, this thesis will explore whether it is

possible to develop a GPU-based JSON query processing system with equivalent or better performance than state-of-the-art systems while making it available to popular dynamic scripting languages.

1.3 Objective

Our project is mainly an exploration of GPU-based parsing and querying algorithms, specialized to JSON. Our main goal is to develop a real-world open-source implementation of a GPU-based JSON analytics framework. In addition, our implementation needs to be evaluated against state-of-the-art JSON parsers to show its performance characteristics. The goal is to outperform state-of-the-art JSON parsers available in low-level languages while making the system available to high-level scripting languages.

1.4 Contributions

We provide a novel approach to evaluating queries on newline-delimited JSON documents on a GPU using CUDA. Our implementation outperforms state-of-the-art JSON processing engines on large datasets by running a large majority of the query processing pipeline on a GPU.

The techniques described in this thesis are based on structural indexes, i.e., auxiliary data structures that are created at runtime by our JSON processing engine and are used to parse and query JSON data. While structural indexes have been used in several data processing domains (including sequential [6] and SIMD-based [4, 7] JSON processing), we present the first GPU-based techniques to build and query structural indexes for JSON using a GPU.

Moreover, to fully exploit the parallelism of GPU-based data processing, we introduce a bytecode-based interpreter that runs entirely on the GPU and can execute arbitrary queries on structural indexes. Our implementation demonstrates that such a bytecode interpreter on a GPU is a feasible option for querying JSON values using structural indexes. We also demonstrate that the querying language for JSON queries is independent of the query evaluation.

We show that our GPU-based implementation consistently outperforms state-of-the-art JSON processing solutions for high-level dynamic languages such as JavaScript, as well as low-level hand-optimized highly-efficient C++ SIMD-based implementations. We also show that query processing in our system can be further extended by implementing query expression evaluations directly on the GPU.

1.5 Outline

Chapter 2 introduces the background of this project and related work. Chapter 3 gives an overview of how we create a JSON query evaluation engine. Chapter 4 describes how to construct structural index on-GPU. In Chapter 5 we describe how the structural index can be used to query JSON documents. Chapter 6 shows how our approach can be implemented in a real-world system, which Chapter 7 will evaluate against state-of-the-art JSON parsers. Finally, we will give our conclusions in Chapter 8.

Chapter 2

Background and related work

This chapter will give background information about common JSON processing techniques and parallel computing, which is necessary to understand the next chapters fully. Section 2.1 starts by giving a definition of the JSON format and exploring past work into querying JSON. Section 2.2 will explore how structural indexes are used in state-of-the-art JSON processing frameworks and libraries. Then, section 2.3 will examine how GPUs are used for massively parallel applications and finally section 2.4 will give some background knowledge about GraalVM.

2.1 JSON processing

In this section, we will first define the JSON format itself, then look at how JSON can be queried, followed by several parsing and querying methods described in previous work.

2.1.1 The JSON format

It is important to define JSON, the JavaScript Object Notation. JSON is specified by ECMA-404 [8] and is a relatively simple, text-based data-interchange format. It has relatively few constructs but can be used to express semi-structured and unstructured data easily. There are only a few types that can be used in JSON documents:

- A string literal: A UTF-8 string enclosed by double quotes (‘`"`’), with some escape sequences allowed as specified in ECMA-404 [8], e.g., `\n` for a newline character.
- A number literal: All numbers can have decimals and, as such, every number in JSON is a floating-point number. Exponential notation is also allowed.
- `true`, `false`, `null`: Three constant values.
- An object: Objects are enclosed by curly brackets (‘`{`’ and ‘`}`’) and contain several key-value pairs, separated by commas. Key-value pairs consist of a string literal key, a colon and a value of any type.
- An array: Arrays are enclosed by square brackets (‘`[`’ and ‘`]`’) and contain several values, separated by commas. Items can be of any type.

A complete overview of the grammar is given in listing 1, with an example of a JSON object given in listing 2.

Since we will be processing relatively large datasets, it is important to note that multiple JSON-based formats are used for large datasets. The most trivial approach would be to create one JSON document containing an array. For example, if there are 1000 tweets within a Twitter dataset, the top-level JSON value would be an array with 1000 objects. An example of this format for a small dataset is shown in listing 3. This is easy to output; however, there are significant drawbacks to this approach. When using

Listing 1 JSON grammar in ANTLR format

```
json : value ;
obj : '{' pair (',' pair)* '}' | '{' '}' ;
pair : STRING ':' value ;
arr : '[' value (',' value)* ']' | '[' '] ;
value : STRING | NUMBER | obj | arr | 'true' | 'false' | 'null' ;
STRING : '"' (ESC | SAFECODEPOINT)* '"' ;
fragment ESC : '\\' (["\\/bfnrt] | UNICODE) ;
fragment UNICODE : 'u' HEX HEX HEX HEX ;
fragment HEX : [0-9a-fA-F] ;
fragment SAFECODEPOINT : ~ ["\\\u0000-\u001F] ;
```

Listing 2 JSON example from Twitter [9]

```
{
  "created_at": "Mon Oct 17 11:56:58 +0000 2016",
  "id": 787985754408484865,
  "id_str": "787985754408484865",
  "truncated": false,
  "user": {
    "id": 787518866763067392,
    "location": null
  },
  "entities": {
    "user_mentions": [{
      "name": "emilycrockett",
      "id": 29372259
    }]
  }
}
```

such a large dataset, it is essentially required to parse all data at once since the only way to extract a single record is by parsing all data. Parsing just a single record requires knowledge of the start and end location of the record within the document. The only way to locate the record is thus to parse the complete dataset, or at least a significant part of it. Therefore, another approach is usually used to store large JSON datasets into single files.

Line-delimited JSON (LDJSON) is an alternative format for representing a large array of JSON records. Instead of containing one large JSON array, a file will contain many smaller JSON documents, each of which is separated by a newline. An example is shown in listing 4. This approach makes parsing easier since less contextual data is required to find different records within the dataset; the only character which needs to be parsed is the newline character. The only drawback of this approach is that it does not allow for formatted JSON records, such as those usually intended for humans. There is no conflict between newline characters in string literals since those are always escaped in a JSON record. Therefore, this makes this approach ideal for large datasets. While LDJSON is the focus of this work, our techniques could easily work for large arrays of JSON objects as well.

2.1.2 Querying JSON

Querying JSON is essential to using JSON data, just like querying XML using XPath is an essential part of working with XML. Therefore, JSON has an equivalent to XPath, which is used to query JSON

Listing 3 An example of the large document approach for storing large JSON datasets

```
[{"id":1},{"id":2},{"id":3},{"id":4},{"id":5}]
```

Listing 4 An example of the line-delimited approach for storing large JSON datasets

```
{"id":1}
{"id":2}
{"id":3}
{"id":4}
{"id":5}
```

Listing 5 An XML representation of the JSON document of listing 2

```
<?xml version="1.0" encoding="UTF-8"?>
<tweet>
  <created_at>Mon Oct 17 11:56:58 +0000 2016</created_at>
  <id>787985754408484865</id>
  <truncated>false</truncated>
  <user>
    <id>787518866763067392</id>
    <location>null</location>
  </user>
  <entities>
    <user_mention>
      <name>emilycrockett</name>
      <id>29372259</id>
    </user_mention>
    <user_mention>
      <name>foobar</name>
      <id>293722382</id>
    </user_mention>
  </entities>
</tweet>
```

data from a JSON document. First, we will explain what XPath is and how it is used, followed by explaining the JSON equivalent and its syntax.

XPath [10] is a World Wide Web Consortium (W3C) standard which specifies a standard querying language for XML documents. An XPath expression allows traversing an XML tree, as well as specifying conditions on keys and values. For example, the XPath expression `/tweet/user/id` would select the value `787518866763067392` from the XML document in listing 5. Additionally, more expressions may be added to an XPath expression, such as `text()`, which adds an assertion that the final node in the XML document is a text node. If we apply this to the previous expression, we get `/tweet/user/id/text()` which extracts exactly the same value, unless our our document's `id` node would not be a text node; in that case it would not return a value. Expressions can also be applied on values, such as `/tweet/user/id[contains(text(), '392')]/text()`, which would return all nodes which contain `392` in their text at this specific path.

XPath can also be used to query JSON documents. The latest specification for XPath, version 3.1, has added support for JSON querying. However, there are a several critical differences between JSON and XML which make XPath unsuitable for JSON querying.

First, the data model of XML and JSON is different enough such that there are data types that are not common to both types. In XML, all values are represented as string, and it does not have native support for numeric values or booleans. While this can be added by defining a schema, a schema is domain-specific and is thus not widely applicable to all XML documents.

Second, XML and JSON have different representations of arrays and objects. XML represents both arrays and objects as nodes, which may be repeated any number of times. JSON has distinct and specific data types for arrays and objects. XML also implements attributes on nodes, whereas JSON

does not make a distinction between nodes and attributes. Another difference between XML and JSON is that JSON does not have a name for the root object in a document, while XML always names its top-level nodes. Therefore, it would never be possible to apply the same XPath query to a JSON and an XML document, even if they represent equivalent data in the same structure.

Thus, while XPath can be used to query JSON, XPath contains many features which do not apply to JSON documents, such as querying by attributes and extracting comments. For these reasons, as well as the lack of XPath support for JSON until 2017, an alternative querying language for JSON was proposed, JSONPath [11]. JSONPath aims to be suited more towards the specific characteristics of JSON and only covers essential parts of XPath.

JSONPath's syntax is similar in structure to XPath, although the exact syntax differs. A JSONPath expression always starts with a `$`, the notation for the root JSON document, since a JSON document only has an anonymous root node. Then, JSONPath allows two different notations for querying paths: the dot-notation or the bracket-notation. The dot-notation may look like `$.users[5].lang`, while the equivalent bracket-notation is `$['users'][5]['lang']`. The dot and bracket-notation may also be mixed, such as `$.users[5]['lang']`.

Like XPath, JSONPath supports expressions for selecting values. For example, to express a query selecting all users whose language is `nl`, we would use `$.users[?(@.lang == 'nl')]`. Unlike XPath, JSONPath does not have a formal specification, and as such, the expression supported differ between implementations. The original proposal specifies expressions as using the "underlying script engine", which would make expressions non-interoperable between different execution platforms. To more formally specify the expressions that our implementation could support, we will use the expressions supported by the JsonPath Java implementation [12] as a reference.

2.1.3 Sequential parsing of JSON data

While JSON is a simple encoding format, parsing performance is often poor, and as such, some research has been done into speeding up JSON parsing. In V8, the most popular JavaScript execution engine, JSON parsing performance was improved by switching from a recursive parser to an iterative parser [13], resulting in much better parsing performance. However, the performance of the V8 engine is still slow compared to recent research.

NoDB [6] can query JSON without first loading it into a database processing system. FAD.js [14] uses selective access to data and just-in-time compilation to efficiently access and process JSON structures.

For further examples of sequential parsing, we may look at other, similar, data formats, such as XML. XML is older than JSON, and as such, more research into parsing it has been done. One of the earliest techniques uses selective data access to only parse values needed by the application. Noga et al. [15] proposed one of the earliest techniques, making a distinction between push-based and pull-based XML parsing. Utilizing a pull-based model, where the application requests data on an as-is basis, improves XML parsing times by 65%. Fernando et al. [16] partition one XML file into multiple XML files, allowing querying of fields within an XML file with less overhead and smaller input files.

Kostoulas et al. [17] show that even with an integrated schema validation in the parsing pipeline, they achieve speeds higher than other parsers by reducing the separation between different layers and reducing the number of data copies.

These XML parsers show that using a pull-based model or selective data access are viable strategies to reduce the parsing time for not only XML but also for JSON. Currently available JSON parsers in Node.js applications can only fully parse a JSON document, while usually only a subset of the data is required. Therefore, integrating a pull-based model into a Node.js engine could have substantial performance gains compared to the standard `JSON.parse`.

2.1.4 JSON querying

To query subsets of structured, semi-structured, and unstructured documents, various techniques exist, both for XML, as well as for JSON. Usually, these techniques rely on an index to improve query performance.

Gottlob et al. [18] show that existing XPath query evaluation systems in 2005 required time exponential in the size of the queries in the worst case. By improving the XML document traversal, they were able to show that worst-case polynomial-time performance was also possible, with best-case performance being linear in the size of the query.

Arroyuele et al. [19] uses compressed indexes to optimize XPath expression evaluation performance. They use a bit array of structural characters, and separate storage for the text nodes of the document. Using these techniques and by using a more optimized query engine, they outperform most then-state-of-the-art XPath evaluation engines.

Green et al. [20] were able to use a Deterministic Finite Automaton (DFA) to parse streaming XML data efficiently. However, it is limited to linear XPath expressions, which limits its usefulness to real-world workloads.

There are fewer JSONPath-specific evaluation engines than XPath evaluation engines. However, there are alternative query languages for JSON which have been implemented in database systems. Argo [21] presents a mapping layer from JSON documents to traditional relational databases. DiScala and Abadi [22] expand on this by automatically generating a relational schema from any nested key-value pairs, which reduces data repetition and allows the use of traditional relational database tools for analyzing the data. Other techniques for inferring schemas also exist, such as Sinew [23] and the technique used in the Oracle RDBMS [24].

2.2 Structural indexes for JSON analytics

In this section, we will define structural indexes and how and why they are used. We will also explore existing methods that leverage structural indexes to query JSON documents. Our technique uses parts of the structural indexes presented but are built and stored on the GPU, unlike existing approaches, as discussed in chapters 4 and 5.

2.2.1 Definition

Structural indexes in general are a known technique that is used to accelerate data lookups. They are usually a metadata abstraction of some input data and contain relevant information for finding specific tokens within the input data. For example, a structural string index can be used to index the locations of string literals within a JSON file. In most cases, these are stored as bitmap indexes, allowing for a large amount of data to be represented in a smaller amount of storage than usually required. Bitmap indexes use only 1 bit to represent the data for 1 byte in the data input, allowing a compression ratio of 8 to be achieved without any additional effort. In figure 2.1, an example structural string index is shown, where a 1 represents a location in which a bit is set in the bitmap.

```
{"alpha": { "beta": "gamma" }}
0011111100000011110000111110000
```

Figure 2.1: An example of a structural index which indexes the locations of string literals

Structural indexes can also be used to represent more complex structures in the index data, and they can also be extended to encompass data on multiple levels. Multi-leveled structural indexes can be used to store data about nested data structures, such as nested JSON objects and arrays.

One of the advantages of structural indexes is that Single Instruction, Multiple Data (SIMD) instructions can be used on their contents. SIMD is an extension to the standard instruction set of modern CPUs

that allows a single instruction to operate on multiple elements simultaneously, i.e., it allows executing the same manipulations on multiple data elements. SIMD allows a CPU to perform calculations simultaneously, reducing the time it takes to manipulate data.

There is a multitude of operations that can be accelerated using SIMD in a structural index. For example, SIMD allows an AND or OR operation to be calculated quickly. Depending on the supported instruction set, SIMD instructions execute the AND operation on 128, 256, or 512 bits at once, in comparison to the usual 8, 16, 32, or 64 bits of a single instruction on a single integer type. Therefore, SIMD is often used in systems using structural indexes, and, especially when combined with other acceleration techniques, it provides a significant speed-up compared to traditional CPU instructions.

2.2.2 SIMD JSON processing

Mison [4] was the first system to use SIMD to process JSON data. It is one of the most interesting systems that can access nested fields within a JSON structure without parsing unused content. It can process JSON documents at over 2 GB/s but only accepts strictly ASCII documents and does not validate them. It assumes all supplied documents are valid JSON and does not throw exceptions if the input JSON data is not valid. It functions by scanning the document several times, discovering different structural elements, such as `{` and `,` in every scan. Using these scans, it can quickly locate a given key and extract its value.

It is crucial to understand how Mison functions since our work is an adaptation of their structural index modified to be built and queried on a GPU. Mison constructs different structural indexes, which denote how a particular structural character is used. This would be straightforward if the structural characters would only occur where they are used as structural elements. However, these structural characters can also occur within string literals. Therefore, it is necessary to first construct an index that allows us to determine whether a particular character is part of a string. Since characters can also be escaped within strings, this is not trivial. For example, the string literal `\"foo\"bar"` only ends at the last quote character, so it is crucial to determine which quote characters are escaped. It is impossible to do this by only looking at the previous character of a quote character since the escape backslash can also be escaped itself, so it is necessary to fully understand the context in which the quote character is used.

For constructing the string index, Mison uses multiple bitmap indexes and bit-parallel bitmap SIMD CPU instructions to create a string bitmap which contains for every character whether it is contained within a string, as in figure 2.2. It then uses this bitmap to construct a leveled-colon bitmap, which contains whether a specific colon denotes a comma within the level. A *level* is defined as the depth within a JSON record, and as such, the root object is defined as level 1, an object contained within the root object as level 2, etc.. An example of a leveled-colon bitmap is shown in figure 2.3.

```
{"foo": { "object": "bar" }}
00111100000111111100011110000
```

Figure 2.2: An example of a string index created by Mison, non-mirrored

```
  {"foo": { "object": "bar" }}
L0 0000001000000000000000000000
L1 0000000000000000001000000000
```

Figure 2.3: The leveled colon index created by Mison for a simple record, with levels denoted at the left and 1's representing the structural characters

To query the leveled-colon bitmap index, Mison uses a relatively simple algorithm. It will iteratively scan over the line, finding every colon. Once it has found a colon character, it will look back at the key located before it and check if it matches. If it does, it will go down one level and repeat the process,

looking for a colon in the next level. If the key does not match the queried key, it will simply continue to the following colon and repeat the process.

Mison can also be extended with support for array queries by building a second index on commas and using that index for finding a specific array index. This addition is only necessary if the query contains an array index query and is unnecessary if only the JSON record contains an array.

simdjson [7] is another SIMD JSON parser that can parse at the same speed as Mison. However, it fully parses and validates the document, rather than only extracting some fields without validating them. simdjson works similarly to Mison, but with only two passes over the input. In the first pass, all structural elements are found, just like in Mison. Using the result of the first pass, it constructs the structure of the JSON document in the second step.

simdjson's first stage is used to identify all JSON value starting locations, as well as the location of structural characters. It does this in a similar way as Mison, but in comparison to Mison it uses fewer branches and more SIMD instructions, resulting in a higher throughput. The second stage of simdjson is used to construct the JSON tree, the tape, which can then be queried. In contrast to Mison, simdjson does not implement querying as part of their strategy; simdjson only constructs the JSON tree. Therefore, users manually need to query the JSON tree to extract data from it; it does not support a querying language such as JSONPath and does not provide accelerated operations for it.

Both Mison and simdjson use SIMD instructions to operate on large strings quickly. simdjson is much faster than Mison because it uses these instructions more efficiently and with fewer loops. It also changes the discovery model to be almost entirely branchless. This results in a higher throughput in the index building phase. However, a direct comparison is not given due to the different goals of the techniques: Mison queries a JSON structure, while simdjson builds a JSON node tree and leaves query execution to the user.

A recent promising system is Pison by Jiang et al. [1]. It is based on the techniques introduced by Mison and simdjson, but builds the structural index in parallel on the CPU using SIMD instructions on multiple threads. Pison uses parallelization techniques to solve dependencies between different steps in the index construction, most notably partitioning the JSON record into multiple records such that they can be processed in parallel. This allows all other stages to be effectively parallelized, as shown in figure 2.4. Pison uses various optimization techniques which make index construction faster on the CPU, while our work focuses on constructing the index on the GPU, for which different optimization techniques are applicable. We will discuss this difference more in chapter 4. The authors claim a speedup of 4.6X over simdjson when all records can be processed in parallel.

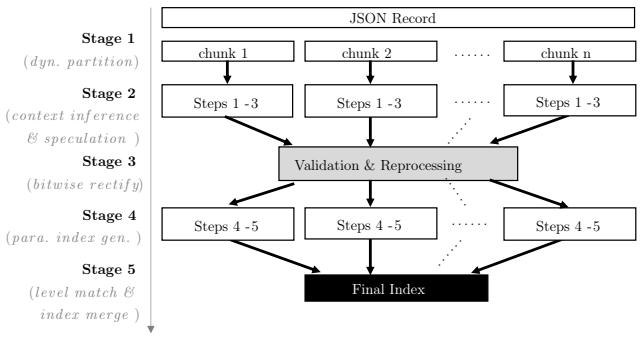

Figure 2.4: Workflow of Pison index construction [1]

2.3 GPU-based parallel processing

2.3.1 GPU Overview

Parallel computing can be used to speed up many everyday tasks by executing multiple tasks in parallel. Graphics processing units (GPUs) can to massively parallelize the execution of tasks due to their large

number of cores.

Graphics processing units (GPUs) were designed to accelerate the creation and rasterization of graphics. To do so, they have massively parallel processing cores with SIMD capabilities. In recent years, these GPUs have as many 10 752 general-purpose cores with hundreds of specialized cores [2]. While these cores were initially designed for graphics processing, they can also be used to process data in a massively parallel way.

To fully understand how GPUs can be used to parallelize operations, we will describe the architecture of recent GPUs and how their parallelization operates. A recent NVIDIA GPU consists of five distinct types of processing cores [2]: Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), Raster Operators (ROPS), and memory controllers. A high-level overview is depicted in figure 2.5, showing 7 GPCs, 42 TPCs, and 84 SMs. While the GPC is the highest-level hardware block, each containing a full graphics pipeline, the primary method by which general-purpose processing on the GPU can be used is by using the SMs. A full NVIDIA Ampere GA102 GPU contains 84 SMs, each of which contains 128 CUDA cores. In addition, each SM also contains additional processing power, memory storage, and cache.

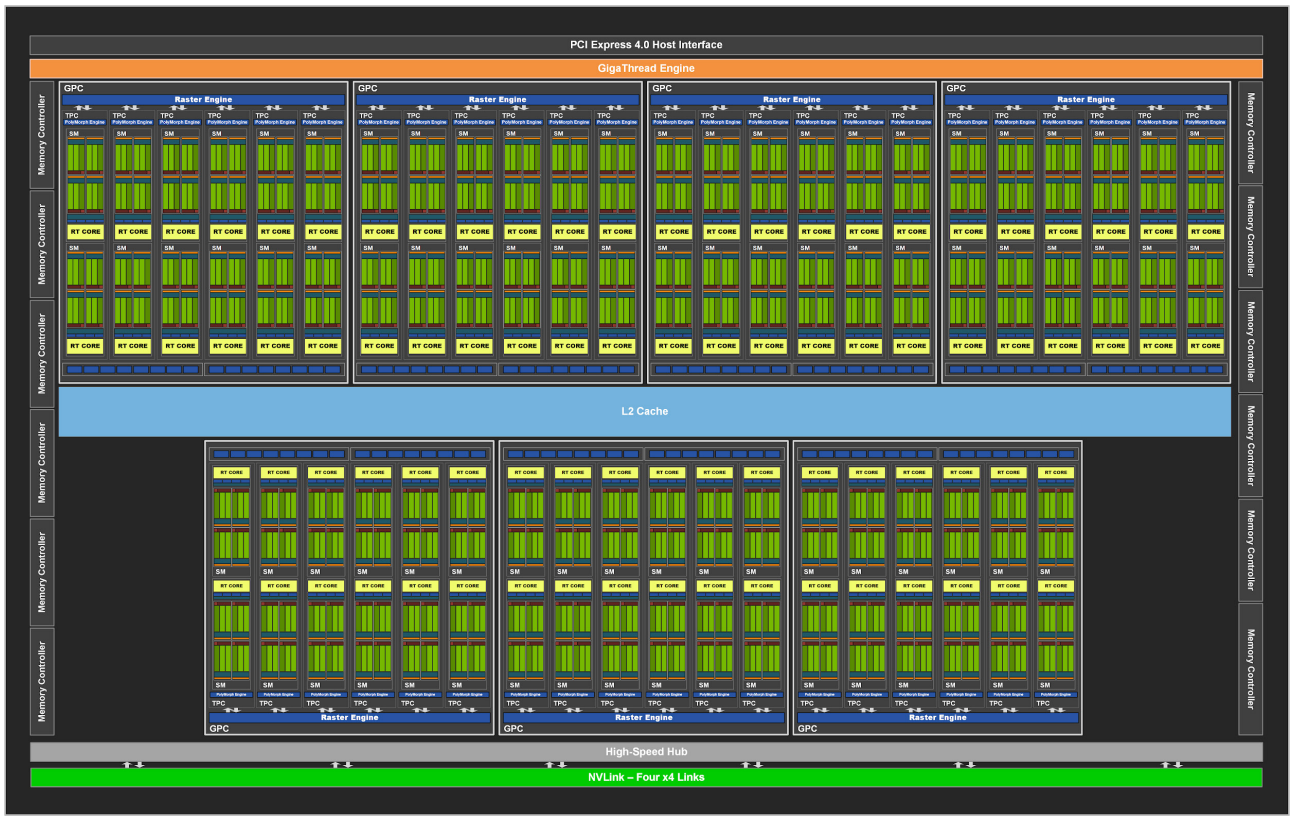

Figure 2.5: NVIDIA Ampere high-level architecture [2]

CUDA abstracts all of these modules into a 1D, 2D, or 3D grid, containing multiple thread blocks, which in turn contains threads [25]. Threads run on the SMs and allow parallelization. Each thread within a thread block runs the same instructions, making it similar to SIMD. However, it uses a Single Instruction, Multiple Thread (SIMT) architecture, since not only is the same instruction executed on multiple data elements (SIMD), the same instruction is also executed on multiple threads.

NVIDIA's SIMT architecture always operates on warps, which contain 32 parallel threads. Each thread in a warp starts execution at the same instruction but can diverge due to different branches. However, a warp can only run optimally when all threads take the same branch since an instruction must always be executed at the same time within a warp. So, if a branch is taken, it will essentially half the performance of all instructions after it until the threads converge again. Therefore, care needs to be taken to reduce the number of diverging threads in a warp.

CUDA programming model is based on compute kernels, which run on threads on the GPU. It can be compiled from C or C++, making it easy to program against since the programming language is the same. However, the standard library usually found on more traditional platforms, such as the POSIX interface or the C++ standard library, is not available.

A kernel within the CUDA programming model runs on many threads on the GPU, each locked to a single core. For optimal performance, the kernel should have as few diversions as possible, and as such, branches can be problematic. However, if the same branch is taken on all threads within a group, which usually consists of at least 32 threads, performance is not impacted. This is consistent with the challenges of SIMD programming experienced on CPUs, since branches are also not always possible in SIMD programming. It is also important to note that a kernel is usually run on thousands of threads simultaneously, resulting in a performance improvement over CPU-based processes.

Another challenge is that memory access patterns can significantly effect performance since data needs to be copied from the main GPU memory to the thread group's memory. Strided memory access is usually better for performance due to the transaction-based memory model of CUDA. One other challenge in the memory model of CUDA is that only very small amounts of memory can be allocated on the GPU itself; large memory allocations can only happen on the CPU, especially if the memory needs to be retained or accessed after the kernel has finished running.

An open alternative to the proprietary CUDA Toolkit is OpenCL. OpenCL is more widely applicable than CUDA and is an open platform, as its graphical counterpart OpenGL. NVIDIA's market share in new GPUs shipped is 83%, with its primary competitor AMD possessing the other 17% of the market [26]. Higher performance can be reached when using CUDA on NVIDIA GPUs, compared to using OpenCL due to better support by NVIDIA and better optimizations in the compiler [27, 28].

2.3.2 GPU-based data processing

Since these GPUs have so many cores with SIMT capabilities, it is possible to process data much quicker than on the CPU. For example, Stehle and Jacobsen propose using ParPaRaw [3] to parse CSV files. Their method is able to achieve 14.2 GB/s of CSV file parsing using a GPU with 3584 cores. It uses a Deterministic Finite Automaton (DFA) for parsing, such as the one in figure 2.6. However, they cannot apply the DFA sequentially since each input data chunk is processed independently by the GPU. Therefore, it is undetermined what the starting state is when the parsing starts. Therefore, each chunk is parsed with all possible starting states and later combined to the correct parsing context, as shown in figure 2.7. The threads can then determine what they have parsed and start discovering rows and columns using a structural index. This technique has not been generalized or adapted to JSON parsing. Therefore, it is interesting to explore whether JSON can also be parsed on the GPU using SIMT methods.

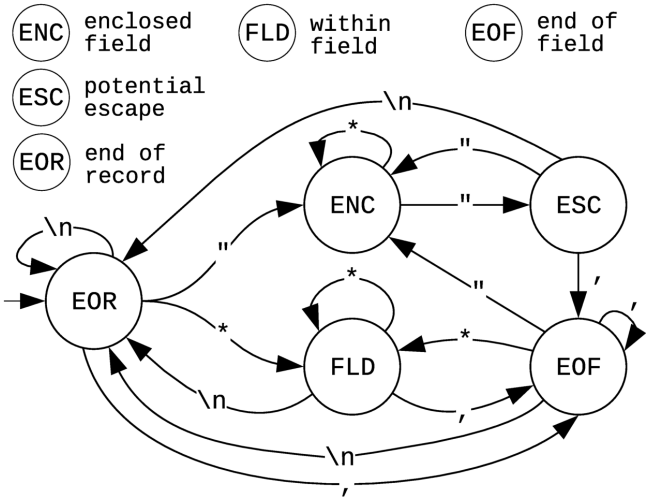

Figure 2.6: Example of a simple DFA for parsing CSV files [3]

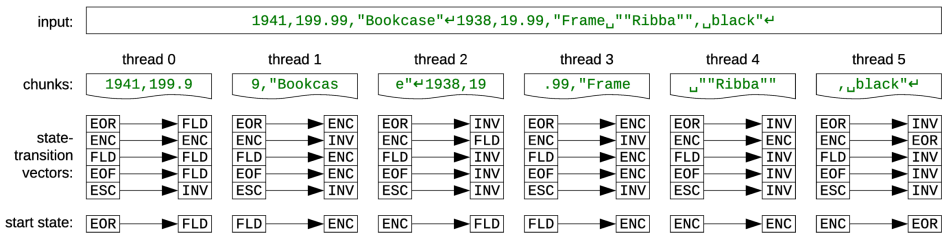

Figure 2.7: A representation of how ParPaRaw is able to determine the parsing context [3]

GPUs have already shown to be quite promising in accelerating general-purpose databases, for example, in the works of He et al. [29], Bakkum and Skadron [30], Simion et al. [31] and Krolik et al. [32]. They demonstrate significant speedup compared to traditional CPU-based databases due to the massive parallelism achievable using a GPU-based algorithm. Bakkum and Skadron demonstrate a speedup of at least 20x in SQLite by only accelerating SELECT SQL queries, while Simion et al. demonstrate a speedup of at least 6x. However, on larger queries, this speedup is 62-240x due to the relatively lower cost of memory transfer from the CPU to the GPU. These works show that it is possible to achieve significant speedups by adapting traditionally CPU-based applications to use GPU-based algorithms.

2.3.3 Other parallel parsing techniques

In addition to GPU-specific parallel parsing techniques, several CPU-based parallel parsers also exist. Barenghi et al. propose PAPAGENO, a parser generator that can generate a parallel parser for JSON [33]. It achieves speedups of 5× compared to a regular parser generator. However, it is not compared with a hand-tuned sequential parser. Pavlopoulou et al. propose a parallel JSON query processor that can quickly process a query and extract data from a JSON document [34]. However, this is focused more on the parallelization of the query processing rather than parsing the JSON data itself.

2.4 GraalVM

This project aims to expose fast GPU-based JSON querying to high-level scripting languages, such as JavaScript and Python. This allows our work to be used without using a compiled language such as C++ or Java. To accomplish this, we will be using GraalVM.

GraalVM is a virtual machine with support for multiple programming languages, with a unified execution model. It supports languages such as JavaScript, Python, and Ruby while also allowing interoperability between the different languages, using polyglot programming. It is built on the Java JDK and, as such, it also supports Java-based programs.

GraalVM already supports CUDA programming using the grCUDA project [35], by NVIDIA, which proves that accessing GPUs on GraalVM is possible. It is also a case in favor of CUDA, rather than OpenCL, since there is currently no implementation to access OpenCL on GraalVM.

By using GraalVM, it would not only be possible to access the implementation from multiple scripting and compiled languages, but it is also possible to do runtime optimizations, as done by FAD.js [14] in the case of accessing JSON. This would allow the implementation to change depending on current, previous, or future expected input, and for more optimizations to be executed.

Chapter 3

GPJSON System Overview

3.1 Introduction

This chapter will give a complete overview of the project and how we will approach the problem. Our system is called GPJSON, is open-source, and can be found on GitHub at `https://github.com/koesie10/gpjson`.

As stated before, it is interesting to explore whether JSON can be parsed on the GPU and how much faster it can be made than a CPU-based sequential or parallel parser. However, usually, only a small subset of the data is required for processing, so the focus will be on extracting a subset of the data from the JSON document.

In section 3.2 we will give a specification of what input we will allow in our implementation. Then, in section 3.3 we will discuss the querying language to use. Finally, in section 3.4 we will give an overview of how we will use GraalVM for our implementation.

3.2 Input

As stated in section 2.1.1, a popular data format for large JSON datasets is newline-delimited JSON, or LDJSON. LDJSON requires less context for determining different JSON values than an array of JSON values. This is ideal for a GPU environment since many threads run in parallel, and determining context requires a synchronization step. Therefore, we will only allow LDJSON in our implementation.

Furthermore, we expect all lines in the LDJSON file to be valid JSON, as defined by the ECMA-404 specification [8]. This allows us to skip validation of the JSON documents, which would add overhead since this requires synchronization between threads to determine the parsing context. In future work, this limitation could easily be lifted, as will be discussed in chapter 8.

While we will support UTF-8 by not making any assumptions on multi-byte values, we will not compare escaped and non-escaped keys. This means that a JSONPath query like `$['/']` will not match with a key `\u002f` (which is the UTF-8 escape sequence for the slash character). This is a reasonable limitation since keys in JSON documents are usually only composed of ASCII characters and rarely escaped. This is also a limitation of simdjson [7]. It is still possible to find these escaped values by using the literal value in the JSONPath, like `$['\u002f']`, so it does not make it impossible to query escaped keys.

Our implementation will happen completely in-memory and, as such, will not support files larger than the GPU's memory. The typical memory size of a high-end consumer GPU is 10-12GB, with top-of-the-line consumer GPUs having 24GB of memory. Top-of-the-line datacenter GPUs have up to 80GB of memory [36]. The GPU's memory will also need to store some structural indexes, so the maximum size of the file is even smaller. However, this is only a limitation of our implementation, and a streaming implementation of our technique is feasible, which would lift this restriction.

3.3 Querying languages

As a query language, we will use JSONPath, due to its popularity for querying JSON values and the availability of multiple implementations for both Java and Node.js. We will, however, only support a subset of JSONPath, due to the complexity of implementing all operations. Specifically, we will only support JSONPath queries for which the number of results is known in advance. Therefore, recursive descent operations such as `$..author` are not implemented.

JSONPath also supports many expressions that allow a user to make many assertions on queried values. Due to the number of expressions available, we will only support one expression which shows that expressions can be implemented in our GPU-based querying engine.

For all unsupported JSONPath queries, we will use an already existing JSONPath querying engine. Since our implementation is exposed via Java, we will use JsonPath Java [12] for our fallback implementation. This allows the user to execute any query using GPJSON without worrying about the performance of the query. Even though JsonPath Java is slower than our final implementation, it will still be faster than other libraries available in Node.js, as will be shown in chapter 7.

It is also important to note that the limitations on JSONPath queries mentioned here are only limitations in our implementation and not for the techniques presented in general. A streaming implementation of our technique is feasible, which would allow recursive descent queries and other queries with an indefinite number of results. The same can be said for other JSONPath expressions; these need to be implemented in our implementation and are not a limitation of our technique.

While we have chosen to use JSONPath for our implementation, this is merely an implementation detail. All JSONPath queries are converted to a domain-specific bytecode format which is independent of the JSONPath query structure. Therefore, a SQL query such as listing 6 could be converted to the same bytecode instructions and executed in precisely the same way, without modifying any of the index construction or query execution details.

Listing 6 An equivalent SQL query of `$.user.lang`

```
SELECT "lang" FROM "user"
```

3.4 Runtime environment

For our implementation, we will use the GraalVM runtime environment. This will allow the implementation to be called from multiple high-level scripting languages, including JavaScript, Python, and Ruby.

GraalVM allows exposing custom proxy objects (using the Truffle API [37]), which are called on-demand when a call to get data is made. For example, it is possible to create a lazy array which only retrieves values from some external data source when a request for an item is made. Our implementation uses this technique only to retrieve the JSON values from the file when they are requested. All indexes to the JSON values are still retrieved from the GPU first, but retrieving file thus happens lazily. This means that our implementation is faster when only a subset of the queried values is retrieved.

Other functionalities of GraalVM, such as runtime specialization, are also interesting to explore. However, our implementation will not use such features, which means that we will show the performance of a straightforward implementation of a GPU-based JSON query evaluation engine rather than a fully optimized one. In other words, GPJSON is exposed to GraalVM as a built-in library, but does not directly interact with the VM runtime components.

Chapter 4

Structural index construction

4.1 Introduction

In this chapter, we will introduce our novel way of constructing a structural index on a GPU. Structural indexes are an auxiliary data structure containing metadata about the elements of a JSON object and are used to query JSON data efficiently. The actual structural index is built entirely in parallel using multiple GPU kernels and is stored in the GPU memory space in our approach. In this chapter, we first present the main characteristics of the structural index of our system. Then, we will show how the index can be assembled using three other auxiliary indexes, namely a newline index, a string index, and a structural leveled bitmap index.

In section 4.2 we will present the structural index that is used in our technique and discuss how it differs from existing structural indexes. In section 4.3 we will present the construction of a newline index, followed by section 4.4 which show how we build a string index. Lastly, in section 4.5 we show how the structural leveled bitmaps index is built.

4.2 Structural indexes

Structural indexes are the core auxiliary data structures used to query JSON data efficiently. Unlike existing approaches, the structural indexes used by GPJSON are computed entirely on the GPU in parallel. There are multiple variations of this structure. simdjson [7] constructs a tape, while Mison [4] and Pison [1] use a structured leveled bitmaps index. We will use a variation of the structured leveled bitmaps index optimized for GPU construction and storage.

```
   {"foo": ["alpha", "beta"], "gamma": {"lambda": "kappa"}}
L0 1000001000000000000000000001000000001000000000000000000001
L1 0000000010000000100000001000000000001000000001000000001 0
```

Figure 4.1: An example of a structural index, indicating the places where a bit is set for each level

A structured leveled bitmaps index contains information about structural characters in a JSON object: ‘{’, ‘[’, ‘}’, ‘]’, ‘:’, and ‘,’. These characters identify the start and end of nested structures within a JSON object and can be used to query it. There are multiple levels in such a structural index, each containing a particular level within a JSON object, where a level is defined as the number of arrays or objects that a character is nested within. To efficiently store the structured leveled bitmaps index, it is stored as a bitmap, containing for each character whether it is structural or not. An example of a representation of a structural leveled bitmaps index is shown in figure 4.1. This bitmap can be constructed by comparing bytes in the files against the structural characters and keeping track of the current level by checking against the opening and closing literals. However, one also needs to consider string literals, which may contain these characters, while they are not structural. For example, when the colon character is found inside a string literal, it should not be considered as a key-value

separator. For this reason, it is necessary first to build a string index that can be used to filter out any structural characters present within string literals. To build this index, several other intermediate indexes are necessary; a full overview is shown in figure 4.2. Each step will also be explained separately in sections 4.4 and 4.5.

Our leveled bitmaps index differs from those used in Mison [4] and Pison [1] in several ways. First, unlike Mison, we do not build different bitmaps for different characters initially. This reduces the amount of memory required by approximately 20% in the best case. Memory is usually more constrained on a GPU device than on a host device due to less expandability and a fixed amount of memory for a specific device model, so this brings tangible benefits. Second, our final structural index contains all structural characters, including characters that start or end a level ('{', '[', '}', and ']'). Mison only saves the positions of colons and, in the case array indexes are part of a query, commas. By also saving the position of other structural characters, we can only use one index compared to Mison's two indexes, which again reduces the amount of memory, in this case by approximately 10% in the best case compared to storing two different indexes. This also allows us to determine the end of a level more quickly since it is unnecessary to iterate over two indexes to find it. Pison builds on the structural leveled colons bitmap index proposed by Mison, and uses SIMD instructions to speed up their construction. Such SIMD techniques are less applicable to GPU-based SIMT, since CUDA does not have SIMD operations that operate on vectors more than 64 bits, while Pison uses 256-bit SIMD instructions and uses operations that are unavailable for CUDA, such as `pclmulqdq` (carry-less multiplication).

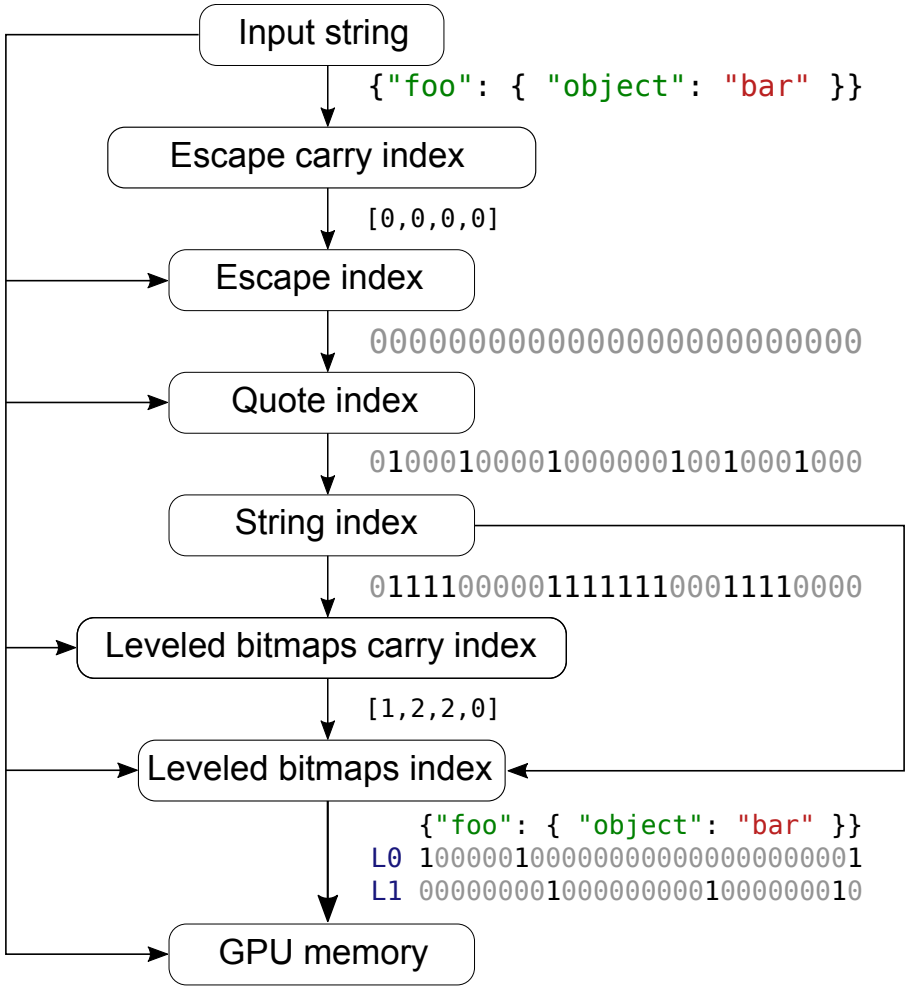

Figure 4.2: Data flow between different steps for structural index construction

4.3 Newline index

Separate from the leveled bitmaps index, it is also necessary to construct a newline index. While the leveled bitmaps index contains information on record level, our technique deals with newline-delimited JSON files, and as such, there are multiple records in each document. The newline index allows traversal of all JSON records within a file by storing the position of every newline ('`\n`') within the file. Between these positions, the leveled bitmaps index can be used for further traversal within the JSON record.

A parallel newline index is non-trivial on a GPU, since the number of newlines within a file is variable and highly dependent on the input data. Therefore, it is unknown how many newlines there are in the file, and as such, it is unknown how much memory is required to store the complete newline index.

One of the challenges of the GPU is that it is not possible to allocate large amounts of memory within a kernel dynamically. Therefore, the host device needs to allocate the memory beforehand to make sure there is a contiguous block of memory that can be used in later steps of the algorithm. This has been accomplished through the use of two separate but dependent kernels.

The first kernel only scans the number of newlines in each thread and stores that information in a block of memory that is dependent on the number of threads running the kernel rather than on the size of the input data. This data will then be used to compute the total amount of memory required to store the newline index. This amount can be computed by simply summing all of the values and multiplying it by the amount of memory required for every position. Since only the numerical position is stored, this would usually be an 8-byte number. Thus, the amount of memory required to store the newline index can be computed as in equation (4.3.1). Once the required amount of memory is known, this amount should be allocated on the GPU, such that it can be used to store the newline index.

$$\text{newline index size} = 8 \cdot \sum_{i=1}^{\text{thread count}} \text{counts}[i] \tag{4.3.1}$$

The second kernel will store the actual positions of all newlines within the file. One of the most important aspects is ensuring that the newlines are in non-overlapping places in the index. One of the other aspects is making sure they are in ascending order. While the order is mostly irrelevant to querying the values, it is important to ensure results can be correlated back to lines. If it not necessary that the results of lines are in the same order as their position in the file, it would be possible to place newlines in an arbitrary order in the newline index, which could potentially improve the performance since a more efficient access pattern could be used. However, we will assume that correlating results back to their line is essential and, as such, will describe all following operations under that assumption.

To ensure that every thread can place their discovered newlines in the correct position in the newline index, the previously discovered count of newlines is used. This is used to create an array of offsets, which will determine where each thread can start writing its discovered newline positions. For example, if there are 4 threads and the result is count $= \{1, 6, 3, 1\}$, then the offsets would be offsets $= \{0, 1, 7, 10\}$ and the size of the newline index would be 11 elements, resulting in a memory size of 88 bytes. When all of this information is passed to the kernel, it is trivial to discover the newlines and place their positions in the result, as shown in algorithm 1. Once the newline index has been computed, it will be kept in memory to be used in a later stage to find the starting points of JSON queries.

Algorithm 1 Newline index kernel

Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: An array of characters f representing the file
Input: The starting index in the result o_s of the kernel
Input: The result newline index r

```
1: o ← o_s
2: for p ← p_s to p_e do
3:     if f[p] == '\n' then
4:         r[o] ← p
5:         o ← o + 1
```

4.4 String index

The second index required is the string index, which will contain for every character within the file whether it is part of a string literal, or outside of it. It will do so using a bitmap index, and as such, it is only an eighth of the size of the input data. However, to construct the string index, other indexes need to be built. Each one uses a temporary amount of memory. The amount of temporary memory used is equal to the amount of memory used for the final string index. The carry indexes only use temporary storage of which the size is dependent on the number of threads running the kernel. The indexes that will be built are in-order: an escape carry index (kernel 1), an escape index (kernel 2), a quote index (kernel 3), and finally the string index (kernel 4), as shown in figure 4.3.

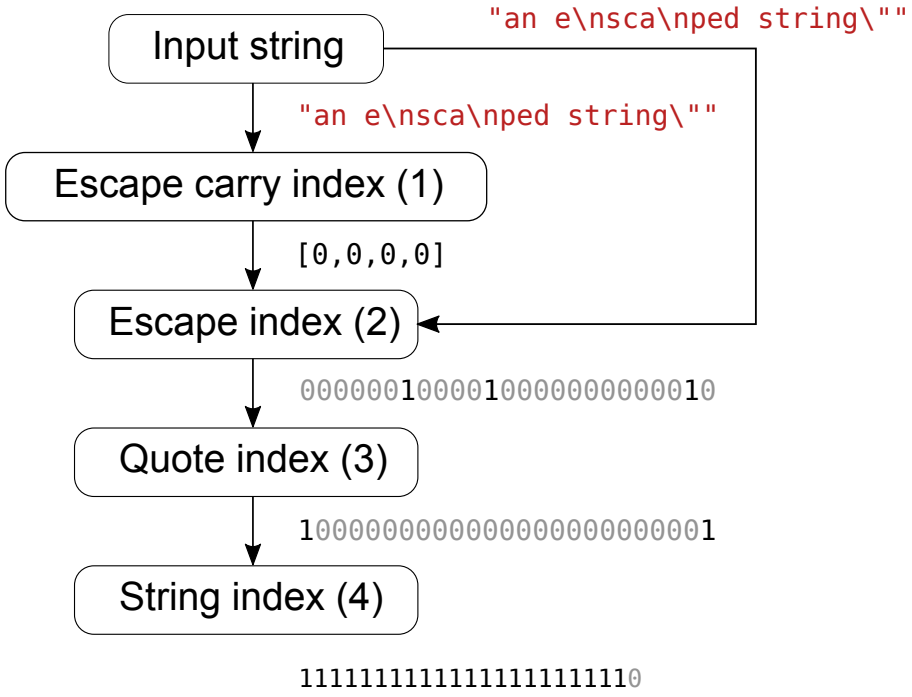

Figure 4.3: Data flow of the string index construction kernels

First, the escape carry index (kernel 1) needs to be built. This index is similar to the first step of the newline index, although in this case, it is used to resolve dependencies between kernels. This index essentially computes whether there is a “carry" of the escape, i.e., whether the first character of the next thread needs to be escaped. This only works when we assume that there are no runs of escapes that run for the kernel duration since that would introduce dependencies between these kernels. However, the input data of one kernel is always at least 64 bytes, and as such, it would already be improbable that all 64 bytes are escape characters. If this does happen, this can easily be detected by counting the number of escapes compared to the total number of bytes processed by the kernel. To compute the

carry index, it is enough to start with an initial carry of FALSE, then looping over all the characters processed by the kernel sequentially. When the character is a backslash character ('\'), the carry needs to be XOR'ed with TRUE. If it is not a backslash character, the carry is reset to FALSE. This is demonstrated in algorithm 2.

Algorithm 2 Escape carry index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: An array of characters f representing the file
Input: The result escape carry index r

```
1: carry ← FALSE
2: for p ← p_s to p_e do
3:     if f[p] == '\\' then
4:         carry ← carry XOR TRUE
5:     else
6:         carry ← FALSE
7: r[t] ← carry
```

Once the escape carry index has been built, the escape index can be built (kernel 2). The escape index is similar to the final string index, in that it encodes using a bitmap for each character whether it is escaped or not. This is shown in figure 4.4. To do so, the iteration is essentially the same, except that instead of starting with FALSE as the carry, we start with the carry of the previous kernel. Moreover, when the carry is equal to TRUE, we store the result in a bitmap index for the current position, which is a simple bitshift combined with an OR operation. This is shown in algorithm 3.

```
"an e\nsca\nped string\""
0000001000010000000000001 0
```

Figure 4.4: A representation of the escape index

Algorithm 3 Escape index kernel

Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: An array of characters f representing the file
Input: The carry c of kernel 1 for this thread
Input: The result escape index r

```
1: carry ← c
2: for p ← p_s to p_e do
3:     if carry == TRUE then
4:         SETBIT(r, ⌊p/64⌋, p mod 64)        ▷ Set the bit p mod 64 in r[⌊p/64⌋] to 1
5:     if f[p] == '\\' then
6:         carry ← carry XOR TRUE
7:     else
8:         carry ← FALSE
```

The following index to create is a quote index (kernel 3), which is again a bitmap index, this time storing the position of quotes within the file, as shown in figure 4.5. It uses the escape index to determine which quotes are escaped. It again uses a very similar sequential loop, except that this time it only sets a bit in the bitmap index when the character is a quote ('"'), and it is not escaped. It also stores a quote carry index, which determines whether the last character in the kernel is escaped, based on the

assumption that it starts non-escaped, i.e., it is TRUE if the quote count in the slice processed by the kernel is odd. Both of these are stored; the quote index has the same size as the resulting string index, while the quote carry index is sized according to the number of threads running the kernel.

```
"an e\nsca\nped string\""
1000000000000000000000001
```

Figure 4.5: A representation of the quote index

The final string index (kernel 4) is constructed from the quote index and the quote carry index, both computed in kernel 3. The quote carry index determines whether the final result needs to be inverted, using the value of the $n-1$th kernel. However, before passing the quote carry index to the GPU, it is processed on the CPU to XOR all values against each other, such that the carry is correct in the context of the complete input JSON. The quote index is used to compute a prefix-XOR sum, of which the result tells which bytes are in a string literal and which bytes are outside of it. For example, given a value $x = \texttt{01001000}$, it computes the result `01111000`. The value of a specific bit within the result can be computed as in equation (4.4.1) [7]. In other words, the prefix-XOR value of a bit i in the result is the XOR of all bits up to and including bit i in input x. simdjson uses the specialized CPU instruction `pclmulqdq` to accelerate this operation. While there are ways to accelerate the normal computation of this value using GPUs [38], CUDA does not have an instruction to calculate the value instantly and, as such, it is manually implemented.

$$\textsc{PrefixXORSum}(x, i) = \bigoplus_{k=0}^{i} x[k] \tag{4.4.1}$$

It is then XOR'ed with a bitstring of all 1's when the previous string ended in an escape, or not changed if the previous string did not end in a string. This bitstring is then computed for the following string, such that it can be used for the next iteration. Each iteration loops over one 8-byte value, and as such, it processes 64 bytes at once. This kernel does not access the original input data and only requires memory access to the quote index. An example of a result may be found in figure 4.6. The complete procedure is shown in algorithm 4.

```
{"foo": { "object": "bar" }}
0111100000111111100011110000
```

Figure 4.6: An example of a final string index

Algorithm 4 String index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: The quote index q of kernel 3
Input: The quote index carry c of kernel 3 for this thread
Input: The result quote index r

```
prev ← 0
if c == 1 then
    prev ← SetAllBits(prev)
for p ← p_s to p_e do
    r[p] ← PrefixXORSum(q, p) XOR prev
    prev ← r[p] >> 63
```

From all of the created indexes, only the string index is kept in memory to create the leveled bitmaps

index. All other indexes can be discarded since those are only used for the construction of the string index.

4.5 Leveled bitmaps index

Once the string index has been computed, the leveled bitmaps index can be computed. The leveled bitmaps index is a bitmap index once again, but this time it stores at which positions structural characters are located. Structural characters are defined as '{', '[', '}', ']', ':', and ','. However, it does not store these results at one level only. For every level, where a level is defined as one level deeper within the JSON tree structure, it stores the bitmap index. For the construction, similarly to the previous indexes, there are two steps; the first step creates a carry index to resolve dependencies between kernels and the second step creates the actual index. An example index is depicted in figure 4.7, where the levels are denoted to the left, and a 1 represents a structural character on that level.

```
   {"foo": { "object": "bar" }}
L0 1000001000000000000000000001
L1 0000000010000000010000000010
```

Figure 4.7: An example of a leveled bitmaps index for a single line, with levels denoted at the left and 1's representing the structural characters

The carry index for the leveled bitmaps index contains the level at which the parallel traversal of the file starts, with one value for each traversal performed by a dedicated kernel. So, it should contain -1 for the first kernel since it is not yet in a JSON structure. To calculate at which level each kernel starts, it is sufficient to count the number of structural open/close characters. The final value of each carry kernel is essentially the number of opening structural characters minus the closing structural characters. Using this, it is possible to calculate the starting level of each kernel for the following step. One important consideration here is that the structural characters could also be located inside strings, so the string index needs to be used to filter out any structural characters inside strings. The full algorithm is shown in algorithm 5.

Algorithm 5 Leveled bitmaps carry index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: The string index s of kernel 4
Input: The result carry index r

```
1: level ← 0
2: for p ← p_s to p_e do
3:     if not InString(s, p) then
4:         if IsStructureOpen(p) then
5:             level ← level + 1
6:         else if IsStructureClose(p) then
7:             level ← level − 1
8: r[t] ← level
```

Once the carry index has been constructed, a process very similar to the offset index of the newline index is used (section 4.3). The previously calculated values are summed, and each intermediate sum is stored in the index. The sum starts at -1, as alluded to before. After this index has been constructed, the actual leveled bitmaps index can be created.

The leveled bitmaps index goes over each character and uses a similar process to the carry index creation. Characters inside strings are skipped, and structural characters are used to create the actual leveled

bitmaps index. When an opening structural character is discovered, the current level is incremented, while when a closing structural character is discovered, the current level is decremented. When another structural character (':' and ',') is discovered, it is stored at the current level, which also happens for the opening and closing structural characters. Storage is again a bitmap for every character, except that there are multiple levels, so the storage is not just 1/8th the size of the input data but rather depends on the number of levels required for the querying. Any structural characters which are deeper inside the JSON tree structure than what is required for the query are not stored to limit memory usage. The entire algorithm is shown in algorithm 6.

Algorithm 6 Leveled bitmaps index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: The string index s of kernel 4
Input: The leveled bitmaps carry index c
Input: The result index r
Input: The size of a level n

```
1:  level ← c[t]
2:  for p ← p_s to p_e do
3:      if not InString(s, p) then
4:          if IsStructureOpen(p) then
5:              level ← level + 1
6:              SetBit(r, n · level + p/64, p mod 64) ▷ Set the bit p mod 64 in r[n · level + p/64] to 1
7:          else if IsStructureClose(p) then
8:              level ← level − 1
9:              SetBit(r, n · level + p/64, p mod 64)
10:         else if IsColonOrComma(p) then
11:             SetBit(r, n · level + p/64, p mod 64)
```

Chapter 5

Querying

5.1 Introduction

Once the structural index for a given JSON object has been computed, it can be used to query the JSON data. Given that the structural index is stored in the GPU memory space, the evaluation of one or more JSONPath queries should ideally happen on the GPU as well. To this end, we have implemented a novel technique in GPJSON capable of executing any arbitrary JSONPath query (or even non-JSONPath ones) on the GPU directly. Our query execution technique is based on the intuition that GPU parallelism can be exploited to run multiple queries simultaneously (one per line) using a dedicated GPU kernel. In more detail, we have implemented a domain-specific bytecode instruction set [39] and a corresponding bytecode interpreter [40] that is capable of evaluating any JSONPath query on the GPU. GVM [40] shows that GPU-based bytecode interpreters can be faster than conventional CPU-based interpreters for Java bytecode, so by using a smaller instruction set and a domain-specific bytecode format, we are able to outperform compiled CPU-based query engines.

The core idea behind our approach is that any arbitrary JSONPath query can be converted to our domain-specific bytecode, which can then be executed on the GPU. In this way, scanning and traversal of the structural indexes happens directly on the GPU and can happen for any arbitrary query that can be compiled to our intermediate domain-specific bytecode.

Not only does this technique allow us to execute arbitrary JSONPath queries on structural indexes, but it also allows us to define "intrinsic" built-in operations in the form of bytecode instructions for specific query evaluation, such as checking whether a string equals a specific value. This is extensible; adding more built-in operations is trivial. Using this model, we can run multiple different queries, and the kernel code does not change. This model would also allow us to target other high-level front-end languages, such as a subset of SQL. We are the first proposing to run a domain-specific data processing bytecode-interpreter on a GPU to the best of our knowledge.

The rest of this chapter will first present our method for querying on the GPU in section 5.2. Then, we will give an end-to-end example of the index construction and subsequent query evaluation in section 5.3.

5.2 Implementation

Querying the final value happens in a single GPU kernel. First, it is important to understand how the query is represented. We represent a JSON query by a domain-specific bytecode instruction list, which is able to represent a structural index traversal. For example, a JSON query like `$.user.lang` is represented by the instruction list shown in listing 7.

This is encoded as bytecode `4, 4, 117, 115, 101, 114, 3, 4, 4, 108, 97, 110, 103, 3, 1, 0`. The mapping from an instruction to an encoding for this example is shwon in listing 8.

Listing 7 Example querying bytecode

```
MOVE_TO_KEY "user"
MOVE_DOWN
MOVE_TO_KEY "lang"
MOVE_DOWN
STORE_RESULT
END
```

Listing 8 Example querying bytecode with their raw byte representation

```
MOVE_TO_KEY "user" ; [4, 4, 117, 115, 101, 114]
MOVE_DOWN ; [3]
MOVE_TO_KEY "lang" ; [4, 4, 108, 97, 110, 103]
MOVE_DOWN ; [3]
STORE_RESULT ; [1]
END ; [0]
```

This abstract bytecode representation also allows different querying languages to be used as a front-end to the query. For example, one could imagine converting an SQL query to the same bytecode and executing it with an unchanged bytecode interpreter. An equivalent SQL query for the `$.user.lang` is shown in listing 9. This would be parsed and mapped to exactly the same bytecode, allowing flexibility for the choice of querying language.

Listing 9 An equivalent SQL query of `$.user.lang`

```
SELECT "lang" FROM "user"
```

The same can be done for any expression. For example, the equivalent SQL query of `$.user.lang[?(@ == 'nl')]` is shown in listing 10.

Listing 10 An equivalent SQL query of `$.user.lang[?(@ == 'nl')]`

```
SELECT "lang" FROM "user" WHERE "lang" = 'nl'
```

Our querying kernel is a domain-specific bytecode interpreter, which can fully execute the given instruction list to traverse the structural index. The bytecode is executed for every individual JSON record, and results are stored per line.

Our domain-specific bytecode consists of seven distinct opcodes, mapping to specific index traversal operations. Basic operations, such as move up, move down, move to key, and move to index may also be found in other state-of-the-art index traversal APIs. For example, Pison's C++ API can be used with just these four operations. Our bytecode instructions contain additional operations to support storing values in GPU memory, as well as expression matching operations.

- Index navigation operations:
  - `MOVE_UP`: This moves up a level in the leveled bitmaps index
  - `MOVE_DOWN`: This moves down a level in the leveled bitmaps index
  - `MOVE_TO_KEY`: Moves to a specific key in a JSON object
  - `MOVE_TO_INDEX`: Moves to a specific array in a JSON array
- Query execution and control flow operations:
  - `END`: This ends the execution of the current line
  - `STORE_RESULT`: This stores the current level result in the result
- Expression execution operations:

- `EXPRESSION_STRING_EQUALS`: If the string value does not match the current value, exits the current line and skips further execution

Each of them is pretty straightforward on its own, although all of them will be explained below in order of execution. However, first, it is crucial to understand how the current path in the search is represented and how each line is found and executed.

In contrast to previous kernels used to build the structural indexes, the kernels for this process all get assigned lines instead of bytes. They are divided evenly, so if there are 80 lines and ten kernels, each kernel will get eight lines. The lines can be discovered from the newline index created previously. While the kernels run in parallel and many lines are processed in parallel, the lines within a kernel are processed sequentially. For each line, its start and end locations are retrieved from the newline index. The end location is simply the start location of the following line, or if it is the last line, it will be the end of the file.

Once the position of the line has been retrieved, processing of the line will start. For each line, the query is executed, which requires keeping some state. First of all, the ending positions of all the levels are stored. These are initialized to be an undetermined value since, at the start of the processing, it is unknown where each level ends, as the position of the actual level differs based on the query. The ending positions can still also change when they have been set when there are multiple results, in which case the query will move up a level, and therefore the ending position of the lower level is once again invalid. To support multiple results, the current number of results for the line is also stored to be able to index the output value.

The actual processing of the line begins by executing the bytecode. The kernel will execute as many bytecode instructions as possible until one of `MOVE_TO_KEY` or `MOVE_TO_INDEX` is found. If one of those is found, the kernel will continue by iterating through all characters of the current line until the required value is found, which will be detailed below. All other operations are determined based on the opcode that is being executed.

5.2.1 `MOVE_TO_KEY`

`MOVE_TO_KEY` is a basic opcode that will take in a property name and move to that property within the current level. It does by using the leveled bitmaps index and iterating over all characters of the line. Once it finds a structural character, as determined by the leveled bitmaps index, it will check that the structural character is a colon, denoting a key-value pair. It will then do a simple string comparison of the key and the required property. If the two strings match, it will execute the next instruction of the bytecode. If they do not match, it will continue looking for the following key-value pair, or if it is the last one, it will stop the execution of this line.

```
   {"foo":"bar", "bar" : "foo"}
L0 1000001000001000000010000001
L1 00000000000000000000000000000
```

Figure 5.1: The final position of a `MOVE_TO_KEY` instruction for property "bar"

5.2.2 `MOVE_TO_INDEX`

`MOVE_TO_INDEX` is used for finding a specific index within an array. While the basic checks are simpler than `MOVE_TO_KEY`, since there is no need for string comparison, it is necessary to keep track of the current index in the array, also among different executions of this opcode to allow finding multiple indexes within the same array.

Once the `MOVE_TO_INDEX` opcode is encountered, it will first find the beginning of the array, which will be at the next opening square bracket (‘[’). It will then set the current index within the array to zero. Then, just like move to key, it will iterate over all the characters in the line and find structural

characters. If the structural character is a comma (‘,’), it will increment the current index in the array and check if the current index is equal to the expected index. If so, it will continue to the next instruction. If it has reached the end of the level but has not found the correct index, it will end the execution of the current line.

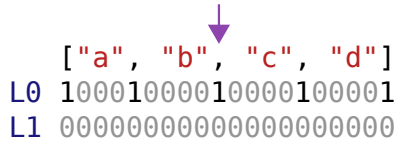

Figure 5.2: The final position of a `MOVE_TO_INDEX` instruction for index 2

5.2.3 `MOVE_DOWN`

`MOVE_DOWN` is used for moving between different levels of the leveled bitmaps index. When it is executed, the current level is set to be one level deeper, and the end of this level is set by iteration through the bitmap index of the previous level. Once it encounters a structural character in the previous level, the level ends there, and this is stored to limit further instructions to the correct level.

5.2.4 `MOVE_UP`

Like `MOVE_DOWN`, `MOVE_UP` is used for moving between different levels of the leveled bitmaps index. Move up will move one level up in the tree and reset the level end of the deeper level, after which it will simply continue execution.

5.2.5 `STORE_RESULT`

This operation is executed to store the current level to the resulting value of the kernels. It will find the start of the value based on the current position in the line by skipping all whitespace since the position in the file is at the structural character and not at the value position. It will then simply store the current position and the ending of the current level in global memory, which can be used later to extract the value from the file. The reason for storing only the positions in memory is based on the limitations of memory allocation, as mentioned before. The amount of memory that would need to be allocated for the results is unknown at the start of the execution. In contrast to other kernels, running this kernel twice will limit the performance by a significant amount. Therefore, only the positions are stored, and the values can be retrieved lazily from the file.

5.2.6 `END`

The final instruction is the end instruction, which will stop the execution of the current line and move on to the following line, which matches the behavior for the previous opcodes when a value was not found. This is used at the end of a sequence of instructions and can be replaced by a check for the number of instructions left in the query.

5.2.7 Expression filters

The expression filter opcodes are used for filtering the encountered value. For simplicity purposes, only one such opcode is implemented, which is the string equals expression (`EXPRESSION_STRING_EQUALS`). This will compare the value in the instruction with the current value of the level and only continue execution if they match. While this is a simple expression, it can be extended to many more expressions.

5.3 End-to-end example

In this section, we will walk through all steps that are necessary for querying, from creating all indexes to running the bytecode interpreter. For the input, we will use the line-delimited JSON shown in

listing 11. The query will be `$.user.lang[?(@ == 'en')]`, i.e. finding all `user.lang` values where the value is `"en"`.

Listing 11 End-to-end input example

```
1 {"user":{"lang":"en"},"text":"Lorem ipsum dolor sit amet."}
2 {"text": "Lorem\" ipsum: dolor sit amet.", "user":  { "lang":  "en" }}
3 {"user": {"id": "12345","lang":  "en"}}
4 {"id":  "1235"}
5 {"entities": ["123", "46"], "user":  {"lang":  "nl"}}
```

The first index that will be created is the string index, which is shown in figure 5.3. The string index is created by dividing the input string into 64-byte chunks and assigning these to CUDA threads, in sequential order. Therefore, if four threads are used to create the index of figure 5.3, each thread will process one 64-byte chunk, while when there are two threads, each thread will process two 64-byte chunks. The index for each 64-byte chunk consists of one 64-bit integer and contains for every character whether it is inside a string literal or not.

```
{"user":{"lang":"en"},"text":"Lorem ipsum dolor sit amet."}↵{"te
0111110001111100111000111110011111111111111111111111111110000111

xt": "Lorem\" ipsum: dolor sit amet.", "user":  { "lang":  "en"
1100011111111111111111111111111111000111110000001111100001100

}}↵{"user": {"id": "12345","lang":  "en"}}↵{"id":  "1235"}↵{"ent
0000111110000111000111111001111100001110000011100001111100001111

ities": ["123", "46"], "user":  {"lang":  "nl"}}
1111100001111000111000011111000001111100001110000
```

Figure 5.3: String index for listing 11

Another index which is created simultaneously is the newline index, which contains the positions of the newline characters. For our example, the newline index would consist of an array of positions, which are shown in table 5.1. The line numbers are not stored, but are the indexes of each position in the array. The first position is always 0, since it also starts a new line within the file, and this is necessary for easier processing in a later step. The newline index is parallelized on byte-level, and, as such, if there are four threads this 240-byte input will be divided into four 60-byte chunks, while if there are two threads this input would be divided into two 120-byte chunks.

Line	Position
1	0
2	59
3	130
4	170
5	186

Table 5.1: Newline index for our example input

Using the string index, a structural leveled bitmaps index can be created for this input, which is shown in figure 5.4. This index construction kernel operates on 64-byte chunks, similar to the string index kernel construction. The structural leveled bitmaps index contains in which positions a structural character is located in each level of the JSON input.

Figure 5.4: Structural leveled bitmaps index for listing 11, with levels listed on the left side

With all indexes created, the bytecode interpreter GPU kernel can be used to query for a value. First, our query needs to be converted into our domain-specific bytecode. Our query `$.user.lang[?(@ == 'en')]` can be converted into bytecode as shown in table 5.2. This results in a full "program" to be executed as shown in listing 12.

Part	Instruction
$.	-
user	MOVE_TO_KEY "user"
.	MOVE_DOWN
lang	MOVE_TO_KEY "lang"
[	MOVE_DOWN
(@ == 'en')	EXPRESSION_STRING_EQUALS "en"
]	-
-	STORE_RESULT
-	END

Table 5.2: Bytecode conversion of our input query

Listing 12 End-to-end query bytecode

```
1 MOVE_TO_KEY "user"
2 MOVE_DOWN
3 MOVE_TO_KEY "lang"
4 MOVE_DOWN
5 EXPRESSION_STRING_EQUALS "en"
6 STORE_RESULT
7 END
```

The full bytecode is executed once for each line, based on the newline index. When there are two threads running in parallel, the first thread will get three lines, with the second thread needing to process the additional two lines. When using four threads, the first and second thread will process two lines, the third thread will process one line, and the fourth thread will process no lines.

We will now run through the instructions one-by-one to demonstrate how the bytecode is interpreted and how the various indexes are used. We will demonstrate this for lines 2, 4, and 5, to show how looking up a value works, what happens when a key is not found, and what happens when an expression is not matched.

5.3.1 Line 2

The full structural leveled bitmaps index for this line is shown in figure 5.5, without a division into 64-byte chunks.

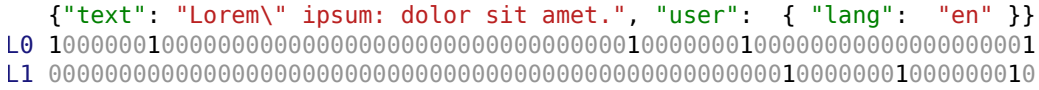

Figure 5.5: Structural leveled bitmaps index for line 2 of listing 11

5.3.1.1 MOVE_TO_KEY "user"

This instruction will iterate over all bits set to 1 in level 0 of the structural leveled bitmaps index. When it finds a :, it will check that the key matches "user" by comparing the previous string. Once the key matches, it will continue to the next instruction, leaving the current position pointer on the colon and not changing the level. The final state of this operation is shown in figure 5.6. If there were to be no match, it would continue to the next colon and do the same string comparison. If there are no further colons, and, as such, no more key-value pairs on this level, it will stop processing of this line and continue to the next line. When this happens, it will re-initialize the state of the bytecode interpreter to the starting state.

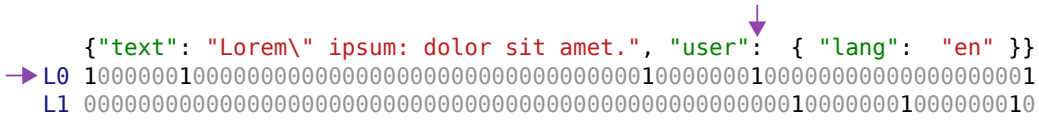

Figure 5.6: Position in line 2 after executing instruction 1

5.3.1.2 MOVE_DOWN

This will move the level pointer one level down, leaving the state of this operation as in figure 5.7.

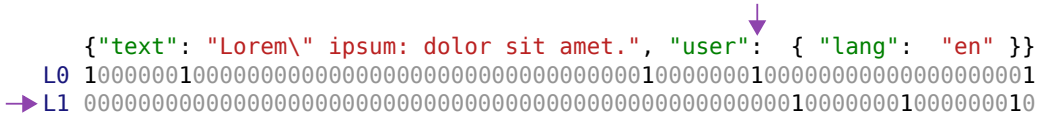

Figure 5.7: Position in line 2 after executing instruction 2

5.3.1.3 MOVE_TO_KEY "lang"

This will execute exactly the same operations as the first instruction, except that it does so in level 1, since that is the current level, and looks for the key "lang". The final state after this instruction is shown in figure 5.8.

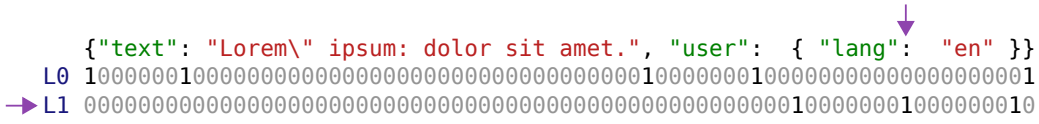

Figure 5.8: Position in line 2 after executing instruction 3

5.3.1.4 MOVE_DOWN

This instruction will again move the level pointer one level down. Since there is no further level, this results in an invalid state of the level pointer. However, the structural leveled bitmaps index will not be used after this instruction, so this is irrelevant in this case. The reason for this instruction is because it also calculates the end of the level, so it allows us to limit the boundaries of the value, which will be used in the next instruction.

5.3.1.5 EXPRESSION_STRING_EQUALS "en"

This instruction will find the value that is located at the current position by skipping whitespace and then comparing the value to the requested value. This is again a simple string comparison. This operation modifies the current position in the line since it skips all whitespace characters. The final state after this instruction is shown in figure 5.9. It will then continue to the next instruction, since the value matches.

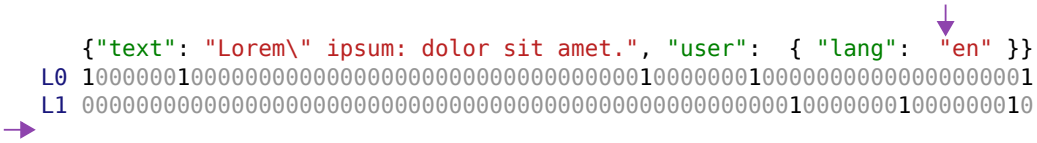

Figure 5.9: Position in line 2 after executing instruction 5

5.3.1.6 STORE_RESULT

This instruction will store the current position and the current end of the level in the result array. In this case, it would store positions 123 and 128 to the result array. These can be used to extract the value `"en"` from the input. The state is not modified.

5.3.1.7 END

This instruction moves the bytecode interpreter to the next line and does not modify the state further.

5.3.2 Line 4

The full structural leveled bitmaps index for this line is shown in figure 5.10.

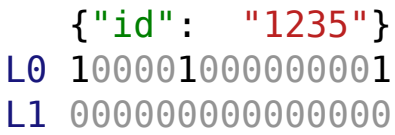

Figure 5.10: Structural leveled bitmaps index for line 4 of listing 11

5.3.2.1 MOVE_TO_KEY "user"

This instruction will again iterate over all bits set to 1 and do a string comparison to check keys. However, in this case there is no matching key, so when the complete line has been scanned, it will stop processing this line and move to the next line without retaining any state.

5.3.3 Line 5

The full structural leveled bitmaps index for this line is shown in figure 5.11. The first four instructions will execute exactly the same as for line 2, so we will not show the full steps here. Instead, we will show the state it is at after these instructions in figure 5.12.

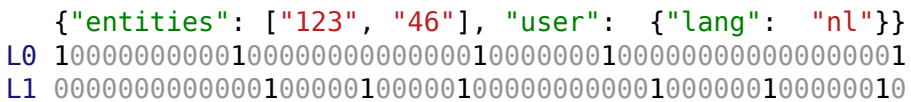

Figure 5.11: Structural leveled bitmaps index for line 5 of listing 11

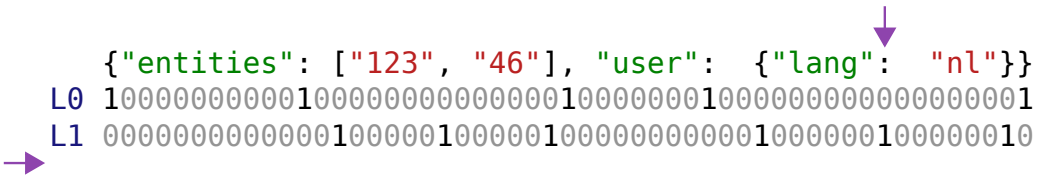

Figure 5.12: Position in line 5 after executing instruction 4

5.3.3.1 `EXPRESSION_STRING_EQUALS "en"`

This instruction will first skip whitespace, to arrive at the first byte of the value. Then, it will do a string comparison. For this line, the value will not match the requested string. This results in the bytecode interpreter stopping execution for this line, and continuing to the next line without retaining any state. This is the same process as what happens when a property or index instruction does not match.

Chapter 6

Implementation

6.1 Introduction

In this chapter, we will explore how the techniques explained in chapters 4 and 5 can be applied in the context of the GraalVM and implemented on the GPU using CUDA.

First, we show how our implementation, GPJSON, uses the techniques presented in previous chapters in section 6.2. Based on this initial construction of the kernels, there are many possible optimizations that can be performed, which will be explored in section 6.3.

6.2 Kernels

In figure 6.1, we show the high-level data flow of all GPU kernels used by GPJSON. These mostly follow the algorithms outlined in chapters 4 and 5, although some details are important to note.

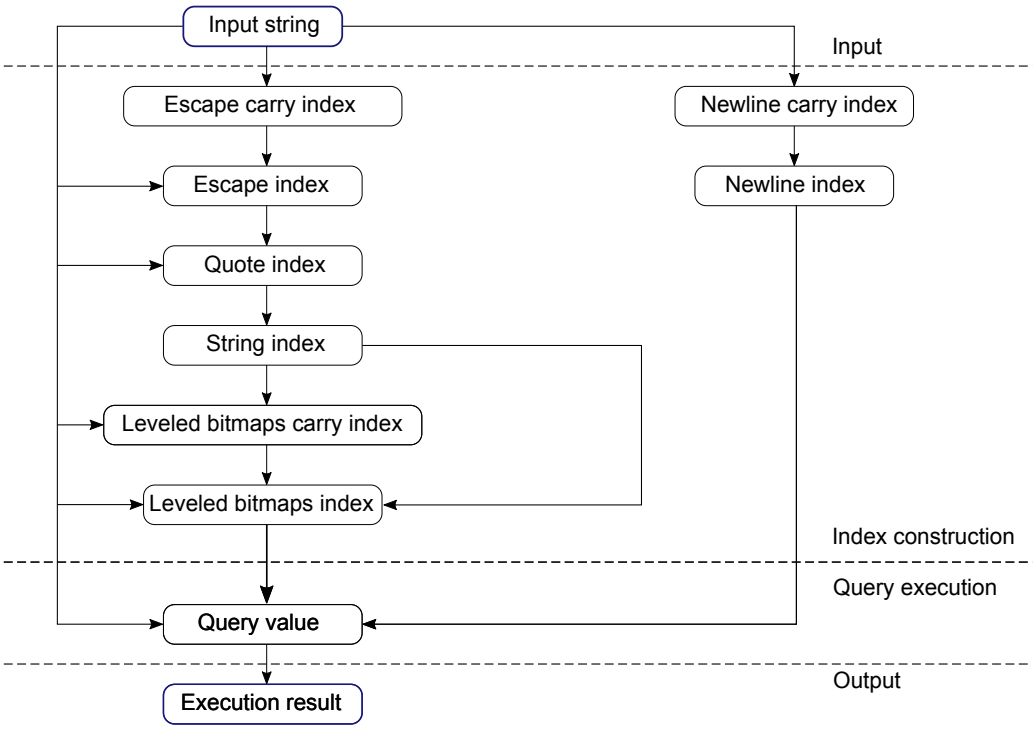

Figure 6.1: Data flow between kernels

First, for all bitmap indexes, it is important to make sure every kernel can fill in at least 64 bits at a time since the bitmap indexes are stored as 8-byte long integers. Therefore, the starting and ending positions

of each kernel need to be aligned to 64 characters. This can be calculated using equation (6.2.1), where n is the size of the file in bytes and t is the number of threads. This does mean that the GPU is only fully utilized with very large files. For all kernels, there are at least 8192 threads running in parallel, and there are 64 times more characters necessary than threads to even get to full usage of all the kernels. Therefore, a file will need to be at least $8192 \cdot 64 = 524\,288 \approx 500$KB.

$$l = \left\lfloor \frac{\lfloor n/t \rfloor}{64} \right\rfloor \cdot 64 \tag{6.2.1}$$

Second, it is vital to understand how the kernels are run. Every kernel is run in parallel, but the amount of parallelization can be controlled using CUDA's grid size. This allows setting the kernels to different amounts of parallelization based on the optimal performance for a specific kernel. However, not all of the kernels are independent with respect to their amount of parallelization. The carry indexes created are only applicable when the same amount of parallelization is used to run the following kernels. Therefore, the number of parameters that can be tweaked decreases with the number of kernels that are not fully independent with respect to their data. The amount of parallelization will be referred to as the number of threads, as that is what is ultimately controlled, albeit the exact configuration depends on both grid and block size.

The initial amount of threads that was selected was 8192 threads, which gives a reasonably good performance. Increasing the amount of throughput by varying the number of threads will be explored in section 6.3. At a file size of 500 KB, which is shown to be the minimum file size for full utilization, the overhead of starting many different kernels is quite significant, so even at this file size, the full throughput is not achieved. As will be evaluated in chapter 7, it requires multiple dozens of megabytes of data to achieve full throughput, which is one of the limitations. This could be improved by using runtime optimization and dynamically modifying the number of threads; this has, however, not been explored.

6.3 Optimizations

For optimizing the performance of the kernels, there are many possibilities. One of the first possibilities we explored is to reduce the number of kernels that are run. As shown in figure 6.1, the newline index and leveled bitmaps index are fully independent with respect to their data. Therefore, it is possible to run these in the same kernel, which is why we have combined the newline carry index and escape carry index and the newline index and escape index. This leads to a control flow as shown in figure 6.2, where combined kernels are outlined in red, and decreases the number of kernels run by two, resulting in less overhead of switching between CPU and GPU.

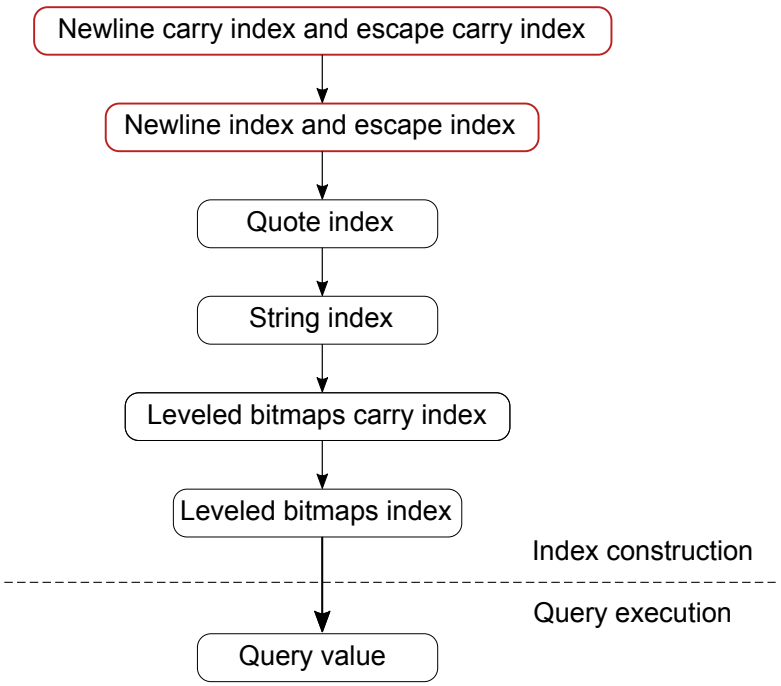

Figure 6.2: Control flow between optimized kernels

Another optimization is only to compute the leveled bitmaps index for levels required to execute the query. By limiting the number of stored levels, both the amount of memory used and the number of memory writes are reduced. To do so, the JSONPath parser stores the depth of the query, which is then passed to the individual kernels and the GPU memory allocation code.

Some optimizations reduce the amount of memory accesses on the input data when finding a value. For example, when finding the end of a level, which is necessary to limit the querying and to calculate the size of the result, a simple loop would look like shown in listing 13. This results in retrieving the same value, namely the current value of the leveled bitmaps index 64 times. This is not a problem since the GPU caches it. However, it is possible to derive information from the value returned by `current_level`. If this value is 0, that means that none of the characters in the current 8-byte long are the end of the level. This can be applied by skipping to the next 8-byte long, as shown in listing 14. Other optimizations might also be possible here, such as using the `__ffsll` compiler intrinsic [41], which would find the first bit set to 1. However, this was not implemented due to time constraints.

Listing 13 Unoptimized loop for finding the end of a level

```
for (long m = j + 1; m < level_ends[current_level - 1]; m += 1) {
  int current_level_index = level_size * (current_level - 1) + m / 64;
  long current_level = leveled_bitmaps_index[current_level_index];
  long required_bit = (1L << m % 64);
  bool is_structural = (current_level & required_bit) != 0;
  if (is_structural) {
    level_ends[current_level] = m;
    break;
  }
}
```

Lastly, it is also possible to change the number of threads that are running a specific kernel. This would unfortunately also increase memory usage for all index construction kernels since an index is built based on the number of threads. However, for the kernel that queries values, changing the number of running threads is possible. In our experiments, increasing the number of threads from 8192 to 524 288 resulted in slightly better performance while also making the kernel more scalable. In this kernel, each thread

Listing 14 Optimized loop for finding the end of a level

```
for (long m = j + 1; m < level_ends[current_level - 1]; m += 1) {
  int current_level_index = level_size * (current_level - 1) + m / 64;
  long current_level = leveled_bitmaps_index[current_level_index];
  if (current_level == 0) {
    m += 64 - (m % 64) - 1;
    continue;
  }
  long required_bit = (1L << m % 64);
  bool is_structural = (current_level & required_bit) != 0;
  if (is_structural) {
    level_ends[current_level] = m;
    break;
  }
}
```

processes a single line, so full utilization is only reached when 524 288 lines are present in the file, which is not the case for our 800MB+ datasets.

Chapter 7

Evaluation

7.1 Introduction

In this chapter, we will evaluate the implementation of the described method for several datasets and queries. We will compare our implementation against other state-of-the-art JSON parsers and query processing systems, as well as to Node.js-based JSONPath query evaluation engines.

In section 7.2 we will describe which implementation we compared against and how the benchmarks were run. In section 7.3 we will present our results, and in section 7.4 we will discuss the results.

7.2 Methodology

In this section, we will describe how the benchmarks were run and which implementation we compared against.

As stated in chapter 6, our implementation can be used from multiple languages, as available in GraalVM. We have chosen to use Node.js for these benchmarks since there are many JSONPath libraries available for Node.js, which allows direct comparison between these libraries and our implementation.

The libraries we compare against have been selected based on their performance, as shown in previous work. In addition, we have added implementations with a higher-level interface in Node.js. The implementations we will be comparing against are listed below. A short overview is provided in table 7.1.

- simdjson [7]: simdjson uses SIMD instructions to parse JSON documents into JSON trees, which can then be queried manually by the user. It is implemented in C++ and exposes a C++ API. simdjson is used in its default configuration using the document stream interface, allowing loading a newline-delimited JSON file directly. JSONPath queries have been manually converted to their corresponding C++ API calls.
- Pison [1]: Pison is a parallel SIMD JSON query execution engine implemented in C++. Pison is used as published on GitHub (`https://github.com/AutomataLab/Pison`). The number of parallel constructors and queries running is adjusted based on the logical core count of the machine. The number of levels in the bitmap that is constructed depends on the query and the minimal amount necessary to run the query. JSONPath queries are not directly supported, and, as such, they have been manually converted to the Pison C++ API.
- Java JsonPath [12]: Java JsonPath is a popular JsonPath implementation in Java, supporting JSONPath queries directly. This JsonPath is parallelized using the Java Streams API, allowing multiple lines to be queried in parallel. The Java Streams API is an abstraction over Java Threads, which in turn uses operating system threads and the Java Fork-Join Pool [42].

- RapidJSON [43]: RapidJSON is a popular C++ JSON parser developed by Tencent and is used in its default configuration. It does not support JSONPath queries, and, as such, all JSONPath queries have been manually converted to the corresponding C++ API calls.
- Node jsonpath [44]: This is a JSONPath implementation for Node.js. This jsonpath implementation is used together with Node's readline interface and as a result of how the library functions, it will use the default `JSON.parse` function available in Node.js. It is run on Node 14.16 using the Node.js distribution, which ships with the V8 JavaScript engine.
- Node jsonpath-plus [45]: This is an alternative JSONPath implementation for Node.js which is based on the original JavaScript implementation described in the JSONPath proposal [11]. This implementation is used in the same way as the jsonpath library and will thus also use the default `JSON.path` and Node's readline interface.
- Node manual extraction: This implementation does not use a library and instead has handwritten functions to extract the data from a parsed JSON object, removing the overhead of parsing the JSONPath query and showing the performance impact of using a library like jsonpath and jsonpath-plus. It uses the default `JSON.parse` available in Node.js, like the other two Node.js JSONPath libraries, and is thus able to show the performance overhead of using a library.
- Node simdjson: This implementation is a binding from Node.js to simdjson, allowing simdjson to be used from Node.js. It provides a lazy parsing API that parses a JSON string using simdjson, after which values can be queried using a custom query language. The custom query language is very similar to JSONPath, and for all queries we will evaluate, it suffices to remove the initial `$.`.

Name	**Platform/language**	**Supports JSONPath**
GPJSON	GraalVM	Yes
simdjson	C++	No
Pison	C++	No
Java JsonPath	Java	Yes
RapidJSON	C++	No
Node jsonpath	Node.js/JavaScript	Yes
Node jsonpath-plus	Node.js/JavaScript	Yes
Node manual extraction	Node.js/JavaScript	No
Node simdjson	Node.js/JavaScript/C++	Partially

Table 7.1: Evaluated implementations

As listed, for almost all implementations, the queries needed to be converted from their JSONPath representation to the programming language in which the implementation is implemented manually since these implementations do not support a standardized query language. The only implementations that support JSONPath are Java JsonPath, Node jsonpath, Node jsonpath-plus, and our implementation. This also means that there are multiple possible choices for how the querying is implemented. In all cases, we have chosen to count the number of results and, if possible, to not retrieve the value at a specific path. For our implementation, we retrieve all values since that is required to count the number of values that are not null. This means that by slightly changing the kernel to only retrieve the number of found results, rather than the result indexes, it is possible to achieve higher performance than shown here. However, this would not be fair to other implementations for which it is not possible to only retrieve the number of results, hence our choice for retrieving all values for GPJSON.

As a comparison between having support for JSONPath and not having it, we will compare the code required to query a simple value for our implementation and simdjson. Our implementation is shown in listing 15, while simdjson's required code is shown in listing 16. For more complex queries which also include expressions or need to retrieve multiple values, the difference between the two becomes even more pronounced since it is only a one-line change for GPJSON, while simdjson requires more code to extract and compare these values fully.

Listing 15 GPJSON's required code for querying `$.user.lang` in a dataset

```
const gpjson = Polyglot.eval('gpjson', 'jsonpath');

const result = gpjson.query('dataset.json', ['$.user.lang']);

// Use result
```

Listing 16 simdjson's required code for querying `$.user.lang` in a dataset

```
#include "simdjson.h"

int user_lang_query(const simdjson::dom::element &doc) {
  simdjson::error_code error = doc["user"]["lang"].error();
  if (error == simdjson::error_code::SUCCESS) {
    // Process element
  }
}

void main() {
  simdjson::dom::parser parser;

  simdjson::dom::document_stream stream = parser.load_many("dataset.json");

  for (simdjson::dom::element doc : stream) {
    user_lang_query(doc);
  }
}
```

An additional difference between existing implementations and GPJSON is that our implementation does not return a parsed JSON structure but instead returns a JSON string. Therefore, to ensure we have the same level of access between the other implementations and our implementation, we also have an implementation that includes the time to parse the JSON in Node.js, using `JSON.parse`. Since our implementation runs on GraalVM, the `JSON.parse` implementation used is different from the one included in the Node.js benchmarks since those use the V8 JavaScript engine.

For the datasets, we have chosen to use a subset of the benchmarks used by Pison [1]. These datasets represent real-world, representative LDJSON datasets that contain semi-structured data. The datasets used are a Best Buy (BB) product dataset [46], a stream of tweets from the Twitter (TT) developer API [9], and a Walmart (WM) product dataset [47]. Each full dataset is of approximately the same size, about 1GB. In contrast to the datasets used by Pison [1], we are using newline-delimited records rather than one large JSON record due to our implementation functions. Some statistics for each dataset can be found in table 7.2.

The queries that were evaluated are based on the Pison queries, as listed on their GitHub page [48]. We have selected the queries that are compatible with our implementation, which does not support queries with an indeterminate number of results per line. In addition, to evaluate the performance of expressions, we have selected two additional queries. These are based on one of the queries used by Pison, with an additional condition added; one condition with low selectivity and one condition with

Dataset	Number of lines	Size (MB)	Depth
BB	230 089	996	6
TT	155 489	804	10
WM	272 499	949	3

Table 7.2: Evaluated datasets

high selectivity. table 7.3 lists the queries that were used.

Dataset	Queries	Number of results
TT	`{$.user.lang}`	155 490
TT	`{$.user.lang, $.lang}`	300 270
WM	`{$.bestMarketplacePrice.price, $.name}`	288 391
BB	`{$.categoryPath[1:3].id}`	459 332
TT	`{$.user.lang[?(@ == 'nl')]}`	405
TT	`{$.user.lang[?(@ == 'en')]}`	137 559

Table 7.3: Evaluated queries

All benchmarks were run on a single machine containing an Intel i7-5820K, 32GB of DDR4 RAM, and running an NVIDIA GeForce GTX 1080 (2560 CUDA cores) for our implementation. The CPU supports all prerequisites for Pison: 64-bit CPU, the carry-less multiplication instruction, and 256-bit SIMD instruction set support. The machine runs Ubuntu 20.04, uses GCC 9.3.0, CMake 3.20.2, JDK 11.0.9, and GraalVM CE 21.0.0.2 using Java 8. All C++ programs were compiled in default CMake release mode and use vcpkg for installing RapidJSON. The disk that was used is a Samsung SSD 970 EVO 1TB NVMe.

Each run in a benchmark for a specific implementation was run 15 times. The first five times were warm-up runs and were not recorded for evaluation, allowing caches to warm up. The following ten times were recorded and averaged. All 15 runs happen within the same process. The times were recorded in the same process as the benchmark, reducing errors due to kernel pauses when communicating between processes. We also record the number of results for correctness purposes.

7.3 Benchmark results

In this section, we will evaluate the performance of all implementations in several different conditions and with different queries. The goal is to evaluate the performance of GPJSON compared to existing implementations and determine whether our novel approach leads to a performance improvement compared to existing techniques. First, we will evaluate the performance baseline performance when the datasets and queries are used unmodified.

The performance baseline for all experiments will be GPJSON running in a Node.js context on GraalVM. When performance differences are shown relatively, GPJSON will be the performance baseline to show performance gains or regressions compared to existing implementations. It is also important to note that the currently most advanced library is simdjson, so we will be comparing GPJSON against simdjson in many our experiments. However, there is a significant difference in the usability of GPJSON and simdjson; simdjson is only available from C++ and does not support JSONPath, while GPJSON is available from a multitude of high-level scripting languages and support JSONPath. Therefore, we will also be comparing against the Node.js implementations to show the performance gains over implementations on the same platform.

7.3.1 End-to-end performance

The queries without expressions will be evaluated first. Figure 7.1 reports the time to query the full dataset. First, we can see that all Node.js implementations take much longer to query the data than the other implementations. The JSONPath parsing and querying overhead is relatively small for the jsonpath and jsonpath-plus datasets, since the Node manual extraction shows the optimal performance of such a scenario. However, this is still bound by the performance of `JSON.parse`, which is the dominating factor in these implementations. This shows that even with optimal JSONPath parsing and querying, the Node.js implementations are not able to match the speed of Java or C(++) implementations due to the poor performance of `JSON.parse`.

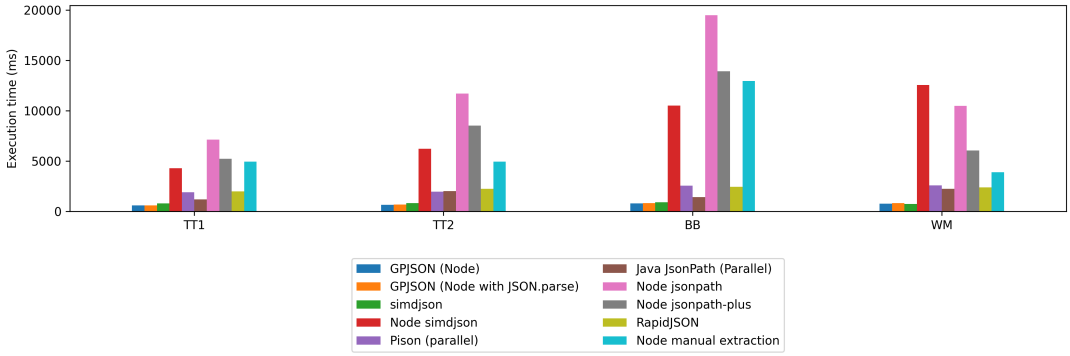

Figure 7.1: Comparison of JSON query time for queries without expressions (lower is better)

One surprising result is that for our configuration, the Pison implementation seems to be slower than simdjson and closely resembles the performance of RapidJSON, in contrast to what the authors report. As reported above, the machine used for these benchmarks supports all extensions that Pison requires. We are also running in parallel mode, which should improve the performance compared to their serial performance, which was shown for many small records. Therefore, we would have expected a higher throughput than the authors, rather than a lower throughput. Even when running in single-threaded mode, we were unable to reproduce the performance shown in their evaluation. Further evaluation is necessary as to why this performance difference exists. All other libraries and frameworks used in our evaluation perform in line with our expectation.

The performance of the parallel Java JsonPath implementation is better than expected since it even outperforms RapidJSON. However, that difference could be explained since we are not running RapidJSON in parallel; the work required to get RapidJSON to work in parallel is much larger than for the Java JsonPath implementation, where only a few lines needed to be modified.

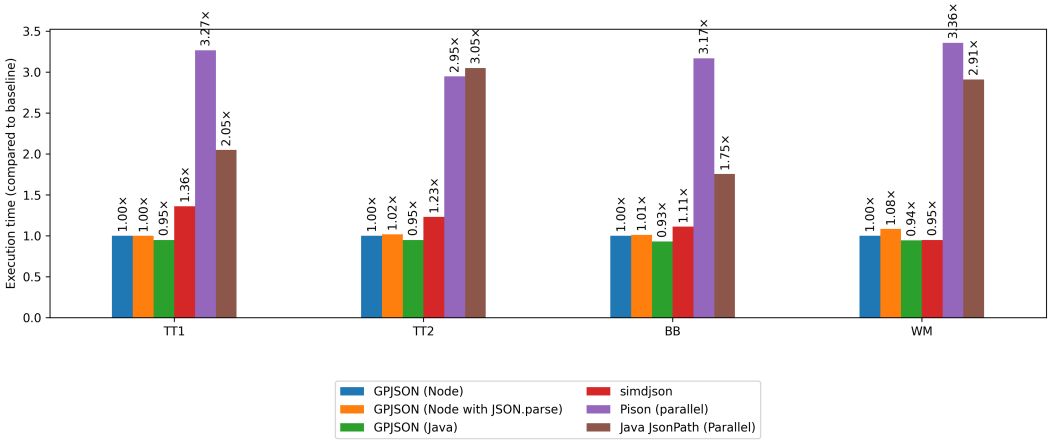

Figure 7.2: JSON query times for queries without expressions compared to the performance baseline (lower is better)

Our implementation shows a 15% improvement over simdjson for the case of a single query, with a much smaller difference for multiple queries. This shows that index construction and querying on a GPU is a viable and performant option. The performance difference between one and two queries can be

explained by our implementation needing to run the querying kernel twice, while simdjson can iterate over all results sequentially, essentially executing both queries simultaneously. With more optimization in our JSONPath parser, this would certainly also be possible in our implementation since our bytecode interpreter can move up and down within the levels of the JSON structure.

For the Walmart dataset, simdjson is slightly faster than our implementation. However, this difference is negligible and is caused entirely by the overhead of running in Node.js; when we run our implementation on GraalVM using Java, we are slightly faster than simdjson, while still running in a higher-level environment. This is shown in figure 7.2.

The performance of Node simdjson shows that there is significant overhead when using simdjson in Node.js due to implementation differences and fewer possible optimizations. Our implementation is at least seven times faster than Node simdjson, while using the same high-level scripting language.

Other Node.js implementations show poor performance as well, as shown in figure 7.3. None of the Node.js implementations come close to the performance of our implementation. The fastest implementation is Node.js manual extraction, but this does not support JSONPath since it is manually written to extract the correct values. The JSONPath-compatible implementations are even slower due to the overhead of using the library and parsing JSONPath.

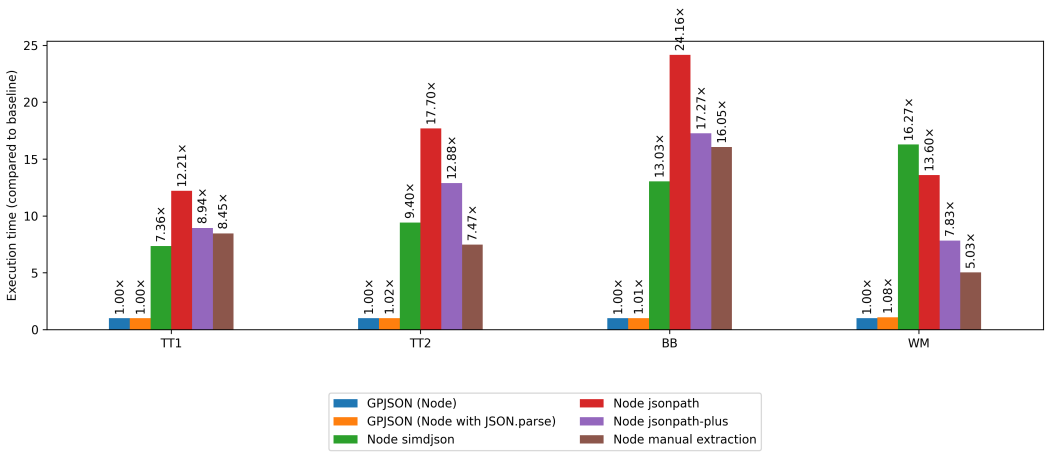

Figure 7.3: Node.js JSON query times for queries without expressions compared to the performance baseline (lower is better)

So far, we have only evaluated large datasets. Now, we will evaluate the impact of dataset size on the performance of the implementations. To do so, we will use the TT dataset with the `$.user.lang` query, and modify the size of the dataset to be between 100kB and 500MB. The results of our evaluation are shown in figure 7.4.

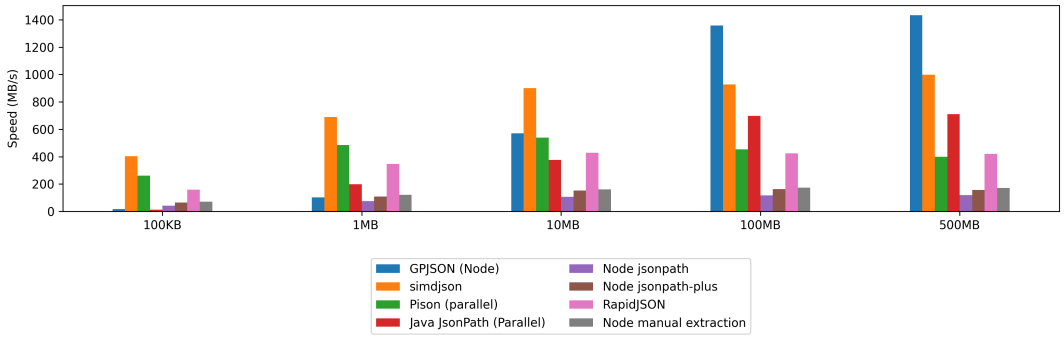

Figure 7.4: Comparison of JSON query throughput for different dataset sizes (higher is better)

For tiny dataset sizes, the performance of all implementations is lower than for larger files. simdjson is least affected, at only half the speed, while our implementation is most affected and drops down to 7.3 MB/s, compared to 1.3 GB/s for the largest files. This can be explained by the large overhead of switching between the CPU and GPU and the fact that our implementation runs 8192 threads at a time, each requiring at least 64 bytes. Therefore, our implementation requires at least 500kB to even utilize all threads, resulting in the 100kB dataset not using all threads. As the size of the dataset increases, so does the performance of our implementation due to less relative overhead and more utilization of the GPUs. Above 500MB, the speed does not increase further, indicating that the maximum performance of our implementation has been reached. The same can be concluded for simdjson and the other implementations.

Next, we will evaluate the impact of selectivity, which we evaluated by keeping the same dataset size, but removing the `$.user.lang` value from a variable number of lines, until the required selectivity was reached. Each line from which the value was removed, was modified in one of three different ways:

- Only removing the `$.user.lang` key, resulting in the `user` key still being present
- Removing the `$.user` key, resulting in no `user` key being present
- Removing both the `$.user` and the `$.entities` key, which results in both not being present and in a much smaller record

These modifications were all done equally as often, and the benchmark results for all tested selectivities can be found in figure 7.5.

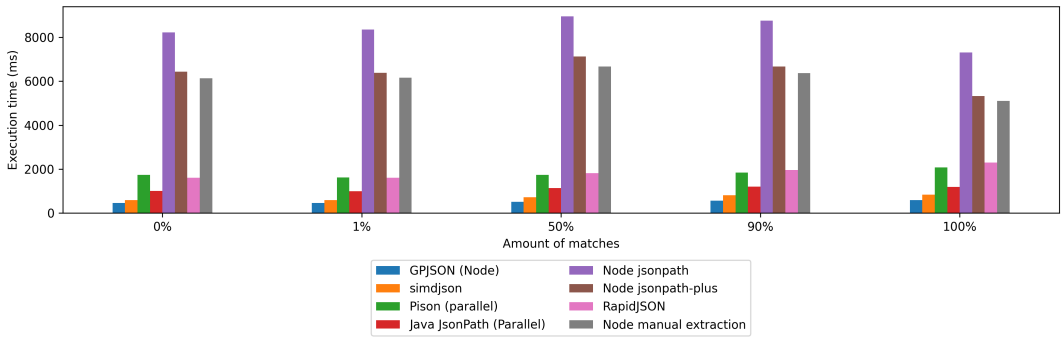

Figure 7.5: Comparison of JSON query time for different dataset selectivities (lower is better)

These results show that the selectivity of the datasets does not have a significant effect on all implemen-

tations. There are minor differences between the runs, but small variations in CPU activity probably cause these. This shows that our method's performance is independent of the selectivity and is still faster than simdjson and all other implementations. This is expected, as the full kernel is run regardless of the selectivity and most of the querying time is spent in finding the top-level value. A full scan of the top-level structured leveled bitmaps index is still executed when this value is not present.

Lastly, we will evaluate the performance of the queries, which include expressions. We have only evaluated our implementation against simdjson for this evaluation since it shows the most competitive performance in our previous benchmarks. In addition, we have added a C++ implementation of our method, which is incomplete since it does not return the values from the results but does show the performance of just executing queries without retrieving values or parsing the returned JSON values. We will show one query with low selectivity and one with high selectivity, as shown in table 7.3. The results are shown in figures 7.6 and 7.7.

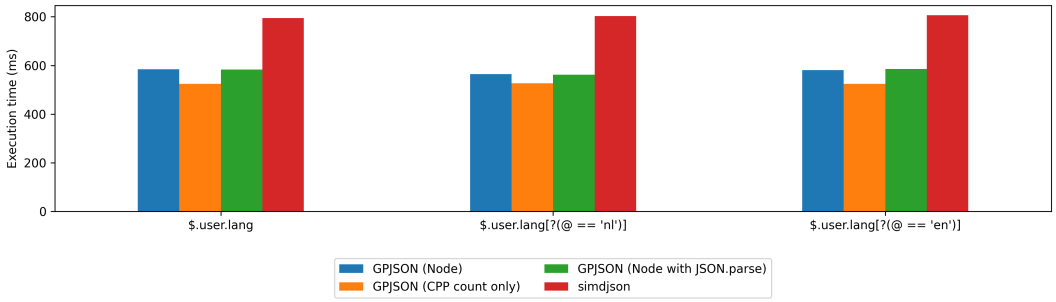

Figure 7.6: Comparison of JSON query time for different expression queries (lower is better)

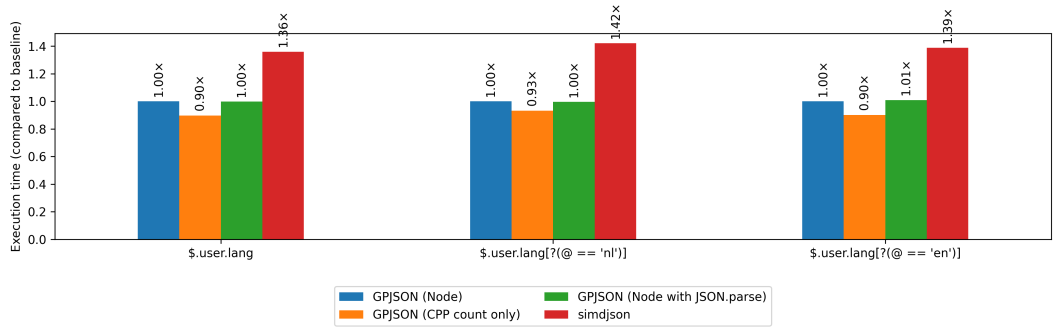

Figure 7.7: JSON query time for different expression queries compared to performance baseline (lower is better)

These results show fewer rows lead to a slight increase in performance, although this is only 2% for our implementation and 1% for simdjson. The advantage of our method is that the expression can be fully evaluated on the GPU using our bytecode interpreter without any performance impact. Even when the resulting strings are parsed to a JavaScript value and usable from a scripting language, the speed-up is still more than 25% compared to simdjson.

7.3.2 Kernel performance breakdown

So far, we have only measured the end-to-end performance of GPJSON. To fully evaluate our implementation's performance, we will also show the performance of the separate kernels. This shows which steps in the process have the largest impact on the performance and shows the querying step's impact, which was shown to be somewhat significant when multiple queries need to be run on an already constructed index. In other words, with this experiment, we want to assess the performance of the GPU alone, without any overhead caused by IO to read the file or due to switching kernels. Tables 7.4 to 7.7 show the performance of each kernel with the previously reported datasets and queries. These times were measured by taken the times measured by GPJSON for the 10th run of the same query on the same file. The times shown are only the times spent in CUDA kernels; the total duration is shown as the total duration from calling into GPJSON until the result is returned.

Step	**Duration (ms)**	**Percentage**
1. Read and copy file	107.68	19.31%
2. Create escape carry and newline count index	62.6	11.23%
3. Create escape and newline index	71.41	12.81%
4. Create quote index	74.47	13.36%
5. Create string index	4.62	0.83%
6. Create leveled bitmaps carry index	41.28	7.40%
7. Create leveled bitmaps index	68.54	12.29%
8. Query value	85	15.24%
Total duration	**557.6**	**100.00%**

Table 7.4: Costs of different steps in our implementation (TT)

Step	**Duration (ms)**	**Percentage**
1. Read and copy file	107.4	21.02%
2. Create escape carry and newline count index	61.74	12.08%
3. Create escape and newline index	71.24	13.94%
4. Create quote index	74.79	14.64%
5. Create string index	5.01	0.98%
6. Create leveled bitmaps carry index	41.29	8.08%
7. Create leveled bitmaps index	68.62	13.43%
8. Query value	40.64	7.95%
Total duration	**510.93**	**100.00%**

Table 7.5: Costs of different steps in our implementation (TT1)

As expected, the most significant impact on performance is reading the file and copying memory from the CPU to the GPU, even though the disk on the machine can achieve a sequential throughput of over

Step	**Duration (ms)**	**Percentage**
1. Read and copy file	139	20.01%
2. Create escape carry and newline count index	77.76	11.19%
3. Create escape and newline index	83.75	12.05%
4. Create quote index	90.52	13.03%
5. Create string index	4.25	19.82%
6. Create leveled bitmaps carry index	52.62	19.82%
7. Create leveled bitmaps index	137.69	19.82%
8. Query value	57.36	8.26%
Total duration	**694.82**	**100.00%**

Table 7.6: Costs of different steps in our implementation (BB)

Step	Duration (ms)	Percentage
1. Read and copy file	129.55	20.75%
2. Create escape carry and newline count index	74.79	11.98%
3. Create escape and newline index	85.45	13.69%
4. Create quote index	88.81	14.22%
5. Create string index	5.47	6.57%
6. Create leveled bitmaps carry index	40.99	11.51%
7. Create leveled bitmaps index	71.84	11.51%
8. Query value	88.58	14.19%
Total duration	**624.33**	**100.00%**

Table 7.7: Costs of different steps in our implementation (WM)

2GB/s. Still, all other steps also have a significant performance impact. All kernels are bound by the GPU's memory speed and the memory access patterns since data is being read from global memory in all kernels. Only the string index creation is much faster because it does not need to access the file contents since it only performs the prefix-XOR sum.

By optimizing the memory access patterns and minimizing the number of global memory accesses, these kernels may be optimized further. Nevertheless, we believe that a large portion of our performance is limited by just reading and copying the file, which is hard to optimize further.

We also report the index construction time and querying time for both simdjson and Pison. To measure the performance of index construction, we ran our benchmark implementations without any queries being executed. We only ran these benchmarks for the TT dataset with 1 query (TT1) since these results are very similar between the datasets. The results are shown in tables 7.8 and 7.9.

Step	Duration (ms)	Percentage
Index construction	832.4	98.47%
Querying	12.9	1.53%
Total duration	**845.3**	**100.00%**

Table 7.8: Costs of different steps in simdjson (TT1)

Step	Duration (ms)	Percentage
Index construction	2008.3	95.89%
Querying	86.1	4.11%
Total duration	**2094.4**	**100.00%**

Table 7.9: Costs of different steps in Pison (TT1)

These results show that querying has minimal impact on simdjson and Pison, while taking up a much more significant percentage for our implementation. Therefore, for many queries on the same dataset, it may be beneficial to use simdjson instead of GPJSON. However, simdjson is not available from Node.js and uses a much lower-level interface than JSONPath for querying values.

7.4 Discussion

The above results show that our method outperforms other low-level, hand-optimized, state-of-the-art JSON querying methods. Since this is a novel implementation of GPU-based JSON querying, further optimizations can lead to further reductions in querying times, showing a promising role for GPU-based JSON querying methods.

We show that the selectivity of a query does not impact our querying performance in line with other state-of-the-art JSON parsers. However, GPJSON is only beneficial for large datasets; small datasets

show a significant regression compared to simdjson. This is caused by the high cost of switching between the CPU and GPU and not being able to fully utilize the processing power of a GPU when there is not enough data.

Compared to other JSONPath query engines available to high-level scripting languages, our implementation shows a more than 5× performance improvement with a similar or simpler API. This shows the benefits of exposing fast implementations of common operations via GraalVM to high-level scripting languages. Another benefit is that our implementation is not available for JavaScript but also for Python and Ruby, along with any other supported languages available on GraalVM.

Chapter 8

Conclusions

In this thesis, we explored the feasibility of a GPU-based JSON query evaluation engine. We introduced a novel technique for constructing a structural index on-GPU and show how to use this structural index to query values. We provide an experimental evaluation of GPJSON, an open source implementation of our technique available at `https://github.com/koesie10/gpjson`, which is shown to have significant speed-ups compared to state-of-the-art JSON parsers and query engines on large datasets.

8.1 Contributions

We showed how existing techniques for structural index construction using SIMD can be adapted to SIMT methods on the GPU. We show how this can be applied to structural string index and structural leveled bitmaps index construction and how this can be applied to newline-delimited JSON documents.

For query evaluation, we show that an on-GPU bytecode interpreter is a feasible abstraction of different query languages and how our bytecode interpreter can query JSON values. We show that a bytecode interpreter for query evaluation may be extended with support for expressions and that expressions can be evaluated on a GPU.

Our experiments show that the combination of structural index construction and query evaluation on a GPU is faster than state-of-the-art JSON parsers and query evaluation engines when the size of the dataset is sufficiently large ($\geq$ 100MB). For smaller datasets, we discussed why a GPU-based query evaluation engine is likely to be infeasible. We also show that the selectivity of the dataset is not a factor in our implementation's performance.

8.2 Limitations

All queries that we evaluate and support are queries with a pre-determined number of results per line. Queries without a pre-determined number of results per line (indefinite queries), such as queries that contain a recursive descent operator (`..`) are not supported by our current implementation. However, this is a limitation of our implementation, and by using streaming, this can be achieved with only a small adaptation to our shown algorithms.

The implementation we have shown is also only able to operate on datasets that fully fit into the GPU's memory, including the size of all indexes. By using a streaming approach and minor adaptations to our algorithms, this limitation can be lifted. This was not explored in this thesis because adding support for this would not have any significant consequences for the core techniques and would only present an additional challenge in the implementation phase.

A peculiar aspect of our implementation is that it only supports line-delimited JSON. To support other types of JSON datasets, such as those represented as a top-level JSON array or even as a top-level JSON

object containing a JSON array, somewhat larger adaptations to our algorithm would be required. This is still a feasible option since we have already shown that calculating the current level of a byte in the input is feasible. The same technique can be used to determine the delimiters between records.

Like other JSON query evaluation engines such as Mison and Pison, the algorithms we have shown do not validate the input; they assume all input is valid JSON. Therefore, when the input is not valid JSON, the output is undefined and possibly incorrect. Validating the correctness of the JSON documents would be a great addition to the implementation. A simple validation to add would be validating that the number of curly braces and square brackets is balanced for each record, for which no additional scanning over the file is necessary; it is only necessary to check that the level computed in the leveled bitmaps index is at level 0 when a new line is encountered. Other validations, such as UTF-8 input validation, would require additional work but can also be validated on-GPU by looking beyond the specific bytes assigned to a CUDA thread.

8.3 Future work

Multiple interesting angles can be explored. We showed how one structural index can be constructed for both objects and arrays. However, it would be interesting to evaluate whether building separate indexes for every character would lead to a speed-up due to fewer look-ups in the original input data. This will come with the cost of additional memory usage; however, this can be somewhat prevented by also implementing a streaming approach.

Another obvious addition would be to implement support for multiple expression operators, in addition to the one our implementation currently supports. More complex operators are especially challenging, even more so whether they can have a more significant edge over existing state-of-the-art implementations.

Bibliography

[1] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable Structural Index Construction for JSON Analytics. *Proc. VLDB Endow.*, 14(4):694–707, December 2020.

[2] NVIDIA. NVIDIA Ampere GA102 GPU Architecture, 2020. Whitepaper.

[3] Elias Stehle and Hans-Arno Jacobsen. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *Proc. VLDB Endow.*, 13(5):616–628, January 2020.

[4] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A Fast JSON Parser for Data Analytics. In *The 43rd International Conference on Very Large Data Bases (VLDB 2017)*, June 2017.

[5] X. Si, A. Yin, X. Huang, X. Yuan, X. Liu, and G. Wang. Parallel Optimization of Queries in XML Dataset Using GPU. In *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, pages 190–194, 2011.

[6] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB in Action: Adaptive Query Processing on Raw Data. *Proc. VLDB Endow.*, 5(12):1942–1945, August 2012.

[7] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *The VLDB Journal*, 28, 12 2019.

[8] ECMA TC39. ECMA-404: The JSON data interchange syntax, December 2017.

[9] Twitter Developer API. `https://developer.twitter.com/en/docs/`. Retrieved: 2019-07-01.

[10] W3C. XML Path Language (XPath) 3.1, March 2017.

[11] Stefan Goessner. JSONPath - XPath for JSON, 2007. `https://goessner.net/articles/JsonPath/`. Retrieved: 2021-03-15.

[12] JsonPath: Java JsonPath implementation. `https://github.com/json-path/JsonPath`. Retrieved: 2021-03-18.

[13] V8 release v7.6, June 2019. `https://v8.dev/blog/v8-release-76`. Retrieved: 2021-04-02.

[14] Daniele Bonetta and Matthias Brantner. FAD.Js: Fast JSON Data Access Using JIT-Based Speculative Optimizations. *Proc. VLDB Endow.*, 10(12):1778–1789, August 2017.

[15] Markus L. Noga, Steffen Schott, and Welf Löwe. Lazy XML Processing. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 88–94, New York, NY, USA, 2002. Association for Computing Machinery.

[16] Fernando Farfán, Vagelis Hristidis, and Raju Rangaswami. Beyond Lazy XML Parsing. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications*, pages 75–86, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[17] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. XML Screamer: An Integrated Approach to High Performance XML Parsing,

Validation and Deserialization. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, page 93–102, New York, NY, USA, 2006. Association for Computing Machinery.

[18] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 30(2):444–491, June 2005.

[19] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyên, Jouni Sirén, and Niko Välimäki. Fast in-memory XPath search using compressed indexes. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 417–428, 2010.

[20] Todd Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29:752–788, 01 2004.

[21] Craig Chasseur, Yinan Li, and J. Patel. Enabling JSON Document Stores in Relational Systems. In *WebDB*, 2013.

[22] M. DiScala and D. Abadi. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[23] Daniel Tahara, Thaddeus Diamond, and Daniel Abadi. Sinew: A SQL system for multi-structured data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 06 2014.

[24] Zhen Hua Liu, B. Hammerschmidt, and Doug McMahon. JSON data management: supporting schema-less development in RDBMS. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[25] NVIDIA. Programming Guide - CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation`. Retrieved: 2021-03-12.

[26] Robert Dow Jon Peddie. Supply shortages and mining batter the AIB market, March 2021. `https://www.jonpeddie.com/press-releases/supply-shortages-and-mining-batter-the-aib-market`. Retrieved: 2021-05-07.

[27] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[28] Jianbin Fang, Ana Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. pages 216–225, 09 2011.

[29] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 511–524, New York, NY, USA, 2008. Association for Computing Machinery.

[30] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 94–103, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Bogdan Simion, Suprio Ray, and Angela Demke Brown. Speeding up Spatial Database Query Execution using GPUs. *Procedia Computer Science*, 9:1870–1879, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[32] Alexander Krolik, Clark Verbrugge, and Laurie Hendren. r3d3: Optimized Query Compilation on GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 277–288, 2021.

[33] Alessandro Barenghi, Ermes Viviani, Stefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella. PAPAGENO: A Parallel Parser Generator for Operator Precedence Grammars. pages 264–274, 01 2013.

[34] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. A Parallel and Scalable Processor for JSON Data. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 576–587. OpenProceedings.org, 2018.

[35] Rene Mueller. grCUDA: A Polyglot Language Binding for CUDA in GraalVM, November 2019. `https://developer.nvidia.com/blog/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm/`. Retrieved: 2021-03-12.

[36] NVIDIA. NVIDIA A100 Tensor Core GPU, January 2021. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/a100-80gb-datasheet-update-nvidia-us-1521051-r2-web.pdf`. Retrieved: 2021-06-03.

[37] The GraalVM Community. The Truffle Language Implementation Framework. `https://github.com/oracle/graal/tree/master/truffle`. Retrieved: 2021-06-20.

[38] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Christopher Fraser and Todd Proebsting. Custom Instruction Sets for Code Compression. 01 1995.

[40] Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[41] NVIDIA. CUDA Math API. `https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT`. Retrieved: 2021-03-15.

[42] Doug Lea. A Java Fork/Join Framework. 2000.

[43] RapidJSON. `http://rapidjson.org/`. Retrieved: 2021-05-11.

[44] jsonpath: Query and manipulate JavaScript objects with JSONPath expressions. Robust JSONPath engine for Node.js. `https://github.com/dchester/jsonpath`. Retrieved: 2021-05-11.

[45] JSONPath Plus. `https://github.com/JSONPath-Plus/JSONPath`. Retrieved: 2021-05-11.

[46] Best Buy Developer API. `https://bestbuyapis.github.io/api-documentation/`. Retrieved: 2019-05-01.

[47] Product Lookup API. `https://developer.walmartlabs.com/docs`. Retrieved: 2019-05-10.

[48] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Pison: Scalable Structural Index Constructor for JSON Analytics, 2020. `https://github.com/AutomataLab/Pison`. Retrieved: 2021-05-11.