Parsing Gigabytes of JSON per Second

Geoff Langdale · Daniel Lemire

Abstract JavaScript Object Notation or JSON is a ubiquitous data exchange format on the Web. Ingesting JSON documents can become a performance bottleneck due to the sheer volume of data. We are thus motivated to make JSON parsing as fast as possible.

Despite the maturity of the problem of JSON parsing, we show that substantial speedups are possible. We present the first standard-compliant JSON parser to process gigabytes of data per second on a single core, using commodity processors. We can use a quarter or fewer instructions than a state-of-the-art reference parser like RapidJSON. Unlike other validating parsers, our software (simdjson) makes extensive use of Single Instruction, Multiple Data (SIMD) instructions. To ensure reproducibility, simdjson is freely available as open-source software under a liberal license.

1 Introduction

JavaScript Object Notation (JSON) is a text format used to represent data [4]. It is commonly used for browser-server communication on the Web. It is supported by many database systems such as MySQL, PostgreSQL, IBM DB2, SQL Server, Oracle, and data-science frameworks such as Pandas. Many document-oriented databases are centered around JSON such as CouchDB or RethinkDB.

Geoff Langdale
branchfree.org
Sydney, NSW
Australia
E-mail: geoff.langdale@gmail.com

Daniel Lemire
Université du Québec (TELUQ)
Montreal, Quebec
Canada
E-mail: lemire@gmail.com

The JSON syntax can be viewed as a restricted form of JavaScript, but it is used in many programming languages. JSON has four primitive types or atoms (string, number, Boolean, null) that can be embedded within composed types (arrays and objects). An object takes the form of a series of key-value pairs between braces, where keys are strings (e.g., {`"name":"Jack","age":22`}). An array is a list of comma-separated values between brackets (e.g., `[1,"abc",null]`). Composed types can contain primitive types or arbitrarily deeply nested composed types as values. See Fig. 1 for an example. The JSON specification defines six *structural characters* (‘`[`’, ‘{’, ‘`]`’, ‘}’, ‘`:`’, ‘`,`’): they serve to delimit the locations and structure of objects and arrays.

To access the data contained in a JSON document from software, it is typical to transform the JSON text into a tree-like logical representation, akin to the right-hand-side of Fig. 1, an operation we call JSON parsing. We refer to each value, object and array as a *node* in the parsed tree. After parsing, the programmer can access each node in turn and navigate to its siblings or its children without need for complicated and error-prone string parsing.

Parsing large JSON documents is a common task. Palkar et al. state that big-data applications can spend 80–90% of their time parsing JSON documents [19]. Boncz et al. identified the acceleration of JSON parsing as a topic of interest for speeding up database processing [2].

JSON parsing implies error checking: arrays must start and end with a bracket, objects must start and end with a brace, objects must be made of comma-separated pairs of values (separated by a colon) where all keys are strings. Numbers must follow the specification and fit within a valid range. Outside of string values, only a few ASCII characters are allowed. Within string values, several characters (like ASCII line endings) must

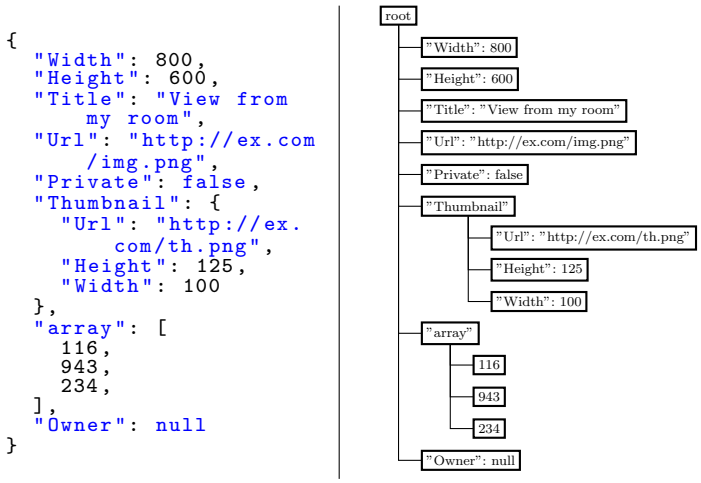


Fig. 1: JSON example

be escaped. The JSON specification requires that documents use a unicode character encoding (UTF-8, UTF-16, or UTF-32), with UTF-8 being the default. Thus we must validate the character encoding of all strings. JSON parsing is therefore more onerous than merely locating nodes. Our contention is that a parser that accepts erroneous JSON is both dangerous—in that it will silently accept malformed JSON whether this has been generated accidentally or maliciously—and poorly specified—it is difficult to anticipate or widely agree on what the semantics of malformed JSON files should be.

To accelerate processing, we should use our processors as efficiently as possible. Commodity processors (Intel, AMD, ARM, POWER) support single-instruction-multiple-data (SIMD) instructions. These SIMD instructions operate on several words at once unlike regular instructions. For example, starting with the Haswell microarchitecture (2013), Intel and AMD processors support the AVX2 instruction set and 256-bit vector registers. Hence, on recent x64 processors, we can compare two strings of 32 characters in a single instruction. It is thus straightforward to use SIMD instructions to locate significant characters (e.g., '"', '=') using few instructions. We refer to the application of SIMD instructions as *vectorization*. Vectorized software tends to use fewer instructions than conventional software. Everything else being equal, code that generates fewer instructions is faster.

A closely related concept to vectorization is branchless processing: whenever the processor must choose between two code paths (a branch), there is a risk of incurring several cycles of penalty due to a mispredicted branch on current pipelined processors. In our experience, SIMD instructions are most likely to be beneficial in a branchless setting.

To our knowledge, publicly available JSON validating parsers make little use of SIMD instructions. Due to its complexity, the full JSON parsing problem may not appear immediately amenable to vectorization.

One of our core results is that SIMD instructions combined with minimal branching can lead to new speed records for JSON parsing—often processing gigabytes of data per second on a single core. We present several specific performance-oriented strategies that are of general interest.

- We detect quoted strings, using solely arithmetic and logical operations and a fixed number of instructions per input bytes, while omitting escaped quotes (§ 3.1.1).
- We differentiate between sets of code-point values using vectorized classification thus avoiding the burden of doing N comparisons to recognize that a value is part of a set of size N (§ 3.1.2).
- We validate UTF-8 strings using solely SIMD instructions (§ 3.1.5).

2 Related Work

A common strategy to accelerate JSON parsing in the literature is to parse selectively. Alagiannis et al. [1] presented NoDB, an approach where one queries the JSON data without first loading it in the database. It relies in part on selective parsing of the input. Bonetta and Brantner use speculative just-in-time (JIT) compilation and selective data access to speed up JSON processing [3]. They find repeated constant structures and generate code targeting these structures.

Li et al. present their fast parser, Mison which can jump directly to a queried field without parsing intermediate content [12]. Mison uses SIMD instructions to

quickly identify some structural characters but otherwise works by processing bit-vectors in general purpose registers with branch-heavy loops. Mison does not attempt to validate documents; it assumes that documents are pure ASCII as opposed to unicode (UTF-8). In some instances, Mison can exceed a parsing speed of 2 GB/s on a 3.5 GHz Intel processor, which is 4–6 times faster than a conventional validating parser like RapidJSON.

Pavlopoulou et al. [20] propose a parallelized JSON processor that supports advanced queries and rewrite rules. Sparser filters quickly an unprocessed document to find mostly just the relevant information [19].

2.1 XML Parsing

Before JSON, there has been a lot of similar work done on parsing XML. Noga et al. [18] report that when fewer than 80% of the values need to be parsed, it is more economical to parse just the needed values. Marian et al. [14] propose to "project" XML documents, down to a smaller document before executing queries. Green et al. [9] show that we can parse XML quickly using a Deterministic Finite Automaton (DFA) where the states are computed lazily, during parsing. Farfán et al. [8] go further and skip entire sections of the XML document, using internal physical pointers. Takase et al. [22] accelerate XML parsing by avoiding syntactic analysis when subsets of text have been previously encountered. Kostoulas et al. designed a fast validating XML parser called Screamer: it achieves higher speed by reducing the number of distinct processing steps [10]. Cameron et al. show that we can parse XML faster using SIMD instructions [5], in their parser (called Parabix). Zhang et al. [23] show how we can parse XML documents in parallel by first indexing the document, and then separately parsing partitions of the document.

Mytkowicz et al. [16] show how to vectorize finite-state machines using SIMD instructions. They demonstrate good results with HTML tokenization, being more than twice as fast as a baseline.

3 Parser Architecture and Implementation

In our experience, most JSON parsers proceed by top-down recursive descent [6] that makes a single pass through the input bytes, doing character-by-character decoding. We adopt a different strategy, using two distinct passes. We briefly describe the two stages before covering them in detail in subsequent sections.

1. In stage 1, we validate the character encoding and identify the starting location of all JSON nodes (e.g., numbers, strings, null, true, false, arrays, objects). We also need the location of all structural characters (‘`[`’, ‘`{`’, ‘`]`’, ‘`}`’, ‘`:`’, ‘`,`’) defined in the JSON specification [4]. These locations are written as integer indexes in a separate array.
   During this stage, it is necessary to distinguish the characters that are between quotes, and thus inside a string value, from other characters. For example, the JSON document `"[1,2]"` is a single string despite the appearance of brackets. That is, these brackets should not be identified as relevant structural characters. Because quotes can be escaped (e.g., ‘`\"`’), it is necessary to identify backslash characters as well. Outside of strings, only four specific white-space characters are allowed (space, tab, line feed, carriage return). Any other white-space character needs to be identified.
   The first stage involves either SIMD processing over bytes or the manipulation of bitsets (arrays of bits) that have one bit corresponding to one byte of input. As such, it can be inefficient for some inputs—we can observe dozens of operations taking place to discover that there are in fact no odd-numbered sequences of backslashes or quotes in a given block of input. However, this inefficiency on such inputs is balanced by the fact that it costs no more to run this code over complex structured input, and the alternatives would generally involve running a number of unpredictable branches.
2. In stage 2, we process all of the nodes and structural characters. We distinguish the nodes based on their starting character. When a quote (‘`"`’) is encountered, we parse a string; when a digit or a hyphen is found, we parse a number; when the letters ‘`t`’, ‘`f`’, ‘`n`’ are found, we look for the values `true`, `false` and `null`.
   Strings in JSON cannot contain some characters unescaped, i.e., ASCII characters with code points less than 0x20, and they may contain many sorts of escaped characters. It is thus necessary to *normalize* the strings: convert them to valid UTF-8 sequences.
   Encountered numbers must be converted to either integers or floating-point values. They can take many forms (e.g., `12`, `3.1416`, `1.2e+1`). However, we must check many rules while parsing numbers. For example, the following strings are invalid numbers: `012`, `1E+`, and `.1`. We also check for overflows: we refuse to parse integers that do not fit in the 64-bit range: $[-2^{63}, 2^{63})$.
   We validate objects as sequences of strings, colons (‘`:`’) and values; we validate arrays as sequences of values separated by commas (‘`,`’). We ensure that all objects started with an open brace (‘{’) are ter-

minated with a closing brace (‘}’). We ensure that all arrays started with an open square bracket (‘[’) are terminated with a closing square bracket (‘]’). The result is written in document order on a *tape*: an array of 64-bit words. The tape contains a word for each node value (string, number, true, false, null) and a word at the beginning and at the end of each object or array. To ensure fast navigation, the words on the tape corresponding to braces or brackets are annotated so that we can go from the word at the start of an object or array to the word at the end of the array without reading the content of the array or object.

We have a secondary array where normalized string values are stored. Other parsers like RapidJSON or sajson may store the normalized strings directly in the input bytes.

At the end of the two stages, we report whether the JSON document is valid [4]. All strings are normalized and all numbers have been parsed and validated.

Our two-stage design is motivated by performance concerns. Stage 1 operates directly on the input bytes, processing the data in batches of 64 bytes. In this manner, we can make full use of the SIMD instructions that are key to our good performance. Except for unicode validation, we deliberately delay number and string validation to stage 2, as these tasks are comparatively expensive and difficult to perform unconditionally and cheaply over our entire input.

3.1 Stage 1: Structural and Pseudo-Structural Elements

The first stage of our processing must identify key points in our input: the *structural characters* of JSON (brace, bracket, colon and comma), the start and end of strings as delineated by double quote characters, other JSON *atoms* that are not distinguishable by simple characters (`true`, `false`, `null` and numbers), as well as discovering these characters and atoms in the presence of both quoting conventions and backslash escaping conventions.

In JSON, a first pass over the input can efficiently discover the significant characters that delineate syntactic elements (objects and arrays). Unfortunately, these characters may also appear between quotes, so we need to identify quotes. It is also necessary to identify the backslash character because JSON allows escaped characters: ‘\”’, ‘\\’, ‘\/’, ‘\b’, ‘\f’, ‘\n’, ‘\r’, ‘\t’, as well as escaped unicode characters (e.g. `\uDD1E`).

A point of reference is Mison [12], a fast parser in C++. Mison uses vector instructions to identify the colons, braces, quotes and backslashes. The detected quotes and backslashes are used to filter out the insignificant colons and braces. We follow the broad outline of the construction of a structural index as set forth in Mison; first, the discovery of odd-length sequences of backslash characters—which will cause quote characters immediately following to be escaped and not serve their quoting role but instead be literal characters, second, the discovery of quote pairs—which cause structural characters within the quote pairs to also be merely literal characters and have no function as structural characters, then finally the discovery of structural characters not contained within the quote pairs. We depart from the Mison paper in method and overall design. The Mison authors loop over the results of their initial SIMD identification of characters, while we propose branchless sequences to accomplish similar tasks. For example, to locate escaped quote characters, they iterate over the repeated quote characters. Their Algorithm 1 identifies the location of the quoted characters by iterating through the unescaped quote characters. We have no such loops in our stage 1: it is essentially branchless, with a fixed cost per input bytes (except for character-encoding validation, § 3.1.5). Furthermore, Mison's processing is more limited by design as it does not identify the locations of the atoms, it does not process the white-space characters and it does not validate the character encoding.

3.1.1 Identification of the quoted substrings

Identifying escaped quotes is less trivial than it appears. While it is easy to recognize that the string “\"” is made of an escaped quote since a quote character immediately preceded by a backslash, if a quote is preceded by an even number of backslashes (e.g., “\\"”), then it is not escaped since \\ is an escaped backslash. We distinguish sequences of backslash characters starting at an odd index location from sequences starting at even index location. A sequence of characters that starts at an odd (resp. even) index location and ends at an odd (resp. even) index location must have an even length, and it is therefore a sequence of escaped backslashes. Otherwise, the sequence contains an odd number of backslashes and any quote character following it must be considered escaped. We provide the code sequence with an example in Fig. 2 where two quote characters are escaped.[1]

[1] We simplify this sequence for clarity. Our results are affected by the previous iteration over the preceding 64 byte input if any. Suppose a single backslash ended the previous 64 byte input; this alters the results of the previous algorithm. We similarly elide the full details of the adjustments for previous loop state in our presentation of subsequent algorithms.

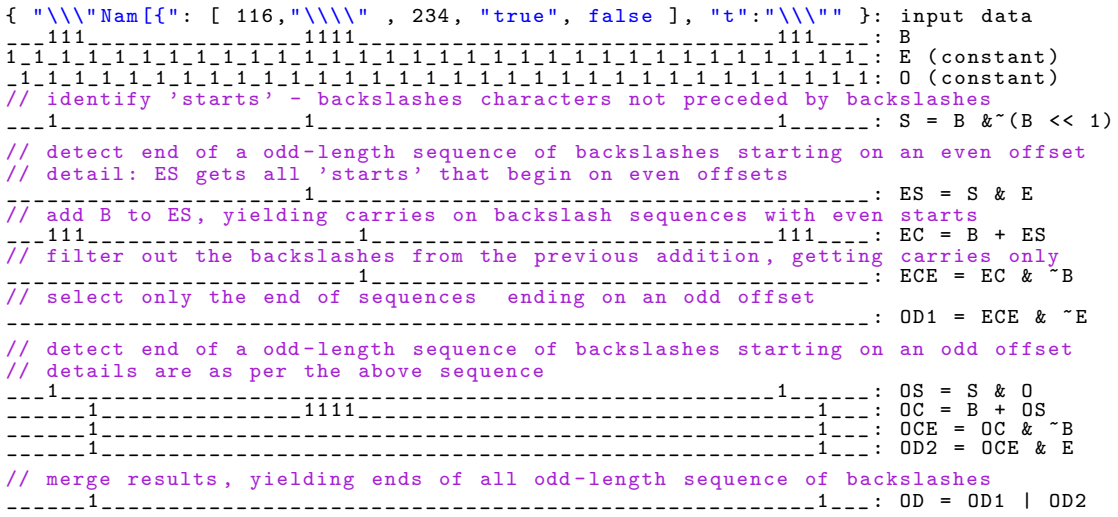


Fig. 2: Branchless code sequence to identify escaped quote characters (with example). We use the convention of the C language: '`&`' denotes the bitwise AND, '`|`' the bitwise OR, '`<<`' is a left shift, '`~`' is a bitwise negation.

With the backslash and quote characters identified, we can locate the unescaped quote characters efficiently, with a shift followed by a bitwise ANDNOT, eliminating the *escaped* quote characters. However, we are interested in finding the location between quotes (the strings), so we can find the actual structural characters. The desired bit pattern would be 1 if there are an odd-numbered number of unescaped quotes at or before our location and zero otherwise; this is the parallel prefix sum of the XOR operation over our bit vector representing unescaped quotes. This prefix sum can be implemented as one instruction by using the carry-less multiplication [11] (implemented with the `pclmulqdq` instruction) of our unescaped quote bit vector by another 64-bit word made entirely of ones, yielding the desired scope (the interior of the quoted region plus the initial quote character). E.g., if we compute the carry-less multiplication of 0b100010000 with a word containing only ones, we get 0b011110000.[2]

3.1.2 Vectorized Classification

Mison does one SIMD comparison per character ('`:`', '`\`', '`:`', '`"`', '`{`', '`}`'). We proceed similarly to identify the quotes and the backslash characters. However, there are six structural characters, and, for purposes of further analysis, we also need to discover the four permissible white-space characters. Doing ten comparisons and accompanying bitwise OR operations would be expensive. Instead of a comparison, we use the AVX2 `vpshufb` instruction to acts as a vectorized table lookup to do a *vectorized classification* [15]. The `vpshufb` instruction uses the least significant 4 bits of each byte (low nibble) as an index into a 16-byte table. Other processor architectures (ARM and POWER) have similar SIMD instructions.

By doing one lookup, followed by a 4-bit right shift and a second lookup (using a different table), we can separate the characters into one of two categories: structural characters and white-space characters. The first lookup maps the low nibbles (least significant 4 bits) of each byte to a byte value; the second lookup maps the high nibble (most significant 4 bits) of each byte to a byte value. The two byte values are combined with a bitwise AND.

To see how this can be used to identify sets of characters, suppose that we want to identify the byte values 0x9, 0xa and 0xd. The low nibbles are 9, a and d, and the high nibbles are all zeroes. In the first 16-byte lookup table, we set the fourth least significant bit to 1 for the values corresponding to indexes 9, a and d, and only for these three values. In the second 16-byte lookup table, set the fourth least significant bit to 1 for the value at index 0, and only for this value. Then we have that whenever the input values 0x9, 0xa and 0xd are encountered, and only for these values, the fourth least significant bit of the result of the bitwise AND is 1. Hence, using two `vpshufb` instructions, a shift and a few bitwise logical operations, we can identify a set of characters. If we could only identify one set of characters, this approach would not be necessarily advantageous, but we can identify many different sets with the same two `vpshufb` instructions. We can repeat the same

[2] We use the convention that 0b100010000 is the binary value with the fifth and ninth least significant bits set to 1.

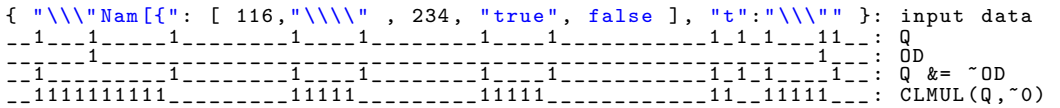


Fig. 3: Branchless code sequence to identify quoted range excluding the final quote character. CLMUL refers to the carry-less multiplication. We assume that OD is computed using the code sequence from Fig. 2.

strategy with new sets of input values, always making them match a given bit index (the fourth in our example). To avoid misclassifications, we need to ensure that each set of input values corresponding to a bit index is uniquely characterized by a set of low nibbles and high nibbles. The set {0x9, 0xa, 0xd} works since it is the set of all values with low nibbles 9, a, d, and high nibble 0. The set {0x5b,0x5d, 0x7b, 0x7d} also works since it is the set of all values with the low nibbles b and d and the high nibbles 5 and 7. The set {0x21, 0x33} would not work since 0x31 and 0x23 also share the same high nibbles and low nibbles: we would need to break the set {0x21, 0x33} into two subsets ({0x21}, {0x33}). We can support as many sets as we have bit indexes (8).

We break the set of code-point values corresponding to structural characters into three sets: {0x2c}, {0x3a}, {0x5b,0x5d, 0x7b, 0x7d}. We match them to the first three bit indexes. We break the set of code-point values corresponding to white-space characters into two sets {0x9, 0xa, 0xd}, and {0x20}. We match them to the fourth and fifth bit indexes. These sets are all uniquely characterized by their low and high nibbles. Our solution is not unique: e.g., we could have broken {0x5b,0x5d, 0x7b, 0x7d} into two sets ({0x5b,0x5d}, {0x7b, 0x7d}).

See Table 1. The table for the low nibbles is 16, 0, 0, 0, 0, 0, 0, 0, 8, 10, 4, 1, 12, 0, 0; and the table for the high nibbles is 8, 0, 17, 2, 0, 4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0. Applying our algorithm, we get the following results:

- For the comma (code-point value 0x2c), we get 1, as the bitwise AND between 17 and 1.
- For the colon (code-point value 0x3a), we get 0b10 (2 in binary).
- For the other structural characters (code-point values 0x5b,0x5d, 0x7b, 0x7d), we get 0b100 (4 in binary).
- For the first three white-space characters (with code-point values 0x9, 0xa and 0xd), we get the value 0b1000 (8 in binary).
- For the space character (code-point value 0x20), we get 0b10000 (16 in binary).
- All other character inputs will yield zero.

We can recognize the structural characters by computing a bitwise OR with 0xb111 and the white-space characters with a bitwise OR with 0b11000. That is, with only two `vpshufb` instructions and a few logical instructions, we can classify all code-point values into one of three sets: structural (comma, colon, braces, brackets), ASCII white-space (‘\r’, ‘\n’, ‘\t’, ‘ ’) and others. No branching is required.

3.1.3 Identification of White-Space and Pseudo-Structural Characters

We also make use of our ability to quickly detect white space in this early stage. We can use another bitset-based transformation to discover locations in our data that follow a structural character or quote followed by zero or more characters of white space; excluding locations within strings, and the structural characters we have already discovered, these locations are the only place that we can expect to see the starts of the JSON *atoms* (such as numbers whether or not starting with a minus sign, `null`, `true`, and `false`). These locations are thus treated as *structural* and we term them *pseudo-structural characters*. Formally, we define pseudo-structural characters as non-white-space characters that are

1. outside quotes and
2. have a predecessor that is either a white-space character or a structural character.

We use a feature of JSON: the legal atoms can all be distinguished from each other by their first character: ‘`t`’ for `true`, ‘`f`’ for `false`, ‘`n`’ for `null` and the character class `[0-9-]` for numerical values.

As a side-effect, identifying pseudo-structural characters helps validate documents. For example, only some ASCII white-space characters are allowed unescaped outside a quoted range in JSON. An isolated disallowed character would be flagged as a pseudo-structural character and subsequently rejected in stage 2. Furthermore, dangling atoms are automatically identified (as the `a` in `[12 a]`) and will be similarly rejected. The key insight is that stage 1 need not discover whether such out-of-place characters are legal JSON—it only needs to *expose* them in the stream of structural and pseudo-structural characters that will be parsed in stage 2.

Fig. 4 illustrates a final sequence in stage 1 where given the (unescaped) quotes and the quoted ranges, as

high nibble	low nibble	0	···	9	a	b	c	d	e	f
		16	···	8	10	4	1	12	0	0
0	8		···	8	8			8		
1	0		···							
2	17	16	···				1			
3	2		···		2					
4	0		···							
5	4		···			4		4		
6	0		···							
7	4		···			4		4		

code points	desired value
0x2c	1
0x3a	2
0x5b,0x5d, 0x7b, 0x7d	4
0x09, 0x0a, 0x0d	8
0x20	16
others	0

Table 1: Table describing the vectorized classification of the code points. The first column and first row are indexes corresponding to the high and low nibbles. The second column and the second row are the looked up table values. The main table values are the bitwise AND result of the two table values (e.g., 10 `AND` 8 is 8). The omitted values are zeroes. On the right, we give the desired classification.

well as the structural and white-space characters, we identify the pseudo-structural characters.

3.1.4 Index Extraction

During stage 1, we process blocks of 64 input bytes. The end product is a 64-bit bitset with the bits corresponding to a structural or pseudo-structural characters set to 1. Our structural and pseudo-structural characters are relatively rare and can sometimes, but not always, be infrequent. E.g., we can construct plausible JSON inputs that have such a character once ever 40 characters or once every 4 characters. As such, continuing to process the structural characters as bitsets involves manipulating data structures that are unpredictably spaced. We choose to transform these bitsets into indexes. That is, we seek a list of the locations of the 1-bits. Once we are done with the extraction of the indexes, we can discard the bitset. In contrast, Mison does not have such an extraction step and iterates directly over the 1-bits.

Our implementation involves a transformation of bitsets to indexes by use of the *count trailing zeroes* operation (via the `tzcnt` instruction) and an operation to clear the lowest set bit (via the `blsr` instruction). This strategy introduces an unpredictable branch; unless there is a regular pattern in our bitsets, we would expect to have at least one branch miss for each word. However, we employ a technique whereby we extract 8 indexes from our bitset unconditionally, then ignore any indexes that were extracted excessively by means of overwriting those indexes with the next iteration of the index extraction loop. See Fig. 5. This means that as long as the frequency of our set bits is below 8 bits out of 64 we expect few unpredictable branches. The choice of the number 8 is a heuristic based on our experience with JSON documents; a larger unconditional extraction procedure would be more expensive due to having to use more operations, but even less likely to cause a branch miss as a wider range of bit densities could be handled by extracting, say, 8 indexes from our bitset.

3.1.5 Character-Encoding Validation

In our experience, JSON documents are served using the unicode format UTF-8: we could not find a single instance of JSON document published using another character encoding. Indeed, the JSON specification indicates that many implementation do not support encodings other than UTF-8. Parsers like Mison assume that the character encoding is ASCII [12]. Though it is reasonable, a safer assumption is that unicode (UTF-8) is used. Not all sequences of bytes are valid UTF-8 and thus a validating parser needs to ensure that the character encoding is correct. We assume that the incoming data is meant to follow UTF-8, and that the parser should produce UTF-8 strings.

UTF-8 is an ASCII superset. The ASCII characters can be represented using a single byte, as a numerical value called *code point* between 0 and 127 inclusively. That is, ASCII code points are an 8-bit integer with the most significant bit set to zero. UTF-8 extends these 128 code points to a total of 1,114,112 code points. Non-ASCII code points are represented using from two to four bytes, each with the most significant bit set to one. Non-ASCII code points cannot contain ASCII characters: we can therefore remove from an UTF-8 stream of bytes any number of ASCII characters without affecting its validation.

Outside of strings in JSON, all characters must be ASCII. Only the strings require potentially expensive validation. However, there may be many small strings in a document, so it is unclear whether vectorized unicode

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
__1_________1________1____1________1____1___________1_1_1____1__: Q
__1111111111________11111________11111___________11__11111___: R
1_________11_1_1____1_______1____1_______1_______11____1_______1: S
_1___________1_1_________1_1____1_______1_____1__1_________1_: W
// eliminate quoted regions from our structural characters
1___________1_1____1_______1____1_______1_______11____1_______1: S = S&~R
// restore ending quotes to our structural characters
// (for purposes of building pseudo-structural characters)
1_1_________11_1____11____1_1____1_1____11_______11_1_111____1_1: S = S|Q
// begin to calculate pseudo-structural characters
// initially; pseudo-structural characters are structural or white space
111_________11111___11____1111___111____111_____11111_111____111: P = S|W
// now move our mask for candidate pseudo-structural characters forward by one
1111_________11111___11____1111___111____111_____11111_111____11: P = P<<1
// eliminate white-space and quoted characters from our candidates
1___________1_1_1_________1_1_________1_1_____11____1_______1: P &= ~W&~R
// merge pseudo-structural characters into structural character mask
1_1_________11_1_1__11____1_1_1__1_1____11_1_____11_1_111____1_1: S = S|P
// eliminate ending quotes from our final structural characters
1_1_________1_1_1__11______1_1__1_1_____1_1_____11_1__11______1: S&~(Q&~R)
```

Fig. 4: Branchless code sequence to identify the structural and pseudo-structural characters. The quotes Q and the quoted range R are computed using the code sequence from Fig. 3. The structural (S) and white-space (W) characters are identified vectorized classification.

```
// we decode the set bits from 's'
// to array 'b'
uint64_t s = ...
uint32_t * b = ...
// net line => popcnt instruction
uint32_t cnt = popcount(s);
uint32_t next_base = b + cnt;
while (s) {
  // next line => tzcnt instruction
  *b++ = idx + trailingzeroes(s);
  // next line => blsr instruction
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
  *b++ = idx + trailingzeroes(s);
  s = s & (s - 1);
}
b = new_base;
```

Fig. 5: Code sequence to extract set bits out of a bitset

validation would be beneficial at the individual string level. Thus we validate the input bytes as a whole.

We first test if a block of 64 bytes is made entirely of ASCII characters. It suffices to verify that the most significant bit of all bytes is zero. This optimization might trigger some unpredictable branches, but given how frequently JSON documents might be almost entirely composed of ASCII characters, it is a necessary risk.

If there are non-ASCII characters, we apply a vectorized UTF-8 validation algorithm. We need to check that sequences of bytes are made of valid UTF-8 code points (see Tables 2). It involves several steps, but each one is efficient. We work exclusively with SIMD instructions.

- We need to verify that all byte values are no larger than 0xF4 (or 244): we can achieve this check with an 8-bit *saturated* subtraction with 0xF4. The result of the subtraction is zero if and only if the value is no larger than 0xF4.
- When the byte value 0xED is found, the next byte must be no larger than 0x9F; when the byte value 0xF4 is found, the next byte must be no larger than 0x8F. We can check these conditions with vectorized byte comparisons and byte shifts.
- The byte values 0xC0 and 0xC1 are forbidden. When the byte value is 0xE0, the next byte value is larger than 0xA0. When the byte value is 0xF0, the next byte value is at least 0x90.
- When a byte value is outside the range of ASCII values, it belongs to one out of four classes, depending on the value of its high nibble:
  - If the high nibble is 8, 9, a or b (in hexadecimal) then the byte is the second, third of fourth byte in a code point.
  - If the high nibble is c or d then the byte must be the first of a sequence of two bytes forming a code point.
  - If the high nibble is e then the byte is the first out of a code point made of three bytes.

– Finally, if the high nibble is f, then the byte is first in a sequence of four bytes.

We use the `vpshufb` instruction to quickly map bytes to one of these categories using values 0, 2, 3, and 4. We map ASCII characters to the value 1. If the value 4 is found (corresponding to a nibble value of f), it should be followed by three values 0. Given such a vector of integers, we can check that it matches a valid sequence of code points in the following manner. Shift values by 1 and subtract 1 using saturated subtraction, add the result to the original vector. Repeat the same process with a factor of two: shift values by 2 and subtract 2, add the result to the original vector. Starting with the sequence 4 0 0 0 2 0 1 1 3 0 0, you first get 4 3 0 0 2 1 1 1 3 2 0 and then 4 3 2 1 2 1 1 1 3 2 1. If the sequence came from valid UTF-8, all final values should be greater than zero, and be no larger than the original vector.

All these checks are done using SIMD registers solely, without branching. At the beginning of the processing, we initialize an *error variable* (as a 32-byte vector) with zeroes. We compute in-place the bitwise OR of the result of each check with our error variable. Should any check fail, the error variable will become non-zero. We only check at the end of the processing (once) that the variable is zero. If a diagnosis is required to determine where the error occurs, we can do a second pass over the input.

```
0  : r  // pointing to 38
1  : {  // pointing to next tape
     location 38
2  : string "Image"
3  : {  // pointing to next tape
     location 37
4  : string "Width"
5  : integer 800
7  : string "Height"
8  : integer 600
10 : string "Title"
11 : string "View from 15th Floor"
12 : string "Thumbnail"
13 : {  // pointing to next tape
     location 23
14 : string "Url"
15 : string "http://www.example.com/
     image/481989943"
16 : string "Height"
17 : integer 125
19 : string "Width"
20 : integer 100
22 : }  // pointing to previous tape
     location 13
23 : string "Animated"
24 : false
25 : string "IDs"
26 : [  // pointing to next tape
     location 36
27 : integer 116
29 : integer 943
31 : integer 234
33 : integer 38793
35 : ]  // pointing to previous tape
     location 26
36 : }  // pointing to previous tape
     location 3
37 : }  // pointing to previous tape
     location 1
38 : r  // pointing to 0 (start root)
```

Fig. 6: JSON tape corresponding to the example in Fig. 1

3.2 Stage 2: Building the Tape

In the final stage, we iterate through the indexes found in the first stage. To handle objects and arrays that can be nested, we use a goto-based state machine. Our state is recorded as a stack indicating whether we are in an array or an object, we append our new state to the stack whenever we encounter an embedded array or object. When the embedded object or array terminates, we use the stored state from the stack and a goto command to resume the parsing from the appropriate state in the containing scope. Values such as `true`, `false`, `null` are handled as simple string comparisons. We parse numbers and strings using dedicated functions. See Fig. 6 for an example of the resulting tape. Without much effort, we could support streaming processing without materializing JSON documents objects as in-memory tapes [13].

3.2.1 Number Parsing

It is difficult to do number parsing without proceeding in a standard character-by-character manner. Thus we proceed in such a manner as do most parsers. However, we found it useful to test for the common case where there are at least eight digits as part of the fractional portion of the number. Given the eight characters interpreted as a 64-bit integer `val`, we can check whether it is made of eight digits with an inexpensive comparison:

```
(((val & 0xF0F0F0F0F0F0F0F0)
| (((val + 0x0606060606060606)
      & 0xF0F0F0F0F0F0F0F0) >> 4))
      == 0x3333333333333333).
```

When this check is successful, we can invoke a fast vectorized function to compute the equivalent integer value (see Fig. 7).

3.2.2 String Validation and Normalization

When encountering a quote character, we always read 32 bytes in a vector register, then look for the quote and the escape characters. If an escape character is found before the first quote character, we use a conventional code path to process the escaped character, otherwise we just write the 32-byte register to our string buffer.

code points	1st byte	2nd byte	3rd byte	4th byte
0x000000...0x00007F	00...7F			
0x000080...0x0007FF	C2...DF	80...BF		
0x000800...0x000FFF	E0	A0...BF	80...BF	
0x001000...0x00CFFF	E1...EC	80...BF	80...BF	
0x00D000...0x00D7FF	ED	80...9F	80...BF	
0x00E000...0x00FFFF	EE...EF	80...BF	80...BF	
0x010000...0x03FFFF	F0	90...BF	80...BF	80...BF
0x040000...0x0FFFFF	F1...F3	80...BF	80...BF	80...BF
0x100000...0x10FFFF	F4	80...8F	80...BF	80...BF

Table 2: UTF-8 code-points and their representation into sequences of up to four bytes.

```
uint32_t parse_eight_digits_unrolled(char *
    chars) {
  __m128i ascii0 = _mm_set1_epi8('0');
  __m128i mul_1_10 =
      _mm_setr_epi8(10, 1, 10, 1, 10, 1, 10,
           1, 10, 1, 10, 1, 10, 1, 10, 1);
  __m128i mul_1_100 = _mm_setr_epi16(100, 1,
       100, 1, 100, 1, 100, 1);
  __m128i mul_1_10000 =
      _mm_setr_epi16(10000, 1, 10000, 1,
          10000, 1, 10000, 1);
  __m128i in = _mm_sub_epi8(_mm_loadu_si128
      ((__m128i *)chars), ascii0);
  __m128i t1 = _mm_maddubs_epi16(in,
      mul_1_10);
  __m128i t2 = _mm_madd_epi16(t1, mul_1_100)
      ;
  __m128i t3 = _mm_packus_epi32(t2, t2);
  __m128i t4 = _mm_madd_epi16(t3,
      mul_1_10000);
  return _mm_cvtsi128_si32(t4);
}
```

Fig. 7: Code sequence using Intel intrinsics to convert eight digits to their integer value.

Our string buffer is made of null-terminated strings, so we add a null character where the terminating quote would be. Strictly speaking the JSON specification allows string characters containing null characters, but we do not know of any application that would require null characters inside strings. As part of the string validation, we must check that no code-point value less than 0x20 is found: we use vectorized comparison.

4 Experiments

We validate our results through a set of reproducible experiments over varied data.[3] In § 4.3, we report that the running time during parsing is split evenly between our two stages. In § 4.4, we show that we use half as many instructions during parsing as our best competitor. In § 4.5, we show that this reduced instruction count translates into a comparable runtime advantage.

4.1 Hardware and Software

Most recent Intel processors are based on the Skylake microarchitecture. We also include a computer with the more recent Cannonlake microarchitecture in our tests. We summarize the characteristics of our hardware platforms in Table 3.

Our software was written using C++17. We tested it under several recent compilers (LLVM's clang, GNU GCC, Microsoft Visual Studio 2017). For our testing, we use GNU GCC 7 to compile all software under Linux using the `-O3` flag. We compile the code as is, without profile-guided optimization. All code is single-threaded. We disable hyper-threading.

Our experiments assume that the JSON document is in memory; we omit disk and network accesses. In

[3] Scripts and code is available online: `https://github.com/lemire/simdjson`.

Table 3: Hardware

Processor	Frequency	Microarchitecture	Memory	Compiler
Intel i7-6700	3.4 GHz	Skylake (x64, 2015)	DDR4 (2133 MT/s)	GCC 7
Intel i3-8121U	2.2 GHz	Cannonlake (x64, 2018)	LPDDR4 (3200 MT/s)	GCC 7

practice, JSON documents are frequently ingested from the network. Yet current networking standards allow for speeds exceeding 10 GB/s [7] and modern networking hardware can allow network data to be read directly into a cache line, so a high performance implementation of JSON scanning is desirable even for data coming from the network. While we focus on speed, we also expect that more efficient parsers reduce energy consumption.

After reviewing several parsers, we selected RapidJSON and sajson, two open-source C++ parsers, as references (see Table 4). Palkar et al. describe RapidJSON as *the fastest traditional state-machine-based parser available* [19]. In practice, we find that another C++ parser, sajson, is faster. They are both mature and highly optimized: they were created in 2011 and 2012 respectively. The sajson parser can be used with either static or dynamic memory allocations: the static version is faster, so we adopt it.

Counting our own parser (simdjson), all three parsers can parse 64-bit floating-point numbers as well as integers. However, sajson only supports 32-bit integers whereas both RapidJSON and simdjson support 64-bit integers. RapidJSON represents overly large integers as 64-bit floating-point numbers (in a lossy manner) whereas both our parser (simdjson) and sajson reject documents with integers that they cannot exactly represent.

RapidJSON can either normalize strings in a new buffer or within the input bytes (*insitu*). We find that the parsing speed is greater in *insitu* mode, so we present these better numbers. In contrast, sajson only supports *insitu* parsing. Our own parser does not modify the input bytes: it has no *insitu* mode. All three parsers do UTF-8 validation of the input.

We consider other open-source parsers but we find that they are either slower than RapidJSON, or that they failed to abide by the JSON specification (see § 4.5). For example, parsers like gason, jsmn and ultrajson accept `[0e+]` as valid JSON. Parsers like fastjson and ultrajson accept unescaped line breaks in strings. Other parsers are tightly integrated into larger frameworks, making it difficult to benchmark them fairly. For methodological simplicity, we also do not consider parsers written in Java or other languages.

RapidJSON has compile-time options to enable optimized code paths making use of SIMD optimizations: these optimizations skip spaces between values or structural characters. However, we found both of these compile-time macros (RAPIDJSON_SSE2 and RAPIDJSON_SSE42) to be systematically detrimental to performance in our tests. Moreover, they are disabled by default in the library. Thus we do not make use of these optimizations.

Other than RapidJSON, we find that none of the libraries under consideration make deliberate use of SIMD instructions. However, we expect that all libraries benefit of SIMD instructions in our tests: many functions from the standard libraries are vectorized, and the compiler translates some conventional code to SIMD instructions (e.g., via autovectorization [17]).

We cannot directly compare with Mison since their software is not available publicly [12]. However, the authors of Mison reports speeds up to slightly over 2 GB/s on a 3.5 GHz Intel Xeon Broadwell-EP (E5-1620 v3): e.g., while parsing partially Twitter data. We know that Mison does not attempt to validate the documents nor to parse them entirely.

4.2 Datasets

Parsing speed is necessarily dependent on the content of the JSON document. For a fair assessment, we chose a wide range of documents. See Table 5 for detailed statistics concerning the chosen files. In Table 6, we present the number of bytes of both the original document and the version without extraneous white-space characters outside strings.

From the author of RapidJSON[4], we acquired three data files. We have canada.json which is a description of the Canadian contour in GeoJSON: it contains many numbers. We have citm_catalog.json which is commonly used benchmark file. Finally, we have twitter.json which is the result of a search for the character one in Japanese and Chinese using the Twitter API: it contains many non-ASCII characters. From the author of sajson[5], we retrieved several more files: apache_builds.json, github_-

[4] `https://github.com/miloyip/nativejson-benchmark`

[5] `https://github.com/chadaustin/sajson/tree/master/testdata`

Table 4: Competitive parsers

Processor	snapshot	link
simdjson	January 5th 2019	`https://github.com/lemire/simdjson`
RapidJSON	version 1.1.0	`https://github.com/Tencent/rapidjson`
sajson	September 20th 2018	`https://github.com/chadaustin/sajson`

events.json, instruments.json, mesh.json, mesh.pretty.json, update-center.json.

We also generated number.json as a large array of random floating-point numbers. We also created twitterescaped.json which is a minified version of the twitter.json where all non-ASCII characters have been escaped.

Many of these documents require much number parsing or much string normalization. We deliberately did not consider tiny documents (smaller than 1 kB). The task of parsing many tiny documents is outside our scope.

4.3 Running Time Distribution

Most of the vector processing (with SIMD instructions) occurs during stage 1. In Fig. 8, we present the distribution of cycles per stage, for each test file. About half the CPU cycles per input byte (between 0.5 and 3 cycles) are spent in stage 1. Thus at least half of the processing time is directly related to SIMD instructions and branchless processing.

Roughly a third of the CPU cycles are spent parsing numbers in the files canada, marine_jk, mesh, mesh.pretty and numbers. In other files, the time spent parsing numbers is negligible.

The string parsing time is a sizeable cost in the twitterescaped file. In this file, all non-ASCII characters have been escaped which makes string normalization more difficult.

In the random file, UTF-8 validation is a significant cost, but a relatively small cost in all other instances. This file has a relatively high fraction of non-ASCII characters (20%). In comparison, the twitter file has only 3% of non-ASCII characters.

The running time of the parser depends on the characteristics of the JSON document. It can be useful to model the performance: e.g., an engineer could predict the execution time and budget accordingly. For this purpose, we used linear regression based on our dataset of files. Our dataset is relatively small, but we expect that it is large enough for a simple linear regression.

- Let F the number of floating-point numbers,
- S be the number of structural and semi-structural elements and
- B the number of bytes.

With a high accuracy ($R^2 \geq 0.99$), we have the following cost models:

- The stage 1 running time (in CPU cycles) is $1.8 \times S + 0.62 \times B$ on Skylake and $1.9 \times S + 0.63 \times B$ on Cannonlake.
- The stage 2 running time is $19 \times F + 9.5 \times S + 0.33 \times B$ on Skylake and $19 \times F + 9 \times S + 0.36 \times B$ on Cannonlake.
- The total running time is $19 \times F + 11 \times S + 0.95 \times B$ on Skylake and $19 \times F + 11 \times S + 0.98 \times B$ on Cannonlake.

The number of input bytes as a small coefficient (less than 1 cycle per input byte) but its contribution to the cost is still significant because there are many more bytes than structural elements or floating-point numbers.

In our model, a floating-point number not only has a direct cost, but also generates one pseudo-structural character and comprises several bytes: thus each number costs dozens of cycles.

4.4 Fewer Instructions

The main benefit of SIMD instructions is to do more work with fewer instructions. Thus we expect our parser to user fewer instructions. We present in Table 7 the number of instructions needed to parse various file using our three competitive parsers.

On average, simdjson uses half as many instructions as sajson and four times fewer than RapidJSON. Our fastest competitor, sajson, uses between 1.7 and 3.3 more instructions. The files where our advantage over sajson is relatively smallest are marine_ik mesh.pretty and twitterescaped. These files involve either much number parsing, or expensive string normalization.

4.5 Speed Comparison

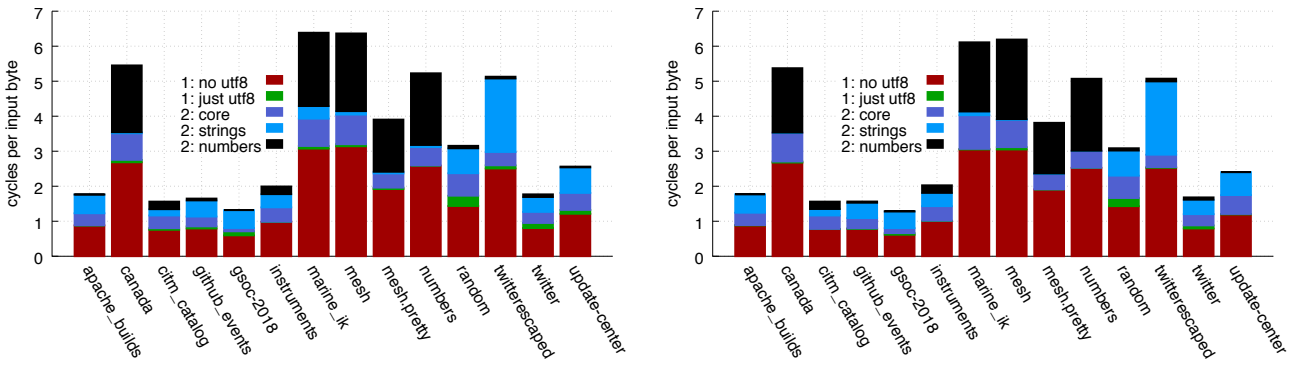
We present raw parsing speeds in Fig. 9. In only a few instances, sajson can slightly surpass 1 GB/s and RapidJSON can slightly surpass 0.5 GB/s. On our Skylake (3.4 GHz) processor, our parser (simdjson) can achieve

Table 5: Datasets statistics. The last column (struct.) is the number of structural and pseudo-structural characters.

file	integer	float	string	non-ascii	object	array	null	true	false	struct.
apache_builds	2	0	5289	0	884	3	0	2	1	12365
canada	46	111080	12	0	4	56045	0	0	0	334374
citm_catalog	14392	0	26604	348	10937	10451	1263	0	0	135991
github_events	149	0	1891	4	180	19	24	57	7	4657
gsoc-2018	0	0	34128	0	3793	0	0	0	0	75842
instruments	4935	0	6889	0	1012	194	431	17	109	27174
marine_ik	130225	114950	38268	0	9680	28377	0	6	0	643013
mesh	40613	32400	11	0	3	3610	0	0	0	153275
mesh.pretty	40613	32400	11	0	3	3610	0	0	0	153275
numbers	0	10001	0	0	0	1	0	0	0	20004
random	5002	0	33005	103482	4001	1001	0	495	505	88018
twitterescaped	2108	1	18099	0	1264	1050	1946	345	2446	55264
twitter	2108	1	18099	95406	1264	1050	1946	345	2446	55264
update-center	0	0	27229	49	1896	1937	0	134	252	63420

Table 6: Datasets sizes: minified size omits white-space characters outside quotes.

file	bytes (minified)	bytes (original)	ratio
apache_builds	94653	127275	74%
canada	2251027	2251027	100%
citm_catalog	500299	1727204	29%
github_events	53329	65132	82%
gsoc-2018	3073766	3327831	92%
instruments	108313	220346	49%
marine_ik	1834197	2983466	61%
mesh	650573	723597	90%
mesh.pretty	753399	1577353	48%
numbers	150121	150124	100%
random	461466	510476	90%
twitterescaped	562408	562408	100%
twitter	466906	631514	74%
update-center	533177	533178	100%



(a) Skylake

(b) Cannonlake

Fig. 8: Time required in cycles per input byte to process our test files, timings are decomposed in the time needed to execute various components of the parsing.

Table 7: Instructions per byte required to parse and validate documents (Skylake).

file	simdjson	RapidJSON	sajson	RapidJSON/ simdjson	sajson/ simdjson
apache_builds	5.3	28.2	9.5	5.3	1.8
canada	12.4	29.2	22.0	2.4	1.8
citm_catalog	5.0	15.1	10.4	3.0	2.0
github_events	4.7	28.7	9.5	6.1	2.0
gsoc-2018	3.2	29.7	10.4	9.3	3.3
instruments	5.9	22.6	12.3	3.8	2.1
marine_ik	12.7	27.4	21.3	2.2	1.7
mesh	13.5	30.2	24.4	2.2	1.8
mesh.pretty	8.7	17.9	15.0	2.1	1.7
numbers	10.9	27.3	20.8	2.5	1.9
random	8.4	33.8	15.4	4.0	1.8
twitterescaped	8.7	29.6	14.4	3.4	1.7
twitter	5.2	24.8	11.0	4.8	2.1
update-center	5.9	35.2	11.5	6.0	1.9
average	7.9	27.1	14.9		
geometric mean				3.7	2.0

Table 8: Time required in cycles per input byte to parse and then select all distinct user.id from the parsed tree, using the file twitter.

(a) Skylake

parser	cycles/byte
simdjson	2.4
RapidJSON	10.0
sajson	4.8

(b) Cannonlake

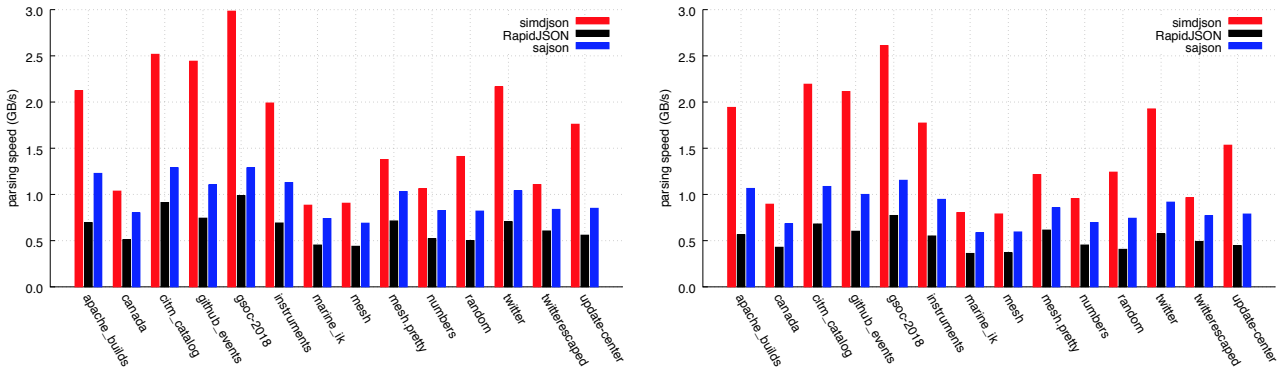
parser	cycles/byte
simdjson	3.6
RapidJSON	7.4
sajson	6.4

and even surpass 2 GB/s in five instances, and for gsoc-2018, we reach 3 GB/s.

The purpose of parsing is to access the data contained in the document. It would not be helpful to quickly parse documents if we could not, later on, access the parsed tree quickly. In Table 8, we present our results while parsing the twitter document and finding all unique user.id (SELECT DISTINCT "user.id" FROM tweets), a query from Tahara et al. [21]. We report the time in cycles per byte to fully parse and scan the parsed tree. Our parser is again twice as fast as the reference parsers.

In Table 9, we present the parsing speed in gigabytes per second (GB/s) for several different parsers. In particular, we present results regarding RapidJSON using both the default configuration and the faster version that we use elsewhere (with insitu processing). In several cases, insitu processing is faster (apache_build, gsoc-2018, twitter, etc.) up to a factor of two, while in other cases, such as all files made mostly of numbers, the difference is negligible. Compared with RapidJSON without insitu string processing, our parser (simdjson) can be more than five times faster. RapidJSON further allows us to disable character encoding validation or to use higher precision number parsing, we do not report the results for these cases. For sajson, we use both the default dynamic-memory allocation and the faster version with static-memory allocation which we use elsewhere. The dynamic-memory allocation leads to a significant performance penalty, but we expect that it makes the parser more conservative in its memory usage. For reference, we also include several other popular C/C++ parsers even though we found them all to be lacking regarding their validation: the Dropbox parser[6], fastjson[7], gason[8], ujson4c: a wrapper around the UltraJSON library[9], jsmn[10], cJSON[11], and jsoncpp[12]. In all cases, we used the latest available version and we tried to benchmark to get the best speed. Out of these other parsers, the most competitive regarding speed is gason, as it is close to the best performance of sajson.

[6] https://github.com/dropbox/json11
[7] https://github.com/mikeando/fastjson
[8] https://github.com/vivkin/gason
[9] https://github.com/esnme/ujson4c
[10] https://github.com/zserge/jsmn
[11] https://github.com/DaveGamble/cJSON
[12] https://github.com/open-source-parsers/jsoncpp



(a) Skylake (3.4 GHz)

(b) Cannonlake (2.2 GHz)

Fig. 9: Speed of the three parsers (simdjson, RapidJSON and sajson) while parsing our different files (in GB/s).

Table 9: Parsing speed in gigabytes per second (GB/s) for several different parsers (Skylake)

	simdjson	RapidJSON	RapidJSON ins.	sajson dyn.	sajson	Dropbox	fastjson	gason	ultrajson	jsmn	cJSON	jsoncpp
apache_builds	2.2	0.52	0.69	0.83	1.2	0.17	0.27	0.93	0.41	0.07	0.30	0.14
canada	1.0	0.51	0.50	0.59	0.80	0.07	0.20	0.91	0.47	0.01	0.08	0.04
citm_catalog	2.5	0.83	0.88	0.85	1.1	0.24	0.37	1.2	0.71	0.21	0.38	0.18
github_events	2.4	0.51	0.75	0.96	1.1	0.15	0.26	0.93	0.38	0.57	0.28	0.13
gsoc-2018	3.0	0.55	1.0	1.1	1.3	0.23	0.31	1.1	0.44	0.17	0.54	0.26
instruments	2.0	0.61	0.69	0.67	0.99	0.14	0.31	0.99	0.45	0.26	0.26	0.12
marine_ik	0.92	0.45	0.46	0.51	0.68	0.07	0.20	0.76	0.40	0.18	0.08	0.04
mesh	0.92	0.45	0.44	0.51	0.69	0.09	0.18	0.72	0.40	0.05	0.07	0.03
mesh.pretty	1.4	0.73	0.72	0.67	1.00	0.16	0.31	1.0	0.71	0.10	0.14	0.07
numbers	1.1	0.53	0.53	0.57	0.83	0.09	0.22	0.83	0.48	0.56	0.08	0.03
random	1.4	0.40	0.50	0.44	0.82	0.10	0.22	0.82	0.29	0.03	0.19	0.08
twitter	2.2	0.51	0.71	0.70	0.97	0.14	0.26	0.85	0.42	0.28	0.34	0.13
twitterescaped	1.1	0.42	0.59	0.60	0.86	0.11	0.24	0.72	0.36	0.26	0.27	0.11
update-center	1.7	0.40	0.55	0.48	0.84	0.11	0.20	0.72	0.31	0.06	0.25	0.10

5 Conclusion and Future Work

Though the application of SIMD instructions for parsing is not novel [5], our results suggest that they are underutilized in popular JSON parsers. We expect that many of our strategies could benefit existing JSON parsers like RapidJSON. It may even be possible to integrate the code of our parser (simdjson) directly into existing libraries.

JSON is one of several popular data formats such as Protocol Buffers, XML, YAML, MessagePack, BSON, CSV, or CBOR. We expect that many of our ideas would apply to other formats.

JSON documents are all text. Yet we frequently need to embed binary content inside such documents. The standard approach involves using base64 encoding. Base64 data can be decoded quickly using SIMD instructions [15]. Because number parsing from text is expensive, it might be fruitful to store large arrays of numbers in binary format using base64.

Intel has produced a new family of instruction sets with wider vector registers and more powerful instructions (AVX-512). Our Cannonlake processor supports these instructions, including the AVX512-VBMI extension, which is relevant to the byte processing required for this work. Future research should assess the benefits of AVX-512 instructions.

Many of our strategies are agnostic to the specific architecture of the processor. Future research should try to replicate our performance improvements with other processor, such as those of the ARM or POWER families.

Acknowledgements The vectorized UTF-8 validation was motivated by a blog post by O. Goffart. K. Willets helped design the current vectorized UTF-8 validation. In particular, he provided the algorithm and code to check that sequences of two, three and four non-ASCII bytes match the leading byte. The authors are grateful to W. Muła for sharing related number-parsing code online.

The work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant RGPIN-2017-03910.

References

1. Alagiannis I, Borovica R, Branco M, Idreos S, Ailamaki A (2012) NoDB in Action: Adaptive Query Processing on Raw Data. Proc VLDB Endow 5(12):1942–1945, DOI 10.14778/2367502.2367543
2. Boncz PA, Graefe G, He B, Sattler KU (2019) Database architectures for modern hardware. Tech. Rep. 18251, Dagstuhl Seminar
3. Bonetta D, Brantner M (2017) FAD.Js: Fast JSON Data Access Using JIT-based Speculative Optimizations. Proc VLDB Endow 10(12):1778–1789, DOI 10.14778/3137765.3137782
4. Bray T (2017) The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc8259`, internet Engineering Task Force, Request for Comments: 8259
5. Cameron RD, Herdy KS, Lin D (2008) High Performance XML Parsing Using Parallel Bit Stream Technology. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, ACM, New York, NY, USA, CASCON '08, pp 17:222–17:235, DOI 10.1145/1463788.1463811
6. Cohen J, Roth MS (1978) Analyses of deterministic parsing algorithms. Commun ACM 21(6):448–458, DOI 10.1145/359511.359517
7. Cole CR (2011) 100-Gb/s and beyond transceiver technologies. Optical Fiber Technology 17(5):472–479
8. Farfán F, Hristidis V, Rangaswami R (2007) Beyond Lazy XML Parsing. In: Proceedings of the 18th International Conference on Database and Expert Systems Applications, Springer-Verlag, Berlin, Heidelberg, DEXA'07, pp 75–86
9. Green TJ, Gupta A, Miklau G, Onizuka M, Suciu D (2004) Processing XML Streams with Deterministic Automata and Stream Indexes. ACM Trans Database Syst 29(4):752–788, DOI 10.1145/1042046.1042051
10. Kostoulas MG, Matsa M, Mendelsohn N, Perkins E, Heifets A, Mercaldi M (2006) XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization. In: Proceedings of the 15th International Conference on World Wide Web, ACM, New York, NY, USA, WWW '06, pp 93–102, DOI 10.1145/1135777.1135796
11. Lemire D, Kaser O (2016) Faster 64-bit universal hashing using carry-less multiplications. Journal of Cryptographic Engineering 6(3):171–185, DOI 10.1007/s13389-015-0110-5
12. Li Y, Katsipoulakis NR, Chandramouli B, Goldstein J, Kossmann D (2017) Mison: A Fast JSON Parser for Data Analytics. Proc VLDB Endow 10(10):1118–1129, DOI 10.14778/3115404.3115416
13. Liu ZH, Hammerschmidt B, McMahon D (2014) Json data management: Supporting schema-less development in rdbms. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '14, pp 1247–1258, DOI 10.1145/2588555.2595628
14. Marian A, Siméon J (2003) Projecting XML Documents. In: Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB Endowment, VLDB '03, pp 213–224
15. Muła W, Lemire D (2018) Faster Base64 Encoding and Decoding Using AVX2 Instructions. ACM Trans Web 12(3):20:1–20:26, DOI 10.1145/3132709
16. Mytkowicz T, Musuvathi M, Schulte W (2014) Data-parallel finite-state machines. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS '14, pp 529–542, DOI 10.1145/2541940.2541988
17. Naishlos D (2004) Autovectorization in GCC. In: Proceedings of the 2004 GCC Developers Summit, pp 105–118
18. Noga ML, Schott S, Löwe W (2002) Lazy XML Processing. In: Proceedings of the 2002 ACM Symposium on Document Engineering, ACM, New York, NY, USA, DocEng '02, pp 88–94, DOI 10.1145/585058.585075
19. Palkar S, Abuzaid F, Bailis P, Zaharia M (2018) Filter before you parse: faster analytics on raw data with Sparser. Proceedings of the VLDB Endowment 11(11):1576–1589
20. Pavlopoulou C, Carman Jr EP, Westmann T, Carey MJ, Tsotras VJ (2018) A Parallel and Scalable Processor for JSON Data. In: EDBT'18
21. Tahara D, Diamond T, Abadi DJ (2014) Sinew: A SQL System for Multi-structured Data. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, New

York, NY, USA, SIGMOD '14, pp 815–826, DOI 10.1145/2588555.2612183

22. Takase T, Miyashita H, Suzumura T, Tatsubori M (2005) An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization. In: Proceedings of the 14th International Conference on World Wide Web, ACM, New York, NY, USA, WWW '05, pp 692–701, DOI 10.1145/1060745.1060845
23. Zhang Y, Pan Y, Chiu K (2009) Speculative p-DFAs for parallel XML parsing. In: High Performance Computing (HiPC), 2009 International Conference on, IEEE, pp 388–397