Unicode at Gigabytes per Second

Daniel Lemire

DOT-Lab Research Center, University of Quebec (TELUQ), Montreal, Canada
daniel.lemire@teluq.ca

Abstract. We often represent text using Unicode formats (UTF-8 and UTF-16). The UTF-8 format is increasingly popular, especially on the web (XML, HTML, JSON, Rust, Go, Swift, Ruby). The UTF-16 format is most common in Java, .NET, and inside operating systems such as Windows.

Software systems frequently have to convert text from one Unicode format to the other. While recent disks have bandwidths of 5 GiB/s or more, conventional approaches transcode non-ASCII text at a fraction of a gigabyte per second.

We show that we can validate and transcode Unicode text at gigabytes per second on current systems (x64 and ARM) without sacrificing safety. Our open-source library can be ten times faster than the popular ICU library on non-ASCII strings and even faster on ASCII strings.

Keywords: Unicode · Vectorization · Internationalization.

1 Introduction

From the early days of computing, programmers have had to represent characters in software. They needed to agree to standards so that different software written by different vendors would be interoperable. One of the earliest such standards is ASCII—first specified in the early 1960s. The ASCII standard is still popular today: it uses one byte per character—with the most significant bit set to zero. Unfortunately, ASCII could only ever represent up to 128 characters—far less than needed.

Thus many diverging standards emerged for representing characters in software. The existence of multiple incompatible formats made the production of interoperable localized software difficult. Conversion between some formats could sometimes be lossy or ambiguous.

Unicode arose in the late 1980s as an attempt to provide a single agreed-upon standard. Initially, it was believed that using 16 bits per character would be sufficient. However, engineers realized over time that a wider range of characters should be supported—if the standard was to be universal. Thus the Unicode standard was extended to potentially include up to 1 114 112 characters. Characters are sometimes called code points and represented as integer values between

This manuscript is based on a forthcoming long-form article written with Wojciech Muła, *Transcoding Billions of Unicode Characters per Second with SIMD Instructions*.

0 and 1 114 112. In practice, only a small fraction of all possible code points have been assigned, but more are assigned over time with each Unicode revision. The Unicode standard is an extension of the ASCII standard in the sense that the first 128 Unicode code-point values match the ASCII characters.

There are several ways to represent Unicode characters in bytes. Due to the original expectation that Unicode would fit in 16-bit space, a format based on 16-bit words (UTF-16) format was published in 1996 and formalized in 2000 [3]. It may use either 16-bit or 32-bit per character. The UTF-16 format was adopted by programming languages such as Java, and became a default under Windows.

Unfortunately, UTF-16 is not backward compatible with ASCII at a byte level. Thus an ASCII compatible format was proposed and formalized in 2003: UTF-8 [8]. Over time, it became widely used. Text interchange formats such as JSON, HTML or XML are expected to be in the UTF-8 format. Programming languages such as Go, Rust and Swift use UTF-8 by default. When used as part of data interchange documents, UTF-8 is commonly more concise due to its ability to use one byte per character to represent ASCII text.

Though UTF-8 dominates in many applications, it does not make UTF-16 obsolete. The UTF-16 format has advantages. Indeed, most text represented in the UTF-16 format has exactly 2-byte per character, except for the occasional special character (e.g., an emoji). Having a flat 2-byte per character makes some operations faster. Both formats require validation: not all arrays of bytes are valid. However, the UTF-8 format is more expensive to validate. In some cases, UTF-16 may be even more concise as well (e.g., when representing Chinese text).

Thus, for the foreseeable future, we need to validate both UTF-16 and UTF-8 strings, and to convert (transcode) text between the two formats.

We should expect these operations to be fast, and they are. However, speed and efficiency are relative. Cloud vendors offer high bandwidth between node instances (e.g., 3.3 GiB/s) and the bandwidth of disks is rising fast (e.g., 5 GiB/s with PCIe 4.0) [1]. We believe that we should be able to match such high speeds inside the processor when transcoding text.

For fast processing, we should seek to make the best possible use out of our processors. Commodity processors support single-instruction-multiple-data (SIMD) instructions. These instructions operate on several words at once unlike regular instructions. Starting with the Haswell microarchitecture (2013), Intel and AMD processors support the AVX2 instruction set and 256-bit vector registers. Most mobile phones tablets have 64-bit ARM processors (`aarch64`) with NEON instructions (128-bit registers). Hence, on recent x64 processors, we can compare two strings of 32 characters in a single instruction. Algorithms designed for SIMD instructions typically require fewer instructions per byte. Software that retires fewer instructions tends to use less power and to be faster.

2 Related Work

There are many ways to validate and transcode Unicode text. One may use series of branches or finite-state approaches [4]. We are most interested in the fastest techniques. Keiser and Lemire [6] describe a fast SIMD-based UTF-8 validation directly on byte streams. We make use of their approach (see § 4).

Cameron [2] proposed that we transform text using *bit streams*. Given byte arrays, the bit-stream approach creates eight bit arrays, each one corresponding to a bit position within a byte. There is one bit stream for the least significant bit, one for the second least significant bit, and so forth. We must first transform the input data into such bit streams and then convert the data back from bit streams to an array of bytes. Cameron applied this strategy to UTF-8 to UTF-16 transcoding.

Independently, Inoue et al. [5] proposed a UTF-8 to UTF-16 SIMD-accelerated transcoder, but it does not validate its inputs. The authors did not make their implementation (for PowerPC processors) available.

We are not aware of any further work in the scientific literature regarding the application of SIMD instruction to the validation or transcoding of Unicode text. Except for fast ASCII paths, we do not know of any widespread use of SIMD instructions for validating or transcoding Unicode text.

3 The Unicode Formats

ASCII characters require one byte with UTF-8 and two bytes with UTF-16. UTF-16 can represent all characters—except for the supplemental characters such as emojis—using two bytes. The UTF-8 format uses two bytes for Latin, Hebrew and Arabic alphabets. Asiatic characters (including Chinese and Japanese) require three UTF-8 bytes. Both UTF-8 and UTF-16 require 4 bytes for the supplemental characters. We often represent Unicode characters using its integer value in hexadecimal as, for example, `U+7F` (for 127).

UTF-8 encodes values in sequences of one to four bytes. We refer to the first byte of a sequence as a leading byte; the most significant bits of the leading byte indicate the length of the sequence:

- If the most significant bit is zero, we have a sequence of one byte (ASCII).
- If the three most significant bits are `110`, we have a two-byte sequence.
- If the four most significant bits are `1110`, we have a three-byte sequence.
- Finally, if the five most significant bits are `11110`, we have a four-byte sequence.

All bytes following the leading byte in a sequence are continuation bytes, and they must have their two most significant bits as `10`. Except for the required most significant bit sequences, other bits (from 7 bits to 21 bits) provide the code-point value. The most significant bits of the code-point value are in the leading byte, followed by lesser significant bits in the second byte and so forth, with the least significant bits in the last byte of the sequence. Valid UTF-8 sequences must follow the following exhaustive rules:

1. The five most significant bits of any byte cannot be all ones.
2. The leading byte must be followed by the right number of continuation bytes.

3. A continuation byte must be preceded by a leading byte.
4. The decoded character must be larger than `U+7F` for two-byte sequences, larger than `U+7FF` for three-byte sequences, and larger than `U+FFFF` for four-byte sequences.
5. The decoded code-point value must be less than 1 114 112.
6. The code-point value must not be in the range `U+D800...DFFF`.

In the UTF-16 format, characters in `U+0000...D7FF` and `U+E000...FFFF` are stored as 16-bit values—using two bytes. The characters in the range `U+010000...10FFFF` require two 16-bit words (a surrogate pair). The first word in the pair is in `0xD800...DBFF` whereas the second word is in `0xDC00...DFFF`. The code-point value is made of the 10 least significant bits of the two words—using the second word as least significant—adding `0x10000` to the result. During validation, only the possible surrogate pairs require attention.

4 Algorithms

All commodity software with SIMD instructions (e.g., x64, ARM, POWER) have fast instructions to permute bytes within a SIMD register according to a sequence of indexes. Our transcoding techniques depend critically on this feature: we code in a table the necessary parameters—including the indexes (sometimes called shuffle masks)—necessary to process a variety of incoming characters.

Our accelerated UTF-8 to UTF-16 transcoding algorithm processes up to 12 input UTF-8 bytes at a time. From the input bytes, we can quickly determine the leading bytes and thus the end of each character. We use a 12-bit word as a key in a 1024-entry table. Each entry in the table contains the number of UTF-8 bytes that will be consumed and an index into another table where we find shuffle masks. The value of the index into the other table also determines one of three possible code paths. The first 64 index values indicate that we have 6 characters spanning between one and two bytes. Index values in $[64, 145)$ indicate that we have 4 characters spanning between one and three bytes. The remaining indexes represent the general case: 3 characters spanning between one and four bytes. The shuffle mask can then be applied to the 12 input bytes to form a vector register that can be transformed efficiently. We use this 12-byte routine inside 64-byte blocks. After loading a 64-byte block, we apply the Keiser-Lemire validation routine [6]. Afterward, we identify the leading bytes, and then process the block in multiple iterations, using up to 12 bytes each time. In the special case where all 64 bytes are ASCII, we use a fast path. For even greater efficiency, we have three other fast paths within the 12-byte routine: we check whether the next 16 bytes are ASCII bytes, whether they are all two-byte characters, or all three-byte characters.

Our UTF-16 to UTF-8 algorithm iteratively reads a block of input bytes in a SIMD register. If all 16-bit words in the loaded SIMD register are in the range `U+0000...007F`, we use a fast routine to convert the 16 input bytes into eight equivalent ASCII bytes. If all 16-bit words are in the range `U+0000...07FF`, then we use a fast routine to produce sequences of one-byte or two-byte UTF-8 characters. Given an 8-bit bitset which indicates which 16-bit words are ASCII, we load a byte value from a table indicating how many bytes will be written, and a 16-byte shuffle mask. If all 16-bit words are in the ranges `U+0000...D7FF`, `U+E000...FFFF`, we use another similar specialized routine to produce sequences of one-byte, two-byte and three-byte UTF-8 characters. Otherwise, when we detect that the input register contains at least one part of a surrogate pair, we fall back to a conventional code path.

5 Experiments

We make available our software as a portable open-source C++ library. As a benchmarking system, we use a recent AMD processor (AMD EPYC 7262, Zen 2 microarchitecture, 3.39 GHz) and GCC 10. We compare against a popular library: International Components for Unicode (ICU) [7] (version 67.1). We also use the `u8u16` library [2]. Unlike ICU and our own work, the `u8u16` library only provides UTF-8 to UTF-16 transcoding. For our experiments, we use lipsum text in various languages. All of our transcoding tests include validation. To measure the speed, we record the time by repeating the task 2000 times. We compare the average time with the minimal time and find that we have an accuracy of at least 1%. We divide the input volume by the time required for the transcoding. Fig. 1 shows our results. Our UTF-8 to UTF-16 transcoding speed exceeds 4 GiB/s for Chinese and Japanese texts, which is about four times faster than ICU. In our tests, the `u8u16` library only surpasses ICU significantly for Arabic. Our UTF-16 to UTF-8 transcoding speed is nearly 6 GiB/s in all tests which is nearly ten times faster than ICU.

For ASCII transcoding (not shown in the figures), we achieve 36 GiB/s for UTF-16 to UTF-8 transcoding, and 20 GiB/s for UTF-8 to UTF-16 transcoding. Effectively, we are so fast that we are nearly limited by memory bandwidth. Comparatively, ICU delivers 2 GiB/s and 1 GiB/s in our tests.

6 Conclusion

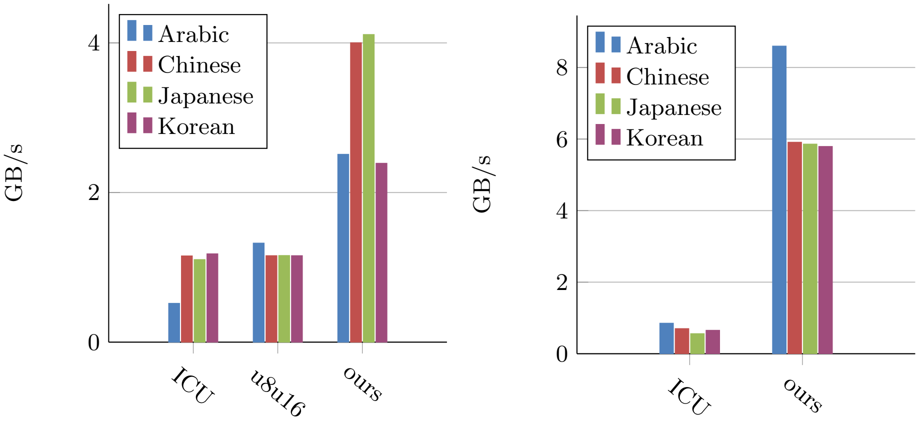
Our SIMD-based transcoders can surpass popular transcoders (e.g., ICU) by a wide margin (e.g., 4×). Our UTF-16 to UTF-8 transcoder achieves speed of about 6 GiB/s for many Asiatic languages using a recent x64 processor. In some cases, we achieve 4 GiB/s for UTF-8 to UTF-16 transcoding with full validation. For ASCII inputs, we achieve tens of gigabytes per second.

References

1. Barr, J.: The Floodgates Are Open – Increased Network Bandwidth for EC2 Instances. `https://aws.amazon.com/blogs/aws/`

`https://github.com/simdutf/simdutf`

`https://github.com/rusticstuff/simdutf8`



(a) UTF-8 to UTF-16 transcoding

(b) UTF-16 to UTF-8 transcoding

Fig. 1: Transcoding speeds for various test files.

the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/ [last checked July 2021] (2018)

2. Cameron, R.D.: 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. pp. 91–98. ACM (2008)
3. Hoffman, P., Yergeau, F.: UTF-16, an encoding of ISO 10646. https://tools.ietf.org/html/rfc2781 [last checked July 2021] (2000), Internet Engineering Task Force, Request for Comments: 3629
4. Höhrmann, B.: Flexible and Economical UTF-8 Decoder. http://bjoern.hoehrmann.de/utf-8/decoder/dfa/ [last checked July 2021] (2010)
5. Inoue, H., Komatsu, H., Nakatani, T.: Accelerating UTF-8 Decoding Using SIMD Instructions (in Japanese). Information Processing Society of Japan Transactions on Programming **1**(2), 1–8 (2008)
6. Keiser, J., Lemire, D.: Validating utf-8 in less than one instruction per byte. Software: Practice and Experience **51**(5) (2021)
7. International Components for Unicode (ICU). http://site.icu-project.org [last checked July 2021] (2010)
8. Yergeau, F.: UTF-8, a transformation format of ISO 10646. https://tools.ietf.org/html/rfc3629 [last checked July 2021] (2003), Internet Engineering Task Force, Request for Comments: 3629