

A Case Study in SIMD Text Processing with Parallel Bit Streams

UTF-8 to UTF-16 Transcoding

Robert D. Cameron

School of Computing Science, Simon Fraser University
cameron@cs.sfu.ca

Abstract

High performance SIMD text processing using the method of parallel bit streams is introduced with a case study of UTF-8 to UTF-16 transcoding. A forward transform converts byte-oriented character stream data into eight parallel bit streams. Decoding, validation and computation of UTF-8 indexed UTF-16 bit streams are performed using bit-parallel logic and shifting operations. Conversion from UTF-8 indexing to UTF-16 indexing is performed using parallel bit deletion. The inverse transform is applied to yield high and low UTF-16 byte streams which are then merged. Combined with optimization techniques for blocks of ASCII data, speed-ups of 3 to 25 times are achieved on commodity processors compared with optimized byte-at-a-time code. Further applications of the method of parallel bit streams to bulk text processing applications are briefly discussed along with future prospects for the combination of intraregister and intrachip parallelism on multicore processors.

Categories and Subject Descriptors C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: Single-instruction-stream, multiple-data-stream processors (SIMD); D.1.3 [Concurrent Programming]: Parallel programming

General Terms Algorithms, Performance

Keywords SIMD text processing, parallel bit streams, UTF-8, UTF-16, transcoding

1. Introduction

Although the intraregister SIMD capabilities of commodity processors (e.g., SSE on Intel, AltiVec on Power PC) have proven useful for graphics, audio, video, encryption and other applications over the last decade, they have seen little application to high-performance text processing. The mismatch between the fixed-size data decompositions that work well with SIMD technology and the variable-length characters, tokens or words common to textual data is one factor; another is that byte-at-a-time text processing performance is often judged to be “good enough” for most applications. Nevertheless, there has been recent interest in the possibility of using SIMD techniques to address the performance challenges of XML parsing [1, 5]. New string-handling instructions have been announced as part of the SSE4 extensions to Intel processor architecture [9]. Although the new string handling capabilities are

limited to a few variable-length comparison instructions, they do reflect a demand for high performance string processing.

Consider instead a new approach to SIMD character processing based on the concept of parallel bit streams. The idea is to transform byte-oriented character stream data into eight parallel bit streams, each bit stream comprising one bit per character code unit. Code units may be ASCII characters or UTF-8 bytes, for example, with one parallel bit stream defined for each of bit 0 through bit 7 of each code unit. Given such a representation, the 128-bit SIMD registers of the SSE or AltiVec registers can be used to process 128 code unit positions at a time. For example, given a set of parallel bit streams for UTF-8 code units, a single bitwise and operation applied to blocks of 128 bit values from the bit 0 and bit 1 streams produces a block classifying these 128 positions as UTF-8 prefix bytes (bytes in the range 0xC0 to 0xFF) or not.

Although there are many technical challenges and certain gaps in the capabilities of present-day SIMD instruction set architectures, the method of parallel bit streams can yield surprisingly good results. This paper presents a case study of the problem of UTF-8 to UTF-16 transcoding. This is a challenging problem for parallel methods, involving conversion between two different variable-length encoding schemes. It is also a practically significant problem, widely cited as one of the principal bottlenecks in XML processing [6, 7, 8]. Even with Intel’s announced string-processing extensions, there seems little hope of a viable byte-oriented SIMD solution to this problem. Applying the techniques described in this paper, however, 3X to 25X speed-ups over byte-at-a-time processing can be achieved using MMX, SSE or AltiVec technology, depending on input data characteristics. These results are obtained with the open source `u8u16` software, implementing UTF-8 to UTF-16 transcoding with an `iconv`-compatible interface [3].

The remainder of this paper is organized as follows. Section 2 illustrates the nature of parallel bit stream programming with the example of UTF-8 decoding and validation, the first stage of UTF-8 to UTF-16 transcoding. Section 3 moves on to the calculation of UTF-16 bit streams based on UTF-8 byte and scope classifications. Section 4 then describes parallel bit deletion methods that can be applied to convert from bit streams indexed in parallel with the UTF-8 byte representation to ones indexed in parallel with the desired UTF-16 code unit representation. Section 5 then describes the essential transform and inverse transform pair that allow conversion between the byte stream domain and the corresponding parallel bit stream domain. Section 6 then briefly describes the block processing structure of the `u8u16` transcoder and then Section 7 goes on to present performance results on existing commodity processors. Section 8 briefly describes further applications of bit stream technology being investigated in ongoing research. Section 9 discusses the open-source patent commercialization of this technology and Section 10 concludes the paper with a discussion of prospects for further development of related technologies.

Unicode Range	Code Point Pattern	UTF-8 Pattern
0000–007F	00000 00000000 0tuvwxyz	0tuvwxyz
0080–07FF	00000 00000pqr stuvwxyz	110pqrst 10uvwxyz
0800–FFFF	00000 jklmnpqr stuvwxyz	1110jklm 10npqrst 10uvwxyz
10000–10FFFF	efghi jklmnpqr stuvwxyz	11110efg 10hijklm 10npqrst 10uvwxyz

Table 1. The UTF-8 Encoding Form of Unicode

2. UTF-8 Byte Classification and Validation

As a first example of parallel bit stream manipulation, consider the example of UTF-8 byte classification and validation. UTF-8 is a variable-length encoding form of Unicode in which one to four 8-bit *code units* (bytes) are used to represent each character. Unicode itself is a system for encoding all the world’s characters as integer *code point* values having up to 21 significant bits. Table 1 shows the structure of UTF-8 byte sequences corresponding to ranges of Unicode code points. The four rows of the table correspond to the four possible lengths of UTF-8 byte sequences. In each row, the first column shows the Unicode code point range that applies in hexadecimal notation. The second column shows the 21-bit pattern for values in the range, with letters standing for bit values at particular positions. The third column shows the bit pattern of the corresponding UTF-8 byte sequence.

As shown in Table 1, the high-order bit (bit 0) of each UTF-8 byte distinguishes whether the byte may occur as a single byte sequence (row 1) or whether it may only occur within multibyte sequences (rows 2 through 4). In the case of single byte sequences, the byte values are in the range 0x00–0x7F and correspond directly to ASCII character codes. In the case of multibyte sequences, the byte values are further partitioned by the value of the second bit (bit 1) into the ranges 0xC0–0xFF for UTF-8 prefix bytes and 0x80–0xBF for UTF-8 suffix bytes. As shown, the first byte of a multibyte sequence must be a prefix byte, while the remaining bytes must be suffix bytes. Bits 2 and 3 of prefix bytes further partition the byte values into the range 0xC0–0xDF for prefixes of two-byte sequences, the range 0xE0–0xEF for prefixes of three-byte sequences, and the range 0xF0 and above for four-byte sequences.

Proper formation of UTF-8 byte sequences requires that the correct number of suffix bytes always follow a UTF-8 prefix byte and that certain illegal combinations are ruled out. For example, row 2 is used for encoding Unicode values only if at least one of the pqr bits is set; two-byte sequences with the prefix codes 0xC0 and 0xC1 are hence considered illegal. Similarly, sequences beginning with the prefix bytes 0xF5 through 0xFF are also illegal as they would represent code point values above 10FFFF. In addition, there are constraints on the first suffix byte following certain special prefixes, namely that a suffix following the prefix 0xE0 must fall in the range 0xA0–0xBF, a suffix following the prefix 0xED must fall in the range 0x80–0x9F, a suffix following the prefix 0xF0 must fall in the range 0x90–0xBF and a suffix following the prefix 0xF4 must fall in the range 0x80–0x8F. The task of ensuring that each of these constraints hold is known as UTF-8 validation.

Now consider how the tasks of UTF-8 byte classification and validation may be implemented using parallel bit streams, assuming that the byte data has already been transformed to parallel bit stream form. For concreteness, assume that bit streams are

processed in blocks of 128 bits at a time (*bitblocks*) and that u8bit0 through u8bit7 are the bitblocks for bit 0 (most significant) through bit 7 of each code unit. Remember that each set of eight parallel bitblocks represents the UTF-8 byte data for 128 consecutive code units.

Producing new UTF-8 classification bitblocks that identify each of the 128 code units as single bytes (unibyte), prefix bytes or suffix bytes requires simple bitwise logic. The following code fragment performs the relevant SIMD operations using function names consisting of the prefix `simd_` and the name of the logical operation. The `simd_andc` operation performs “and-complement,” logically negating its second argument before performing the bitwise conjunction.

```
u8unibyte = simd_not(u8bit0);
u8prefix = simd_and(u8bit0, u8bit1);
u8suffix = simd_andc(u8bit0, u8bit1);
u8prefix3or4 = simd_and(u8prefix, u8bit2);
u8prefix2 = simd_andc(u8prefix, u8bit2);
u8prefix3 = simd_andc(u8prefix3or4, u8bit3);
u8prefix4 = simd_and(u8prefix3or4, u8bit3);
```

Given these byte classifications, the second step is to define *scope* bitblocks that represent expectations for suffix bytes in terms of preceding prefix bytes. These are formed using an operation that shifts forward an entire block of bits by a known immediate number of bit positions (`bitblock_sfli`). The shift forward operation is either a left or right shift operation depending on endianness of the underlying processor architecture. For example, `u8scope22` represents the positions of expected second suffix bytes of two-byte UTF-8 sequences.

```
u8scope22 = simd_sfli(u8prefix2, 1);
u8scope33 = simd_sfli(u8prefix3, 2);
u8scope44 = simd_sfli(u8prefix4, 3);
u8lastscope = simd_or(simd_or(u8scope22, u8scope33),
                      u8scope44);
u8scope32 = simd_sfli(u8prefix3, 1);
u8scope42 = simd_sfli(u8prefix4, 1);
u8scope43 = simd_sfli(u8prefix4, 2);
u8anyscope = simd_or(simd_or(u8lastscope, u8scope32),
                    simd_or(u8scope42, u8scope43));
```

Overall classification of UTF-8 bytes and calculation of scope streams requires 12 logical operations and 6 bitblock shifts. Working with 128 code unit positions at a time, this represents less than 0.2 operations per byte.

Given the classification and byte streams, the task of UTF-8 validation can now be tackled in accord with the constraints described above. Validation is carried out by computing an `err_mask` identifying those positions at which an error is positively identified. Blocks are assumed to start with complete UTF-8 sequences; any suffix found at the beginning of a block is marked as an error. An incomplete sequence at the end of the block is not marked as an error if it is possible to produce a legal sequence by adding one or more bytes.

The error mask is initialized by checking for prefix-suffix mismatches using a simple xor operation.

```
err_mask = simd_xor(u8anyscope, u8suffix);
```

Note that this will set a one bit in `err_mask` whenever either a suffix byte does not occur when one is expected or when a suffix byte does occur and one is not expected.

The error mask is then extended by checking for invalid prefix bytes 0xC0, 0xC1 or 0xF5 through 0xFF.

```
C0C1 = simd_andc(u8prefix2,
                simd_or(simd_or(u8bit3, u8bit4),
                        simd_or(u8bit5, u8bit6)));
F5FF = simd_and(u8prefix4,
                simd_or(u8bit4,
```

```

        simd_and(simd_or(u8bit6,u8bit7),
                u8bit5));
err_mask = simd_or(err_mask(simd_or(C0C1, F5FF)));

```

Finally, the special requirements on suffix bytes in the contexts of 0xE0, 0xED, 0xF0 or 0xF4 prefix bytes must be checked.

```

EOED = simd_andc(u8prefix3,
                simd_or(simd_or(u8bit6,
                                simd_xor(u8bit4,
                                           u8bit7)),
                        simd_xor(u8bit4, u8bit5)));
EOED_reqt = simd_xor(simd_sfli(u8bit5, 1), u8bit2);
err_mask = simd_or(err_mask,
                  simd_andc(simd_sfli(EOED, 1),
                             EOED_reqt));
FOF4 = simd_andc(u8prefix4,
                simd_or(u8bit4,
                        simd_or(u8bit6, u8bit7)));
FOF4_reqt = simd_xor(simd_sfli(u8bit5, 1),
                    simd_or(u8bit2, u8bit3));
err_mask = simd_or(err_mask,
                  simd_andc(simd_sfli(FOF4, 1),
                             FOF4_reqt));

```

If the final value of `err_mask` computed in this way contains any one bit, a UTF-8 validation error has been identified at that position. If there are no such bits, then there are no UTF-8 errors within the block.

As shown, 26 additional logic operations and 4 additional shift operations suffice to calculate `err_mask`, for a total of 38 logic and 10 shift operations per 128 UTF-8 input bytes. As straight-line code without branching, this process requires less than 0.5 cycles per byte on commodity pipelined processors with 128-bit SIMD units. Assuming that transposition to parallel bit stream form is comparably fast, the overall cost of UTF-8 validation is about an order of magnitude less than byte-at-a-time processing.

The `u8u16` program performs UTF-8 validation using logic substantially as shown here. However, certain optimizations are applied, based on the maximum UTF-8 sequence length found within a block. If the block is confined to entirely ASCII, no validation is required. If the block contains multibyte sequences of at most two bytes in length (no `u8prefix3or4` bit set), then logic for three and four byte sequences is bypassed. Similarly, the logic for four byte sequences is bypassed if there are no four byte sequences. The optimizations are hardly worth applying for validation alone, but become worthwhile when combined with other transcoding tasks.

3. U8-Indexed UTF-16 Bit Streams

Now we consider the problem of transcoding UTF-8 data to UTF-16, the Unicode encoding form based on 16-bit code units. UTF-16 is also a variable-length encoding form, requiring one or two code units (two or four bytes) per character. Table 2 shows how UTF-8 byte sequences translate to corresponding UTF-16 code unit sequences, with each UTF-16 code unit being specified as a pair consisting of a high byte and a low byte.

As shown in the table, one-, two-, and three-byte UTF-8 sequences always translate into a single UTF-16 code unit. In each of these cases, the UTF-16 value is just the Unicode code point value represented as a sixteen bit quantity. The set of Unicode code points that may be represented in this way is known as the basic multilingual plane of Unicode and contains most of the characters that are used in practice. As shown in the table, UTF-16 values in the basic multilingual plane may be produced by distributing bits taken directly from the UTF-8 byte representation.

Beyond the multilingual plane, code point values have more than 16 significant bits. In UTF-8, these values are represented by four-byte sequences. UTF-16 uses *surrogate pairs* to represent such

UTF-8		UTF-16	
Byte Class	Pattern	High Byte	Low Byte
u8unibyte	0tuvwxyz	00000000	0tuvwxyz
u8prefix2	110pqrst	-	-
u8scope22	10uvwxyz	00000pqr	stuvwxyz
u8prefix3	1110jklm	-	-
u8scope32	10npqrst	-	-
u8scope33	10uvwxyz	jklmnpqr	stuvwxyz
u8prefix4	11110efg	-	-
u8scope42	10hijklm	110110ab	cdjklmnp
u8scope43	10npqrst	-	-
u8scope44	10uvwxyz	110111qr	stuvwxyz

(where abcd = efghi - 1)

Table 2. UTF-8 to UTF-16 Mapping

values, pairs of code units having particular bit patterns marked in the high byte of each code unit as shown in row 4 of Table 2. Production of UTF-16 surrogate pairs from UTF-8 requires some bit manipulation: the `abcd` bits of the first code unit of the pair are determined by subtracting 1 from the `efghi` bits taken from the UTF-8 representation (considering the `efghi` bit pattern as a natural 5-bit binary quantity).

As one may appreciate, implementation of byte-at-a-time validation and transcoding logic requires a number of operations per byte to recognize the appropriate cases and decompose UTF-8 byte data into appropriate bit fields, as well as to compose the appropriate UTF-16 values. Now let us consider how this work may be sped up considerably by making use of parallelization in the form of parallel bit streams.

Given UTF-8 bit streams that have been validated in accord with the previous section, conversion to UTF-16 proceeds by first determining a parallel set of 16 bit streams that comprise a *u8-indexed* representation of UTF-16. These are grouped into two sets `u16hi0` through `u16hi7` for the high byte of each unit and `u16lo0` through `u16lo7` for the low byte. The *u8-indexed* representation defines the correct UTF-16 bit values at the following UTF-8 positions: at the single byte of a single-byte sequence (`u8unibyte`), at the second byte of a two-byte sequence (`u8scope22`), at the third byte of a three byte sequence (`u8scope33`), and at the second and fourth bytes of a four-byte sequence (`u8scope42` and `u8scope44`). Table 2 reflects this structure by aligning the UTF-16 code units for each type of sequence with the appropriate index bytes in UTF-8.

Using similar bitwise logic and shifting operations to those shown for UTF-8 validation, the correct values of `u16hi0` through `u16hi7` and `u16lo0` through `u16lo7` are calculated at *u8-indexed* positions. The UTF-16 bit stream values at other positions (that is, `u8prefix2`, `u8prefix3`, `u8prefix4`, `u8scope32`, `u8scope43`) are not significant; no UTF-16 output is to be generated from these positions. Prior to generation of output, data bits at these positions are to be deleted using the parallel deletion operations of the subsequent section. These deletions produce the UTF-16 bit streams in *u16-indexed* form.

Potentially, each UTF-16 bit stream value involves a conditional calculation depending on the various possible byte and scope classifications. However, common logic applies in several cases. For example, except for the first unit of a surrogate pair, bits 2 through 7 of the low UTF-16 byte always correspond to bits 2 through 7 of the last byte of a UTF-8 sequence. These are the bits labeled `uvwxyz` in the four rows of Table 2.

```

u8lastbyte = simd_or(u8unibyte, u8lastscope);
u16lo2 = simd_and(u8lastbyte, u8bit2);
u16lo3 = simd_and(u8lastbyte, u8bit3);
u16lo4 = simd_and(u8lastbyte, u8bit4);
u16lo5 = simd_and(u8lastbyte, u8bit5);

```

```
u16lo6 = simd_and(u8lastbyte, u8bit6);
u16lo7 = simd_and(u8lastbyte, u8bit7);
```

Bit 1 of the low UTF-16 byte (bit *t* in the Table 2) differs depending on whether the current UTF-8 byte is a singleton or the last suffix (*u8lastscope*) of a multibyte sequence. In the latter case, the bit value is shifted forward from the bit 7 position of the preceding UTF-8 byte.

```
u16lo1 = simd_or(simd_and(u8unibyte, u8bit1),
                 simd_and(u8lastscope,
                           simd_sfli(u8bit7, 1)));
```

Again, except for the for the first unit of a surrogate pair, bits 5 through 7 (bits *pqr* in the table) of the high UTF-16 byte and bit 0 of the low UTF-16 byte have a common pattern.

```
u16hi5 = simd_and(u8lastscope, simd_sfli(u8bit3, 1));
u16hi6 = simd_and(u8lastscope, simd_sfli(u8bit4, 1));
u16hi7 = simd_and(u8lastscope, simd_sfli(u8bit5, 1));
u16lo0 = simd_and(u8lastscope, simd_sfli(u8bit6, 1));
```

For blocks containing three-byte sequences in the basic multi-lingual plane, the high five UTF-16 bit streams only become significant at *u8scope33* positions (bits *ijklmn* in the table)

```
u16hi0 = simd_and(u8scope33, simd_sfli(u8bit4, 2));
u16hi1 = simd_and(u8scope33, simd_sfli(u8bit5, 2));
u16hi2 = simd_and(u8scope33, simd_sfli(u8bit6, 2));
u16hi3 = simd_and(u8scope33, simd_sfli(u8bit7, 2));
u16hi4 = simd_and(u8scope33, simd_sfli(u8bit2, 1));
```

Logic for 4-byte UTF-8 sequences is only applied when at least one *u8prefix4* byte is found within a block. In this case the high six UTF-16 bits are set to a fixed bit pattern of 110110 or 110111 for the respective surrogate pair positions.

```
u16surrogate = simd_or(u8scope42, u8scope44);
u16hi0 = simd_or(u16hi0, u8surrogate);
u16hi1 = simd_or(u16hi1, u8surrogate);
u16hi3 = simd_or(u16hi3, u8surrogate);
u16hi4 = simd_or(u16hi4, u8surrogate);
u16hi5 = simd_or(u16hi5, u8scope44);
```

The values for the low ten bit streams at *u8scope44* positions have already been set according to the common patterns given previously, so it is only necessary to extend the definitions of these ten bit streams with the logic for the first UTF-16 code unit of the surrogate pair at the *u8scope42* position. The logic for the most significant four of these bits is complicated somewhat by the requirement that the UTF-16 bit pattern is formed by subtraction of 1 from the corresponding fields of the UTF-8 pattern.

```
s42lo1 = simd_not(u8bit3); /* subtract 1 */
u16lo1 = simd_or(u16lo1, simd_and(u8scope42, s42lo1));
s42lo0 = simd_xor(u8bit2, s42lo1); /* borrow */
u16lo0 = simd_or(u16lo0, simd_and(u8scope42, s42lo0));
borrow1 = simd_andc(s42lo1, u8bit2);
s42hi7 = simd_xor(simd_sfli(u8bit7, 1), borrow1);
u16hi7 = simd_or(u16hi7, simd_and(u8scope42, s42hi7));
borrow2 = simd_andc(borrow1, simd_sfli(u8bit7, 1));
s42hi6 = simd_xor(simd_sfli(u8bit6, 1), borrow2);
u16hi6 = simd_or(u16hi6, simd_and(u8scope42, s42hi6));
```

The logic for the remaining six bits is straightforward.

```
u16lo2 = simd_or(u16lo2, simd_and(u8scope42, u8bit4));
u16lo3 = simd_or(u16lo3, simd_and(u8scope42, u8bit5));
u16lo4 = simd_or(u16lo4, simd_and(u8scope42, u8bit6));
u16lo5 = simd_or(u16lo5, simd_and(u8scope42, u8bit7));
u16lo6 = simd_or(u16lo6, simd_and(u8scope42,
                                   simd_sbli(u8bit2,1)));
u16lo7 = simd_or(u16lo7, simd_and(u8scope42,
                                   simd_sbli(u8bit3,1)));
```

The calculation of 16 parallel bit streams requires approximately four logic and shift operations per stream. However, following the optimization strategy described for UTF-8 validation, the

computations for three-, or four-byte sequences are typically bypassed whenever it is established that a block is confined to multi-byte sequences of maximum length two or three, respectively.

4. Parallel Bit Deletion

As identified in the previous section, the UTF-16 bit streams are initially defined in *u8*-indexed form, that is, with sets of bits in one-to-one correspondence with UTF-8 bytes. However, only one set of UTF-16 bits is required for encoding two or three-byte UTF-8 sequences and only two sets are required for surrogate pairs corresponding to four-byte UTF-8 positions. The *u8lastbyte* and *u8scope42* streams mark the positions at which the correct UTF-16 bits are computed. The bit sets at other positions must be deleted to compress the streams to *u16*-indexed form. A deletion mask *delmask* is computed to identify these positions.

Three different inductive doubling algorithms can be used for performing parallel deletion. Each algorithm involves a preprocessing phase in which deletion information is prepared to control the deletion process. Preprocessing is applied only once, independent of the number of parallel bit streams to which deletion is applied. In each case, the main deletion algorithm is then applied to each of the parallel bit streams in a process that successively computes the solution for a bit width parameter 2^{j+1} in terms of the solution for bit width parameter 2^j . Seven steps are needed for processing the full 128 bit widths of SSE or AltiVec registers. However, three steps suffice to solve the problem in parallel for each of the 8-bit fields within a register; upon application of the inverse transform this will ensure that each register full of UTF-16 doublebyte data is properly compressed.

The parallel-prefix compress algorithm documented by Warren and attributed to Steele [10] uses only logic and shifts with a constant parameter to carry out the deletion process. However, it requires k^2 preprocessing steps for a final field width parameter of size 2^k , as well as 4 operations per deletion step per stream.

Two other algorithms have been developed to take advantage of SIMD multiplication and SIMD rotation capabilities, respectively [3]. The preprocessing phases for each of these algorithms is considerably simpler, requiring only k steps for a final field width parameter of size 2^k . The multiplicative process requires three operations per deletion step per stream, while the rotation process requires only a single SIMD rotation operation per deletion step per stream.

The multiplicative process produces left deletion results in each inductive doubling step. A left deletion result is one in which all nondeleted bits of a field are left-justified within the field and the remaining bit positions are all zeroed out. Thus, for an initial 8-bit data pattern *abcdefgh* and deletion mask 00110100, the left deletion result is *abegh000* and the updated deletion mask is 00000111. Now consider the 16-bit packed pair of 8-bit left results *abegh000jknop0000* in which the second 8-bit left result arose by deletion from an initial data pattern *ijklmnop* and mask 10011010. A single addition suffices to produce the desired 16-bit left result *abeghjknp00000000*. The quantity to be added is simply the 16-bit unsigned product of the left 8-bit deletion mask 00000111 and the right 8-bit value *jknop0000!* In general, the left deletion mask has the value $2^d - 1$ where d is the number of bits deleted. Thus, adding the product of the left deletion mask and the right field value effectively multiplies that field value by 2^d , equivalent to a d bit left shift.

Unfortunately, existing commodity SIMD architectures support the necessary multiplication operations only for a limited set of field widths. Operations on other field widths may be simulated, but the simulations are too slow for practical application.

The ideal approach to parallel deletion uses rotations in a process of central result induction. A central deletion result is one in which nondeleted bits in an n -bit field are moved together in a single consecutive run that touches or spans the center of the field. Central deletion results naturally arise from shifting the nondeleted bits of the left half of a field to the right and the nondeleted bits of the right half of a field to the left. Thus, for an initial 8-bit data pattern `abcdefgh` and deletion mask `00110100`, the central deletion result is `00abegh0` and the updated deletion mask is `11000001`. The left and right shifts can be performed simultaneously with SIMD rotate instructions such as the AltiVec rotate left instruction for simultaneous rotation of 8, 16 or 32 bit fields by a vector of individual rotation values. For example, given the packed pair of 8-bit central results `00abegh000jkn00`, the 16-bit central result `000abeghjkn0000` is obtained by simultaneous rotate left of the two fields by 7 bits and 2 bits respectively.

Although parallel deletion to the full SIMD register widths may be needed in other applications, the implementation of `u8u16` takes advantage of the fact that deletion is the last step before inverse transformation to byte streams. As each register containing UTF-16 doublebyte values is produced, the nondeleted doublebytes should be contiguously located for direct writing to the output stream. Incrementing the output pointer by the number of nondeleted bytes then achieves the effect of deleting the other positions. As 128-bit registers contain up to 8 doublebyte UTF-16 units, only three steps of deletion in the parallel bit stream domain are required with AltiVec or SSE technology. Only two steps are required in working with MMX technology.

A further innovation is employed with the AltiVec technology to make use of its powerful byte permutation capabilities. Rather than applying bit deletions directly to parallel bit streams, deletion is applied to an index vector that can be used to select nondeleted bytes. However, because the byte indices are confined to the range 0 to 15, only 4 bits per index value are needed. Thus, deletions can be applied to packed permutation vectors with 4 bits per index, after which the indices may be unpacked and used to select nondeleted bytes with a `vec_perm` instruction.

5. Transform and Inverse Transform

Efficient implementation of the transform from byte-oriented character stream data to parallel bit stream form is crucial to any application of the parallel bit stream method. The inverse transform is also crucial to UTF-8 to UTF-16 conversion as well as to other applications that compute new character stream data through bit stream manipulations. Fortunately, the overheads imposed by these transforms are tolerable on existing architectures (MMX, SSE, AltiVec) and can be expected to diminish on future processors.

Although the forward transform can be implemented in an iterative fashion using the `pmovmskb` instruction of the MMX and SSE instruction sets, a faster and more general approach is to use SIMD pack instructions in a three stage process of binary division. The pack instructions combine two consecutive registers of data by extracting $n/2$ bits from each n bit field. In the first stage, packing is used to divide byte streams into two separate streams having 4 bits per each byte. For example, one stream may comprise the low nybble of each byte while the other comprises the high nybble, or one may comprise the odd bits of each byte, while the other comprises the even bits. The bits are selected by appropriate combinations of shifting and masking, as required. In the second stage, the two streams having 4 bits per original byte are further packed into two streams each having only 2 bits per original byte. The final stage completes the process by dividing up the streams having 2 bits per byte into the desired bit streams. Details and algorithm variations are described in the technical report [3].

Using the AltiVec instruction set of the Power PC, this three-stage process can be applied to convert a set of 8 consecutive registers full of byte data (16 bytes per 128 bit register; 128 bytes in all) into 8 registers of parallel bit stream data in a mere 72 instructions. At one cycle per instruction, the overhead of this process is well less than one cycle per byte.

The process using MMX or SSE technology is comparable, although additional instructions are required because of the destructive nature of the two-operand instructions versus the AltiVec three-operand instructions, the limited register availability (8 in 32-bit mode, versus 32 for the AltiVec) and the lack of a bitwise if-then-else instruction (`vec_sel` on the AltiVec). Nevertheless, the process typically requires between one and two cycles per byte on recent generation processors.

The inverse transform can be implemented using a similar three-stage application of SIMD merge instructions. The merge instructions interleave fields from parallel registers to generate merged results. The process mirrors that of the forward transform; see the technical report [3] for details. The performance also mirrors that of the forward transform, requiring 72 instructions to generate a stream of 128 bytes on the AltiVec, for example.

In UTF-8 to UTF-16 transcoding, two applications of the inverse transform produce the low and high bytes of each UTF-16 code unit from the low 8 UTF-16 bit streams and high 8 UTF-16 bit streams, respectively. These byte streams are then merged to produce UTF-16 code units in either the UTF-16LE or UTF-16BE forms.

Overall, UTF-8 to UTF-16 transcoding nominally imposes the cost of one forward transform and two inverse transforms for each UTF-8 input byte. However, the `u8u16` program reduces this cost substantially using certain optimizations. First, whenever a sufficiently long run of ASCII bytes is detected, transcoding to UTF-16 is implemented by simply interleaving null bytes with the UTF-8 input bytes, bypassing bit-stream based transcoding entirely. Second, whenever a 128-position block of UTF-8 input data is confined to the Eurocentric subset having at most two UTF-8 bytes per character, it is known that the high 5 bits of the UTF-16 representation are always zero. In this case, an optimized inverse transform is applied.

Although the transform and inverse transform costs are acceptable on present-day commodity processors, it is reasonable to anticipate a reduction in the cost of both the forward and inverse transforms with advances in SIMD processor architecture. These reductions may arise from general improvements in the SIMD processing units (such as the improvement from Pentium to Core 2 architecture), or due to targeted improvements related to matrix transposition support, for example. There is substantial interest in development of high-performance SIMD implementations of matrix transposition for a variety of applications in graphics, signal processing, encryption and other applications. In essence, each of the forward and inverse transforms to and from the parallel bit stream representation is a form of matrix transposition on a fine-grained scale.

For example, one possible development is the extension of SIMD instruction set architecture to provide direct support for inductive doubling algorithms [2]. Inductive doubling refers to a general property of algorithms that involve doubling transitions in the size or number of data elements. The three-pass transformation algorithms described above are examples; other examples include divide-and-conquer bit counting, bit reversal by parallel swap and the parallel bit deletion by central result induction discussed above. With such an extended instruction set architecture, the number of instructions required to implement transposition in the AltiVec three register model reduces from 72 to 24 [3]. With 128-bit SIMD registers, the overhead of transposition should drop to about a fifth of a cycle per byte, at which point it is unlikely to be a significant

cost component even if invoked repeatedly in bulk text processing applications.

In essence, the transform/inverse transform pair for conversion to and from parallel bit stream form may be likened to the Fourier and inverse Fourier transforms for converting between the time and frequency domains in signal processing. While some signal processing calculations are most conveniently carried out in the time domain, others are best carried out in the frequency domain. Analogously, the byte stream domain and the parallel bit stream domain each may be convenient and efficient for particular types of text processing algorithm and less convenient for others. Ideally, a very efficient pair of transform operations will allow conversion to the most convenient form at will.

6. Block Processing Structure

The `u8u16` is structured to perform the complete processing of one block of 64 (MMX) or 128 (Altivec/SSE) bytes of UTF-8 data at a time. Alternative organizations are possible. For example, blocks may be organized in larger buffers with each of the phases of transposition, byte classification, validation, UTF-16 bit stream calculation, parallel bit deletion and inverse transform being applied to a full buffer of data before control is passed on.

In order to handle the possibility of a multibyte UTF-8 sequence crossing a block boundary, a block shortening strategy is used. In essence, blocks may be reduced by up to 3 bytes so that the shortened block contains only full UTF-8 sequences. This shortening is performed by a rather inelegant test performed in byte space. An alternative strategy is to perform bit stream shifting across block boundaries. However, the shifting operations become considerably more expensive with this alternative and the register pressure increased considerably. In practice, block shortening is relatively infrequent and imposes only a small cost on the overall processing.

Further details of the `u8u16` block processing strategy and other implementation details are available in the technical report [3].

7. Performance Results

To study the performance characteristics of the `u8u16` transcoder in different contexts, a series of measurements were made for each of four input types: pure ASCII text, and XML texts in each of German, Arabic and Japanese. The XML texts used as test data are large files downloaded from the Wikimedia project. Table 3 shows the statistical breakdown of blocks processed for each file, categorized by the maximum UTF-8 sequence length found within the block. These categories reflect the different optimizations used within the transcoder. Of course, the ASCII input file consists entirely of single-byte UTF-8 sequences. The German XML text also has a majority of blocks confined to the ASCII (single-byte) subrange of UTF-8, with most of the rest consisting of blocks having a maximum UTF-8 sequence length of 2. This is expected to be typical of Eurocentric XML files. The Arabic XML file was dominated by blocks having at least one 2-byte UTF-8 sequence and no longer sequence. As expected, the Japanese text file was dominated by blocks containing at least one 3-byte sequence. Both the German and Arabic texts had a small number of blocks containing 4-byte sequences.

	ASCII	German	Arabic	Japanese
ASCII only	100%	53.9%	15.9%	16.5%
2-byte max/block	0%	41.6%	78.4%	0.2%
3-byte max/block	0%	4.4%	5.6%	83.2%
4-byte max/block	0%	<0.1%	<0.1%	0%

Table 3. Input Blocks by Maximum UTF-8 Sequence Length

Tables 4 through 6 present overall performance results of the `u8u16` transcoder versus the built-in `iconv` utility on Power PC and Intel platforms. Results are presented in cycles/UTF-8 input byte, with measurements made using built-in processor time base unit (PPC) and timestamp (Intel) counters. All measurements reported for `u8u16` in these tables include the time for performing both the forward and inverse transforms to and from the parallel bit stream domain.

	ASCII	German	Arabic	Japanese
<code>iconv</code>	46.5	47.1	38.1	32.5
<code>u8u16/Altivec</code>	1.7	3.5	4.4	4.6
Altivec Speed-up	27.3	13.5	8.7	7.1

Table 4. `iconv` vs. `u8u16` on Power PC G4/Mac OS X (cycles/byte)

The performance results are most impressive with the Altivec platform as shown in Table 4. The test environment is a Power PC G4 laptop running Mac OS X 10.4, compiling with Apple's customized `gcc 4.1` with Altivec support. Using the standard `iconv` implementation, UTF-8 to UTF-16 transcoding requires over 30 CPU cycles per UTF-8 input byte. In each case, the parallel bit stream implementation of `u8u16` is much faster, requiring fewer than 5 cycles per byte. Similar results have been measured on a G5 platform. Overall, these results demonstrate that the parallel bit stream technology is capable of delivering order-of-magnitude performance improvements in end-to-end bulk text processing.

Examination of the generated code for the Altivec platform suggests that further improvements are possible with better compiler technology. Even with 32 registers, there is sufficient register pressure that `gcc` generates quite a number of instructions to temporarily save SIMD register values on the stack. However, a great many of the register values used in the algorithm are shift or other control constants that may be recomputed in one or a very few instructions without memory access. With a compiler capable of performing systematic register rematerialization for such values, this stacking of temporary values could be reduced considerably. Analysis of the register requirements suggests that it may even be possible to implement the entire body of the main transcoding loop without any memory access other than reading the initial UTF-8 byte stream and writing out the final UTF-16 results.

	ASCII	German	Arabic	Japanese
<code>iconv</code>	34.2	35.8	29.8	26.4
<code>u8u16/MMX</code>	2.4	9.4	18.5	18.7
MMX Speed-up	14.3	3.8	1.6	1.4
<code>u8u16/SSE</code>	1.8	8.1	12.9	13.2
SSE Speed-up	19	4.4	2.3	2.0

Table 5. `iconv` vs. `u8u16` on Intel P4/Linux (cycles/byte)

Table 5 shows performance results on an Intel Pentium 4 platform running Mandriva Linux 10.1, with `u8u16` compiled in both MMX and SSE versions compiled using `gcc 4.1.2`. The results show that the parallel bit stream methods achieve significant speed-ups in all cases. However, examination of the generated assembly code suggests that there is substantial room for further performance improvement by either improved compiler technology or implementation of core algorithm components by hand.

Table 6 shows performance results on an Intel Core 2 platform running Ubuntu Linux 7.04, using the same compiled versions of `u8u16` evaluated on the Pentium platform. The results show that the improvements in the SIMD execution units with the Core 2 processors have significantly increased the performance ratio

	ASCII	German	Arabic	Japanese
iconv	23.2	23.2	20.3	17.6
u8u16/MMX	1.4	4.6	8.2	8.8
MMX Speed-up	16.6	5.0	2.5	2.0
u8u16/SSE	0.9	3.5	5.6	6.2
SSE Speed-up	25.8	6.6	3.6	2.8

Table 6. iconv vs. u8u16 on Intel Core 2/Linux (cycles/byte)

of parallel bit stream technology over byte-at-a-time processing. With further tuning a minimum ratio of 3X over byte-at-a-time transcoding seems feasible.

Optimizations	ASCII	German	Arabic	Japanese
None	6.3	6.9	7.5	7.6
3-byte	5.8	6.4	6.9	7.1
2-byte	5.2	5.9	6.4	7.3
2-byte, 3-byte	5.2	5.8	6.3	7.0
ASCII only	0.9	4.4	6.7	6.6

Table 7. Optimization Levels in u8u16 Performance

To investigate the relative contributions of the various optimizations to u8u16 performance, several versions of the Intel Core 2/Linux program were evaluated with different combinations of optimizations as shown in Table 7. All figures are reported in CPU cycles per UTF-8 input byte. These may be compared with the results when all optimizations are turned on, as shown in the u8u16/SSE row of Table 6. Significantly, if all optimizations are turned off, u8u16 still performs very well compared to iconv. In this case, performance for the various file types is more consistent, although still better for files with straight ASCII text (which require no block shortening for multibyte sequences and which can maintain alignment of byte data when loaded into SSE registers). The 3-byte optimization, in which code for 4-byte sequences in blocks having UTF-8 sequences of maximum length 3, is consistently useful. The consistency is to be expected because 4-byte sequences are so infrequent in practice. However, the 2-byte optimization is more significant for Eurocentric XML documents. This optimization saves work not only during validation and computation of UTF-16 bit streams, but also eliminates deletion operations for the high 5 bit streams and allows a simplified inverse transformation for the high UTF-16 byte. The 2-byte and 3-byte optimizations can be applied together for combined effect.

The ASCII optimization is most significant for documents having even a small number of pure ASCII blocks. This optimization completely bypasses the bulk of the u8u16 algorithm, including both the forward and inverse transforms as well as the validation, decoding and deletion operations on parallel bit streams. In fact, this optimization does not depend on the parallel bit stream method and has been reported useful by itself for improving the performance of UTF-8 to UTF-16 transcoding [4]. However, application of this technique alone only improves behaviour only in proportion to the percentage of identified ASCII subblocks, whereas combination with the parallel bit stream method provides a substantial speed-up even in the worst case.

The overall u8u16 performance numbers can also be broken down by functional unit. Of these, the most significant cost component is that of the forward and inverse transforms. The forward transform on the Intel Core 2 requires 1.6 cycles/byte. The inverse transform, which generates two output bytes for each input byte and is interwoven with the deletion operations requires approximately 4.0 cycles/byte. The core semantics of UTF-8 validation and com-

putation of UTF-16 bit streams is relatively inexpensive by comparison, requiring only 0.9 cycles per byte together.

8. Further Applications

Further applications of parallel bit stream technology are being explored as part of ongoing research. Principal among these is the application of parallel bit stream methods to XML parsing. In general, the approach is to construct lexical item streams identifying lexically significant characters in XML parsing such as the opening and closing angle brackets, single and double quotes delimiting attribute values, whitespace characters and so on. Bit scan operations can then be used to rapidly identify the positions of these delimiters as required. Current prototypes perform basic XML parsing considerably faster than the fast C-based parsers such as expat.

Another interesting area of research is parallel regular expression matching using bit streams, working with certain constrained classes of regular expression that are either deterministic or can be easily transformed to deterministic form. Promising initial results allow parallel matching of all instances of such a regular expression within a bit block, in a number of steps proportional to the number of concatenated expression elements. Application of these techniques to validation of datatype instances in XML schema is particularly promising.

The use of parallel bit streams in string search applications is also being explored. The basic approach is to simultaneously perform partial matching of all positions within a bit block against a partial bit match data set. On average, a partial bit match data set of 12 bits is sufficient to prove the absence of match candidates in a bit block 31 of 32 times. In the case of false positives above the expected rate, an adaptive process adjusts the partial bit match data set to rule out the string or strings most responsible for false positives.

9. PatentLeft Commercialization

A Simon Fraser University spin-off company, International Characters, has been formed to commercialize the results of the ongoing parallel bit stream research using an open source model. Several patent applications have been filed on various aspects of parallel bit stream technology and inductive doubling architecture. However, any issued patents are being dedicated to free use in research, teaching, experimentation and open-source software. This includes commercial use of open-source software provided that the software is actually publically available. However, commercial licenses are required for proprietary software applications as well as combinations of hardware and software.

From an industry perspective, the growth of software patents and open-source software are both undeniable phenomena. However, these industry trends are often seen to be in conflict, even though both are based in principle on disclosure and publication of technology details. It is hoped that the patentleft model advanced by this commercialization effort will be seen as at least one constructive approach to resolving the conflict. A fine result would ultimately be legislative reform that publication of open source software is a form of knowledge dissemination that is considered fair use of any patented technology.

10. Conclusions

As illustrated in the case of UTF-8 to UTF-16 transcoding, the method of parallel bit streams is a viable technique for high performance text processing on commodity processors with SIMD capabilities. Promising initial results have also been obtained in a number of other application areas. As SIMD register widths and processing technologies improve, ratios of performance over byte-

at-a-time text processing are likely to increase further for existing applications and potential applications are likely to broaden.

In order to make the use of parallel bit stream technology accessible to the broader programming community, new language and compiler technologies are needed as well as run-time libraries integrated with popular technology stacks. It seems unlikely that any form of automated compiler technology can be used to transform traditional byte-at-a-time text processing code to the parallel bit stream form. However, one promising direction is the design of new high-level text processing constructs that are at once easy to translate into efficient code based on parallel bit streams and easy for programmers to use to produce high quality code that is less prone to error. For example, it may be possible to design new grammar-based text processing constructs that move beyond the popular but error-prone features of current regular expression packages.

The intraregister parallelism of the parallel bit stream method may also prove to be a convenient stepping stone to intrachip parallelism with multi-core processors. For example, it is quite easy to imagine the distribution of different functional components of the transcoding process based on parallel bit streams to separate cores. This is not at all a reasonable possibility for traditional byte-at-a-time programming. It is also possible to contemplate a data-oriented distribution, with the transcoding of separate data blocks or buffers carried out in parallel on separate cores. The present block-shortening strategy of the u8u16 transcoder is somewhat inconvenient from this perspective, but other options seem feasible, such as a fixed small overlap for distributed block processing. It is anticipated that parallel bit stream applications in other areas will present similar leveraging opportunities to exploit the intrachip parallelism of multi-core processors.

Acknowledgments

This research is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

References

- [1] Apparao, P. and Bhat, M. "A Detailed Look at the Characteristics of XML Parsing," Proceedings of the 1st Workshop on Building Block Engine Architectures for Computers and Networks (BEACON '04). Boston, MA. October, 2004.
- [2] Cameron, Robert D., "Method and Apparatus for an Inductive Doubling Architecture," U.S. Patent Application 11/834,333, filed August 6, 2007.
- [3] Cameron, Robert D., "u8u16 – A High-Speed UTF-8 to UTF-16 Transcoder Using Parallel Bit Streams," Technical Report TR 2007-18, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada, June 2007.
- [4] Jelliffe, Rick. "Using C++ Intrinsic Functions for Pipelined Text Processing," O'Reilly Network, November 7, 2005.
- [5] Kostoulas, M. G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., and Mercaldi, M. "XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization," Proceedings of the 15th International Conference on World Wide Web (WWW '06), pp. 93-102.
- [6] Nicola, Matthias, and John, Jasmi, "XML Parsing: A Threat to Database Performance," Proceedings of the Twelfth International Conference on Information and Knowledge Management, New Orleans, Louisiana, 2003.
- [7] Perkins, E., Kostoulas, M., Heifets, A., Matsa, M., and Mendelsohn, N., "Performance Analysis of XML APIs," XML 2005, Atlanta, Georgia, November 2005.
- [8] Psaila, Giuseppe, "On the Problem of Coupling Java Algorithms and XML Parsers," (Invited Paper), 17th International Conference on Database and Expert Systems Applications (DEXA'06), September 2006, pp. 487-491.
- [9] Ramanathan, R.M., "Extending the World's Most Popular Processor Architecture," *Technology@Intel Magazine*, October 2006.
- [10] Warren, H.S., "Hacker's Delight," Addison Wesley Professional, Boston, 2002, 320 pages.