

Fixing ill-formed UTF-16 strings with SIMD instructions

Robert Clausecker¹ | Daniel Lemire²

¹Zuse Institute Berlin, Germany

²Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence

Daniel Lemire, 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

UTF-16 is a widely used Unicode encoding representing characters with one or two 16-bit code units. The format relies on surrogate pairs to encode characters beyond the Basic Multilingual Plane, requiring a high surrogate followed by a low surrogate. Ill-formed UTF-16 strings—where surrogates are mismatched—can arise from data corruption or improper encoding, posing security and reliability risks. Consequently, programming languages such as JavaScript include functions to fix ill-formed UTF-16 strings by replacing mismatched surrogates with the Unicode replacement character (U+FFFD). We propose using Single Instruction, Multiple Data (SIMD) instructions to handle multiple code units in parallel, enabling faster and more efficient execution. Our software is part of the Google JavaScript engine (V8) and thus part of several major Web browsers.

1 | INTRODUCTION

Unicode is the standard for text representation in modern software, supporting over one million characters across diverse writing systems. Unicode assigns each character a unique code point from U+0000 to U+10FFFF, organized into 17 planes. The Basic Multilingual Plane (BMP, U+0000–U+FFFF) contains the most commonly used characters, while supplementary planes encode less frequent characters, such as ideographs or emojis. *Unicode Transformation Formats* (UTF) encode these characters in binary form. One of the primary Unicode Transformation Formats, UTF-16, is used by platforms such as Microsoft Windows, Java, and JavaScript for internal string representation. UTF-16 encodes characters in the Basic Multilingual Plane (BMP, U+0000–U+FFFF) using a single 16-bit code unit, while characters in supplementary planes (U+10000–U+10FFFF) are encoded as surrogate pairs: a high surrogate (U+D800–

```

size_t replace_invalid_surrogates(char16_t *dst, const char16_t *src, size_t src_size) {
    size_t src_idx = 0, dst_idx = 0;
    const char16_t replacement = 0xFFFD; /* U+FFFD Replacement Character */

    while (src_idx < src_size) {
        char16_t c = src[src_idx];

        /* not a surrogate */
        if (c < 0xD800 || c > 0xDFFF) {
            dst[dst_idx++] = c;
            src_idx++;
        } else if (c >= 0xD800 && c <= 0xDBFF) { /* high surrogate */
            /* check if there is a next unit and it is a valid low surrogate */
            if (src_idx + 1 < src_size && src[src_idx + 1] >= 0xDC00 && src[src_idx + 1] <= 0xDFFF) {
                /* Valid surrogate pair, copy both units */
                dst[dst_idx++] = c;
                dst[dst_idx++] = src[src_idx + 1];
                src_idx += 2;
            } else {
                /* invalid: lone high surrogate or no low surrogate */
                dst[dst_idx++] = replacement;
                src_idx++;
            }
        } else { /* low surrogate without high surrogate */
            dst[dst_idx++] = replacement;
            src_idx++;
        }
    }

    return dst_idx;
}

```

FIGURE 1 Scalar C function to replace invalid UTF-16 surrogates with the replacement character

U+DBFF) followed by a low surrogate (U+DC00–U+DFFF). The code point is computed from a surrogate pair (h, l) as:

$$\text{Code point} = ((h - 0xD800) \ll 10) + (l - 0xDC00) + 0x10000$$

where h is the high surrogate and l is the low surrogate.

Ill-formed UTF-16 strings occur when surrogate pairs are mismatched, such as a high surrogate not followed by a low surrogate or a low surrogate appearing without a preceding high surrogate. Such errors can result from data corruption, improper encoding conversions, or malicious inputs, potentially leading to security vulnerabilities or application crashes—[1, 2]. For example, the UTF-16 decoder of CPython 3.1 to 3.3 failed to update an internal bound after invoking its error handler—which is triggered by ill-formed sequences such as an unpaired surrogate—so that ill-formed input caused the decoder to read past the end of its buffer, disclosing process memory or crashing the application [2]. To mitigate these issues, mismatched surrogates are typically replaced with the Unicode replacement character (U+FFFD), ensuring robust text processing and signaling errors to developers or users.

Conventional scalar algorithms for fixing ill-formed UTF-16 strings process code units sequentially, as shown in Fig. 1. While straightforward, these methods are computationally expensive for large strings, limiting their suitability for high-throughput applications like web browsers or databases. Modern processors, including ARM and x64 architectures, support Single Instruction, Multiple Data (SIMD) instructions, enabling parallel processing of multiple data elements. We propose a SIMD-based algorithm to accelerate UTF-16 correction, leveraging ARM NEON and x64 Streaming SIMD Extensions (SSE) instructions to process 16-bit code units in blocks, achieving significant performance gains.

Our contributions include a novel SIMD algorithm for in-place and copy-based UTF-16 correction, implementations optimized for ARM NEON and x64 SSE, and a comprehensive experimental evaluation demonstrating up to eight-fold speedups over scalar methods. Our software is part of the open-source `simdutf` library, a widely used li-

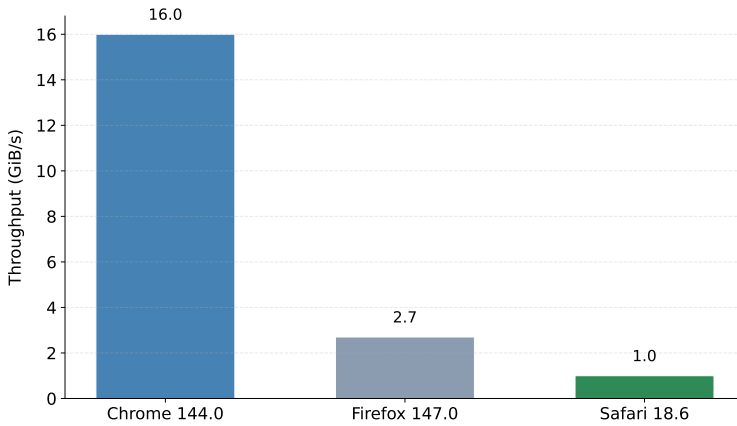


FIGURE 2 Speed of the `String.toWellFormed` JavaScript function using a 1024-character string using-on different browsers with an Apple M4 processor. The string is random, made of 5% low surrogates, 5% high surrogates, the rest being ASCII characters.

brary: it helps ensure reproducibility and practical applicability. Our method has been deployed in Chromium browsers (Chrome, Edge, Brave and others): Fig. 2 illustrates the superior performance (16 GiB/s, several times faster than competing browsers).¹

2 | RELATED WORK

Unicode processing has received attention for tasks like validation and transcoding, but correcting ill-formed UTF-16 strings is less explored. Keiser and Lemire [3] developed SIMD-based UTF-8 validation algorithms, processing multiple bytes in parallel. Lemire and Muła [4] proposed SIMD-accelerated UTF-8 to UTF-16 transcoding, achieving gigabytes-per-second throughput using ARM NEON and x64 SSE instructions. More recently, Schröder et al. [5] proposed a SIMD-based algorithm for validating CESU-8 encoded text, an encoding scheme combining UTF-8’s ASCII compatibility with UTF-16’s binary order. Their work, utilizing x86, ARM, and PowerPC SIMD instructions, achieves a sevenfold performance improvement over conventional validation methods, as demonstrated on datasets with ASCII, Hangul, and random text. These works focus on validation or format conversion, not in-place correction of UTF-16 strings.

Cameron [6] introduced bit-stream-based SIMD processing for UTF-8 to UTF-16 transcoding, but without addressing surrogate pair correction. Similarly, Lemire and Muła [4] proposed SIMD-accelerated transcoding from UTF-8 to UTF-16, achieving gigabyte-per-second throughput using ARM NEON and x64 SSE instructions. Clausecker and Lemire [7] further advanced transcoding performance by leveraging Advanced Vector Extensions 512 (AVX-512) instructions to process 512-bit registers, achieving over 5 GiB/s for UTF-8 to UTF-16 transcoding of Chinese text with fewer than two CPU instructions per character. However, these algorithms only do transcoding and not in-place correction.

Scalar UTF-16 validation is part of standard libraries (e.g., ICU, Boost), but these implementations process code units sequentially, lacking SIMD optimization. Our work builds on SIMD techniques from prior studies, adapting them

¹We make available an online benchmark at <https://lemire.github.io/browserwellformed/>.

to the specific problem of UTF-16 surrogate correction, and introduces novel optimizations for ARM NEON.

More broadly, SIMD instructions are used to accelerate many string operations. For exact string matching, a naive approach leveraging these instructions can be remarkably effective. As demonstrated by Tarhio et al., an algorithm that compares characters in a special order and utilizes SIMD instructions can outperform more complex conventional algorithms [8]. Similarly, Fiori et al. [9] and Chhabra et al. [10] introduce novel algorithms for the approximate matching problem between strings. We can also use SIMD instructions to parse strings more quickly, such as JSON strings [11, 12], XML [13], DNS records [14], and so forth.

3 | SIMD ALGORITHM

SIMD is a parallel processing technique that allows a single instruction to operate on multiple data elements simultaneously. The core motivation behind SIMD is to exploit data-level parallelism, enabling processors to perform the same operation—such as addition or comparison—on multiple data points in a single cycle, thereby reducing execution time. By processing multiple elements in parallel, SIMD not only accelerates computation but also improves power efficiency [15, 16], as it reduces the number of instruction cycles needed.

Our algorithm processes UTF-16 strings in blocks of at least 8 code units (16 bytes) using SIMD instructions, correcting mismatched surrogates by replacing them with U+FFFD. It supports both in-place correction (input equals output) and copy-based correction (input and output are distinct buffers). The algorithm checks for valid surrogate pairs by examining high and low surrogates across block boundaries, using a lookback mechanism to handle pairs spanning blocks.

3.1 | Architecture-independent algorithm

Our algorithm (Fig. 3) works in three steps: first, we load two vectors of 16-bit code units with an offset of one word from the input. The earlier vector (the *lookback*) is checked for the presence of high surrogates, the later vector (the *block*) for the presence of low surrogates, yielding two vectors of booleans *lb_is_high* and *block_is_low*. In the second step, we compute the element-wise exclusive-or of these two vectors of booleans. As the two vectors have an offset of one word, this checks if a high surrogate is not followed by a low surrogate or vice versa. If the result is all zeros, the current block is correctly sequenced and we copy it to the output.²

Otherwise we proceed to the third step, which involves detecting which specific 16-bit code units are illegally sequenced and replacing them with the Unicode replacement character U+FFFD prior to writing the block vector to the output buffer. We compute which high surrogates are not followed by a low surrogate by taking the element-wise and of *lb_is_high* with the complement of *block_is_low* and shifting the resulting vector to the right by one element (of two bytes) to have the result correspond to elements in *block* instead of *lookback*. Likewise, low surrogates not preceded by high surrogates are found by taking the element-wise and of the complement of *lb_is_high* with *block_is_low*. The element-wise or of these two vectors is taken to get *block_illseq*, indicating illegally sequenced code units in *block*, which are replaced with U+FFFD using a blend operation prior to writing them back.

We perform this algorithm on each vector-sized block of input in turn. As it is idempotent, a tail of less than the block size is handled by performing a final iteration aligned to the end of the input, possibly overlapping the penultimate iteration. Two cases (input starts with low surrogate or ends with high surrogate) are not caught by the vectorised procedure and must be taken care of manually. Likewise, input that is shorter than one vector and one

²When the processing is done in-place, the copy is omitted.

```

void utf16fix_block(char16_t *out, const char16_t *in) {
    const char16_t replacement = 0xFFFD; /* U+FFFD Replacement Character */
    using vector_u16 = simd16<uint16_t>;
    auto lookback = vector_u16::load(in - 1);
    auto block = vector_u16::load(in);
    auto lb_masked = lookback & 0xfc00;
    auto block_masked = block & 0xfc00;
    auto lb_is_high = lb_masked == 0xd800;
    auto block_is_low = block_masked == 0xdc00;
    auto illseq = lb_is_high ^ block_is_low;
    if (!illseq.is_zero()) { // can be implemented with movemask inst.
        /* compute the cause of the illegal sequencing */
        auto lb_illseq = ~block_is_low & lb_is_high;
        auto block_illseq =
            (~lb_is_high & block_is_low) | lb_illseq.byte_right_shift<2>();
        /* fix illegal sequencing in the lookback */
        const auto lb = lb_illseq.first();
        out[-1] = char16_t((lb & replacement) | (~lb & out[-1]));
        /* fix illegal sequencing in the main block */
        auto mask = as_vector_u16(block_illseq);
        auto fixed = (~mask & block) | (mask & replacement);
        fixed.store(out);
    } else {
        block.store(out);
    }
}

void to_well_formed(char16_t *dst, const char16_t *src, size_t n) {
    using vector_u16 = simd16<uint16_t>; // Our SIMD vector type
    constexpr size_t N = vector_u16::ELEMENTS; // Number of 16-bit elements
    // in the SIMD vector
    if (n < N + 1) {
        // short input, fallback
        replace_invalid_surrogates(src, n, dst);
        return;
    }
    const char16_t replacement = 0xFFFD; /* U+FFFD Replacement Character */
    dst[0] =
        is_low_surrogate(src[0]) ? replacement : src[0];
    for (size_t i = 1; i + N < n; i += N) {
        utf16fix_block(dst + i, src + i);
    }
    utf16fix_block(dst + n - N, src + n - N);
    dst[n - 1] = is_high_surrogate(dst[n - 1])
        ? replacement
        : dst[n - 1];
}

```

FIGURE 3 Generic SIMD function to replace invalid UTF-16 surrogates with the replacement character

element cannot be processed by our algorithm and must fall back to the scalar procedure from Fig. 1.

Example

We shall demonstrate the operation of `utf16fix_block` function of Fig. 3 with a vector length of 64 bytes on the input string

Hello, wor?ld!Hello, w?orld!Hello, wor?ld!Hello w?orld!He😊

where `[?]` is a mismatched surrogate. The UTF-16 string has a length of 31 code units and is surrounded by U+0000 on both sides. The value-values of the variables of `utf16fix_block` as it processes our string are shown in Fig. 4. The input, shown in the first row-column, contains mismatched surrogates D800 at position 10 and ~~DG000~~DC00 at position 21 marked in red. There is also a correctly sequenced surrogate pair D83D DE0A at positions 29–30 encoding U+1F60A smiling face with smiling eyes marked in green.

The lookback vector with its leading U+0000 is shown in the second row-column. The third and fourth rows-columns

Index	Input	lookback	lb_masked	block_masked	lb_is_high	block_is_low	illseq	Output
0	0048	0000	0000	0000	0	0	0	0048
1	0065	0048	0000	0000	0	0	0	0065
2	006C	0065	0000	0000	0	0	0	006C
3	006C	006C	0000	0000	0	0	0	006C
4	006F	006C	0000	0000	0	0	0	006F
5	002C	006F	0000	0000	0	0	0	002C
6	0020	002C	0000	0000	0	0	0	0020
7	0077	0020	0000	0000	0	0	0	0077
8	006F	0077	0000	0000	0	0	0	006F
9	0072	006F	0000	0000	0	0	0	0072
10	D800	0072	0000	D800	0	0	0	FFFD
11	006C	D800	D800	0000	1	0	1	006C
12	0064	006C	0000	0000	0	0	0	0064
13	0021	0064	0000	0000	0	0	0	0021
14	0048	0021	0000	0000	0	0	0	0048
15	0065	0048	0000	0000	0	0	0	0065
16	006C	0065	0000	0000	0	0	0	006C
17	006C	006C	0000	0000	0	0	0	006C
18	006F	006C	0000	0000	0	0	0	006F
19	0020	006F	0000	0000	0	0	0	0020
20	0077	0020	0000	0000	0	0	0	0077
21	DC00	0077	0000	DC00	0	1	1	FFFD
22	006F	DC00	DC00	0000	0	0	0	006F
23	0072	006F	0000	0000	0	0	0	0072
24	006C	0072	0000	0000	0	0	0	006C
25	0064	006C	0000	0000	0	0	0	0064
26	0021	0064	0000	0000	0	0	0	0021
27	0048	0021	0000	0000	0	0	0	0048
28	0065	0048	0000	0000	0	0	0	0065
29	D83D	0065	0000	D800	0	0	0	D83D
30	DE0A	D83D	D800	DC00	1	1	0	DE0A
31	0000	DE0A	DC00	0000	0	0	0	0000

FIGURE 4 Transposed-UTF-16 AVX-512 Values of the variables of `utf16fix_block` (Fig. 3) while processing the example string with a vector length of 64 bytes. The mismatched surrogates at indices 10 and 21 are marked in red, the correctly sequenced surrogate pair at indices 29–30 in green.

show `lookback` and `block` masked with `D600FC00`, from which are computed the `lb_is_high` and `block_is_low` masks ; showing-shown in the fifth and sixth columns, flagging high surrogates in `lookback` and low surrogates in `block` are computed. The exclusive-or of these masks forms the `illseq` vector, which indicates the presence, but not the precise location of mismatched surrogates. In this case, the surrogate `D800` at position 10 is incorrectly indicated at position 11. Correct location information is only determined if `illseq` is found to be not all zeros. The mismatched surrogates are then replaced with U+FFFD replacement character, leaving the correctly sequenced surrogates at position 29–30 alone.

Discussion

Three subtle design decisions influence the procedure. The first decision is to track high and low surrogates in two overlapping vectors of input instead of using one vector and shifting the resulting masks `lb_is_high` and `block_is_low` to find mismatches. The second decision is to perform two loads from the input buffer to obtain these overlapping

vectors *lookback* and *block* instead of performing only one load per iteration and slicing out *lookback* from the previous and the current *block* vectors. The third decision is to make the algorithm branchy,³ skipping the correction of illegally sequenced surrogates unless those actually appear.

We define the operational intensity of our main routine as the ratio of the number of arithmetic-logic SIMD instructions (e.g., comparisons, bitwise-AND) to the number of SIMD register loads and stores. In the best scenario, we need at least five arithmetic-logic SIMD instructions for each iteration compared to at most two load operations and up to one store operation.

Using overlapping vectors instead of shifting masks avoids having to carry over values from the previous iteration for “lookback” or “lookahead.” Loading twice instead of assembling *lookback* from two *block* vectors removes the need for shuffle operations at the cost of an extra load per iteration. On most processors, this is a good tradeoff. Indeed, current processors are often capable of retiring two or more SIMD load instructions per cycle [17]. Meanwhile, in the best scenario, we need at least five arithmetic-logic SIMD instructions (e.g., comparisons or bitwise-AND operations) for each iteration. Thus we have a relatively high *operational intensity* [18] even with two load operations per iteration: we have many more arithmetic-logic instructions than memory operations. Increasing the operational intensity further would diminish the performance.

A potential downside of loading SIMD registers with an offset of two bytes is that it is not possible to align the memory loads on natural alignment boundaries, i.e., using memory addresses divisible by the length of the register in bytes. However, most modern processor designs can handle unaligned SIMD load and store operations at little to no performance penalty.

Another side effect of our approach is that loop-carried dependencies are eliminated entirely, making it easier for the processor to overlap multiple iterations. In theory, given enough execution units, a processor could execute several iterations simultaneously.

The third choice is based on the expectation that almost all text processed by our procedure is correctly sequenced. Therefore, it is advantageous to skip the expensive step of determining the incorrect surrogates and fixing them up unless a quick check shows a need to do so. As a bonus, for in-place operation, writing to the buffer can be avoided entirely for valid UTF-16.

3.2 | AVX-512 (x64) implementation

Intel and AMD 64-bit processors (x86-64), which dominate the market for Windows PCs and servers, support a range of SIMD instruction set extensions, falling into the four families of MMX (64-bit vectors, of historical interest only), SSE (128-bit vectors), AVX (256-bit vectors), and AVX-512 (512-bit vectors). Within each family, there are multiple levels of support, with each level including all previous levels as well as all previous families. All x64 processors support at least SSE2, though it is generally advantageous to make use of the widest SIMD extension available to maximize the amount of data processed per operation. SSE and its extensions (SSE2, SSE3, SSSE3, SSE4.1, SSE4.2) use 128-bit XMM registers, capable of handling 16 bytes of data (e.g., four 32-bit floats or sixteen 8-bit integers). AVX and AVX2 introduce 256-bit YMM registers, doubling the capacity to 32 bytes (e.g., eight 32-bit floats). AVX-512 further expands to 512-bit ZMM registers, processing 64 bytes (e.g., sixteen 32-bit floats) and introduces new features like mask registers.

All these instruction sets provide fast comparison instructions capable of comparing chunks of 16-byte (SSE), 32-byte (AVX/AVX2), or 64-byte (AVX-512) data at the 16-bit granularity, making them well suited for working with

³A branching algorithm executes different code paths based on the input data for a given input size, while a branchless algorithm consistently follows the same code path regardless of the data content.

```

void utf16fix_block(char16_t *out, const char16_t *in) {
    const char16_t replacement = 0xFFFD;
    __m512i lookback, block, lb_masked, block_masked;
    __mmask32 lb_is_high, block_is_low, illseq;
    lookback = __mm512_loadu_si512(in - 1);
    block = __mm512_loadu_si512(in);
    lb_masked =
        __mm512_and_epi32(lookback, __mm512_set1_epi16(0xFC00));
    block_masked =
        __mm512_and_epi32(block, __mm512_set1_epi16(0xFC00));

    lb_is_high = __mm512_cmpeq_epi16_mask(
        lb_masked, __mm512_set1_epi16(0xD800));
    block_is_low = __mm512_cmpeq_epi16_mask(
        block_masked, __mm512_set1_epi16(0xDC00));
    illseq = _kxor_mask32(lb_is_high, block_is_low);
    if (!_ktestz_mask32_u8(illseq, illseq)) {
        __mmask32 lb_illseq, block_illseq;
        /* compute the cause of the illegal sequencing */
        lb_illseq = _kandn_mask32(block_is_low, lb_is_high);
        block_illseq = _kor_mask32(_kandn_mask32(lb_is_high, block_is_low),
            _kshiftri_mask32(lb_illseq, 1));

        /* fix illegal sequencing in the lookback */
        lb_illseq = _kand_mask32(lb_illseq, _cvtu32_mask32(1));
        __mm512_mask_storeu_epi16(out - 1, lb_illseq,
            __mm512_set1_epi16(replacement));

        /* fix illegal sequencing in the main block */
        __mm512_storeu_epi32(
            out, __mm512_mask_blend_epi16(block_illseq, block,
                __mm512_set1_epi16(replacement)));
    } else {
        __mm512_storeu_si512(out, block);
    }
}

```

FIGURE 5 AVX-512 function to replace invalid UTF-16 surrogates with the replacement character within a 64-byte block

UTF-16 data. In practice, the variety of instruction sets available on x64 processors often requires runtime detection of supported instruction sets. For example, the first time the processing is initiated, we might check for the supported CPU features and pick one out of several precompiled functions.

For instruction sets prior to AVX-512, comparison instructions generate results in SIMD registers as either all-ones (0xFF...FF) or all-zeros (0x00...00) per element, indicating true or false outcomes. These results can be efficiently mapped to a general-purpose register acting as a bitset, where each bit corresponds to a comparison result from the SIMD register. For example, with the SSE2 `pcmpeqw` instruction, comparing two 16-byte vectors produces a 128-bit XMM register with 16 bytes, where each 16-bit subword is either 0xFFFF or 0x0000 depending on whether the comparison was true. Using the SSE2 `pmovmskb` instruction, these bytes are converted into a 16-bit integer, where each bit represents one byte's comparison result. This bitset can then be processed in scalar code. Example: comparing two vectors of 16 bytes might yield an XMM register with bytes [0xFF, 0xFF, 0x00, 0x00, 0xFF, 0xFF, ...], and we can map this to a 16-bit value like 0b110011..., where 1 indicates equal bytes. The process is much the same for AVX2: e.g., the AVX2 `vpmovmskb` instructions act like the SSE2 `pmovmskb` instruction but produce a 32-bit register value instead of a 16-bit register value.

AVX-512 introduces dedicated mask registers (k0–k7), which replace the need for `movemask` operations (`pmovmskb` and `vpmovmskb`) in many cases. These 8-bit to 64-bit mask registers directly store comparison results, with each bit corresponding to an element in the 512-bit ZMM register. For instance, a comparison like the `vpcmpw` instructions on two 512-bit vectors (thirty-two 16-bit integers) produces a 32-bit mask in an AVX-512 mask register, where each bit indicates equality for one integer. These masks can be used directly in subsequent AVX-512 instructions for conditional operations or merged with general-purpose registers for scalar processing. This eliminates the overhead of

movemask instructions, with the caveat that operations on the mask registers are limited (e.g., there are no trailing or counting zero instructions).

Architecture-specific adjustments

With SSE2 and AVX2, the generic procedure can be used without changes. It is also applicable to AVX-512, but benefits from using a specific `utf16fix_block` function with some architecture-specific changes, see Fig. 5. The AVX-512 implementation of the `utf16fix_block` function leverages 512-bit ZMM registers to process thirty-two UTF-16 code units (64 bytes) in parallel, utilizing AVX-512's mask registers for efficient conditional operations. Other than being cast in architecture-specific intrinsics, these are the differences in `utf16fix_block` procedure compared to our generic algorithm:

- instead of using vectors of FFFF or 0000 for boolean results, we use mask registers,
- the various blend operations are realised by mask-driven blend instructions and masked stores instead of bit-operations,
- instead of fixing up `out[-1]` by extracting `lb` from `lb_illseq` into a scalar, we perform a 64 byte store into `out - 1` masked with just the first bit of `lb_illseq`, guaranteeing that it stores either one code unit or none and in particular does not overlap the subsequent store to `out`.

For simplicity, we omit the in-place operation. In production, the case of `in = out` is sped up by replacing the blend step in the correction path with a straight masked store to only the illegally sequenced surrogates and by leaving out the store in the fast path entirely.

Table 1 summarizes the SIMD instructions and their corresponding intrinsics used for UTF-16 processing, as detailed in the provided code and discussion. The table lists each instruction, its associated intrinsic, the instruction set (SSE2, AVX2, or AVX-512), and a brief description of its functionality. Instructions like `pcmpeqw` and `pmovmskb` from SSE2 handle 128-bit vectors, while AVX2's `vpmovmskb` extends to 256-bit vectors. AVX-512 instructions, such as `vpcmpw` and `vpblendmw`, leverage 512-bit ZMM registers and mask registers for efficient 32-element processing, with intrinsics like `_mm512_cmpeq_epi16_mask` and `_kxor_mask32` enabling surrogate pair validation.

3.3 | NEON (Aarch64) implementation

Aarch64 processors are common on mobile devices like smartphones and tablets as well as appliances such as Smart TVs and videogame consoles. Recently, they are becoming more and more common in the server market and, with Apple switching to Aarch64, also on desktops and laptops. Two SIMD instruction set extensions are specified for Aarch64. The NEON extension taken over and extended from the 32 bit ARM architecture provides 128-bit registers capable of processing eight 16-bit words simultaneously. Its feature set is comparable to that of SSSE3 with a full set of integer and floating point operations as well as loads, stores, and shuffles. NEON support is mandatory on all Aarch64 processors with few exceptions and thus the most common target for SIMD-acceleration on Aarch64. The newer SVE/SVE2/SVE2.1 family of instruction set extensions builds on NEON and extends it with variable-length vector registers, predicate mask and many other features, rendering it similar to AVX-512. Unfortunately, adoption of SVE has been slow, with almost all OEMs opting to ship CPUs that have no support. We have therefore decided to ignore SVE and use NEON.

In contrast to x64, hardware supporting NEON has diverse execution characteristics. A low-power Aarch64 CPU might only be able to execute one NEON instruction every three cycles. High-power CPUs, on the other hand, have multiple execution units, often more than x64 processors. The resulting higher throughput compensates for the shorter vector length. For example, the Firestorm Cores of Apple M1 processors can execute four of most NEON

TABLE 1 x64 SIMD instructions and intrinsics for UTF-16 processing

Instruction	Intrinsic	Instruction Set	Description
pcmpeqw	<code>_mm_cmpeq_pi16</code>	SSE2	Compares 16-bit integers in two 128-bit vectors, setting each 16-bit element to 0xFFFF (true) or 0x0000 (false) if equal.
pmovmskb	<code>_mm_movemask_pi8</code>	SSE2	Extracts the most significant bit of each byte in a 128-bit vector, producing a 16-bit integer bitmask.
vpmovmskb	<code>_mm256_movemask_epi8</code>	AVX2	Extracts the most significant bit of each byte in a 256-bit vector, producing a 32-bit integer bitmask.
vpcmpw	<code>_mm512_cmpeq_epi16_mask</code>	AVX-512	Compares 16-bit integers in two 512-bit vectors, producing a 32-bit mask where each bit indicates equality.
vmovdqu	<code>_mm512_loadu_si512</code>	AVX-512	Loads 512 bits of unaligned data into a ZMM register.
vpandd	<code>_mm512_and_epi32</code>	AVX-512	Performs a bitwise AND on two 512-bit vectors, treating data as 32-bit integers.
kxord	<code>_kxor_mask32</code>	AVX-512	Performs a bitwise XOR on two 32-bit mask registers.
ktestzq	<code>_ktestz_mask32_u8</code>	AVX-512	Tests if all bits in a 32-bit mask are zero, returning true if so.
kandnw	<code>_kandn_mask32</code>	AVX-512	Performs a bitwise AND NOT on two 32-bit mask registers (inverts first mask, then ANDs).
korw	<code>_kor_mask32</code>	AVX-512	Performs a bitwise OR on two 32-bit mask registers.
kshiftrw	<code>_kshiftri_mask32</code>	AVX-512	Shifts a 32-bit mask right by one bit, inserting a zero at the most significant bit.
kmovw	<code>_cvtu32_mask32</code>	AVX-512	Converts an unsigned 32-bit integer to a 32-bit mask register.
vmovdqu16	<code>_mm512_mask_storeu_epi16</code>	AVX-512	Stores 16-bit integers from a 512-bit vector to unaligned memory, using a 32-bit mask to select elements.
vmovdqu	<code>_mm512_storeu_si512</code>	AVX-512	Stores a 512-bit vector to unaligned memory.
vpblendmw	<code>_mm512_mask_blend_epi16</code>	AVX-512	Blends 16-bit integers from two 512-bit vectors based on a 32-bit mask.
vmovdqu32	<code>_mm512_storeu_epi32</code>	AVX-512	Stores 32-bit integers from a 512-bit vector to unaligned memory.
vpbroadcastw	<code>_mm512_set1_epi16</code>	AVX-512	Broadcasts a single 16-bit integer to all elements of a 512-bit vector.
vpbroadcastd	<code>_mm512_set1_epi32</code>	AVX-512	Broadcasts a single 32-bit integer to all elements of a 512-bit vector.

```

int veq_non_zero(uint8x16_t v) { return vmaxvq_u32(vreinterpretq_u32_u8(v)); }
void utf16fix_block(char16_t *out, const char16_t *in) {
    // similar to x64/generic
}
uint8x16_t get_mismatch_copy(const char16_t *in, char16_t *out) {
    uint8x16x2_t lb = vld2q_u8((const uint8_t *)in - 1);
    uint8x16x2_t block = vld2q_u8((const uint8_t *)in);
    uint8x16_t lb_masked = vandq_u8(lb.val[1], vdupq_n_u8(0xfc));
    uint8x16_t block_masked = vandq_u8(block.val[1], vdupq_n_u8(0xfc));
    uint8x16_t lb_is_high = vceqq_u8(lb_masked, vdupq_n_u8(0xd8));
    uint8x16_t block_is_low = vceqq_u8(block_masked, vdupq_n_u8(0xdc));
    uint8x16_t illseq = veorq_u8(lb_is_high, block_is_low);
    vst2q_u8((uint8_t *)out, block);
    return illseq;
}
uint64_t get_mask(uint8x16_t illse0, uint8x16_t illse1,
                  uint8x16_t illse2, uint8x16_t illse3) {
    uint8x16_t bit_mask = {0x01, 0x02, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80,
                          0x01, 0x02, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80};
    uint8x16_t sum0 =
        vpaddq_u8(vandq_u8(illse0, bit_mask), vandq_u8(illse1, bit_mask));
    uint8x16_t sum1 =
        vpaddq_u8(vandq_u8(illse2, bit_mask), vandq_u8(illse3, bit_mask));
    sum0 = vpaddq_u8(sum0, sum1);
    sum0 = vpaddq_u8(sum0, sum0);
    return vgetq_lane_u64(vreinterpretq_u64_u8(sum0), 0);
}
bool utf16fix_block64(char16_t *out, const char16_t *in) {
    const char16_t replacement= 0xFFFF; /* U+FFFD Replacement Character */
    uint8x16_t illse0 = get_mismatch_copy(in, out);
    uint8x16_t illse1 = get_mismatch_copy(in + 16, out + 16);
    uint8x16_t illse2 = get_mismatch_copy(in + 32, out + 32);
    uint8x16_t illse3 = get_mismatch_copy(in + 48, out + 48);
    if (veq_non_zero(
        vorrq_u8(vorrq_u8(illse0, illse1), vorrq_u8(illse2, illse3)))) {
        uint64_t matches = get_mask(illse0, illse1, illse2, illse3);
        while (matches != 0) {
            int r = stdc_trailing_zeros_ull(matches); // generates rbit + clz
            bool is_high = is_high_surrogate(in[r - 1]);
            out[r - is_high] = replacement;
            matches = (matches & (matches - 1)); // clear least significant bit
        }
        return false;
    }
    return true;
}
void to_well_formed(char16_t *dst, const char16_t *src, size_t n) {
    const char16_t replacement= 0xFFFF; /* U+FFFD Replacement Character */
    if (n < 17) {
        return replace_invalid_surrogates(src, n, dst);
    }
    dst[0] = is_low_surrogate(src[0]) ? replacement : src[0];
    size_t i = 1;
    for (i = 1; i + 64 < n; i += 64) { utf16fix_block64(dst + i, src + i); }
    for (; i + 16 < n; i += 16) { utf16fix_block(dst + i, src + i); }
    utf16fix_block(dst + n - 16, src + n - 16);
    dst[n - 1] = is_high_surrogate(dst[n - 1]) ? replacement : dst[n - 1];
}

```

FIGURE 6 ARM NEON implementation for UTF-16 correction

instructions per cycle [19].

Architecture-specific adjustments

Fig. 6 shows our NEON implementation. While it is possible to use the generic algorithm (Fig. 3) on NEON, there are two important changes to be made that significantly improve performance. First, it should be noted that NEON does not have an equivalent to SSE's and AVX' mask-moving instructions such as 'pmovmskb'. It also lacks a dedicated instruction to check that a register is non-zero. We found an efficient workaround to check that a 128-bit value is non-zero by computing a vertical maximum (maximum over all elements) using an instruction such as `vmaxvq_u32` and then moving the result to a general-purpose register, which holds zero if and only if all elements of the vector were zero. The performance of these instructions varies depending on the chosen hardware, but both instructions can have several cycles of latency. Therefore, this step can become a bottleneck. We can implement a `utf16fix_block` function which works similarly to the x64 function, but the expected performance is not ideal.

We amortize this cost by processing 4 blocks in parallel and only then checking if any of the blocks require correction. Only if this is the case, do we branch to fix up the illegal surrogates. As we expect most inputs to not require correction (most UTF-16 inputs are valid), we can alleviate the problem by processing the data in larger blocks, spanning 64 code points. As a side effect, the instruction-level parallelism is increased, benefiting microarchitectures with many execution ports like the Apple M1.

The other important change is to make use of the NEON-exclusive deinterleaving load instruction `v1d2q_u8`. The LD2 instruction loads two vectors' worth of bytes from memory, writing the even-numbered bytes to one register and the odd-numbered bytes to another.⁴ With `v1d2q_u8`, it is advantageous to treat UTF-16 code units as pairs of bytes. As the more significant of the two bytes in a UTF-16 code unit suffices to tell if the code unit is a high surrogate, low surrogate, or neither, we use `v1d2q_u8` to load 16 code units into a pair of vectors, and then discard the vector of the less significant bytes as its contents are not required to carry out the algorithm. We carry on with just the high-byte vector, having now reduced the element size from 16 to 8 bits, doubling the number of code units processed per step. As a side effect, having discarded the low-byte vector, we cannot fix up the vectors as in Fig. 3, and use a different approach instead.

Like with the other implementations, the core function, `to_well_formed`, orchestrates the processing of a UTF-16 string of length n . It begins by handling short inputs ($n < 17$) with a scalar fallback. For longer inputs, it ensures the first output element is not an invalid low surrogate by replacing it with `U+FFFD` if necessary. The function then processes the string in blocks of 64 code points using `utf16fix_block64` when possible, falling back to blocks of 16 code points with `utf16fix_block` for smaller remaining segments. A final 16-code-point block is processed to cover the string's end, and the last element is checked to ensure it is not an invalid high surrogate, replacing it with `U+FFFD` if needed.

The `utf16fix_block64` function processes sixty-four UTF-16 code units (128-bytes) by dividing the block into four 16-code-unit segments, each handled by `get_mismatch_copy`. This helper function loads two 128-bit vectors: `lb` (lookback, from $in - 1$) and `block` (from in), using `v1d2q_u8` to deinterleave bytes into high and low parts. The function then masks the high bytes with `0xFC` to focus on the top 6 bits, identifying high surrogates (`0xD8`) and low surrogates (`0xDC`) via a comparison instruction `vceq_u8`. An exclusive-or operation (`veorq_u8`) detects illegal sequences (e. g., a high surrogate not followed by a low surrogate). The function copies the input block to the output using `vst2q_u8` and returns the illegal sequence mask. In `utf16fix_block64`, four such masks (`illse0` to `illse3`) are combined using `vorrq_u8` and checked for non-zero status with `veq_non_zero`, which uses `vmaxvq_u32` to compute the maximum across a 128-bit vector, efficiently detecting any invalid sequences.

If invalid sequences are detected, `get_mask` converts the four 128-bit masks into a 64-bit bitset, where each bit

⁴A companion instruction `vst2q_u8` can undo this transformatin on store, but is not needed here.

corresponds to a code unit's validity. It uses a predefined *bit_mask* vector to extract specific bits, accumulating results with *vpaddq_u8* to produce a single 64-bit value via *vgetq_lane_u64*. This bitset is processed scalarly, using *stdc_trailing_zeros_u11* to find invalid positions, determining whether each is a high or low surrogate, and replacing the appropriate code unit with *U+FFFD*. The function returns false to indicate corrections were made, or true if the block was valid and copied unchanged. The *utf16fix_block* function is omitted for simplicity; it works similarly to the generic implementation. The main difference being that it uses interleaved loads.

Table 2 lists the ARM NEON instructions and their corresponding intrinsics used in the UTF-16 processing code. These instructions operate on 128-bit NEON registers, enabling parallel processing of eight 16-bit words. Each entry includes the instruction, its intrinsic, and a concise description of its role in validating and correcting UTF-16 surrogate pairs.

TABLE 2 ARM NEON instructions and intrinsics for UTF-16 processing

Instruction	Intrinsic	Description
LD2	<i>vld2q_u8</i>	Loads 256 bits of unaligned data from memory, deinterleaving into two 128-bit vectors of 8-bit integers.
ST2	<i>vst2q_u8</i>	Stores two 128-bit vectors of 8-bit integers to unaligned memory, interleaving the data.
AND	<i>vandq_u8</i>	Performs a bitwise AND on two 128-bit vectors of 8-bit integers.
CMEQ	<i>vceqq_u8</i>	Compares two 128-bit vectors of 8-bit integers for equality, producing a 128-bit vector with 0xFF for true and 0x00 for false per element.
EOR	<i>veorq_u8</i>	Performs a bitwise XOR on two 128-bit vectors of 8-bit integers.
DUP	<i>vdupq_n_u8</i>	Broadcasts a single 8-bit integer to all elements of a 128-bit vector.
UMAXV	<i>vmaxvq_u32</i>	Computes the maximum value across all 32-bit elements in a 128-bit vector, returning a 32-bit scalar.
ADDP	<i>vpaddq_u8</i>	Pairwise adds 8-bit integers from two 128-bit vectors, producing a 128-bit vector of 8-bit sums.
FMOV	<i>vgetq_lane_u64</i>	Extracts a 64-bit lane from a 128-bit vector, interpreting the vector as two 64-bit integers.
n/a	<i>vreinterpretq_u64_u8</i>	Reinterprets a 128-bit vector of 8-bit integers as a 128-bit vector of 64-bit integers without changing the data.
n/a	<i>vreinterpretq_u32_u8</i>	Reinterprets a 128-bit vector of 8-bit integers as a 128-bit vector of 32-bit integers without changing the data.

4 | EXPERIMENTS

To evaluate the performance of UTF-16 validation and correction algorithms, we developed a benchmarking tool in C++ (*benchmark_to_well_formed_utf16.cpp*). It is part of the *simutf* library⁵. We build the benchmarking soft-

⁵<https://github.com/simutf/simutf>, commit hash dae7a10

TABLE 3 Systems used for benchmarking

Processor	Apple M4	Intel Xeon Gold 6338
Frequency	4.4 GHz to 4.5 GHz	3.2 GHz
Microarchitecture	M4 (aarch64, 2024)	Ice Lake (x64, 2019)
Memory	LPDDR5X (7500 MT/s)	DDR4 (3200 MT/s)
Compiler	Apple/LLVM 17	GCC 12

ware and the C++ library in release mode (`-O3 -NDEBUG`). On x64 processors, the `simdutf` library uses specialized kernels optimized for different Intel processor microarchitectures, including icelake, haswell, and westmere. The icelake kernel is tailored for Intel's Ice Lake processors (AVX-512 with VBM12). The haswell kernel is optimized for the Haswell microarchitecture with AVX2 support. The westmere kernel targets older Westmere processors (SSE4.2).

We present the systems we use for benchmarking in Table 3. This table details the key specifications of each system, including processor type, clock frequency, microarchitecture, memory configuration, and compiler version. The selected systems represent a mix of modern high-performance architectures, allowing for a comprehensive evaluation of performance across different workloads and computational environments.

We generate random UTF-16 strings with controlled characteristics. Specifically, we configure the input strings with a specified percentage of valid surrogate pairs (code units U+D800 to U+DBFF followed by U+DC00 to U+DFFF) and mismatched surrogates (isolated high or low surrogates). Specifically, if we set the percentage of surrogate pairs to 0.1 %, then 0.1 % of all characters involve surrogate pairs. If we set the percentage of mismatched surrogates to 0.1 %, then 0.1 % of the code units outside the valid surrogate pairs are randomly chosen (50 %) high or low surrogates. It is possible, but unlikely, that some isolated surrogates might form valid pairs (i.e., for 0.1 % the probability is 0.01 %).

The benchmark measures the throughput (in GB/s) and hardware performance counters, such as instructions per byte and cycles per byte, for different implementations, including a baseline from the V8 JavaScript engine and optimized versions using the `simdutf` library. The V8 code was replaced by the `simdutf` library in recent versions. The experiments are conducted with input sizes of up to 1 000 000 code units.

We conducted experiments with two distinct configurations to analyze the algorithms' behavior under different conditions. In the first configuration, we set the input size to 1 000 000 code units, with 0.1 % of the code units forming valid surrogate pairs and 0 % as mismatched surrogates. This setup represents a scenario with minimal supplementary plane characters, focusing on basic UTF-16 characters. In the second configuration, we maintained the same input size but adjusted the parameters to include 0.1 % valid surrogate pairs and 0.1 % mismatched surrogates, introducing a small proportion of invalid UTF-16 sequences. These configurations were chosen to evaluate the performance of the algorithms in both nearly valid and slightly erroneous UTF-16 inputs, providing insights into their efficiency and error-handling capabilities.

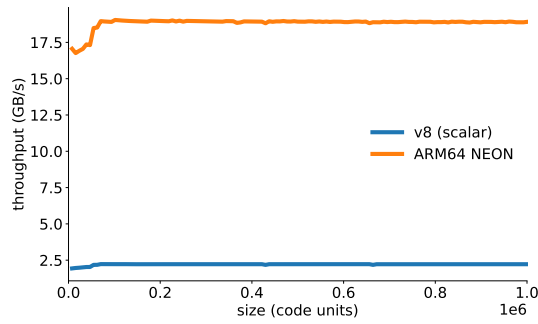


FIGURE 7 Throughput on the Apple system for 0 to a million code units

In addition to the fixed-size experiments, we performed benchmarks across a range of input sizes to assess scalability. The framework divides the maximum input size (1 000 000 code units) into 128 nearly equal chunks, testing each chunk independently. This approach ensures a fine-grained analysis of performance trends as the input size increases. For each chunk, the benchmark generates a random UTF-16 string with the specified surrogate pair and mismatched surrogate percentages (0.1 % and 0 % or 0.1 % and 0.1 %, depending on the configuration). The performance metrics, including throughput and error margins, are collected for each implementation, allowing us to compare their behavior across different input scales and identify any size-dependent bottlenecks. We present this result in Fig. 7 for the Apple platform. We see that the scalar version (V8) and the simdutf version (ARM64) have consistent speeds throughout the range, although they are both slightly slower on a per GB/s basis for tiny strings. The simdutf function is nearly 9 times faster than the scalar function.

Each implementation is tested over 100 iterations, with the input data processed multiple times to achieve measurable execution times (at 1 ms). The software uses an event collector to capture both elapsed time and hardware counters, such as CPU cycles and instructions, when available. The throughput is calculated as the input size divided by the best execution time (in nanoseconds), reported in GB/s. Additionally, we compute the error margin as the percentage difference between the average and best execution times, providing a measure of performance stability. We find a low error margin (about 1%).

The performance results for the Apple and Intel systems are summarized in Table 4 and Table 5, respectively. Table 4 shows the throughput (GB/s), instructions per byte, and instructions per cycle for the V8 baseline and the optimized arm64 implementation on the Apple M4 system, under both 0 % and 0.1 % mismatched surrogate conditions. The arm64 implementation significantly outperforms the V8 baseline, achieving up to 18.9 GB/s with an error rate of 0 % compared to 2.2 GB/s for V8, demonstrating the effectiveness of SIMD optimizations. Table 5 presents similar metrics for the Intel Xeon Gold 6338 system, comparing the V8 baseline with optimized implementations for Ice Lake, Haswell, and Westmere microarchitectures. The Ice Lake implementation achieves the highest throughput at 7.5 GB/s with 0 % errors, while the V8 baseline lags at 1.2 GB/s. Across both systems, the optimized implementations exhibit lower instruction counts per byte and higher efficiency, particularly in error-free scenarios, highlighting the benefits of architecture-specific SIMD optimizations. The performance comparison between icelake and haswell kernels, in Table 5, reveals distinct differences in throughput and efficiency on the Intel Xeon Gold 6338 system. The icelake kernel achieves a throughput of 7.5 GB/s at 0 % error rate, slightly below Haswell's 7.8 GB/s under the same condition. However, the icelake kernel demonstrates superior efficiency, with a lower instruction count per byte (0.4 versus haswell's 0.8) and significantly lower instructions per cycle (1.0 compared to haswell's 1.8).

TABLE 4 Performance metrics for Apple systems at size 1 000 000

Error	V8		arm64	
	0 %	0.1 %	0 %	0.1 %
GB/s	2.2	2.2	18.9	16.3
ins/byte	12.0	12.0	0.9	0.9
ins/cycle	5.9	5.8	3.7	3.3

TABLE 5 Performance metrics for Intel systems at size 1 000 000

Error	V8		icelake		haswell		westmere	
	0 %	0.1 %	0 %	0.1 %	0 %	0.1 %	0 %	0.1 %
GB/s	1.2	1.2	7.5	7.4	7.8	7.6	5.8	5.6
ins/byte	13.0	13.0	0.4	0.4	0.8	0.8	2.0	2.0
ins/cycle	5.0	5.0	1.0	1.0	1.8	1.8	3.6	3.5

TABLE 6 Performance in Node.js of the toWellFormed function (random 1024-character string)

Node.js version	Apple M4	Intel Xeon Goald 6338
24.13.0	2.9 GiB/s	2.0 GiB/s
25.5.0	16 GiB/s	11 GiB/s

4.1 | JavaScript Performance

Our C++ functions have been integrated in the popular Node.js JavaScript runtime engine as of version 25. Prior versions relied a conventional implementation of the toWellFormed function. Using a 1024-character random string made of 5% of low surrogate, 5% high surrogate and otherwise filled with ASCII characters, we get the results in Table 6.⁶ Under both systems, we find a five-fold performance gain, consistent with our C++ benchmarks.

5 | CONCLUSION

By leveraging SIMD instructions, our proposed algorithm achieves significant performance improvements over traditional scalar methods, with up to eightfold speedups on ARM NEON and x64 SSE architectures. Experimental results on Apple M4 and Intel Xeon systems confirm the scalability and effectiveness of our approach, with throughputs reaching 18.9 GB/s on Apple M4 processor and 7.5 GB/s on an Intel Ice Lake processor, alongside reduced instruction counts.

⁶See <https://github.com/lemire/browserwellformed> for our JavaScript benchmark code.

6 | FUNDING

This research was funded by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN-2024-03787 and by the NHR e.V. Graduate School Programme.

references

- [1] Davis M, Suignard M, Unicode Technical Report #36: Unicode Security Considerations; 2014. Unicode Consortium. <https://www.unicode.org/reports/tr36/> [last checked August 2026].
- [2] NIST National Vulnerability Database, CVE-2012-2135: buffer over-read in the UTF-16 decoder of CPython 3.1 through 3.3; 2012. <https://nvd.nist.gov/vuln/detail/CVE-2012-2135> [last checked August 2026].
- [3] Keiser J, Lemire D. Validating UTF-8 in less than one instruction per byte. *Software: Practice and Experience* 2021;51(5). doi:10.1002/spe.2920.
- [4] Lemire D, Muła W. Transcoding billions of Unicode characters per second with SIMD instructions. *Software: Practice and Experience* 2022;52(2):555–575. doi:10.1002/spe.3036.
- [5] 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.
- [6] 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. doi:10.1145/1345206.1345222.
- [7] Clausecker R, Lemire D. Transcoding unicode characters with AVX-512 instructions. *Software: Practice and Experience* 2023;53(12):2430–2462.
- [8] Tarhio J, Holub J, Giaquinta E. Technology beats algorithms (in exact string matching). *Software: Practice and Experience* 2017;47(12):1877–1885.
- [9] Fiori FJ, Pakalén W, Tarhio J. Approximate string matching with SIMD. *The Computer Journal* 2022;65(6):1472–1488.
- [10] Chhabra T, Ghuman SS, Tarhio J. String searching with mismatches using AVX2 and AVX-512 instructions. *Information Processing Letters* 2025;189:106557.
- [11] Keiser J, Lemire D. On-demand JSON: A better way to parse documents? *Software: Practice and Experience* 2023;To appear.
- [12] Langdale G, Lemire D. Parsing gigabytes of JSON per second. *The VLDB Journal* 2019;28(6):941–960.
- [13] 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: Association for Computing Machinery; 2008. <https://doi.org/10.1145/1463788.1463811>.
- [14] Koekkoek J, Lemire D. Parsing Millions of DNS Records Per Second. *Software: Practice and Experience* 2025;55(4):778–788. <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3396>.
- [15] Inoue H. How SIMD width affects energy efficiency: A case study on sorting. In: *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*; 2016. p. 1–3.
- [16] Wassenberg J, Blacher M, Giesen J, Sanders P. Vectorized and performance-portable quicksort. *Software: Practice and Experience* 2022;52(12):2684–2699.

- [17] Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. In: Optimization Manuals, vol. 4 published online at <https://agner.org/optimize>; 2025.
- [18] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. Commun ACM 2009 Apr;52(4):65–76. <https://doi.org/10.1145/1498765.1498785>.
- [19] Johnson D, Apple Microarchitecture Research; 2023. <https://dougallj.github.io/applecpu/firestorm.html> [last checked August 2024].