arXiv:2010.03090v4 [cs.DB] 15 Dec 2021

RESEARCH ARTICLE

Validating UTF-8 In Less Than One Instruction Per Byte

John Keiser[1*] | Daniel Lemire[2*]

[1]Microsoft, Redmond, WA, 98052, USA

[2]DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: lemire@gmail.com

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910

The majority of text is stored in UTF-8, which must be validated on ingestion. We present the `lookup` algorithm, which outperforms UTF-8 validation routines used in many libraries and languages by more than 10 times using commonly available SIMD instructions. To ensure reproducibility, our work is freely available as open source software.

KEYWORDS
Vectorization, Unicode, Text Processing, Character Encoding

1 | INTRODUCTION

Unicode is the ubiquitous standard for text representation in software. It assigns a *code point* (a number from 0 to 1 114 112) to almost every character in every language, as well as formatting and symbols like whitespace characters and emojis. UTF-8, in turn, is the dominant format used to *encode* Unicode text—to store or send it in a series of bytes via memory, disk or network [1]. For example, UTF-8 is in widespread use in XML and JSON documents, as well as in database systems like MySQL. Even more fundamentally, many recently introduced programming languages represent strings as UTF-8 by default (e.g., Rust, Go) while established languages have migrated to UTF-8 (Swift, Ruby). UTF-8 is more concise than other alternative Unicode formats such as UTF-16 and UTF-32.

All of these systems have to *validate* UTF-8 on ingestion. Invalid UTF-8 strings can cause various functions such as search or sort to fail; they may cause display problems in applications or web sites. More critically, invalid UTF-8 is a security risk [2]: e.g., Microsoft's web server (IIS) failed to validate the UTF-8 string used as URI which allowed attackers to access otherwise forbidden paths. Whenever a database system or software program receives bytes that are meant to be UTF-8, they run a validation function.

Validation is not a straightforward problem. UTF-8 uses between 1 and 4 bytes to encode each character, and there are many distinct error cases to check. In our experience, most systems validate UTF-8 using relatively complicated sequences of branches. The speed of a branch-based approach depends on the input. We can exceed speeds of 2 GiB s^{-1}, going as fast as 4 GiB s^{-1} on ASCII content. Though such speeds may seem satisfactory, recent disks can sustain higher throughput (e.g., 5 GiB s^{-1}) with networking speeds being even higher. Generic compression libraries such as LZ4 can decompress text data at 5 GiB s^{-1} [3]. An engineer behind the high-performance ScyllaDB database system [4] concluded that *UTF-8 validation can become a bottleneck under heavy loads* [5].

Going Faster

Starting with the Pentium 4 launched at the beginning of the century, commodity processors have acquired single-instruction-multiple-data (SIMD) instructions capable of working on wide registers (e.g., 128-bit, 256-bit or even 512-bit). These SIMD instructions have become ubiquitous, being available in nearly all mobile processors and in all x64 processors. These instructions enable an efficient form of single-core parallelism that comes in addition to multi-core and memory-level parallelism [6].

There are many different ways to benefit from these SIMD instructions. Optimizing compilers often try to rewrite tight loops so that they use SIMD instructions, a process called autovectorization [7]. Though autovectorization is a powerful approach, the compilers often fail to autovectorize complex routines. Furthermore, compilers cannot produce compiled code that deviates from the semantics of the original source code. The programmer may also rely on libraries that were written with SIMD instructions in mind. Finally, a programmer may design algorithms specifically for SIMD instructions. Though such an approach requires in-depth knowledge of the available instructions and of their performance, our experience is that it provides the best performance, at the cost of greater development time.

Our main contribution is a novel SIMD-based algorithm to validate UTF-8 bytes at high speed. We consistently exceed 10 GiB s^{-1} on x64 processors. To achieve these good results, we have extended an existing technique, vectorized classification, to do most of the validation using few instructions.

2 | UTF-8

UTF-8 encodes a sequence of Unicode characters into variable-length sequences of bytes. We use the word "character" as defined by the Unicode standard: a single character from the *Universal Character Set*, which has been assigned a single code point. However, this convention does not always correspond to a single letter in a word, or a single visible "glyph." Not only are some Unicode characters invisible (e.g., new-line and control characters), glyphs are sometimes formed by combining *multiple* Unicode characters into a `grapheme`. The distinction is irrelevant to UTF-8 validation.

Variable-Length Characters

UTF-8 achieves complete ASCII backward compatibility by encoding ASCII characters (`U+00...7F`) as-is. Further, it ensures that all non-ASCII bytes have a high order bit of 1, so ASCII characters can always be identified by a most significant bit of 0. The character "9" is `00111001` in both ASCII and UTF-8.

Non-ASCII characters (`U+000080...10FFFF`) start with a *leading byte* indicating whether the character is encoded using two, three, or four bytes. It denotes this character length with the number of *header bits*—where the most significant bits are a series of 1's followed by a 0.[1] Thus, `110|00010` is the leading byte of a two-byte character, `1110|1001` starts a three-byte character, and you may expect three more bytes after `11110|000`.

The 1–3 remaining bytes of a multi-byte character are called `continuation bytes`, and have exactly one header bit. The choice to use up the two most significant bits with `10` was a deliberate tradeoff, preserving ASCII compatibility at the cost of space: because their most significant bit is 1, continuation bytes are never mistaken for ASCII. The four-byte character "😀" has leading byte `11110|000` followed by three continuation bytes: `10|011111`, `10|011000` and

[1] We adopt the convention that ASCII bytes are leading bytes with no header bits.

TABLE 1 Valid Unicode characters and corresponding UTF-8 ranges [1]

UTF-8 Bytes	Bits	Description	Code Point	UTF-8
1	7	ASCII	U+0000	00000000
			U+007F	01111111
2	11	Latin	U+0080	110|00010 10|000000
			U+07FF	110|11111 10|111111
3	16	Asiatic	U+0800	1110|0000 10|100000 10|000000
			U+D7FF	1110|1101 10|011111 10|111111
			U+E000	1110|1110 10|000000 10|000000
			U+FFFF	1110|1111 10|111111 10|111111
4	21	Supplementary	U+010000	11110|000 10|010000 10|000000 10|000000
			U+10FFFF	11110|100 10|001111 10|111111 10|111111

10|000000.

UTF-8 tries to concisely represent as many frequently used languages as possible in as few bytes as possible. Two-byte characters (up to U+07FF) can represent most Latin alphabets, and other alphabets like Hebrew and Arabic. Most characters in natural languages (including Chinese and Japanese) fit into at most 3 bytes (up to U+FFFF). Unicode uses 4-byte characters to represent “supplementals” such as emojis.

Encoding the Value

The character value itself is stored by disassembling it bitwise and inserting its bits into the unused (non-header) bits of the byte sequence. The bits are inserted in reverse (“big-endian”) order, with the most significant bits in the leading byte, and the lowest bits in the last continuation byte. See Table 2. Decoding UTF-8 is just reassembling the character value, validating that sequences are well-formed. Thus UTF-8 is independent from the *endianness* of the processor and system. It is still allowed to prefix the string with a byte-order-mask (the byte sequence 0xEF,0xBB,0xBF) but it does not add any difficulty to the validation since this sequence is valid UTF-8 [8].

UTF-8 Sortability

UTF-8 is *normalized*: there is only one way to write a Unicode string in UTF-8. Because of this byte-for-byte stability, UTF-8 strings are byte-sortable and byte-comparable. Two strings form the same sequence of characters if and only if their bytes are all the same. A string is likewise considered larger than another in Unicode lexicographical order if its first non-equal byte is larger.

This helps with compatibility: existing libraries like hash tables, and programs like `grep`, can be easily adapted to UTF-8, and often work without modification. It is also a security feature. There is only one way to represent a given character (such as the null character or the / character) and so validating the content of strings for security is easier.[2]

[2] A single *visual* character, or glyph, may be represented by more than one *sequence* of Unicode characters. This is not relevant to UTF-8, which operates at the Unicode character level.

TABLE 2 UTF-8 Character Examples: ASCII, two, three and four-byte characters

Label	Byte 1	Byte 2	Byte 3	Byte 4
Text	9 (U+0039)			
Binary	00111001			
UTF-8	00111001			
Text	¢ (U+00A3)			
Binary	10	100011		
UTF-8	110|00010	10|100011		
Text	鏡 (U+93E1)			
Binary	1001	001111	100001	
UTF-8	1110|1001	10|001111	10|100001	
Text	😀 (U+1F600)			
Binary	00	011111	011000	000000
UTF-8	11110|000	10|011111	10|011000	10|000000

3 | VALIDATING UTF-8

A validator must step through each character in a UTF-8 document, checking for violations of each of these rules:

a) **5+ Byte.** The leading byte must have fewer than 5 header bits.
b) **Too Short.** The leading byte must be followed by N-1 continuation bytes, where N is the UTF-8 character length.
c) **Too Long.** The leading byte must not be a continuation byte.
d) **Overlong.** The decoded character must be above U+7F for two-byte characters, U+7FF for three-byte characters, and U+FFFF for four-byte characters.
e) **Too Large.** The decoded character must be less than or equal to U+10FFFF.
f) **Surrogate.** The decoded character must be not be in U+D800...DFFF.

The rules can be usefully separated into three kinds: malformed byte sequences, invalid Unicode characters, and overlong byte sequences.

Malformed Byte Sequences

Any UTF-8 character must either be an ASCII byte, or a byte with 2–4 header bits followed by 1–3 continuation bytes—no more, and no less. The easiest such error to detect is 5 or more header bits. These include 111110|, 1111110|, 11111110| and 11111111|. Out-of-order sequences and sequences with the wrong number of bytes are also invalid. See Table 3.

TABLE 3 Examples of Malformed Byte Sequences

type	byte sequence	
Too Long	`00111001 10|000000`	The continuation byte is a “stray”, that is not a part of any character.
Too Short	`1110|1001 10|001111 00111001`	There are only 2 bytes in a 3-byte character.
5-Byte	`111110|10 10|010000 10|010000 10|000000 10|000000`	5-byte character sequences are disallowed.

TABLE 4 Invalid Unicode Character Examples

type	byte sequence	
Surrogate	`1110|1101 10|111000 10|000000`	`U+D83D U+DE00` is the surrogate pair for “😀” (`U+1F600`).
Too Large	`11110|100 10|010000 10|000000 10|000000`	`U+110000` is larger than the largest Unicode character.

Invalid Unicode Characters

A well-formed byte sequence can always be decoded into a code point, but even then, some code points represent *invalid* Unicode characters. For example, Unicode only supports characters from `U+000000...10FFFF`. Anything outside that range is *too large* and therefore invalid. Since 4-byte characters can encode anything up to `U+1FFFFF`, characters from `U+110000...1FFFFF` are too large.

Additionally, UTF-8 disallows Unicode *surrogate* characters (`U+D800...DFFF`), which were designed to encode values larger than 16 bits in UTF-16. UTF-8 disallows these because it already has a way to encode characters larger than 16 bits, and surrogate support would break the normalization rule that there is only one way to encode a given code point. See Table 4.

Overlong Byte Sequences

UTF-8 mandates that each character be encoded in the smallest number of bytes possible. Larger sequences would be well-formed, and represent valid Unicode characters, but they break the normalization rule.

Overlong byte sequences are violations of this rule. For example, the character “a” (`U+61`) in a 3-byte character, padding it with zeroes: `1110|000010|000001 10|100001`. This is the only category of invalid UTF-8 that can occur even when the byte sequence is well-formed and represents a valid Unicode character.

4 | BRANCHY RANGE VALIDATOR

A *Branchy Range Validator* validates without decoding, walking the input character by character and checking that each byte in the character is in a specific range. It branches on the value of the first byte of each character, using it to decide how many continuation bytes are expected, and what range of values those continuation bytes may have. Anything outside these ranges is considered invalid and terminates the algorithm. See Algorithm 1.

Such a relatively simple algorithm is commonly found inside popular software. As a reference, we use the validation function from the Fuchsia operation system [9] by Google. The Fuchsia engineers have benchmarked this function with some care. It follows closely our description.

Algorithm 1 Branchy Range Validator algorithm. Byte values are treated as integers in $[0, 255]$.

```
for each byte $b$ in the UTF-8 sequence do
    switch $b$ do
        case 0b|{00000000...01111111} Continue the loop. end case                ▹ ASCII U+0...7F
        case 0b110|{00010...11111} do                                           ▹ 2-Byte U+80...7FF
            Load the next byte $c_1$, returning an error if it is not a continuation (0b10|).
        case 0b1110|0000 do                                                 ▹ 3-Byte Low U+800...FFF
            Load the next two bytes $c_1$ and $c_2$, returning error on EOF or if not continuations (0b10|).
            if $c_1 \in$ 0b10|{000000...011111} then Return error (Overlong). end if
        case 0b1110|1101 do                                                    ▹ 3-Byte U+D000...D7FF
            Load the next two bytes $c_1$ and $c_2$, returning error on EOF or if not continuations (0b10|).
            if $c_1 \in$ 0b10|{100000...111111} then Return error (Surrogate). end if
        case 0b1110|{0001...1100} or 0b1110|{1110...1111} do    ▹ 3-Byte U+1000...CFFF, U+E000...FFFF
            Load the next two bytes $c_1$ and $c_2$, returning error on EOF or if not continuations (0b10|).
        case 0b11110|000 do                                                  ▹ 4-Byte U+10000...3FFFF
            Load the next three bytes $c_1$, $c_2$ and $c_3$, returning error on EOF or if not continuations (0b10|).
            if $c_1 \in$ 0b10|{000000...001111} then Return error (Overlong). end if
        case 0b11110|{001...011} do                                          ▹ 4-Byte U+40000...FFFFF
            Load the next three bytes $c_1$, $c_2$ and $c_3$, returning error on EOF or if not continuations (0b10|).
        case 0b11110|100 do                                                ▹ 4-Byte U+100000...10FFFF
            Load the next three bytes $c_1$, $c_2$ and $c_3$, returning error on EOF or if not continuations (0b10|).
            if $c_1 \in$ 0b10|{100000...111111} then Return error (Too Large). end if
        else Return error  end else                                                ▹ Too Long, 5+ Byte
Return that the sequence is valid.
```

ASCII Optimization

In many practical instances, UTF-8 contains long strings of ASCII, where a vectorized ASCII check can save us many loop iterations. If there are at least eight bytes to read, we load them into an 8 byte register and quickly check whether any of the characters are non-ASCII (i.e., if they have header bits). This can be done with a simple AND operation against the 8 byte integer value `0x8080808080808080`, followed by a comparison with zero. If we have found eight consecutive ASCII characters, we just advance the byte pointer by eight and resume the loop, checking again for the presence of eight more bytes. We find in practice that it is better to go even wider: we check that the next 16 bytes are ASCII by loading the next 16 bytes into two 8-byte registers, computing their bitwise OR and then using the same 8-byte mask `0x8080808080808080`. We refer to this algorithm, with 16-byte ASCII test, as branchy-ascii. Though we could further widen this approach, we observe poorer performance with wider ASCII checks (i.e., 32 bytes) on realistic data.

5 | FINITE-STATE MACHINE

Even on completely valid input, the Branchy Range Validator branches based on the width of each character. This can cause processor stalls when character widths vary (a frequent occurrence in non-ASCII text). To eliminate this issue, we consider a finite-state machine-based approach.

We could not find an existing finite-state UTF-8 validator. We adapt a UTF-8 decoder proposed by Hoehrmann [10]. This state machine can be in one of nine possible states:

- State `valid` indicates the file is valid to this point. We always begin with `valid`.
- States “1 more”, “2 more” and “3 more” indicate the number of remaining bytes in the character, and that they can be any value.
- Range error states “3-Byte Overlong” and “3-Byte Surrogate”, “4-Byte Overlong”, and “4-Byte Too Large” indicate that there are 2 or 3 bytes remaining in the character, but that the next byte must be checked against a specific range to ensure we do not accept certain invalid values (i.e., it must be a continuation byte, but cannot be just *any* continuation byte).
- The `error` state indicates we have detected an error. Once it reaches this state, it never leaves.

Table 5 describes the transitions. As previously stated, the `error` state is “sticky”, with any byte leading back to `error`. When the state is `valid`, the byte is treated as the first byte of a character: it is possible for the state to transition to any of the nine possible states. From states “3 more”, “4-Byte Overlong”, and “4-Byte Too Large”, we always either transition to an error or to the state “2 more”. When the state is “2 more”, “3-Byte Overlong” or “3-Byte Surrogate”, we always either transition to an error or to the state “1 more”. When the state is “1 more”, we always transition to an error or to the state `valid`

To quickly compute the transition, we need to classify any new byte into one of these categories:

1. Continuation Low (`10|000000...001111`),
2. Continuation (`10|010000...001111`),
3. Continuation High (`10|100000...111111`),

and each of the nine categories corresponding to the last column of Table 5 (ASCII,`110|00010...11111`, etc.). Thus only twelve categories in total are needed. To map any of the 256 possible byte values to one of these twelve categories without branching, we use a 256-entry lookup table. We combine efficiently the resulting category (e.g, as an integer between 0 to 11) with the state (e.g., as a multiple of 12, from 0 to 108) with an addition, so that state + class is always a distinct value. Finally, the combined value is used to look up the next state in another table.

Each byte processed requires two memory loads from small tables: one to categorize the byte and one to determine the new state. The first lookup in the 256-entry table only depends on the character value and may begin before we have the new state.[3] However, there is a critical data dependency between the successive table lookup that update the state.

We could also combine the two small tables into a single large one (with 9×256 entries) to halve the number of memory loads: in our tests, it is no faster and uses more memory. This lack of benefit is expected since we do not remove the critical data dependency tied to state updates.

[3] Current commodity processors can have several memory requests in flight at the same time.

TABLE 5 Finite-State Machine Transitions. We distinguish between three types of continuation bytes: Continuation Low (10|000000...001111), Continuation (10|010000...001111), and Continuation High (10|100000...111111). When in `valid` and encountering a non-continuation byte, we determine the next state by using the last column (1st Byte).

State	Leading	Continuation	Continuation	Continuation	1st Byte
	Byte	Low		High	
`valid`	1st Byte	`error`	`error`	`error`	00000000...01111111
1 more	`error`	`valid`	`valid`	`valid`	110|00010...11111
2 more	`error`	1 more	1 more	1 more	1110|0001...1100 1110|1110...1111
3 more	`error`	2 more	2 more	2 more	11110|001...011
3-Byte Overlong	`error`	`error`	`error`	1 more	1110|0000
3-Byte Surrogate	`error`	1 more	1 more	`error`	1110|1101
4-Byte Overlong	`error`	`error`	2 more	2 more	11110|000
4-Byte Too Large	`error`	2 more	`error`	`error`	11110|100
`error`	`error`	`error`	`error`	`error`	110|00000...00001 1111|0101...1111

Such table-based algorithms are crucially dependent on the latency of loads: at least three cycles on x64 processors. To compensate, when the input string is sufficiently long (32 byte), we divide the strings into three distinct regions of nearly equal size, all of them starting with a leading byte. We then run three interleaved versions of the algorithm, loading three distinct bytes from the three regions, and updating three distinct states. We arrived at the number three experimentally, by trying the different variants (1, 2, 3, 4, ... interleaved versions). We call the resulting algorithm *finite-state*.

It would be possible to add branching to finite-state to accelerate ASCII decoding. However, we would then lose the core conceptual benefit of the finite-state approach: the lack of branches.

6 | THE `LOOKUP` ALGORITHM

The `lookup` algorithm mitigates the finite-state machine's memory latency using small lookup tables that fit in SIMD registers. It also vectorizes the problem, validating many bytes of input at a time.

We rely on a key property of the validation problem: nearly all invalid UTF-8 cases can be detected by looking at the first two bytes of a character (in fact, the first 12 bits—see Table 6). The only cases that cannot be detected in 2 bytes are sequences with extra or missing third, fourth or fifth bytes. All *those* can be detected with 4 bytes (see Table 7).

SIMD registers on a given architecture might span $w = 16$ bytes (e.g., ARM NEON, Intel SSE2), $w = 32$ bytes (e.g., AVX/AVX2) or even wider (e.g., AVX-512), allowing the algorithm to check more bytes at once, but the width is irrelevant for algorithmic purposes.

We load the file w bytes at a time into SIMD register v_1. The previous input is kept in register v_0. On the first iteration, v_0 is filled with zero (the ASCII null character).

TABLE 6 Invalid 1–2 byte UTF-8 Sequences.

Error	UTF-8	
Overlong (2–Byte)	110|00000...00001	
Overlong (3–Byte)	1110|0000	10|0
Overlong (4–Byte)	11110|000	10|0
Too Short (Missing 2nd Byte)	11|	0|
Too Long (ASCII + Continuation)	0|	10|
Surrogate	1110|1101	10|1
Too Large	11110|100	10|1
Too Large	11110|101...111	
Too Large (5+–Byte)	111111|	

TABLE 7 Invalid 3–4 byte UTF-8 Patterns.

Error	UTF-8			
Too Long (Extra 3rd Byte)	11|	10|	10|	
Too Long (Extra 4th Byte)	111|	10|	10|	10|
Too Long (Extra 5th Byte)	10|	10|	10|	10|
Too Short (Missing 3rd Byte)	111|	10|	0|	
Too Short (Missing 3rd Byte)	111|	10|	11|	
Too Short (Missing 4th Byte)	11110|	10|	10|	0|
Too Short (Missing 4th Byte)	11110|	10|	10|	11|

Instead of branching on a error conditions, we use an "error register" that is non-zero if and only if an error is detected. The error register is similar to the state variable in the finite-state algorithm (§ 5). To modify the error register, we use a bitwise OR between an expression that is non-zero if and only if an error is detected. In this manner, we avoid branches. A single check at the end can determine whether there was an error.

6.1 | Invalid 2–Byte Sequences

After loading v_1, we detect all invalid 2-byte sequences at once using vectorized classification, a concept we documented in earlier work [11]. In this scheme, we classify several values at once by doing combining vectorized table lookups. Compared to earlier work, this particular application of vectorized classification uses three different table lookups, instead of merely two. Both ARM and x64 systems have vectorized lookup tables allowing us to use a 4-bit value (nibble) stored in byte as an index into a 16-byte register (e.g., `vpshufb` in AVX2). Even when the source register has 16 or 32 bytes, the 16 or 32 lookups can occur at once, using a single instruction.

For UTF-8, we create three 16-entry lookup tables that map to 8-bit values. Bits 0–6 of these values, when set, indicate a partial match against one of seven error patterns. These patterns were chosen to encompass all possible

TABLE 8 List of 2-Byte error patterns. Any pair of bytes matching one of these patterns is considered invalid except for the last row (bit 7).

Error Bit	Error	Byte 1	Byte 2
0	Too Long (ASCII + Continuation)	`0|`	`10|`
1	Too Short (Missing 2nd Byte)	`11|`	`0|`
	+	`11|`	`11|`
2	Overlong (2–Byte)	`110|00000...00001`	`10|`
3	Surrogate	`1110|1101`	`10|1`
4	Overlong (3–Byte)	`1110|0000`	`10|0`
5	Overlong (4–Byte)	`11110|000`	`10|00`
	+ Too Large	`1111|0101...1111`	`10|00`
6	Too Large	`1111|0100...1111`	`10|01...11`
7	Two Continuations (Not An Error)	`10|`	`10|`

2-byte errors (Table 8). The high and low nibbles of each byte, as well as the low nibble of the next byte, are looked up in their respective tables. To get the low nibble of a byte, we mask an existing register (AND 0xF). To get the high nibble of each byte in a register, we shift its bytes right by 4 bits.[4]

If a bit in the range 0–6 is set in all three looked-up patterns for a byte as checked with the AND instruction,[5] that byte (and the UTF-8) is considered invalid. Bit 7 is used to identify a pair of continuation bytes, which is used in § 6.2 to evaluate long invalid 3–4 byte sequences, but by itself, bit 7 is not considered an error.

There will always be a pair of bytes straddling the two SIMD registers, which need to be validated as well. To get the correct first byte to match against each second byte, we shift the input one byte to the right, "shifting in" the last byte of the previous input as we do so. Under ARM NEON, we use the `ext` instruction. Under x64, a single instruction in the 128-bit case (`palignr`) or two in the 256-bit case (`vpalignr` and `vperm2i128`) suffice.

The original Table 6 contains nine error patterns. We consolidated these into seven error patterns that cover all possible errors, so that we could save a bit for continuation pairs. The problem of finding a minimal cover is NP-hard [12], but is thankfully inexpensive with only seven patterns. Fig. 1 illustrates our routine using pseudocode. It closely matches our C++ code. Table 9 provides a processing example, with the corresponding variable names, starting with the null-terminated string "9¢鏡🙂". Because we assume that it begins the stream, we set the previous input to zeroes. We compute three vectors made of nibbles and from these three vectors we derive three lookup results (byte_1_high, byte_1_low, byte_2_high). The final result is the bitwise AND of the three lookup results. It is made entirely of zeroes except at three locations corresponding to the last byte of the character 鏡 and to the last two bytes of the character 🙂.

[4] Under x64, we lack a byte-wise vectorized shift but we can shift 16-bit words with a vector instruction (e.g., `vpsrlw`) and apply a mask to select the low nibble.

[5] We use two vector AND instructions to combine the three patterns, but for processors supporting it, a single AVX-512 instruction (`vpternlog`) would suffice.

TABLE 9 Vectorized classification example using the notation of Fig. 1 for the null-terminated string “9¢鏡😀”. We use hexadecimal byte values.

	’9’	’¢’		’鏡’			’😀’				0
input	39	C3	A7	E9	8F	A1	F0	9F	98	80	00
previous_input (set to zero)	00	00	00	00	00	00	00	00	00	00	00
prev1 (shifted input)	00	39	C3	A7	E9	8F	A1	F0	9F	98	80
high nibbles: prev1.shift_right<4>()	00	03	0C	0A	0E	08	0A	0F	09	09	08
low nibbles: (prev1 & 0x0F)	00	09	03	07	09	0F	01	00	0F	08	00
high nibbles: input.shift_right<4>()	03	0C	0A	0E	08	0A	0F	09	09	08	00
lookup result: byte_1_high	02	02	21	80	15	80	80	49	80	80	80
lookup result: byte_1_low	E7	CB	83	CB	CB	CB	A3	E7	CB	CB	E7
lookup result: byte_2_high	01	01	BA	01	E6	BA	01	AE	ae	E6	01
(byte_1_high & byte_1_low & byte_2_high)	00	00	00	00	00	80	00	00	80	80	00

6.2 | Invalid 3–4 Byte Sequences

All remaining checks are invalid 3–4 byte sequences, which either have too many continuations, or not enough (Table 7). We first get a list of byte indexes where we expect to find two continuations in a row, which can only be found in 3–4 byte sequences. We can compute these indexes with a pair of shifts and comparisons: we expect two continuations if the previous byte matches `111|`, or if the byte before that matches `1111|`. We then compare these indexes to the locations where we have two consecutive continuations, as detected by bit 7 from our vectorized classification (see § 6.1). If these two lists differ in any respect, the UTF-8 is invalid.

Under x64, we lack unsigned comparison instructions, which are needed to check the `111|` and `1111|` ranges. However, we can emulate them in various efficient ways. For example, to compute the equivalent of $m_3(v_{i-1}, v_i) \geq 0xF0$, we can use the saturated subtraction of $m_3(v_{i-1}, v_i)$ with $0xF0 - 1$ which results in a number greater than 0 where and only where $m_3(v_{i-1}, v_i) \geq 0xF0$. Thus we can compute two saturated subtraction, combine the two results using one bitwise OR. We are then left to apply a mask to set just the most significant bit of each byte where a continuation byte should appear.

```
simd8<uint8_t> classify(simd8<uint8_t> input, simd8<uint8_t> previous_input) {
  // shift the input by 1 byte, shifting in the last byte of the previous input
  auto prev1 = input.prev<1>(previous_input);
  auto byte_1_high = prev1.shift_right<4>().lookup_16(table1);
  auto byte_1_low = (prev1 & 0x0F).lookup_16(table2);
  auto byte_2_high = input.shift_right<4>().lookup_16(table3);
  return (byte_1_high & byte_1_low & byte_2_high);
}
```

FIGURE 1 Pseudocode corresponding to the vectorized classification routine

6.3 | Incomplete Stream

At the end of the stream, we may not have enough bytes to fill an entire SIMD register. If that is the case, we may simply virtually fill the leftover bytes with any ASCII character (such as zero). But even when we have enough bytes to fill a whole register, we still have to check that the data stream does not terminate with an incomplete code point. Furthermore, the 2-byte check (§ 6.1) may allow a byte value larger than the maximum (0xF4) if it occurs as the last byte of a stream. Thankfully, it not difficult to guard against both problems. We just have to check that the last byte in the last register is strictly smaller than 0xC0 (using an unsigned comparison), that the second last byte is strictly smaller than 0xE0, and that the third last byte is strictly smaller than 0xF0. A single vectorized unsigned comparison is sufficient. On x64 processors, there is no unsigned comparison instruction, but we can use an efficient alternative such as an unsigned vectorized maximum instruction followed by a comparison.

6.4 | ASCII

Because a lot of content might be ASCII, it sometimes pay to check whether the current register is made of ASCII bytes. We can efficiently check whether a given register is made of all ASCII bytes: we can check that the byte values are all negative (using two's complement). When they are ASCII, we may then use a fast path: we omit the vectorized classification and the check on the continuation bytes. However, before we do so, we need to check that the previous register did not end with an incomplete code point (§ 6.3). Doing such checks vector register by vector register might be too expensive. When the location of the ASCII blocks is hard to predict, these checks could create many mispredicted branches. Instead, we group the registers in blocks of 64 bytes.[6] We check whether the whole block (64 bytes) is ASCII. In such a case, we also need to verify that the preceding block finished with complete code points (§ 6.3), and if so we do not need any further checks. To validate a whole block of SIMD registers, we could do one comparison per register and then aggregate the result of the comparisons with a bitwise OR. This results in roughly two instructions per register.[7] Instead, it is more advantageous to compute the bitwise OR between all registers and then to do one comparison: the block is all ASCII if and only if the bitwise OR are non-negative.[8] This results in nearly half the number of instructions.

When our input is made entirely of either ASCII characters, or of sequences containing non-ASCII characters, the fast ASCII path is either always called or never called. Thus the branches are easily predicted with high accuracy. In other scenarios, we have to rely on the processors' sophisticated branch predictor for performance.

7 | EXPERIMENTS

We wish to benchmark our algorithms on common x64 processors. Recent Intel processors are often based on the Skylake microarchitecture or similar variations. AMD recently introduced a competitive microarchitecture (Zen 2); we use a server version of this architecture. We summarize the characteristics of our hardware platforms in Table 10. The Intel processor has a slightly higher frequency, but the AMD processor has a more recent microarchitecture. Both processors have 32 kB of (data) L1 cache. The AMD processor has more L2 cache (512 kB vs. 256 kB).

Our software is written using C++ (GNU GCC 9.3) and we use Linux Ubuntu (20.10). We compile the code for

[6] A 64-byte block size matches the length of a cache line on most x64 processors.

[7] A single AVX-512 instruction (`vpternlog`) might also replace two bitwise OR.

[8] Checking that the register is entirely non-negative requires few instructions: e.g., a `pmovmskb`/`vpmovmskb` instruction under x64 or a `umaxv` instruction under ARM NEON, followed by a comparison.

TABLE 10 Hardware

Processor	Base Frequency	Max.	Microarchitecture	Memory	Compiler
Intel i7-6700	3.4 GHz	3.7 GHz	Skylake (x64, 2015)	DDR4 (2133 MT/s)	GCC 9.3
AMD EPYC 7262 (Rome)	3.2 GHz	3.4 GHz	Zen 2 (x64, 2019)	DDR4 (3200 MT/s)	GCC 9.3

best performance with the `-O3` flag. All code is single-threaded and free from disk or network access. We expect all processed inputs to be in CPU cache, by design. Our software, including benchmarking code and corresponding instructions, is freely available.[9]

Our benchmarking code is *instrumented*: we use the performance counters of the processors to record the number instructions retired, the number of cycles and the number of mispredicted branches. Performance timings are heavily skewed to the right and they do not follow a normal distribution [13]. Following our earlier work [11], we run each test many times (1000) and compute both the best (smallest) and the average timing. We find that the average and the smallest timing coincide (within 1 %).

Nevertheless, we still slightly overestimate the number of elapsed cycles and the duration of the tests (by about 10 ns to 30 ns). In effect, we get slightly worse performance numbers. When the tests last a sufficient long time, we can simply ignore the effect since the overhead is negligible. However, we cannot ignore this measurement overhead on small inputs (e.g., less than 1 KiB) when using fast functions like `lookup`. To compensate, we use the following strategy when necessary: we select a string that is twice as long as the desired size, and then we select a valid substring having nearly (within a few bytes) the desired size. We run both benchmarks and subtract the timings. We report the difference as a compensated measure.

We use the AVX2 instruction set and 256-bit vector registers. The Intel processor is subject to *downclocking*: with AVX2 instructions using floating-point operations and multiplications, the processor may reduce its frequency temporarily. However, the `lookup` algorithm does not use multiplication or floating-point operations, and it therefore does not trigger downclocking. We consistently achieve the maximum frequency of the processors. We record the effective processor frequency and find it to be constant within a small margin of error (≈1 %).

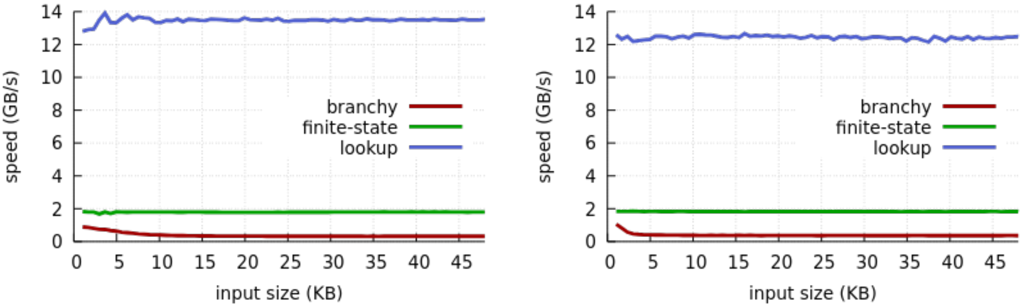
7.1 | Mispredicted Branches

When benchmarking functions involving branches, we must consider the ability of the processor to *learn branches*. When executing the same function, over the same data, repeatedly, we may expect the processor to eventually learn to predict the branches. This is unlikely to happen if the input is large and irregular, but a poorly constructed benchmark made of small or predictable input can lead to spurious results and conclusions.

We generated random UTF-8 strings of various lengths, using random code points. We pick each successive code point to have either one or two-byte length in UTF-8. Once we have generated a string of a given length, we repeatedly validate it (in a tight loop) while measuring speed and number of mispredicted branches. We pick the run with the best speed for each given length. We find that branchy is faster on small inputs: see Fig 2. The reason becomes clear when looking at the number of mispredicted branches (Fig. 3). Because of how we designed our inputs, we should expect a mispredicted branch every three bytes, so ≈333 mispredicted branches per kilobyte.

We find that both the AMD Rome and the Intel Skylake processors have far fewer than 333 mispredicted branches per kilobyte on short inputs, an indication that the branch predictor *has learned* the content of the string. The branch

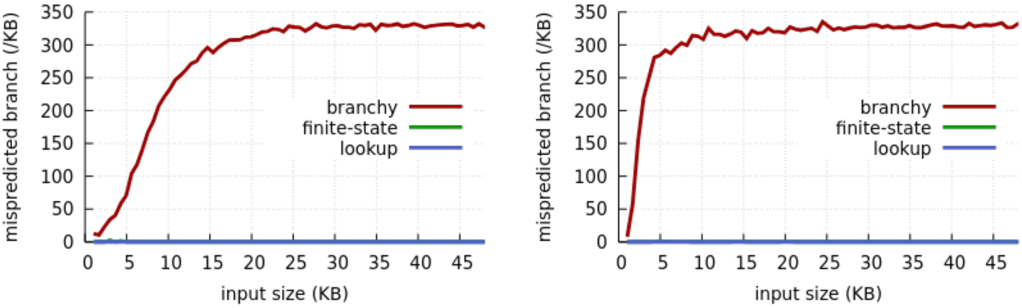
[9] `https://github.com/lemire/validateutf8-experiments`



(a) AMD Rome (Zen 2; 3.4 GHz)

(b) Intel Skylake (3.7 GHz)

FIGURE 2 Processing speed for random UTF-8 inputs of various lengths (one-and-two-byte code points).



(a) AMD Rome

(b) Intel Skylake

FIGURE 3 Number of mispredicted branches per kilobytes for random UTF-8 inputs of various lengths.

predictors work well up until about 30 kB. Observe that the AMD Rome processor is better at predicting branches than the Intel Skylake processor.

7.2 | Realistic Inputs

Of course, the validation performance depends on the input. We use two sizeable input files: a JSON file (twitter.json, 617 KiB) produced from the Twitter API and an HTML file (hongkong.html, 1.8 MiB) captured from the corresponding Wikipedia entry. See Table 11.

The speed of the branchy and of the finite-state validators are similar on the two test files, at roughly 2 GiB s^{-1}. These files are an instance where branchy-ascii is advantageous because they contain long sequences of ASCII strings. It is almost twice as fast as branchy. Though the Intel processor has a higher clock speed (by about 10 %), the AMD processor is more than 50 % faster when running the branchy validator. We also find that the AMD Rome has fewer mispredicted branches per kilobyte: 3.9 versus 4.6 (Intel) for twitter.json and 8.2 versus 7.6 for hongkong.html.

Under both AMD Rome and Intel Skylake, we find that `lookup` retires slightly under 0.4 instructions per byte for both files. Yet the throughput of `lookup` is higher under twitter.json than under hongkong.html. The explanation for

TABLE 11 Throughput in GiB s^{-1} to validate UTF-8 files. The original files are valid UTF-8. We also benchmark the C function `memcpy`, copying the content to a temporary buffer.

(a) AMD Rome (Zen 2; 3.4 GHz)

validator	twitter.json	hongkong.html
`memcpy`	48	48
branchy	2.5	2.3
branchy-ascii	4.4	4.3
finite-state	2.0	2.0
`lookup`	28	18

(b) Intel Skylake (3.7 GHz)

validator	twitter.json	hongkong.html
`memcpy`	36	36
branchy	1.6	1.6
branchy-ascii	4.0	4.4
finite-state	1.8	1.8
`lookup`	24	17

this apparent contradiction lies in the fact that the hongkong.html file triggers many more branch mispredictions.

The number of mispredicted branches per byte is tiny with twitter.json under both processors. For hongkong.html, we observe 2.0 mispredicted branch per kilobyte on AMD Rome, and slightly more on Intel Skylake (2.6). We find that the AMD processors is faster than the Intel processor when running `lookup` (5 % to 15 %) despite a lower clock speed. Under the Intel processor, the `lookup` validator comes close to matching the speed of the `memcpy` function when processing the file twitter.json: 24 GiB s^{-1} versus 36 GiB s^{-1}.

7.3 | Randomized Inputs

To test our functions with different inputs, it is useful to generate synthetic UTF-8 data. If we select to generate code-point values spanning 1–3 bytes, we randomly pick, for each code point, a byte length in the range 1–3, uniformly at random. The generator produces bytes by adding new code-point values until we have generated 16 kB. In general, the final string may exceed 16 kB by up to 3 bytes. A data input 16 kB is long enough to prevent the branch predictor from *learning the input*, but short enough to fit in L1 CPU cache. See Table 12.

The finite-state approach offers a flat performance of 1.8 GiB s^{-1} irrespective of the input source. Such data independence is expected given that the algorithm is essentially free from branches. The branchy-ascii approach does well on the ASCII-only inputs (14 GiB s^{-1} to 15 GiB s^{-1}) and roughly as well as branchy on non-ASCII synthetic inputs. The `lookup` algorithm dominates, being 30 times faster than branchy and branchy-ascii on non-ASCII inputs, and six times faster than finite-state.

On ASCII inputs, the `lookup` function is faster than the `memcpy` function, achieving 66 GiB s^{-1} on the AMD processor and 59 GiB s^{-1} on the Intel processor. In the special case where we expect our strings to be pure ASCII, we could design even faster functions with and without SIMD instructions but our purpose is UTF-8 validation.

In Table 13, we present the number of retired instructions per byte. The retired instructions are counted by the processor by excluding speculative execution. That is, the instructions part of a mispredicted branch are not counted. The processors count some fused instructions such as the comparison and jump of a branch as two instructions. It is therefore possible for some code executing tight loop with branches to have high numbers of instructions retired, if these branches are correctly predicted with high probability.

The AMD Rome and Intel Skylake processors have similar instructions counts, so we only present the numbers for the AMD Rome processors. The reason for the good performance of the `lookup` algorithm is clear: it requires

TABLE 12 Throughput in GiB s^{-1} to validate UTF-8 randomized inputs where code-point values have different byte lengths. We also benchmark the C function `memcpy`, copying the content to a temporary buffer.

(a) AMD Rome (Zen 2; 3.4 GHz)

validator	ASCII	1–2 bytes	1–3 bytes	1–4 bytes
`memcpy`	53	53	53	53
branchy	1.7	0.41	0.39	0.60
branchy-ascii	14	0.33	0.42	0.63
finite-state	1.8	1.8	1.8	1.8
`lookup`	66	13	13	13

(b) Intel Skylake (3.7 GHz)

validator	ASCII	1–2 bytes	1–3 bytes	1–4 bytes
`memcpy`	39	39	39	39
branchy	1.8	0.36	0.35	0.40
branchy-ascii	15	0.30	0.30	0.43
finite-state	1.8	1.8	1.8	1.8
`lookup`	59	12	12	12

TABLE 13 Instruction per byte to validate UTF-8 randomized inputs where code-point values have different byte lengths. As a reference we use the AMD Rome processor.

validator	ASCII	1–2 bytes	1–3 bytes	1–4 bytes
branchy	6.0	11	12	12
branchy-ascii	0.75	16	17	16
finite-state	7.0	7.0	7.0	7.0
`lookup`	0.21	0.97	0.97	0.97

far fewer instructions than the alternatives (often ten times fewer). In all our tests, irrespective of the input, `lookup` requires fewer than one retired instruction per byte.

The number of retired per cycle (Table 14) differs between the two processors with an advantage for the AMD Rome processor with branchy, branchy-ascii and `lookup`. Except for the pure ASCII inputs, the `lookup` function achieves a high 3.6 instructions per cycle on AMD Rome. In all cases, the `lookup` function benefits from a relatively high number of instructions per cycle (at least 3).

We also find that the finite-state function has a consistently high number of instructions retired per cycle (3.5). However, it suffers from a high number of instructions per byte (7).

TABLE 14 Instructions per cycle to validate UTF-8 randomized inputs where code-point values have different byte lengths.

(a) AMD Rome (Zen 2; 3.4 GHz)

validator	ASCII	1–2 bytes	1–3 bytes	1–4 bytes
branchy	3.0	1.3	1.4	2.2
branchy-ascii	4.7	1.4	1.8	2.7
finite-state	3.5	3.5	3.5	3.5
`lookup`	3.2	3.6	3.6	3.6

(b) Intel Skylake (3.7 GHz)

validator	ASCII	1–2 bytes	1–3 bytes	1–4 bytes
branchy	3.0	1.0	1.1	1.3
branchy-ascii	4.7	1.2	1.2	1.7
finite-state	3.5	3.5	3.5	3.5
`lookup`	3.0	3.1	3.1	3.1

8 | RELATED WORK

There has been much work on the acceleration of text content using SIMD instructions (e.g., base64 [14, 15], JSON [11], XML [16], HTML [17], CVS [18]). We are not aware of any published work directly related to Unicode validation using SIMD instructions other than our own [11]. Cameron [19] has worked on the related problem of UTF-8 to UTF-16 transcoding using SIMD instruction, but their approach is not applicable to high-speed validation. There has been some research on the parallelisation of finite-state machines [17, 20] which could be applied to UTF-8 validation.

9 | CONCLUSION

The relatively simple algorithm (`lookup`) can be several times faster than conventional algorithms at a common task using nothing more than the instructions available on commodity processors. It requires fewer than an instruction per input byte in the worst case. This new algorithm has been adopted by the simdjson library with good results.[10] A SIMD-based approach like `lookup` is especially advantageous in a context where the data is loaded in vector registers in any case—as happens in simdjson.

Intel has produced a new family of instruction sets with wider vector registers and more powerful instructions (AVX-512). Future research should assess the benefits of AVX-512 instructions to the problem of UTF-8 validation. In principle, we could expect the performance to double [15]. Similarly, commodity ARM processors may soon benefit from more powerful instructions and wider registers (e.g., SVE and SVE2) [21, 22].

[10] `https://simdjson.org`

Acknowledgements

The authors would like to thank T. Downs for discussions on finite-state validation. Our work is inspired by an early high-speed SIMD validator written by K. Willets that served as a proof-of-principle. We thank Z. Wegner for demonstrating that our early designs were suboptimal and for proposing inspiring new designs. Part of our benchmarking code is derived from code by W. Muła.

references

[1] Yergeau F, UTF-8, a transformation format of ISO 10646; 2015. Internet Engineering Task Force, Request for Comments: 3629. `https://tools.ietf.org/html/rfc3629` [last checked July 2020].

[2] The MITRE Corporation, CAPEC-80: Using UTF-8 Encoding to Bypass Validation Logic; 2019. `https://capec.mitre.org/data/definitions/80.html` [last checked July 2020].

[3] Collet Y, et al., LZ4 - Extremely fast compression; 2020. `https://github.com/lz4/lz4` [last checked July 2020].

[4] Suneja N. ScyllaDB optimizes database architecture to maximize hardware performance. IEEE Software 2019;36(4):96–100.

[5] Cai Y, Utils: optimize UTF-8 validation; 2019. `https://bit.ly/2VrlQ37` [last checked July 2020].

[6] Cebrián JM, Natvig L, Meyer JC. Improving Energy Efficiency through Parallelization and Vectorization on Intel Core i5 and i7 Processors. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis; 2012. p. 675–684.

[7] Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD. ACM SIGPLAN Notices 2006;41(6):132–143.

[8] Xia X, Lo D, Zhu F, Wang X, Zhou B. Software internationalization and localization: An industrial experience. In: 2013 18th International Conference on Engineering of Complex Computer Systems IEEE; 2013. p. 222–231.

[9] Singh T, Bhardwaj R. Fuchsia OS-A threat to Android. IITM Journal of Management and IT 2019;10(1):65–67.

[10] Höhrmann B, Flexible and Economical UTF-8 Decoder; 2010. `http://bjoern.hoehrmann.de/utf-8/decoder/dfa/` [last checked July 2020].

[11] Langdale G, Lemire D. Parsing gigabytes of JSON per second. The VLDB Journal 2019;28(6):941–960.

[12] Karp RM. Reducibility among combinatorial problems. In: Complexity of computer computations Springer; 1972.p. 85–103.

[13] Hoefler T, Belli R. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In: Proceedings of the international conference for high performance computing, networking, storage and analysis; 2015. p. 1–12.

[14] Muła W, Lemire D. Faster Base64 encoding and decoding using AVX2 instructions. ACM Transactions on the Web 2018;12(3):1–26.

[15] Muła W, Lemire D. Base64 encoding and decoding at almost the speed of a memory copy. Software: Practice and Experience 2020;50(2):89–97.

[16] Cameron RD, Herdy KS, Lin D. 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 CASCON '08, New York, NY, USA: ACM; 2008. p. 17:222–17:235.

[17] Mytkowicz T, Musuvathi M, Schulte W. Data-parallel Finite-state Machines. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS '14, New York, NY, USA: ACM; 2014. p. 529–542.

[18] Mühlbauer T, Rödiger W, Seilbeck R, Reiser A, Kemper A, Neumann T. Instant Loading for Main Memory Databases. Proc VLDB Endow 2013 Sep;6(14):1702–1713.

[19] Cameron RD. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming ACM; 2008. p. 91–98.

[20] Jiang P, Agrawal G. Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2017. p. 179–191.

[21] Stephens N, Biles S, Boettcher M, Eapen J, Eyole M, Gabrielli G, et al. The ARM scalable vector extension. IEEE Micro 2017;37(2):26–39.

[22] Pohl A, Cosenza B, Juurlink B. Vectorization cost modeling for NEON, AVX and SVE. Performance Evaluation 2020;140–141:102106.