arXiv:2601.06349v1 [cs.OH] 9 Jan 2026

使用 SIMD 指令修复格式错误的 UTF-16 字符串

Robert Clausecker* Daniel Lemire†

2026年1月13日

UTF-16 是一种广泛使用的 Unicode 编码，通过一个或两个 16 位代码单元表示字符。该格式依赖代理项对来编码基本多文种平面之外的字符，要求先出现高代理项，随后出现低代理项。格式错误的 UTF-16 字符串——其中代理项彼此不匹配——可能源于数据损坏或编码不当，从而带来安全性和可靠性风险。因此，JavaScript 等编程语言包含用于修复格式错误的 UTF-16 字符串的函数，方法是将不匹配的代理项替换为 Unicode 替换字符（U+FFFD）。我们提出使用单指令多数据（SIMD）指令并行处理多个代码单元，从而实现更快速、更高效的执行。我们的软件是 Google JavaScript 引擎（V8）的一部分，因此也属于多个主流网页浏览器的一部分。

1 引言

Unicode 是现代软件中文本表示的标准，支持跨多种书写系统的一百多万个字符。Unicode 为每个字符分配一个唯一的码点，范围从 U+0000 到 U+10FFFF，并划分为 17 个平面。基本多文种平面（BMP，U+0000–U+FFFF）包含最常用的字符，而补充平面则用于编码较少见的字符，例如表意文字或表情符号。UTF-16 是其主要编码之一，Microsoft Windows、Java 和 JavaScript 等平台都使用它来表示内部字符串。UTF-16 使用单个 16 位代码单元编码基本多文种平面（BMP，U+0000–U+FFFF）中的字符，而补充平面（U+10000–U+10FFFF）中的字符则编码为代理项对：一个高代理项（U+D800–U+DBFF），后接一个低代理项（U+DC00–U+DFFF）。代理项对（h，l）所对应的码点计算如下：

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

*clausecker@zib.de，柏林蔡泽研究所，德国
†daniel.lemire@teluq.ca，魁北克大学（TELUQ），蒙特利尔，魁北克，H2S 3L5，加拿大

其中， h 是高代理项， l 是低代理项。

当代理项对不匹配时，就会出现格式错误的 UTF-16 字符串，例如高代理项后面没有低代理项，或者低代理项前面没有高代理项。这些错误可能由数据损坏、不当的编码转换或恶意输入导致，并可能引发安全漏洞或应用程序崩溃。为缓解这些问题，通常会将不匹配的代理项替换为 Unicode 替换字符（U+FFFD），从而确保文本处理健壮，并向开发者或用户指出错误。

用于修复格式错误的 UTF-16 字符串的传统标量算法会按顺序处理代码单元，如图 1 所示。虽然这些方法简单直接，但处理较长字符串时计算开销很大，因此不适用于网页浏览器或数据库等需要高吞吐量的应用。包括 ARM 和 x64 架构在内的现代处理器支持单指令多数据（SIMD）指令，可并行处理多个数据元素。我们提出了一种基于 SIMD 的算法来加速 UTF-16 校正，利用 ARM NEON 和 x64 SSE 指令以块为单位处理 16 位代码单元，从而显著提升性能。

我们的贡献包括一种新颖的单指令多数据算法，用于原位和基于复制的 UTF-16 校正；针对 ARM NEON 和 x64 SSE 优化的实现；以及全面的实验评估，结果表明其速度最高可达标量方法的八倍。我们的软件是开源 simdutf 库的一部分。该库应用广泛，确保了结果的可复现性和实际适用性。

2 相关工作

Unicode 处理在验证和转码等任务中备受关注，但对格式不正确的 UTF-16 字符串进行校正则较少受到探索。Keiser 和 Lemire [9]开发了基于单指令多数据的 UTF-8 验证算法，可并行处理多个字节。Lemire 和 Muła [13]提出了由单指令多数据加速的 UTF-8 到 UTF-16 转码方法，利用 ARM NEON 和 x64 SSE 指令实现每秒数 GB 的吞吐量。最近，Schröder 等人[14] 提出了一种基于单指令多数据的算法，用于验证 CESU-8 编码文本。CESU-8 是一种结合了 UTF-8 的 ASCII 兼容性与 UTF-16 二进制顺序的编码方案。他们的研究利用 x86、ARM 和 PowerPC 单指令多数据指令，在包含 ASCII、韩文和随机文本的数据集上进行测试，结果显示其性能比传统验证方法提升了七倍。这些研究聚焦于验证或格式转换，而非对 UTF-16 字符串进行原位校正。

Cameron[1]提出了用于 UTF-8 到 UTF-16 转码的基于位流的 SIMD 处理方法，但未涉及代理项对校正。类似地，Lemire 和 Muła [13] 提出了由 SIMD 加速的 UTF-8 到 UTF-16 转码方法，利用 ARM NEON 和 x64 SSE 指令实现了每秒数 GB 的吞吐量。Clausecker 和 Lemire [4] 进一步利用 AVX-512 指令处理 512 位寄存器，提升了转码性能；对于中文文本的 UTF-8 到 UTF-16 转码，其速度超过 5 GiB/s，且每个字符所需的中央处理器指令少于两条。然而，这些算法只能执行转码，不能进行原地校正。

```c
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];

        /* Valid single unit or lead surrogate */
        if (c < 0xD800 || c > 0xDFFF) {
            dst[dst_idx++] = c;
            src_idx++;
        } else if (c >= 0xD800 && c <= 0xDBFF) { /* Lead surrogate */
            /* Check if there's a next unit and it's a valid trail 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 lead surrogate or invalid trail */
                dst[dst_idx++] = replacement;
                src_idx++;
            }
        } else { /* Trail surrogate without lead */
            dst[dst_idx++] = replacement;
            src_idx++;
        }
    }
    return dst_idx;
}
```

图 1：使用标量 C 函数将无效 UTF-16代理项替换为替换字符

标量 UTF-16 验证是标准库（例如 ICU、Boost）的一部分，但这些实现按顺序处理代码单元，缺乏单指令多数据优化。我们的工作建立在先前研究中的单指令多数据技术之上，将其应用于 UTF-16代理项校正这一特定问题，并针对 ARM NEON 引入了新的优化。

更广泛地说，单指令多数据指令用于加速许多字符串操作。对于精确字符串匹配，利用这些指令的朴素方法可能非常有效。正如 Tarhio 等人所展示的，一种按特殊顺序比较字符并利用单指令多数据指令的算法，其性能可以超过更复杂的传统算法 [15]。类似地，Fiori 等人 [5] 和 Chhabra 等人 [3] 针对字符串之间的近似匹配问题提出了新颖的算法。我们还可以使用单指令多数据指令更快地解析字符串，例如 JSON 字符串 [10, 12], XML [2], DNS 记录 [11], 等。

3 单指令多数据算法

单指令多数据是一种并行处理技术，允许单条指令同时处理多个数据元素。单指令多数据的核心动机是利用数据级并行，使处理器能够在一个周期内对多个数据点执行相同的操作——例如加法或比较——从而减少执行

```
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 vectorif (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];}
```

图 2：将无效 UTF-16代理项替换为替换字符的通用单指令多数据函数

时间。通过并行处理多个元素，单指令多数据不仅能加速计算，还能提高能效 [7, 16], ，因为它减少了所需的指令周期数。

我们的算法使用 SIMD 指令，以每块至少 8 个代码单元（16 字节）的方式处理 UTF-16 字符串，并将不匹配的代理项替换为 U+FFFD。它支持原地校正（输入等于输出）和基于复制的校正（输入和输出是不同的缓冲区）。算法通过检查块边界两侧的高代理项和低代理项来确认代理项对是否有效，并使用回看机制处理跨块的代理项对。

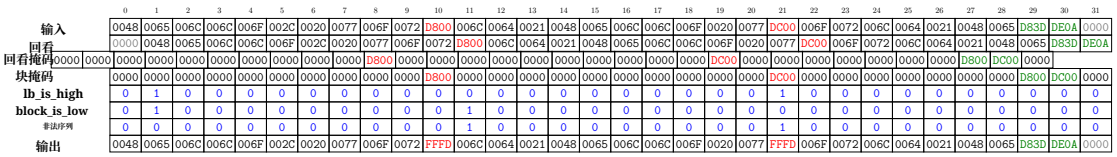


图 3：UTF-16 AVX-512 处理示意图

3.1 架构无关算法

我们的算法（图 2）分三步进行：首先，我们从输入中加载两个相隔一个字的 16 位代码单元向量。较早的向量（回看）用于检查是否存在高代理项，较晚的向量（块）用于检查是否存在低代理项，从而得到两个布尔值向量 *lb_是_高代理项且块_是_低代理项*。第二步，我们对这两个布尔值向量逐元素执行异或。由于这两个向量相隔一个字，这一步会检查高代理项后面是否没有低代理项，或反之。如果结果全为零，则当前块的顺序正确，我们就将其复制到输出。[1]

否则，我们将继续执行第三步：检测具体哪些 16 位代码单元的排列非法，并在将块向量写入输出缓冲区之前，用 Unicode 替换字符替换这些代码单元 `U+FFFD` 。我们通过对 *lb_is_high* 与 *block_is_low* 的补码执行逐元素与运算，并将所得向量向右移动一个元素（即两个字节），使结果对应于*block* 中的元素，而不是 *lookback*中的元素，从而计算出后面没有低代理项的高代理项。同样地，通过对 *lb_is_high* 的补码与 *block_is_-low*执行逐元素与运算，可以找出前面没有高代理项的低代理项。对这两个向量执行逐元素或运算，得到 *block_illseq*，表示*block*中排列非法的代码单元；在将它们写回之前，使用混合操作将其替换为 `U+FFFD` 。

我们依次对输入中的每个向量大小的块执行此算法。由于该算法具有幂等性，对于小于块大小的尾部，我们会进行最后一次迭代，并使其与输入末尾对齐；这可能会与倒数第二次迭代重叠。两种情况（输入以低代理项开头，或以高代理项结尾）无法通过向量化过程检测到，必须手动处理。同样，长度不足一个向量加一个元素的输入无法通过我们的算法处理，必须回退到图1中的标量过程。

示例 我们将演示图 `utf16fix_block` 中的函数如何对长度为 64 字节的向量2处理输入字符串

`Hello, wor?ld!Hello, w? orld!He`😊

其中 `?` 是不匹配的代理项。该 UTF-16 字符串的长度为 31 个代码单元，两侧均由 `U+0000` 包围。变量 `utf16fix_-` 的值

[1]原位处理时，会省略复制。

block 其处理字符串时的状态如图 3所示。第一行显示的输入包含顺序错误的代理项 D800 ，位于第 10 个位置，以及 DC000 ，位于第 21 个位置，均以红色标记。此外，还有一个顺序正确的代理项对 D83D DE0A ，位于第 29–30 个位置，用于编码 U+1F60A 带微笑眼睛的笑脸 ，并以绿色标记。

带有前导 U+0000 的 回看 向量如第二行所示。第三行和第四行分别显示与 DC00 进行掩码运算后的回看 和块 ，由此计算出$lb_is_$高 和块$_is_$低 掩码，分别标示回看 中的高代理项和块 中的低代理项。这些掩码的异或结果构成非法序列 向量，它指示存在代理项顺序错误，但不指出其确切位置。在此例中，位置 10 处顺序错误的代理项 D800 被错误地标记在位置 11。只有在确认非法序列 不全为零时，才会确定错误的确切位置。随后，顺序错误的代理项会被替换为 U+FFFD 替换字符，而位置 29–30 处顺序正确的代理项则保持不变。

讨论 有三个细微的设计决策会影响这一过程。第一个决策是，在两个相互重叠的输入向量中跟踪高代理项和低代理项，而不是使用一个向量并移动生成的掩码$lb_is_$高 和块$_is_$低 来查找不匹配项。第二个决策是，从输入缓冲区加载两次，以获取这两个重叠向量回看 和块 ，而不是每次迭代只加载一次，再从前一个和当前块 向量中切分出回看 。第三个决策是让算法采用分支式实现，[2] 只有在确实出现代理项顺序错误时，才进行纠正。

我们将主例程的运算强度定义为算术逻辑 SIMD 指令（例如比较、按位与）的数量与 SIMD 寄存器加载和存储次数之比。在最佳情况下，与最多两次加载操作和至多一次存储操作相比，每次迭代至少需要五条算术逻辑 SIMD 指令。

使用重叠向量而不是移位掩码，可以避免为了“回看”或“前瞻”而在迭代间传递数值。通过两次加载而不是从两个回看块向量中组装数据，可以省去重排操作，但代价是每次迭代多进行一次加载。对大多数处理器来说，这是值得的权衡。事实上，当前的处理器通常能够每周期完成两条或更多 SIMD 加载指令[6]。与此同时，在最佳情况下，每次迭代至少需要五条算术逻辑 SIMD 指令（例如比较或按位与操作）。因此，即使每次迭代执行两次加载操作，我们的运算强度[17] 仍然相对较高：算术逻辑指令的数量远多于内存操作。进一步提高运算强度反而会降低性能。

以偏移两个字节的方式加载 SIMD 寄存器，一个潜在的缺点是无法让内存加载在自然对齐边界上对齐，也就是说，无法使用能被寄存器长度（以字节为单位）整除的内存地址。不过，大多数现代处理器设计都能以极低的代价处理非对齐 SIMD 加载和存储操作，甚至几乎不需要额外开销

[2]对于给定的输入规模，分支算法会根据输入数据执行不同的代码路径，而无分支算法则始终遵循相同的代码路径，不受数据内容影响。

```
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);}}
```

图 4：在 64 字节块内将无效 UTF-16 代理项替换为替换字符的 AVX-512 函数

性能损失。

我们的方法还有一个附带效果：循环携带依赖被彻底消除，因此处理器可以更轻松地重叠执行多个迭代。理论上，只要执行单元足够多，处理器就可以同时执行多个迭代。

第三种选择基于这样的预期：我们处理的文本几乎都具有正确的代理项顺序。因此，除非快速检查表明有必要，否则跳过确定错误代理项并修正它们这一开销较大的步骤是有利的。作为额外好处，对于原地操作，处理有效 UTF-16 时完全可以避免向缓冲区写入数据。

3.2 AVX-512（x64）实现

英特尔和 AMD 的 64 位处理器（x86-64）主导着 Windows 个人电脑和服务器市场，并支持一系列 SIMD 指令集扩展，可分为四个系列：MMX（64 位向量，仅具有历史意义）、SSE（128 位向量）、AVX（256 位向量）和 AVX-512（512位向量）。每个系列都包含多个支持级别，每个级别除了包含所有先前级别外，也包含所有先前系列。所有 x64处理器至少支持 SSE2，不过通常而言，

利用可用的最宽 SIMD 扩展通常更有优势，这样可以最大限度地增加每次操作处理的数据量。SSE 及其扩展（SSE2、SSE3、SSSE3、SSE4.1、SSE4.2）使用 128 位 XMM寄存器，可处理 16 字节数据（例如，四个 32 位浮点数或十六个 8 位整数）。AVX 和 AVX2 引入了 256 位 YMM寄存器，将容量加倍至 32 字节（例如，八个 32 位浮点数）。AVX-512 则进一步扩展至 512 位 ZMM 寄存器，可处理 64 字节数据（例如，十六个 32 位浮点数），并引入了掩码寄存器等新功能。

所有这些指令集都提供了快速比较指令，能够以 16 位为粒度比较 16 字节（SSE）、32 字节（AVX/AVX2）或 64 字节（AVX-512）的数据，因此非常适合处理 UTF-16数据。实际上，x64处理器支持的指令集种类繁多，通常需要在运行时检测支持的指令集。例如，首次启动处理时，我们可以检查支持的 CPU特性，并从多个预编译函数中选择一个。

对于 AVX-512 之前的指令集，比较指令会在 SIMD 寄存器中为每个元素生成全 1（`0xFF...FF`）或全 0（`0x00...00`）的结果，分别表示真或假。这些结果可以高效地映射到充当位集的通用寄存器中，其中每一位都对应 SIMD 寄存器中的一个比较结果。例如，使用 SSE2 `pcmpeqw` 指令比较两个 16 字节向量，会得到一个包含 16 个字节的 128 位 XMM 寄存器，其中每个 16 位子字会根据比较结果为真或假，取值为 0xFFFF 或 0x0000。使用 SSE2 `pmovmskb` 指令，可以将这些字节转换为一个 16位整数，其中每一位表示一个字节的比较结果。随后即可在标量代码中处理这个位集。例如，比较两个各含 16 字节的向量，可能会得到一个 XMM 寄存器，其字节为 [0xFF、0xFF、0x00、0x00、0xFF、0xFF、...], ，我们可以将其映射为类似 0b110011... 的 16 位值，其中 1 表示字节相等。AVX2 的处理过程大致相同：例如，AVX2 `vpmovmskb` 指令的作用类似于 SSE2 `pmovmskb` 指令，但生成的是 32 位寄存器值，而不是 16 位寄存器值。

AVX-512 引入了专用掩码寄存器（k0–k7），在许多情况下无需再使用 movemask 操作（`pmovmskb` 和 `vpmovmskb`）。这些 8 位到 64 位的掩码寄存器可直接存储比较结果，每一位都对应 512 位 ZMM寄存器中的一个元素。例如，对两个 512 位向量（三十二个 16 位整数）执行 `vpcmpw`指令这样的比较，会在 AVX-512掩码寄存器中生成一个 32位掩码，其中每一位表示一个整数是否相等。这些掩码可直接用于后续 AVX-512指令执行条件操作，也可与通用寄存器合并以进行标量处理。这样便消除了 movemask 指令带来的开销，但需要注意的是，掩码寄存器支持的操作有限（例如，没有尾随零或计数零指令）。

架构特定调整 对于 SSE2 和 AVX2，可以直接使用通用过程，无需修改。它同样适用于 AVX-512，但得益于使用特定的 `utf16fix_block` 函数并进行一些架构特定调整，见图 4。AVX-512 实现的 `utf16fix_block` 函数利用 512 位 ZMM 寄存

器并行处理 32 个 UTF-16 代码单元（64 字节），利用 AVX-512 的掩码寄存器执行高效的条件操作。除了转换为架构特定的内在函数外，这些就是该 `utf16fix_block` 过程与我们的通用算法之间的差异：

- 我们不使用向量表示布尔结果，而是使用掩码寄存器， `FFFF` 或 0000
- 各种混合操作通过掩码驱动的混合指令和掩码存储来实现，而不是通过位运算，
- 我们不通过将lb_illseq 中的lb 提取到标量来修正$out[-1]$ ，而是对$out-1$ 执行 64 字节存储，并仅用lb_illseq的第一位进行掩码，从而确保存储一个或零个代码单元，尤其是不会与后续对out的存储重叠。

为简单起见，我们省略原地操作。在生产环境中，通过将纠正路径中的混合步骤替换为仅针对非法序列代理项的直接掩码存储，并完全省去快速路径中的存储，可以加速$in = out$ 的情况。

表1 总结了 UTF-16 处理所用的 SIMD 指令及其对应内在函数，具体内容详见所提供的代码和讨论。该表列出了每条指令、对应的内在函数、指令集（SSE2、AVX2 或 AVX-512）以及其功能的简要说明。SSE2 的指令（如 `pcmpeqw` 和 `pmovmskb` ）用于处理 128 位向量，而 AVX2 的 `vpmovmskb` 则扩展到 256 位向量。AVX-512 指令（如 `vpcmpw` 和 `vpblendmw`）利用 512 位 ZMM 寄存器和掩码寄存器高效处理 32 个元素，`_mm512_cmpeq_epi16_-mask` 和 `_kxor_mask32` 等内在函数则可用于代理对验证。

3.3 NEON（Aarch64）实现

Aarch64 处理器常见于智能手机和平板电脑等移动设备，也常见于智能电视和电子游戏主机等家用电器。近年来，Aarch64 在服务器市场也日益普及；随着苹果转向 Aarch64，它也开始出现在台式机和笔记本电脑上。Aarch64 定义了两种单指令多数据指令集扩展。NEON 扩展沿用了 32 位 ARM 架构的设计并在此基础上加以扩展，提供了 128 位寄存器，可同时处理八个 16 位字。它的功能集与 SSSE3 相当，涵盖完整的整数和浮点运算，以及加载、存储和重排操作。除了少数例外，所有 Aarch64 处理器都必须支持 NEON，因此它是 Aarch64 上最常见的 SIMD 加速目标。较新的 SVE/SVE2/SVE2.1 指令集扩展系列以 NEON 为基础，进一步引入了可变长度向量寄存器、谓词掩码和许多其他特性，使其与 AVX-512 相似。遗憾的是，SVE 的普及进展缓慢，几乎所有原始设备制造商都选择推出不支持 SVE 的 CPU。因此，我们决定不考虑 SVE，而采用 NEON。

与 x64 不同，支持 NEON 的硬件具有多样化的执行特性。低功耗 Aarch64 中央处理器可能每次只能执行一条 NEON 指令

表 1：用于 UTF-16 处理的 x64 SIMD 指令和内在函数

指令	内在函数	指令集	说明
pcmpeqw	`_mm_cmpeq_pi16`	SSE2	比较两个 128 位向量，并将每个 16 位元素设置为若相等则设为 0xFFFF（真），否则设为 0x0000（假）。
pmovmskb	`_mm_movemask_pi8`	SSE2	提取每个.....的最高有效位128位向量中的每个字节，生成一个16位整数位掩码。
vpmovmskb	`_mm256_movemask_epi8`	AVX2	提取每个256位向量中的字节，生成一个32位整数位掩码。
vpcmpw	`_mm512_cmpeq_epi16_mask`	AVX-512	比较两个512位向量，并生成一个32位掩码，其中每个位表示是否相等。
vmovdqu	`_mm512_loadu_si512`	AVX-512	将 512 位未对齐数据加载到ZMM寄存器中。
vpandd	`_mm512_and_epi32`	AVX-512	对两个 512 位值执行按位与将数据视为 32 位整数的向量。
kxord	`_kxor_mask32`	AVX-512	对两个 32 位掩码寄存器。
ktestzq	`_ktestz_mask32_u8`	AVX-512	测试 32 位掩码中的所有位是否均为零，若是，则返回 true。
kandnw	`_kandn_mask32`	AVX-512	对两个 32 位值执行按位与非位掩码寄存器（先反转第一个掩码，再执行按位与）。
korw	`_kor_mask32`	AVX-512	对两个 32 位掩码寄存器执行按位或。
kshiftriw	`_kshiftri_mask32`	AVX-512	将32位掩码右移一位，并在最高有效位处插入一个零。
kmovw	`_cvtu32_mask32`	AVX-512	将无符号32位整数转换为32 位掩码寄存器。
vmovdqu16	`_mm512_mask_storeu_epi16`	AVX-512	从 512位向量存储 16 位整数到非对齐内存中，使用 32位掩码来选择元素。
vmovdqu	`_mm512_storeu_si512`	AVX-512	将一个 512位向量存储到未对齐的内存中ory.
vpblendmw	`_mm512_mask_blend_epi16`	AVX-512	从两个 512位向量中混合 16 位整数基于 32位掩码的存储。
vmovdqu32	`_mm512_storeu_epi32`	AVX-512	从 512位向量存储 32位整数到非对齐内存。
vpbroadcastw	`_mm512_set1_epi16`	AVX-512	将单个16位整数广播到所有向量元素中，组成一个512位向量。
vpbroadcastd	`_mm512_set1_epi32`	AVX-512	将单个32位整数广播到所有向量元512位向量的元素。

三个周期。另一方面，高性能 CPU 拥有多个执行单元，通常比 x64处理器更多。由此带来的更高吞吐量弥补了较短的向量长度。例如，Apple M1 处理器的 Firestorm 核心每周期可执行四条大多数 NEON 指令 [8]。

```
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= 0xFFFD; /* 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= 0xFFFD; /* 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];}
```

图 5：用于 UTF-16 校正的 ARM NEON 实现

架构特定调整 图 5 展示了我们的 NEON 实现。虽然可以在 NEON 上使用通用算法（图 2），但有两项重要改动可以显著提升性能。首先需要注意的是，NEON 没有与 SSE 和 AVX 的掩码移动指令（如“pmovmskb”）等效的指令。此外，它也缺少专门用于检查寄存器是否非零的指令。我们发现了一种高效的变通方法：使用类似以下的指令计算垂直最大值（所有元素中的最大值），从而检查一个 128 位值是否非零`vmaxvq_u32`，然后将结果移至通用寄存器；当且仅当向量的所有元素均为零时，该寄存器才会保存零。这些指令的性能因所选硬件而异，但两条指令都可能有数个周期的延迟。因此，这一步可能成为瓶颈。我们可以实现一个 `utf16fix_-block` 函数，其工作方式类似于 x64 函数，但预期性能并不理想。

我们通过并行处理 4 个块，然后才检查是否有块需要修正，来摊销这项开销。只有在确实需要修正时，我们才会跳转去修复非法代理项。我们预计大多数输入都不需要修正（大多数 UTF-16 输入都是有效的），因此可以通过处理更大的数据块来缓解这个问题，每块涵盖 64 个码点。副作用是指令级并行度有所提高，这有利于像 Apple M1 这样拥有多个执行端口的微架构。

另一个重要的改进是使用 NEON 独有的反交错加载指令 `vld2q_u8`。LD2 指令会从内存中加载相当于两个向量的数据，将偶数位置的字节写入一个寄存器，将奇数位置的字节写入另一个寄存器。[3] 在 `vld2q_u8`中，将 UTF-16 代码单元视为字节对很有优势。由于 UTF-16 代码单元中较高有效字节足以判断该代码单元是高代理项、低代理项还是两者皆非，因此我们使用 `vld2q_u8` 将 16 个代码单元加载到一对向量中，然后丢弃较低有效字节向量，因为执行该算法不需要其中的内容。接下来我们只使用高字节向量，此时元素大小已从 16 位减小到 8 位，因此每步处理的代码单元数量翻倍。副作用是，丢弃低字节向量后，我们无法像图 2中那样修正这些向量，因此改用另一种方法。

与其他实现一样，核心函数 `to_well_formed`负责协调对长度为 n 的 UTF-16 字符串的处理。它首先通过标量回退处理较短的输入（$n < 17$）。对于较长的输入，它会确保第一个输出元素不是无效低代理项，并在必要时将其替换为 `U+FFFD` 。随后，函数会尽可能使用 `utf16fix_block64` 按每块 64 个码点处理字符串；对于较小的剩余片段，则使用 `utf16fix_block` 按每块 16 个码点处理。最后会处理一个包含 16 个码点的块，以覆盖字符串末尾，并检查最后一个元素是否为无效高代理项，必要时将其替换为 `U+FFFD`。

`utf16fix_block64` 函数通过将块划分为四个各含 16 个代码单元的片段来处理 64 个 UTF-16 代码单元（128 字节），每个片段都由 `get_mismatch_-copy`处理。此辅助函数会加载两个 128 位向量： `lb` （回看，从 $in\ -1$）以及

[3]配套指令 `vst2q_u8`可以在存储时撤销此转换，但此处并不需要。

block （来自 *in*），并使用 vld2q_u8 将字节解交错为高位部分和低位部分。随后，函数使用 0xFC 对高字节进行掩码处理，以聚焦于最高 6 位，并通过比较指令 vceqq_u8识别高代理项（0xD8）和低代理项（0xDC）。异或运算（veorq_u8）用于检测非法序列（例如，高代理项后面没有跟随低代理项）。函数使用 vst2q_u8 将输入块复制到输出，并返回非法序列掩码。在 utf16fix_block64中，使用 vorrq_u8 将四个此类掩码（*illse0* 至 *illse3*）合并，并通过 veq_non_zero检查结果是否非零； veq_non_zero使用 vmaxvq_u32 计算 128 位向量中的最大值，从而高效检测任何无效序列。

如果检测到无效序列， get_mask 会将四个 128 位掩码转换为一个 64 位位集，其中每一位对应一个代码单元的有效性。它使用预定义的位_掩码 向量提取特定位，并通过 vpaddq_u8 累积结果，借助 vgetq_lane_u64生成单个 64 位值。随后对该位集进行标量处理，使用stdc_trailing_zeros_ull 查找无效位置，判断每个位置对应的是高代理项还是低代理项，并将相应的代码单元替换为 U+FFFD。如果进行了修正，该函数返回 false；如果该块有效且未经更改地复制，则返回 true。为简洁起见，此处省略了 utf16fix_block 函数；它的工作方式与通用实现类似。主要区别在于，它使用交错加载。

表 2 列出了 UTF-16 处理代码中使用的 ARM NEON 指令及其对应的内在函数。这些指令在 128 位 NEON 寄存器上运行，可并行处理八个 16 位字。每个条目都包含指令、对应的内在函数，以及对其在验证和修正 UTF-16 代理项对时所起作用的简要说明。

4 实验

为了评估 UTF-16 验证和纠正算法的性能，我们使用 C++ (benchmark_to_well_formed_utf16.cpp) 开发了一款基准测试工具。它是 simdutf 库.[4] 的一部分。我们以发布模式（-O3 -NDEBUG）构建基准测试软件和 C++ 库。在 x64 处理器上，simdutf 库使用专门的内核，针对不同的英特尔处理器微架构进行了优化，包括 Ice Lake、Haswell 和 Westmere。Ice Lake 内核专为英特尔 Ice Lake 处理器（AVX-512 和 VB MI2）定制。Haswell 内核针对支持 AVX2 的 Haswell 微架构进行了优化。Westmere 内核则面向较旧的 Westmere 处理器（SSE4.2）。

我们在表 3中介绍了用于基准测试的系统。该表详细列出了每个系统的关键规格，包括处理器类型、时钟频率、微架构、内存配置和编译器版本。所选系统涵盖了多种现代高性能架构，使我们能够全面评估不同工作负载和计算环境下的性能。

我们会生成具有受控特征的随机 UTF-16 字符串。具体来说，我们会为输入字符串配置指定百分比的有效代理项对（代码单元 U+D800 至 U+DBFF 后跟 U+DC00 至 U+DFFF）和不匹配的代理项

[4] https://github.com/simdutf/simdutf，提交哈希 dae7a10

表 2：用于 UTF-16 处理的 ARM NEON 指令和内在函数

指令	内在的	说明
LD2	vld2q_u8	从以下位置加载 256 位未对齐数据内存，反交错为两个 128 位 8 位整数向量。
ST2	vst2q_u8	将两个 128 位向量中的 8 位整数存储到非对齐内存中，并交错数据。
AND	vandq_u8	对两个 128 位 8 位整数向量。
CMEQ	vceqq_u8	比较两个 128 位向量中的 8 位整数是否相等，并生成一个 128 位向量，其中每个元素为真时取 0xFF，假时取 0x00。
EOR	veorq_u8	对两个 128 位向量执行按位异或 8 位整数向量。
DUP	vdupq_n_u8	将单个 8 位整数广播到所有元—素中，组成一个 128 位向量。
UMAXV	vmaxvq_u32	计算所有元素中的最大值 128 位向量中的 32 位元素，返回—入一个 32 位标量。
ADDP	vpaddq_u8	成对地将两个 128 位 位向量中的 8 位整数相加，生成一个由 8 位和组成的 128 位向量。
FMOV	vgetq_lane_u64	从一个 128 位向量中提取一个 64 位通道，并将该向量视为两个 64 位整数。
n/a	vreinterpretq_u64_u8	将一个由 8 位整数构成的 128位向量重新解释为由 64 位整数构成的 128位向量而不改变数据。
n/a	vreinterpretq_u32_u8	将一个由 8 位整数构成的 128位向量重新解释为由 32位整数构成的 128位向量而不更改数据。

表3：用于基准测试的系统

处理器		Apple M4	Intel Xeon Gold 6338
微架构	频率	4.4 GHz 到 4.5 GHz	3.2 GHz
		M4（Aarch64，2024）	Ice Lake（x64，2019）
	内存	LPDDR5X (7500 MT/s)	DDR4 (3200 MT/s)
	编译器	Apple/LLVM 17	GCC 12

（孤立的高代理项或低代理项）。具体来说，如果我们将代理项对的百分比设为 0.1%，那么所有字符中有 0.1% 涉及代理项对。如果我们将不匹配的代理项百分比设为 0.1%，那么有效代理项对之外的代码单元中，有 0.1% 会被随机选为高代理项或低代理项（各占 50%）。某些孤立的代理项可能会形成有效的代理项对，但这种情况不太可能发生（即，对于 0.1%，其概率为

0。01 %)

该基准测试测量不同实现的吞吐量（单位为 GB/s）和硬件性能计数器，例如每字节指令数和每字节周期数，其中包括来自 V8 JavaScript 引擎的基线实现，以及使用 simdutf 库的优化版本。近期版本已将 V8 代码替换为 simdutf 库。实验使用的输入大小最多为 1 000 000 个代码单元。

我们使用两种不同的配置进行了实验，以分析算法在不同条件下的行为。在第一种配置中，我们将输入大小设为 1 000 000 个代码单元，其中 0.1 % 的代码单元组成有效的代理项对，0 % 为不匹配的代理项。该设置代表补充平面字符极少的场景，重点关注基本 UTF-16 字符。在第二种配置中，我们保持相同的输入大小，但调整参数，使 0.1 % 为有效的代理项对，0.1 % 为不匹配的代理项，从而引入少量无效的 UTF-16 序列。我们选择这些配置，是为了评估算法在几乎有效和存在轻微误差的 UTF-16 输入下的性能，并深入了解其效率和误差处理能力。

除固定大小的实验外，我们还针对一系列输入大小进行了基准测试，以评估可扩展性。该框架将最大输入大小（1000 000 代码单元）划分为 128 个大小几乎相等的块，并对每个块单独进行测试。这种方法能够细致地分析输入大小增加时的性能趋势。对于每个块，基准测试都会生成一个随机 UTF-16 字符串，其中代理项对和不匹配的代理项所占比例按指定值设置（0。1% 和 0% 或0。1% 和 0。1%，具体取决于配置）。我们收集每种实现的性能指标，包括吞吐量和误差范围，从而比较它们在不同输入规模下的表现，并找出任何依赖于输入大小的瓶颈。我们在苹果平台的图 6 中展示了这一结果。可以看到，标量版本（V8）和 simdutf 版本（ARM64）在整个范围内的速度都很稳定，不过对于极短的字符串，两者的每 GB/s 性能都略低。simdutf 函数的速度几乎是标量函数的 9 倍。

每种实现均经过 100 次迭代测试，并多次处理输入数据，以获得可测量的执行时间（1 毫秒）。软件使用事件收集器记录耗时以及可用时的硬件计数器，例如 CPU 周期和指令。吞吐量的计算方式为输入大小除以最佳执行时间（以纳秒为单位），结果以 GB/s 表示。此外，我们还将平均执行时间与最佳执行时间之间的百分比差异计算为误差范围，以此衡量性能稳定性。我们发现误差范围很小（约为 1%）。

苹果和英特尔系统的性能结果分别汇总于表4和表5中。表4 展示了 Apple M4 系统在0% 和0.1% 不匹配代理项条件下，V8 基线与优化后的 arm64 实现的吞吐量（GB/s）、每字节指令数和每周期指令数。arm64 实现的性能显著优于 V8 基线，最高达到 18.9GB/s，错误率为0%，而 V8 为2.2GB/s，充分体现了单指令多数据优化的有效性。表5 展示了 Intel Xeon Gold

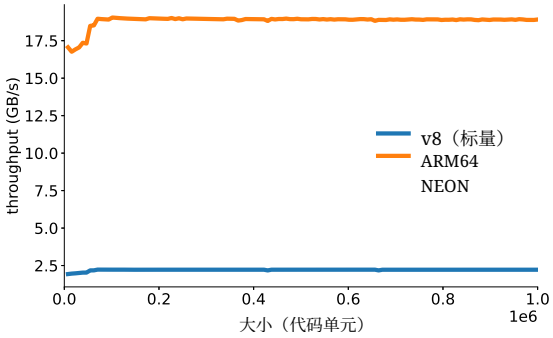


图 6：Apple 系统上处理 0 到一百万个代码单元的吞吐量

表 4：Apple 系统在大小为 1 000 000 时的性能指标

	V8		arm64	
误差	0 %	0.1 %	0 %	0。1 %
GB/s	2.2	2.2	18.9	16.3
指令/字节	12.0	12.0	0.9	0.9
指令/周期	5.9	5.8	3.7	3.3

表 5：Intel 系统在大小为 1 000 000 时的性能指标

	V8		Ice Lake		Haswell		Westmere	
误差	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
指令/字节	13.0	13.0	0.4	0.4	0.8	0.8	2.0	2.0
指令/周期	5.0	5.0	1.0	1.0	1.8	1.8	3.6	3.5

6338 系统上 Ice Lake、Haswell 和 Westmere 微架构的优化实现与 V8 基线的对比结果。Ice Lake 实现的吞吐量最高，达到7.5GB/s，错误率为0%；而 V8 基线仅为 1.2GB/s。在两个系统上，优化后的实现都表现出更低的每字节指令数和更高的效率，尤其是在无错误场景下，这凸显了针对特定架构的单指令多数据优化的优势。表5中对 icelake 和 haswell 内核的性能比较显示，在 Intel Xeon Gold 6338 系统上，两者在吞吐量和效率方面存在明显差异。icelake 内核的吞吐量为7.5GB/s，在0% 错误率下略低于 Haswell 的7.8GB/s。然而，icelake 内核展现出更高的效率，其每字节指令数更低（0.4，而 haswell 为 0.8），每周期指令数也显著更低（1.0，而 haswell 为 1.8）。

5 结论

通过利用单指令多数据指令，我们提出的算法相比传统标量方法取得了显著的性能提升，在 ARM NEON 和 x64 SSE 架构上的加速比最高可达八倍。Apple M4 和 Intel Xeon 系统上的实验结果证实了该方法的可扩展性和有效性；Apple M4 处理器上的吞吐量达到 18.9GB/s，Intel Ice Lake 处理器上的吞吐量达到 7.5GB/s，同时指令数量也有所减少。

6 资助

本研究由加拿大自然科学与工程研究委员会通过编号为 RGPIN-2024-03787 的资助项目以及 NHR e.V. 研究生院计划提供资金支持。

参考文献

[1] R. D. Cameron。使用并行位流进行 SIMD 文本处理的案例研究：UTF-8 到 UTF-16 转码。载于第 *13* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，第 91–98 页。ACM，2008 年。doi:10.1145/1345206.1345222。[2] R. D. Cameron、K. S. Herdy 和 D. Lin。使用并行位流技术进行高性能 XML 解析。载于协作研究高级研究中心 *2008* 年会议论文集：思想汇聚，CASCON '08，美国纽约州纽约市，2008 年。计算机协会。[3] T. Chhabra、S. S. Ghuman 和 J. Tarhio。使用 AVX2 和 AVX-512 指令进行带错配的字符串搜索。《信息处理快报》，189:106557，2025 年。[4] R. Clausecker 和 D. Lemire。使用 AVX-512指令转码 Unicode 字符。*Software: Practice and Experience*，53(12):2430–2462，2023 年。[5] F. J. Fiori、W. Pakalén 和 J. Tarhio。使用单指令多数据进行近似字符串匹配。《计算机杂志》，65(6):1472–1488，2022 年。[6] A. Fog。指令表：英特尔、AMD 和 VIA CPU 的指令延迟、吞吐量及微操作分解列表。载于《优化手册》，第 4 卷。在线发表于 `https://agner.org/optimize`，2025 年 7 月。[7] H. Inoue。SIMD 宽度如何影响能效：排序案例研究。载于*2016* 年 *IEEE* 低功耗与高速芯片研讨会（*COOL CHIPS XIX*），第 1–3 页，2016 年。[8] D. Johnson。苹果 微架构研究。`https://dougallj.github.io/applecpu/firestorm.html` [上次检查于 8 月 2024], 2023。

[9] J. Keiser 和 D. Lemire。每字节少于一条指令即可验证 UTF-8。*Software: Practice and Experience*，51(5)，2021 年。doi:10.1002/spe.2920。[10] J. Keiser 和 D. Lemire。按需 JSON：一种更好的文档解析方式？*Software: Practice and Experience*，2023 年。即将出版。[11] J. Koekkoek 和 D. Lemire。每秒解析数百万条 DNS 记录。*Software:Practice and Experience*，55(4):778–788，2025 年。[12] G. Langdale 和 D. Lemire。每秒解析数 GB 的 JSON。*The VLDBJournal*，28(6):941–960，2019 年。[13] D. Lemire 和 W. Muła。使用 SIMD 指令每秒转码数十亿个 Unicode 字符。*Software: Practice and Experience*，52(2):555–575，2022 年。doi:10.1002/spe.3036。[14] M. Schröder、S. Machmeier、S. Maeng 和 V. Heuveline。利用 SIMD 指令验证 CESU-8 编码文本。载于*2024* 年第 *13* 届软件与计算机应用国际会议论文集，第 102–111 页，2024 年。[15] J. Tarhio、J. Holub 和 E. Giaquinta。技术胜过算法（在精确字符串匹配中）。*Software: Practice and Experience*，47(12):1877–1885，2017 年。[16] J. Wassenberg、M. Blacher、J. Giesen 和 P. Sanders。向量化且具有性能可移植性的快速排序。*Software: Practice and Experience*，52(12):2684–2699，2022 年。[17] S. Williams、A. Waterman 和 D. Patterson。Roofline：面向多核架构的富有洞察力的可视化性能模型。*Commun. ACM*，52(4):65–76，2009 年 4 月。