Scanning HTML at Tens of Gigabytes per Second on ARM Processors

Daniel Lemire[1]

[1]Universite TELUQ

November 18, 2024

Abstract

Modern processors have instructions to process 16 bytes or more at once. These instructions are called SIMD, for single instruction, multiple data. Recent advances have leveraged SIMD instructions to accelerate parsing of common Internet formats such as JSON and base64. The two major Web browser engines (WebKit and Blink) have adopted SIMD algorithms for parsing HTML on 64-bit ARM processors. During HTML parsing, they quickly identify specific characters with a strategy called *vectorized classification*. We review their techniques and compare them with a faster alternative. We measure a 20-fold performance improvement in HTML scanning compared to traditional methods on recent ARM processors. Our findings highlight the potential of SIMD-based algorithms for optimizing Web browser performance.

Posted on 18 Sep 2024 — The copyright holder is the author/funder. All rights reserved. No reuse without permission. — https://doi.org/10.22541/au.172665939.98271085/v1 — This is a preprint and has not been peer-reviewed. Data may be preliminary.

ORIGINAL ARTICLE

Scanning HTML at Tens of Gigabytes per Second on ARM Processors

Daniel Lemire[1]

[1]Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787

Modern processors have instructions to process 16 bytes or more at once. These instructions are called SIMD, for single instruction, multiple data. Recent advances have leveraged SIMD instructions to accelerate parsing of common Internet formats such as JSON and base64. The two major Web browser engines (WebKit and Blink) have adopted SIMD algorithms for parsing HTML on 64-bit ARM processors. During HTML parsing, they quickly identify specific characters with a strategy called *vectorized classification*. We review their techniques and compare them with a faster alternative. We measure a 20-fold performance improvement in HTML scanning compared to traditional methods on recent ARM processors. Our findings highlight the potential of SIMD-based algorithms for optimizing Web browser performance.

KEYWORDS
Text Parsing, Vectorization, Web Performance

1 | INTRODUCTION

SIMD (Single Instruction, Multiple Data) instructions allow operations to be performed on multiple words simultaneously. For example, we can compare sixteen bytes with sixteen other bytes with one instruction. Commodity processors (e.g., Intel, AMD, ARM, POWER) have supported single-instruction-multiple-data (SIMD) instructions for decades. Recent ARM processors have four instructions units capable of executing SIMD instructions [1] and thirty-two SIMD registers.

Nejati and Balasubramanian found that computational tasks such as HTTP parsing may account for for over 60% of the page load latency on mobile browsers [2]. Yet, until recently, mainstream Web browsers did not use explicitly

Web engine	vendor	parent	time
KHTML	KDE	khtmlw library	1998–...
WebKit	Apple	KHTML	2005–...
Blink	Google	WebKit	2013–...

TABLE 1 History of Web Engines

SIMD instructions for critical tasks such as parsing HTML pages. Most browsers today use either the WebKit or Blink Web engines. WebKit is developed primarily by Apple for their Safari browser. WebKit is derived from KHTML, a Web engine developed for Linux and BSD by the KDE project. In turn, Google based their Blink engine on WebKit. It is now part of Google Chrome, Brave and Microsoft Edge. See Table 1.

In May 2024, WebKit adopted SIMDe [3], a framework to help support SIMD instructions across various systems. SIMDe was quickly used to accelerate HTML parsing.[1] Maybe coincidentally, again in May 2024, another major Web engine (Blink) adopted Highway [4], a software framework similar to SIMDe—along with a strikingly similar optimization to the HTML parsing.[2] The net result is that—within a few weeks—nearly the same SIMD-based optimization was adopted by the two main Web engines. The Google engineers reported a 2% performance boost on the Speedometer 3 test [5] following this optimization.

We write Web pages using the Hypertext Markup Language (HTML) [6]. Thus HTML is probably one of the most important document format in software history. HTML is a text format most commonly encoded as UTF-8—a superset of ASCII. A fundamental component of HTML is the element. Typically, an element begins with a starting tag (e.g., <name>) and ends with a closing tag (e.g., </name>). The ampersand '&' represents *HTML entities*: special sequences that are typically used to represent special characters. For example, '<' represents the less-than character, '>' represents the greater-than character, '"' represents the quote.

Both major Web engines (WebKit, Blink) rely on the same fundamental step when parsing HTML as part of the HTML tokenization. They scan the text input for the less-than character '<' which marks the beginning of a tag. Furthermore, three other characters must be detected as they trigger special code paths:

- Whenever the carriage return character is found, browsers normalize the newlines [7]. Specifically, all sequences of the carriage return character followed by the line feed character ('\r\n') are replaced by line feed character ('\n'), and all remaining carriage return characters are replaced by a line feed character.
- The ampersand character must be detected as it may indicate the presence of an entity which must be replaced.
- We must also detect null characters ('\0') as they trigger an error and a replacement.

In practice, we expect the software to jump from one less-than character '<' to another, as they are most common. Fig. 1 illustrates how Web engines solve this problem with conventional C code.

More generally, programmers often write code to scan a string of characters and identify some classes of characters. For example, we might want to find all spaces, and numbers and so forth. The conventional approach is to load the characters one by one and to identify them using comparisons, as in Fig. 1. It is a sufficiently common task that standard libraries provide support for such tasks. For example, the C++ standard library offers the std::string::find_first_of function. In C#, the standard library has a powerful SearchValues class. Thus, even though the problem solved by the engineers working major Web engines is specific, it is also an instance of generic

[1] https://github.com/WebKit/WebKit/pull/28251

[2] https://chromium-review.googlesource.com/c/chromium/src/+/5538407

```
// pos is a pointer inside the document
while (pos != end && *pos != '<') {
  if (*pos_ == '&' || *pos_ == '\r' || *pos == '\0') {
    return SPECIAL; // special handling
  }
  ++pos;
}
```

FIGURE 1 C code to move a pointer to the next less-than character

problem of wide interest.

2 | SIMD ALGORITHM

To scan text faster, we might want to load blocks of consecutive characters and classify them at once. We refer to the general approach where one classify characters using SIMD instructions as *vectorized classification* [8]. It comes in different forms. For example, a single SIMD comparison may be needed: e.g., if we need to compare a single character with every character in the block. In more complex cases, a sequence of instructions may be required: e.g., when validating Unicode strings [9, 10] or parse URLs [11]. We might have conditions, e.g., a character type must be followed by another character.

At a high level, the vectorized classification problem recently solved by the Web engines WebKit and Blink is described by Algorithm 1. We repeatedly load blocks of k consecutive characters in a vector, and check whether one of these characters is in the desired set—where k might be 16, 32, 64, …If not, we advance by k characters and load a new block of k characters. See § 4.1 and § 4.2 for the WebKit and Blink implementations.

In Algorithm 1, we proceed character-by-character when we have fewer than k characters left in the document. Alternatively, we could also load the remaining the characters in a partial vector though it does not make a substantial difference.

Algorithm 1 — Vectorized classification to search for the next character in the set '<', '`\r`', '`&`', '`\0`'.

Require: String s of length n, indexed from 0 to $n - 1$
Require: V a vector containing k elements (characters)
Require: Q a vector containing k elements (Boolean)

1: Set $t \leftarrow \{<, \&, \backslash r, \backslash 0\}$
2: $i \leftarrow 0$
3: **while** $i + k < n$ **do**
4: Load characters at indexes $i, i + 1, \ldots, i + k - 1$ from string s in vector V
5: let $Q_j \leftarrow (V_i \in t)$ for $j = 0, \ldots, k$ ▹ vectorized classification
6: **if** Q contains at least one true element **then**
7: **return** i+ the index of the first true value in Q
8: $i \leftarrow i + k$
9: scan the remaining characters for s from index i to $n - 1$ for characters in the set, if found **return** the index
10: **return** that the string does not contain characters from the set

Algorithm 1 might be suboptimal. Consider an example. Suppose that we have a string that begin with '`<!doctype html><html itemscope="" itemtype="http://schema.o...`'. Henceforth we assume that the input string is stored

as UTF-8 so that each ASCII character uses a byte. We load the first 16 bytes (`<!doctype html><`). We find the target characters: the first character and the sixteenth character are a match ('<'). Algorithm 1 returns the location of the first matching character (the first index). If we must then find the next matching character, we might need to start again from the next position, immediately after the first matching character, e.g., we might load the sequence '`!doctype html><h`'. We then match the same character ('<'), this time as the fifteenth character. We are duplicating the effort, reprocessing many of the same characters twice.

Instead, we can adopt an approach similar to that used by systems like the simdjson library [8]. We load non-overlapping blocks of 64 characters. Each block of 64 characters is turned into a 64-bit register where each bit in correspond to a loaded character. If the character is a match, then the corresponding bit is set to 1. The computed 64-bit word serves as an index for the 64 characters. We can iterate over the set bits of the 64-bit word, as each corresponds to a matching character. Once we have used up this index, we can load another block of 64 characters and so forth. In this manner, we do not duplicate the effort: we load each block of 64 characters only once to match characters. See § 4.3 for details.

3 | INSTRUCTION SETS

There is a wide range of processors supporting vector or SIMD instructions (POWER, RISC-V, etc.). However, two types of commodity processors dominate: x64 and ARM.

Most common on PCs and servers are x64 processors generally made by Intel and AMD. These processors fall into many microarchitecture categories, including the following levels [12]:

- `x86-64-v1`: SSE2 (16-byte registers);
- `x86-64-v2`: SSE3, SSSE3, SSE 4.1, SSE 4.2, POPCNT;
- `x86-64-v3`: AVX, AVX2 (32-byte registers), BMI1, BMI2, LZCNT;
- `x86-64-v4`: AVX-512 (AVX512F, AVX512BW, AVX512CD, AVX512DQ, AVX512VL) (64-byte registers).

The features are additive: a processor of level `x86-64-v2` has the features of level `x86-64-v1`, and so forth. From an historical perspective, `x86-64-v1` goes back to 2003 (AMD K8), `x86-64-v2` to 2008 (Intel Nehalem), `x86-64-v3` to 2013 (Intel Haswell) and `x86-64-v4` to 2017 (Intel Skylake-X). Recent releases of the Red Hat and SUSE Linux systems are built for systems supporting `x86-64-v2`. Recent releases of Windows 11 (version 24H2 and following) require SSE 4.2 which matches level `x86-64-v2`. More recent processors from Intel and AMD support even more advanced SIMD instructions, but there is not yet an agreement on the `x86-64-v5` level at this time. When compiling software for an x64 processor, the simplest approach might be to pick one of the lowest levels as a target (e.g., `x86-64-v1`). It is also common to determine the features of the CPU at runtime, but it requires more care.

Most mobile devices (e.g., smartphones, tables) and some servers and laptops run on 64-bit ARM processors (aarch64). All of these processors share the same SIMD instructions with 16-byte registers: ARM NEON also called Advanced SIMD. There are a few extensions: e.g., for dot products and 16-bit floating-point numbers. More recent processors also support other SIMD instruction sets: SVE and SVE2 [13]. However relative few ARM systems support SVE or SVE2 at this time.

ARM processors only offered a 64-bit mode starting with ARMv8-A architecture in 2011. It was first broadly available in the iPhone 5s, launched in 2013. Thus, whereas x64 systems have different levels of SIMD instruction sets, the 64-bit ARM processors are more uniform from an instruction-set perspective. In part due to their shorter history, 64-bit ARM processors have a more predictable set of instructions: they all support ARM NEON with little variation.

The WebKit engine is developed by Apple. In turn, current Apple systems use 64-bit ARM processors, whether they are laptops, phones or tables. The Blink engine is developed by Google. It is designed to build on a wide range of platforms, including Android devices and various versions of Windows. Most Android devices have ARM processors. As of July 2024, the vectorized classification is only enabled under ARM platforms in Blink.

The algorithms and techniques that we consider are not specific to ARM NEON. Yet, for simplicity, we focus on ARM NEON. In our context, it is the most relevant instruction set.

3.1 | ARM Instructions

A detailed discussion of the 64-bit ARM instruction set is beyond our scope. However, it is useful to review a relevant subset. Compiler often produce regular instructions such as the following:

- `ldr`: loads a value from memory into a register.
- `cbnz`: compares a register with zero and branches if it is not zero.
- `rbit`: reverses the bit order in a register.
- `clz`: counts the number of leading zero bits in a register.

Programming languages such as C or C++ do not allow the programmer to directly express SIMD operations. To avoid having to write assembly, we often use intrinsic functions. Intrinsic functions are recognized by the compiler as representing low-level operations, often corresponding directly to CPU instructions. They are not part of the standard library but are provided by the compiler or the vendor. There are many intrinsic functions and many SIMD instructions. The following instructions are relevant to our problems:

- `ld1`: loads elements from memory into a vector register. To emit this instruction, we can use intrinsic functions such as `vld1q_u8` for unsigned 8-bit integers. There are other similar loads and store instructions (`ld2`, `st1`, `st2`, . . .).
- `cmeq`: compares two vectors for equality, it compares each pair of values and sets the result to all ones (`0xF...F`) if they are equal, otherwise to zero. We can use the intrinsic function `vceqq_u8` to compare unsigned 8-bit integers.
- `and`: computes the bitwise and between two vectors. E.g., we may use the `vandq_u8` intrinsic.
- `orn`: performs a bitwise OR operation between a register and the bitwise NOT of another register. E.g., we may use the `vornq_u8` intrinsic.
- `add`: adds the components of two vectors (`vaddq_u8` intrinsic).
- `umaxv`: finds the maximum value in a vector. The intrinsic `vmaxvq_u32` finds the maximum of unsigned 32-bit integers and the intrinsic `vmaxvq_u8` finds the maximum of unsigned 8-bit integers.
- `uminv`: finds the minimum value in a vector. It has intrinsic functions similar to `umaxv`: `vminvq_u32` for unsigned 32-bit integers, `vminvq_u16` for unsigned 16-bit integers, `vminvq_u8` for unsigned 8-bit integers.
- `shrn`: shifts each element in a vector to the right and narrows the result. For example, each 16-bit word might be shifted right and we keep only the least significant 8 bits. A 16-byte inputs results in an 8-byte output. We might use the `vshrn_n_u16` intrinsic for unsigned 16-bit integers.
- `umov`: moves a value from a vector to a general-purpose register, it can be used in conjunction with the instructions `shrn`, `umaxv` and `uminv`. We have several intrinsic functions depending on the desired bit widths such as `vgetq_lane_u64`, `vgetq_lane_u32`, `vgetq_lane_u16`, `vgetq_lane_u8`. Some compilers generate the `fmov` instruction (floating-point move) instead of the `umov` instruction.
- `addp`: adds pairs of adjacent elements in a vector. E.g., adds pairs of 8-bit integers and produce a 16-bit output. Thus a vector of sixteen 8-bit values becomes a vector of eight 16-bit values. We may use intrinsic functions such

	Neoverse V2		Apple Firestorm	
	L	T	L	T
cmeq	2	4	2	4
and	2	4	2	4
orn	2	4	2	4
add	2	4	2	4
uminv/umaxv (32-bit)	2	2	3	4
uminv/umaxv (16-bit)	4	2	3	4
uminv/umaxv (8-bit)	4	1	3	4
umov	2	1	$\leq$ 10	2
shrn	2	2	3	4
addp	2	4	2	4
tbl	2	2	2	4

TABLE 2 Latency and throughput of various SIMD instructions for two microarchitecture. The latency is the time in cycles required to complete the instruction. The throughput is the number of instructions that can be issued per cycle.

as `vpaddq_u8` for unsigned 8-bit integers.

- `tbl`: performs a table lookup on a vector. It uses elements from one vector to index into another vector, effectively performing a gather operation. If the index value is within the bounds of the table vectors (e.g., $0, 1, \ldots, 15$), the corresponding byte from the table is copied to the result vector. If the index is out of bounds, a zero is placed in the result for that element. We may use the intrinsic function `vtbl1q_u8` to generate the `tbl` instruction.

Table 2 presents the latency and throughput of these SIMD instructions on two ARM microarchitectures: Apple Avalanche [1] and Neoverse V2 [14]. To our knowledge, Apple does not release official latencies and throughputs for its processors, so these values are estimates. The Apple Avalanche is likely representative of Apple's performance cores. We believe that the Neoverse V2 is the basis for Amazon's Graviton 4 server processor. The Apple microarchitecture has high throughput: most instructions can be issued four times per cycle while only simple instructions such as `cmeq`, `and`, `orn`, `add` and `addp` have a comparable throughput on Neoverse V2. The `uminv` and `umaxv` instructions on the Apple microarchitecture have the same performance irrespective of the bit width (8-bit, 16-bit, 32-bit) but on the Neoverse V2, the 8-bit version is relatively slow: one instruction of throughput and 4 cycles of latency. There are many more ARM-based microarchitecture and a complete survey is outside our scope.

4 | EFFICIENT SIMD IMPLEMENTATION

We can use SIMD instructions deliberately from a language like C++ in various ways: inline assembly, intrinsic functions, etc. Both WebKit and Blink use dedicated SIMD libraries. These libraries abstract away the instruction sets, allowing the engineers to focus on the programming logic. Furthermore, they allow the engineers to quickly support

a wider range of platforms with less programming effort. We describe the WebKit and Blink implementations as of July 2024. The source code is freely available online.[3]

4.1 | WebKit Implementation

We present in Fig. 2 a slightly simplified version of the WebKit implementation. The full version also supports UTF-16 inputs but most Web sites use UTF-8. Further, we also apply the simplifying assumption that the input contains at least 16 bytes. WebKit relies on SIMDe [3]. The SIMDe `simde_uint8x16_t` type represents a vector of 16 eight-bit unsigned integers.

The `scanText` function is part of WebKit's HTML parser. It scans a buffer of text for specific characters and extracts a substring based on those findings. If a null terminator is found within the buffer or if the first character encountered is '`&`' or '`\r`', it signals a parsing error or the need for special handling and returns an empty string. The `vectorEquals8Bit` function does the vectorized classification, returning is the SIMD equivalent. The `vectorMatch` lambda function uses the `vectorEquals8Bit` function to find the first occurrence of the target characters within a SIMD vector.

The `vectorEquals8Bit` function identifies the matching character with as few as three instructions: a bitwise AND (`and` instruction), a vectorized table lookup (`tbl` instruction), and a byte-wise comparison (`cmeq` instruction). The algorithm is based on the observation that the four characters that we need to identify ('<', '`\r`', '`&`', '`\0`') have code point values are `0x00`, `0x26`, `0x3c`, and `0x0d`: they can be distinguished by the least significant 4 bits (`0x0`, `0x6`, `0xc`, and `0xd`). Hence the algorithm proceed as follows:

1. Take the character value c as an 8-bit integer.
2. Use a 16-element table t where all values are zero except for the those corresponding to indexes `0x0`, `0x6`, `0xc`, and `0xd` where we store `0x00`, `0x26`, `0x3c`, and `0x0d`.
3. Use the least significant bits of c, computed as c AND 0xF to look up a value in the table. We have that $t[c \text{ AND } 0xF] = c$ if and only if c is one of `0x00`, `0x26`, `0x3c`, and `0x0d`.

Because it is implemented using SIMD instructions, sixteen bytes are processed at once. Thus using as few as three instructions, we get to a SIMD register where the matching characters correspond to a true value (`0xFF` by convention).

Given this SIMD register, we need to find the index of the first match, if any. We provide the implementation in Fig. 3. The WebKit engineers first compute the maximum byte value in the register using a function from the SIMDe [3] framework (`simde_vmaxvq_u8` corresponding to the `vmaxvq` instruction): this function returns 0 if and only if there is no match. Otherwise the value `0xFF` is returned. In this case, the function continues: it computes a byte-wise OR NOT (using the `orn` instruction) between the values 0, 1, ...15 and our SIMD register. It then calls `simde_vminvq_u8` (corresponding to the `vminvq` instruction). These two instructions (`orn` and `vminvq`) are sufficient to return the smallest index. E.g., suppose that only the second and last character were a match, then we have the SIMD register (0, `0xFF`, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, `0xFF`). After the bitwise OR NOT with the values 0, 1, ...15, we get (`0xFF`, 1, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, `0xFF`, 15), and the minimum is 1, indicating the first match.

The function `SIMD::find` wearches the buffer for a match using the `vectorMatch` function. See Fig. 4. Under ARM, the core of the find function might compile to the assembly instruction in Fig. 5. When a 16-byte input does not contain any matching character, we use about seven instructions—and slightly more in practice: (`ldr`, `and`, `tbl`, `cmeq`, `umaxv`, `cbnz`). Overall, we use fewer than 16 instructions to process 16 bytes. When the 16 bytes contain a

[3] `https://github.com/WebKit/WebKit` and `https://github.com/chromium/chromium/tree/main/third_party/blink`

```
String scanText() {
  auto* start = m_parsingBuffer.position();
  auto* end = start + m_parsingBuffer.lengthRemaining();
  auto vectorEquals8Bit = [&](auto input) {
    simde_uint8x16_t lowNibbleMask { '\0', 0, 0, 0, 0, 0, '&', 0, 0, 0, 0, 0, '<', '\r', 0, 0 };
    simde_uint8x16_t v0f = SIMD::splat<uint8_t>(0x0f);
    return SIMD::equal(simde_vqtbl1q_u8(lowNibbleMask, SIMD::bitAnd(input, v0f)), input);
  };
  CharacterType* cursor = nullptr;
  auto vectorMatch = [&](auto input) {
    return SIMD::findFirstNonZeroIndex(vectorEquals8Bit(input));
  };
  cursor = SIMD::find(std::span { start, end }, vectorMatch);
  m_parsingBuffer.setPosition(cursor);
  if (cursor != end) {
    if (*cursor == '\0')
      return didFail(HTMLFastPathResult::FailedContainsNull, String());

    if (*cursor == '&' || *cursor == '\r') {
      m_parsingBuffer.setPosition(start);
      return scanEscapedText();
    }
  }
  unsigned length = cursor - start;
  return length ? String({ start, length }) : String();
}
```

FIGURE 2 WebKit Implementation

```
std::optional<uint8_t> findFirstNonZeroIndex(simde_uint8x16_t value) {
    if (!simde_vmaxvq_u8(value))
        return std::nullopt;
    constexpr simde_uint8x16_t indexMask { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    return simde_vminvq_u8(simde_vornq_u8(indexMask, value));
}
```

FIGURE 3 WebKit's Find-First Implementation

```
const char* find(std::span<const char> span, const auto& vectorMatch) {
    auto* cursor = span.data();
    auto* end = span.data() + span.size();
    for (; cursor + (16 - 1) < end; cursor += 16) {
            if (auto index = vectorMatch(SIMD::load(cursor)))
               return cursor + index.value();
    }
    if (cursor < end) {
        if (auto index = vectorMatch(SIMD::load(end - stride)))
            return end - stride + index.value();
    }
    return end;
}
```

FIGURE 4 WebKit's Find Implementation

```
        and v3.16b, v2.16b, v0.16b; AND 0xF
        tbl v3.16b, { v1.16b }, v3.16b ; table lookup
        cmeq v2.16b, v3.16b, v2.16b ; compare
        umaxv b3, v2.16b ; check if the result is zero
        fmov w9, s3 ; could be umov
        cbnz w9, MATCH
; ... insert code to loop back
MATCH: ; we have a match
        orn v0.16b, v0.16b, v2.16b
        uminv b0, v0.16b
        fmov w8, s0 ; could be umov
        add x0, x0, x8 ; x0 contains the index of a match
```

FIGURE 5 WebKit's Possible ARM NEON Assembly

```
and v30.16b, v29.16b, v30.16b
tbl v31.16b, {v31.16b}, v30.16b
cmeq v31.16b, v31.16b, v29.16b
shrn v31.8b, v31.8h, 4
umov x0, v31.d[0]
cbz x0, .NOMATCH ; does not match
rbit x0, x0
clz x0, x0
lsr x0, x0, 2; division by 4
```

FIGURE 6 Blink's Possible ARM NEON Assembly

match, we must add another series of instructions (`orn`, `uminv`, `umov/fmov`, `add`), making the process more expensive: we get closer to one instruction per byte. The `umov` or `fmov` instructions move the data from a SIMD register to a general-purpose register while the `add` instruction increments our index.

The exact performance of the code depends on the processor microarchitecture. However, we should pay attention to the `uminv` and `umaxv` instructions which can require many cycles of latency (e.g., four) on some processors.

4.2 | Blink Implementation

The Blink implementation resembles the WebKit implementation. Instead of SIMDe, it relies on the Highway library [4]. The Highway library provide a `FindFirstTrue` function which is equivalent in functionality to WebKit's `findFirstNonZeroIndex` function (see Fig 3). Unlike the WebKit implementation, the `FindFirstTrue` function in Highway does not use the `umaxv` and `uminv` instructions. Instead, Highway uses the `vshrn_n_u16` intrinsic function. This intrinsic function corresponds to the `shrn` instruction. It takes each 16-bit word, shifts them to the right by 4 bits, and then keeps only the least significant 8 bits out of each 16-bit word. The net result covers 64 bits and can be moved to a general-purpose register. To see why it is a helpful operation, consider again the mask where only the first and last character were a match (`0xFF`, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, `0xFF`).

1. We first represent it as eight 16-bit words with the little-endian convention: `0x00FF`, `0x0000`, `0x0000`, `0x0000`, `0x0000`, `0x0000`, `0x0000` , `0xFF00`.
2. We shift by 4 bits: `0x000F`, `0x0000`, `0x0000`, `0x0000`, `0x0000`, `0x0000`, `0x0000` , `0x0FF0`.
3. We keep only the least significant byte of each 16-bit word: `0x0F`, `0x00`, `0x00`, `0x00`, `0x00`, `0x00`, `0x00` , `0xF0`.
4. We convert the result to a 64-bit word: `0xF0000000000000F0`.

The final 64-bit still contain the information concerning the matches: each matched character correspond to a 4-bit subword. To find the index of the first match, it suffices to compute the number of trailing zeros and divide this number by four. Fig. 6 presents a possible compiled assembly. Observe how computing the number of trailing zeros is done using two instructions: `rbit` which reverses the bits, and `clz` which computes the number of leading zeros.[4]

4.3 | 64-bit Vectorized Classification

Both the WebKit and Blink implementations do some unnecessary processing, potentially reloading and reclassifying the same characters many times. Further, they might be limited by the latency of the instructions used to indentify the first matching character in a block of 16 bytes.

Instead, let us load non-overlapping blocks of 64 bytes and identify the matching characters in a 64-bit word, as in Fig. 7. Given this 64-bit index, we can check whether it is zero, and if so, we know that must advance forward by

[4] ARM added new relevant functionality to its designs as an extension, called *Common Short Sequence Compression* (`FEAT_CSSC`). This extension adds a trailing-zero instruction as well as a population count function, among others. We expect that it will be supported by upcoming processors.

64 bytes in the input string. Otherwise, we just compute the number of trailing zeros and advance the string by the count. We can shift the index by the corresponding number of trailing zeros plus one, and if the result is non-zero then the index can be used to advance in the string once more, and so forth. If the result is zero, we need to load 64 new bytes: we always load the bytes in a non-overlapping manner, to the next 64 bytes. At the end of the string, when fewer than 64 bytes remain, we may need to either fallback on a conventional routine.

The routine in Fig. 7 begins by filling four 16-byte registers corresponding to a block of 64 bytes using the `vld1q_u8` intrinsic—corresponding to the `ld1` instructions. We then use the same vectorized classification as in the WebKit and Blink routines: we select the least significant 4 bits of each byte with a bitwise AND (`&` operator) followed by a vectorized table lookup (`vqtbl1q_u8` intrinsic, `tbl` instructions) followed by a comparison (`vceqq_u8` intrinsic, `cmeq` instruction). We then have 64 bytes where the matching characters have been identified stored in `matchesones1`, `matchesones2`, `matchesones3`, and `matchesones4`. If the microarchitecture allows it, these computations can be executed in parallel: many ARM processors have 4 SIMD execution units. Finally, we must then aggregate these four 16-byte registers counting byte values `0x00` and `0xFF` into a 64-bit integer. See Fig. 8 for an illustration.

1. We compute the bitwise AND of the four SIMD variables with (`0x01`, `0x02`, `0x4`, `0x8`, `0x10`, `0x20`, `0x40`, `0x80`, `0x01`, `0x02`, `0x4`, `0x8`, `0x10`, `0x20`, `0x40`, `0x80`). These values correspond to bits at position 1, 2, 3, . . . 8, repeated twice.
2. We do horizontal pairwise byte sums (`vpaddq_u8` intrinsic, `addp` instruction): starting with two 16-byte values, treated as a sequence of 32 bytes, we sum the first two bytes and write the result in the output, we sum the next two bytes and write the result as a byte and so forth. By doing it four times, we collapse the initially 64 bytes into the desired 64-bit integer. To illustrate the algorithm, suppose that all of the characters matched. Then we would do a horizontal pairwise byte sum on the 32-byte sequence (`0x01`, `0x02`, . . . , `0x80`). We want to show that we can compute the 64-bit word `0xFFFFFFFFFFFFFFFF`. Initially, we get the 16-byte sequence (`0x03`, `0x0c`, `0x30`, `0xc0`, `0x03`, `0x0c`, `0x30`, `0xc0`, `0x03`, `0x0c`, `0x30`, `0xc0`, `0x03`, `0x0c`, `0x30`, `0xc0`). We get two such sequences (`sum0` and `sum1` in Fig. 7). Combining the once more we get the 16-byte sequence `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`, `0x0F`, `0xF0`. Repeating one last time, this time by repeating the same sequence of 16-byte values twice, we finally get the 64-bit value `0xFFFFFFFFFFFFFFFF` twice, as expected. We keep only one of the two 64-bit values. If some characters were unmatched, then zero bits would appear. Depending on the microarchitecture, the first two `addp` instructions can be executed simultaneously, whereas the next two must be done in sequence. Though there is likely some non-trivial latency, it is amortized over an input of 64 bytes of input data.

Because we need to compute the number of trailing zeros if the output is non-zero, we might reverse the bits (`rbit`) once. In this manner, we do not need to repeatedly call the sequence `rbit/clz` to compute the number of trailing zeros, if there are several matches: it suffices to call `rbit` once followed by a single call to `clz`. However, this may not be a significant optimization in practice.

5 | EXPERIMENTS

We implemented the WebKit and Blink approaches, as well as the 64-bit indexing approach. As a reference implementation, we also use the standard C++ function (`string::find_first_of`) as provided by the runtime library. We make our code freely available to ease reproducibility.[5] We use two different ARM-based systems for benchmarking. See Table 3.

We collected three HTML files by saving the HTML pages of the BBC, Microsoft Office, and Google Web sites.

[5] https://github.com/lemire/htmlscanning

```
// constants:
static uint8x16_t low_nibble_mask = {0, 0, 0, 0, 0, 0, 0x26, 0, 0, 0, 0, 0, 0x3c, 0xd, 0, 0};
static uint8x16_t bit_mask = {0x01, 0x02, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80,
                              0x01, 0x02, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80};
static uint8x16_t v0f = vmovq_n_u8(0xf);

uint64_t update(const uint8_t *buffer) {
    uint8x16_t data1 = vld1q_u8(buffer);
    uint8x16_t data2 = vld1q_u8(buffer + 16);
    uint8x16_t data3 = vld1q_u8(buffer + 32);
    uint8x16_t data4 = vld1q_u8(buffer + 48);
    uint8x16_t lowpart1 = vqtbl1q_u8(low_nibble_mask, data1 & v0f);
    uint8x16_t lowpart2 = vqtbl1q_u8(low_nibble_mask, data2 & v0f);
    uint8x16_t lowpart3 = vqtbl1q_u8(low_nibble_mask, data3 & v0f);
    uint8x16_t lowpart4 = vqtbl1q_u8(low_nibble_mask, data4 & v0f);
    uint8x16_t matchesones1 = vceqq_u8(lowpart1, data1);
    uint8x16_t matchesones2 = vceqq_u8(lowpart2, data2);
    uint8x16_t matchesones3 = vceqq_u8(lowpart3, data3);
    uint8x16_t matchesones4 = vceqq_u8(lowpart4, data4);
    uint8x16_t sum0 = vpaddq_u8(matchesones1 & bit_mask, matchesones2 & bit_mask);
    uint8x16_t sum1 = vpaddq_u8(matchesones3 & bit_mask, matchesones4 & bit_mask);
    sum0 = vpaddq_u8(sum0, sum1);
    sum0 = vpaddq_u8(sum0, sum0);
    return vgetq_lane_u64(vreinterpretq_u64_u8(sum0), 0);
}
```

FIGURE 7 Computation of a 64-bit index over 64 bytes using ARM NEON intrinsic functions

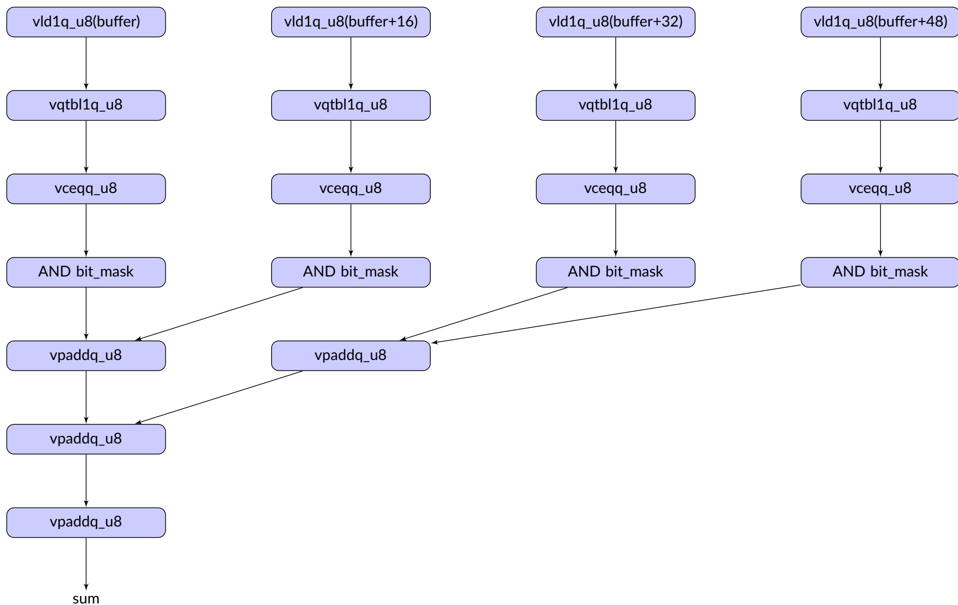


FIGURE 8 Computation of a 64-bit index: observe the obvious parallelism

TABLE 3 Systems

Processor	Frequency	Microarchitecture	Memory	Compiler
Graviton 4	2.8 GHz	Neoverse V2 (aarch64, 2022)	DDR5 (5600 MT/s)	GCC 13.2
Apple M2	3.0 GHz	Avalanche (aarch64, 2022)	LPDDR5 (6400 MT/s)	Apple/LLVM 15

TABLE 4 Test file sizes and number of matches.

file	bytes	matches	ratio
BBC	418 417	4420	1.06%
Office	213 748	2393	1.12%
Google	20 319	380	1.87%

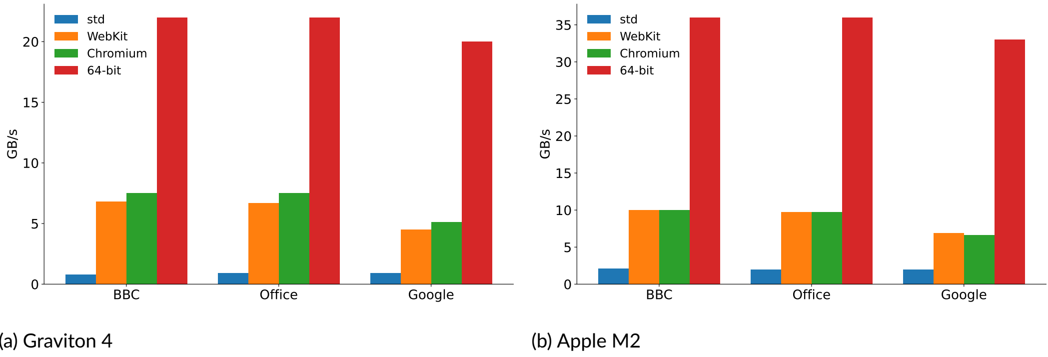
See Table 4. Importantly, the BBC Web site we captured has fewer matching characters relatively speaking than the Google Web site: we should thus expect that scanning the BBC data should be more expensive on a per input byte basis. We make the data available with our software.

To measure the performance, we take the input string and we iterate over the matching characters. We store the matching characters to a volatile variable to prevent undue optimization. In a Web browser, the scanning would be interleaved with other operations—but we wish to measure just the scanning. To help understand our performance, we use performance counters to record the number of instructions retired by cycle as well the number of instructions per cycle and the effective frequency. We present our results in Table 5. We estimate the error of the measured speed to be less than 2% and the reported number of retired instructions per byte is much less than 1%.

We find that on the Graviton 4 system, the Blink approach is faster than the WebKit approach even though they both retire the same number of instructions per byte. The better performance is due to the higher number of retired instructions for the Blink approach. The reverse is true on the Apple M2 system for the Google file. The difference is attributable to the characteristics of the microarchitectures. For example, the performance of the `uminv` and `umaxv` instructions are likely better on the Apple M2 than on the Graviton 4 system [1].

The 64-bit indexing approach is significant faster than the competitors. It is between 15 to 20 times faster than the standard function (`string::find_first_of`). The 64-bit approach is three to four times faster than the WebKit/Blink approaches. The benefits of the 64-bit approach are slightly greater in the Google file which has more matching characters. This is expected because when several matching characters are within a 64-byte range, we can move from one character to another by merely using the index, without reloading input bytes.

The 64-bit approach requires fewer instructions (40% less than WebKit/Blink), and it allows to retire many more instructions per cycle. We save instructions in part by amortizing the processing over 64-byte blocks. In effect, we use a form of batching.



(a) Graviton 4

(b) Apple M2

FIGURE 9 Parsing speed in gigabytes per second

TABLE 5 Speed in gigabytes per second, number of instructions per cycle and instructions retired per cycle for various data inputs and systems.

(a) BBC – Graviton 4

function	GB/s	ins./byte	ins./cycle
std	0.78	7.8	2.16
WebKit	6.8	1.0	2.51
Blink	7.5	1.0	2.74
64-bit	22	0.66	5.21

(b) BBC – Apple M2

function	GB/s	ins./byte	ins./cycle
std	2.1	9.0	5.42
WebKit	10	0.95	2.74
Blink	10	0.93	2.65
64-bit	36	0.61	6.29

(c) Office – Graviton 4

function	GB/s	ins./byte	ins./cycle
std	0.90	8.7	2.82
WebKit	6.7	1.0	2.46
Blink	7.5	1.0	2.71
64-bit	22	0.66	5.27

(d) Office – Apple M2

function	GB/s	ins./byte	ins./cycle
std	2.0	9.0	5.12
WebKit	9.7	0.95	2.72
Chromium	9.7	0.93	2.57
64-bit	36	0.62	6.44

(e) Google – Graviton 4

function	GB/s	ins./byte	ins./cycle
std	0.93	7.7	2.53
WebKit	4.5	1.3	2.02
Chromium	5.1	1.2	2.02
64-bit	20	0.76	5.47

(f) Google – Apple M2

function	GB/s	ins./byte	ins./cycle
std	2.0	9.1	5.11
WebKit	6.9	1.1	2.21
Chromium	6.6	1.1	2.06
64-bit	33	0.70	6.62

6 | CONCLUSION

Our experiments suggest that both the WebKit and Chromium approaches are several times faster than a naive approach (`string::find_first_of`) for the HTML scanning problem. We also find that an approach that indexes 64 bytes can be several times faster once again. However, our approach cannot serve as a simple drop-in replacement: it would require some reengineering to keep track of the indexes. Future work should investigate its application.

The accelerated HTML parsing function is not the sole application of SIMD parsing in major Web engine. The WebKit engine had accelerated some functions having to do with JSON parsing while both the Chromium and the WebKit engine are considering faster routines for CSS parsing. Future work should examine these algorithms.

The HTML scanning problem is a specific task but it is generalizable. For example, the programmer might provide an arbitrary set of characters, and ask for a function to find all occurrences. Or they might be several classes of characters and we may want to have the ability to scan the text quickly while receiving information about the character types. In the HTML scanning scenario, the characters are part of the ASCII character set. The more general problem where the characters are arbitrary Unicode characters could prove more challenging.

Future work should consider other processor architectures. Our approach based on blocks of 64 bytes should be even more favorable on processors with wider SIMD registers (e.g., AVX-512). Some intructions sets (RISC-V and SVE/SVE2) have registers of sizes that are potentially unknown at compile-time. We might need new algorithms for such intructions sets.

references

[1] Johnson D, Apple Microarchitecture Research; 2023. https://dougallj.github.io/applecpu/firestorm.html [last checked August 2024].

[2] Nejati J, Balasubramanian A. An In-depth Study of Mobile Browser Performance. In: Proceedings of the 25th International Conference on World Wide Web WWW '16, Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee; 2016. p. 1305–1315. https://doi.org/10.1145/2872427.2883014.

[3] Nemerson E, SIMDe — SIMD Everywhere; 2024. https://github.com/simd-everywhere/simde [last checked August 2024].

[4] Wassenberg J, Highway — Efficient and performance-portable vector software; 2024. https://github.com/google/highway [last checked August 2024].

[5] John I, Speedometer 3; 2024. https://issackjohn.github.io/SpeedometerJSWebComponents/ [last checked August 2024].

[6] The Web Hypertext Application Technology Working Group, HTML; 2024. https://html.spec.whatwg.org/ [last checked August 2024].

[7] The Web Hypertext Application Technology Working Group, Infra — normalize newlines; 2024. https://infra.spec.whatwg.org/#normalize-newlines [last checked August 2024].

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

[9] Keiser J, Lemire D. Validating UTF-8 in less than one instruction per byte. Software: Practice and Experience 2021;51(5):950–964.

[10] Schröder M, Machmeier S, Maeng S, Heuveline V. Validating CESU-8 Encoded Text Utilising SIMD Instructions. In: Proceedings of the 2024 13th International Conference on Software and Computer Applications; 2024. p. 102–111.

[11] Nizipli Y, Lemire D. Parsing millions of URLs per second. Software: Practice and Experience 2024;To appear.

[12] Weimer F, Building Red Hat Enterprise Linux 9 for the x86-64-v2 microarchitecture level; 2021. https://developers.redhat.com/blog/2021/01/05/building-red-hat-enterprise-linux-9-for-the-x86-64-v2-microarchitecture-level [last checked August 2024].

[13] 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.

[14] ARM Limited, Arm Neoverse V2 Core Software Optimization Guide; 2021. https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-v2 [last checked August 2024].