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

PPoPP '08
Salt Lake City, Utah
Feb. 20-23, 2008
Outline

- Context: XML processing
- Parallel bit streams
- SIMD notation/idealized instructions
- Fast transform to parallel bit streams
- UTF-8 to UTF-16 problem
- UTF-8 byte classification & validation
- UTF-16 bit streams
- Parallel deletion & putting it together
- Performance results
- XML parsing; regular expression match
- Conclusions
Context: XML Processing

- Goal: fastest possible XML processing on commodity processors.
Context: XML Processing

- Goal: fastest possible XML processing on boring processors.
Context: XML Processing

- Goal: fastest possible XML processing on commodity processors.
Context: XML Processing

- **Goal**: fastest possible XML processing on commodity processors.
- **Approach**: use SIMD capabilities of commodity processors to speed-up various tasks.
  - UTF-8 to UTF-16 conversion is a known bottleneck.  [u8u16.costar.sfu.ca](http://u8u16.costar.sfu.ca)
  - XML parsing proper: replace byte-at-a-time scan with bitscan.  [parabix.costar.sfu.ca](http://parabix.costar.sfu.ca)
  - Schema datatype validation with parallel regular expression matching.
Parallel Bit Streams

- Transform 8-bit character (byte) streams to 8 parallel bit streams.
- Ex: u8bit0 stream has bit 0 of each UTF-8 byte.
Parallel Bit Streams

- Transform 8-bit character (byte) streams to 8 parallel bit streams.
- Ex: u8bit0 stream has bit 0 of each UTF-8 byte.
- Process bit streams with 128-bit SIMD registers (e.g. SSE registers on Intel).
Parallel Bit Streams

- Transform 8-bit character (byte) streams to 8 parallel bit streams.
- Ex: u8bit0 stream has bit 0 of each UTF-8 byte.
- Process bit streams with 128-bit SIMD registers (e.g. SSE registers on Intel).
- Ex: classify 128 code units as UTF-8 prefix bytes or not in a single instruction:
  - u8prefix = simd_and(u8bit0, u8bit1);
An idealized SIMD notation simplifies and provides portability to Altivec or SSE (or MMX, ...)

- $r = \text{simd}_\text{op}/w(r1, r2)$
  - simultaneous application of operation $\text{op}$ to all fields of width $w$
- $r = \text{simd}_\text{add}_8(r1, r2)$
  - partition $r$, $r1$ and $r2$ into 8-bit fields
  - add corresponding 8-bit fields of $r1$ and $r2$ to produce fields of $r$
The notation also provides systematic support for *inductive doubling*:
- algorithms that repeatedly double field widths or other data attributes
- SIMD operations defined for all field widths $w = 2, 4, 8, ...$
- Half-operand modifiers may be applied to input operands to select either the high (h) or low (l) $w/2$ bits of each field
Example: compute population count of each 16-bit field of rA → rB

t1 = simd_add/2(rA/l, rA/h)
t2 = simd_add/4(t1/l, t1/h)
t3 = simd_add/8(t2/l, t2/h)
rB = simd_add/16(t3/l, t3/h)
Transposition to Parallel Bit Streams

- Start with 8 consecutive registers $s_0, s_1, s_2, \ldots, s_7$ of serial byte data.
- Produce 8 parallel registers of serial bit stream data $p_0, p_1, \ldots, p_7$.
- Three stage algorithm:
  - produce 2 streams of serial nybble data
  - then 4 streams of serial bitpair data
  - finally 8 streams of serial bit data
- Uses `simd_pack`: $r = \text{simd_pack}/w(a,b)$
  - convert each $w$-bit field of $a$ and $b$ to $w/2$ bits and pack them together consecutively
Idealized Transposition Stages

- High nybble stream (½ of stage 1)
  \[ b_{0123\_0} = \text{simd\_pack}/8(s_{0/h}, s_{1/h}) \]
  \[ b_{0123\_1} = \text{simd\_pack}/8(s_{2/h}, s_{3/h}) \]
  \[ b_{0123\_2} = \text{simd\_pack}/8(s_{4/h}, s_{5/h}) \]
  \[ b_{0123\_3} = \text{simd\_pack}/8(s_{6/h}, s_{7/h}) \]

- Bits 2/3 bitpair stream (¼ of stage 2)
  \[ b_{23\_0} = \text{simd\_pack}/4(b_{0123\_0/l}, b_{0123\_1/l}) \]
  \[ b_{23\_1} = \text{simd\_pack}/4(b_{0123\_2/l}, b_{0123\_3/l}) \]

- Bit 2 and 3 bitstreams (¼ of stage 3)
  \[ \text{bit2} = \text{simd\_pack}/2(b_{23\_0/h}, b_{23\_1/h}) \]
  \[ \text{bit3} = \text{simd\_pack}/2(b_{23\_0/l}, b_{23\_1/l}) \]
Transposition Summary

- Idealized transposition requires 3 stages of 8 operations each.
- Can simulate on Altivec/SSE.
- Better Altivec/SSE algorithms based on pack/16; Altivec: 72 ops/128 bytes.
- Future: CPU support for single-cycle idealized instructions => transposition at 0.2 cycles/byte.
UTF-8 to UTF-16

- UTF-8 is a Unicode format based on 8-bit code units
  - 1 to 4 code units/character
- UTF-16 is based on 16-bit code units
  - 1 or 2 code units/character
- Four translation patterns based on UTF-8 sequence length
  - 1, 2, or 3 byte UTF-8 sequences generate a single UTF-16 code unit
  - 4 byte UTF-8 code sequences generate two UTF-16 code units (a surrogate pair)
<table>
<thead>
<tr>
<th>UTF-8 sequences</th>
<th>UTF-16 sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0tuvwxyz</td>
<td>000000000 0tuvwxyz</td>
</tr>
</tbody>
</table>
## Bit Decoding and Movement Patterns

<table>
<thead>
<tr>
<th>UTF-8 sequences</th>
<th>UTF-16 sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0tuvwxyz</td>
<td>0000000000 0tuvwxyz</td>
</tr>
<tr>
<td>110pqrst</td>
<td>000000pqrst stuvwxyz</td>
</tr>
<tr>
<td>10uvwxyz</td>
<td>000000pqrst stuvwxyz</td>
</tr>
</tbody>
</table>
# Bit Decoding and Movement Patterns

<table>
<thead>
<tr>
<th>UTF-8 sequences</th>
<th>UTF-16 sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0tuvwxyz</td>
<td>0000000000 0tuvwxyz</td>
</tr>
<tr>
<td>110pqrst</td>
<td>000000pqr stuvwxyz</td>
</tr>
<tr>
<td>10uvwxyz</td>
<td>000000pqr stuvwxyz</td>
</tr>
<tr>
<td>1110jklm</td>
<td>jklmnpqrs stuvwxyz</td>
</tr>
<tr>
<td>10npqrst</td>
<td>jklmnpqrs stuvwxyz</td>
</tr>
<tr>
<td>10uvwxyz</td>
<td>jklmnpqrs stuvwxyz</td>
</tr>
</tbody>
</table>
Bit Decoding and Movement Patterns

<table>
<thead>
<tr>
<th>UTF-8 sequences</th>
<th>UTF-16 sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0tuvwxyz</td>
<td>0000000000 0tuvwxyz</td>
</tr>
<tr>
<td>110pqrst</td>
<td>000000pqr stuvwxyz</td>
</tr>
<tr>
<td>10uvwxyz</td>
<td>jklmnpqrs tuvwxyz</td>
</tr>
<tr>
<td>1110jklm</td>
<td>(let abcd = efghi - 1)</td>
</tr>
<tr>
<td>10npqrstuvwxyz</td>
<td>110110ab cdjklmnp</td>
</tr>
<tr>
<td>10npqrst</td>
<td>110111qr stuvwxyz</td>
</tr>
<tr>
<td>10uvwxyz</td>
<td>110111qrstuvwxyz</td>
</tr>
</tbody>
</table>
UTF-8 to UTF-16 with Parallel Bit Streams

- Transform to parallel bit streams
- Classify UTF-8 bytes and validate
- Form u8-indexed UTF-16 bit streams
  - UTF-16 bit streams calculated for every UTF-8 byte position
- Apply parallel bit deletion
  - keep only one UTF-16 position for 1, 2 and 3-byte UTF-8 sequences; 2 positions for 4-byte sequences
- Inverse transform to UTF-16 doublebyte stream
UTF-8 Byte Classification

- UTF-8 bytes are single-byte sequences, or prefixes or suffixes of multibyte sequences.
- Classify 128 at a time.
  
  ```
  u8unibyte = simd_not(u8bit0);
  u8prefix = simd_and(u8bit0, u8bit1);
  u8suffix = simd_andc(u8bit0, u8bit1);
  u8prefix2 = simd_andc(u8prefix, u8bit2);
  u8pfx3or4 = simd_and(u8prefix, u8bit2);
  u8prefix3 = simd_andc(u8pfx3or4, u8bit3);
  u8prefix4 = simd_and(u8pfx3or4, u8bit3);
  ```
- 7 cycles/128 bytes.
UTF-8 Scope Streams

- Identify suffix expectations in terms of prefix bytes.
- Use shift forward logical immediate of 1, 2, or 3 positions.
  
  \[
  \text{scope22} = \text{simd_sfli(u8prefix2, 1)};
  \]
  
  \[
  \text{...}
  \]
  
  \[
  \text{scope43} = \text{simd_sfli(u8prefix4, 2)};
  \]
  
  \[
  \text{scope44} = \text{simd_sfli(u8prefix4, 3)};
  \]
  
  \[
  \text{s_nn} = \text{simd_or(simd_or(scope22, scope33), scope44)};
  \]
  
  \[
  \text{any} = \text{simd_or(simd_or(scope32, scope42), simd_or(scope43, s_nn))};
  \]

- 6 shifts, 5 logic ops/128 bytes.
UTF-8 Validation

- Suffixes must occur where expected.
  \[ \text{err\_mask} = \text{simd\_xor(any, u8suffix)}; \]
- Prefix bytes 0xC0, 0xC1 are illegal.
  \[ \text{C0C1} = \text{simd\_andc(u8prefix2,} \]
  \[ \text{simd\_or(simd\_or(u8bit3, u8bit4),} \]
  \[ \text{simd\_or(u8bit5, u8bit6));} \]
  \[ \text{err\_mask} = \text{simd\_or(err\_mask, C1)}; \]
- Other constraints similar.
- 26 logic and 4 shift operations for validation.
U8-indexed UTF-16 Bit Streams

- Calculate UTF-16 bit streams as follows.
  - At last byte position for 1, 2 and 3-byte sequences.
  - At scope22 and scope44 positions for 4-byte UTF-8 sequences.
  - Values at prefix, scope32 and scope43 positions will ultimately be deleted.
  - Each UTF-16 bit stream is a logical combination dependent on scope streams.
  - Approximately 4 operations per stream.
Example UTF-16 Stream Equations

last = simd_or(u8unibyte, s_nn);
u16lo2 = simd_and(last, u8bit2);
    /* same pattern up to u16lo7 */
u16lo0 = simd_and(s_nn, simd_sfli(u8bit6, 1));
u16lo1 = simd_or(simd_and(u8unibyte, u8bit1),
                 simd_and(last, simd_sfli(u8bit7, 1)));
s42lo1 = simd_not(u8bit3); /* sub 1 */
u16lo1 = simd_or(u16lo1,
                 simd_and(u8scope42, s42lo1));
s42lo0 = simd_xor(u8bit2, s42lo1);
u16lo0 = simd_or(u16lo0, /* borrow */
                 simd_and(u8scope42, s42lo0));
Parallel Bit Deletion

- The most complex aspect of UTF-8 to UTF-16 conversion is compression of the UTF-8 multibyte sequences to one or two UTF-16 code unit positions.
- A deletion mask is formed to mark u8prefix, u8scope32 and u8scope43 positions.
- One of three inductive doubling algorithms can be used for deletion.
Deletion by Central Result Induction

- Deletion steps move bits to the center.
- Left shift the right half; right shift the left.
  - abcdefgh data pattern
  - 00110100 deletion mask
  - 00abegh0 is the central result
  - 110000001 is the updated deletion mask

- Use SIMD rotate to move from central result for n/2 bit fields to n-bit fields.
  - 00abegh0 00jknp00 data pattern d
  - 00000111 00000010 rotate factor f
  - 000abegh jknp00000 simd_rotl_8(d, f)
Putting it All Together

- Inverse transformation produces high and low UTF-16 byte streams.
- These are then merged.
- UTF-8 sequences straddling block boundaries must be handled.
  - block shortening
- Optimizations applied for blocks whose max sequence length is 1, 2 or 3.
- See u8u16.costar.sfu.ca for source code, with documentation in Knuth's literate programming system.
Performance Results

- Measure speed-up from iconv to u8u16.
  - Power PC G4/Altivec running Mac OS X.
  - Intel Core2/SSE running Ubuntu Linux.
- Measure CPU cycles per UTF-8 byte.
- XML: German, Arabic, Japanese, ASCII.
Performance Results

- Measure speed-up from iconv to u8u16.
  - Power PC G4/Altivec running Mac OS X.
  - Intel Core2/SSE running Ubuntu Linux.
- Measure CPU cycles per UTF-8 byte.
- XML: German, Arabic, Japanese, ASCII.
- German:
  - 47 cycles/byte on Altivec to 3.5 (13.5X)
  - 23.2 cycles/byte on SSE to 3.5 (6.6X)
Performance Results

- Measure speed-up from iconv to u8u16.
  - Power PC G4/Altivec running Mac OS X.
  - Intel Core2/SSE running Ubuntu Linux.
- Measure CPU cycles per UTF-8 byte.
- XML: German, Arabic, Japanese, ASCII.
- German:
  - 47 cycles/byte on Altivec to 3.5 (13.5X)
  - 23.2 cycles/byte on SSE to 3.5 (6.6X)
- Japanese:
  - 32.5 cycles/byte on Altivec to 4.6 (7.1X)
  - 17.6 cycles/byte on SSE to 6.2 (2.8X)
Performance Results

- Measure speed-up from iconv to u8u16.
  - Power PC G4/Altivec running Mac OS X.
  - Intel Core2/SSE running Ubuntu Linux.
- Measure CPU cycles per UTF-8 byte.
- XML: German, Arabic, Japanese, ASCII.
- German:
  - 47 cycles/byte on Altivec to 3.5 (13.5X)
  - 23.2 cycles/byte on SSE to 3.5 (6.6X)
- Japanese:
  - 32.5 cycles/byte on Altivec to 4.6 (7.1X)
  - 17.6 cycles/byte on SSE to 6.2 (2.8X)
- ASCII: about 25X speed-up on either.
XML Parsing

- Form lexical item streams for particular XML character classes.
  - Markup_start, NameFollow, Quote, CDend, Hyphen, Qmark, WhiteSpace
- Use bit scan operations to find the next byte in the class, replacing byte-at-a-time looping.
- Initial tests: 7X faster than C-based byte-at-a-time parsing (expat).
- parabix.costar.sfu.ca
Parallel Matching of \([-+]?[0-9]+\) Regular Expression

character stream
5.796953 - 6++ 4+ gnorw 17- 421
character class
00000000001001100100000000010000
[0-9] character class
01011111100010001000000001100111
initial cursor
c0, initial cursor
000000001010001001000001000010000
end_mask
Parallel Matching of [-+]?[0-9]+ Regular Expression

; 5.796953 - 6++ 4+ gnorw 17- 421 character stream

00000000001001100100000000010000 [-+] character class

00000000101000100100000100010001 c0, initial cursor

00000000110001001000000100100001 c1 = c0 + (c0 & [-+])
Parallel Matching of [-+]?[0-9]+ Regular Expression

;5.796953 - 6++ 4+ gnorw 17- 421 character stream

01011111000100010000001100111 [0-9] character class

00000000110001001000000100100001 c1 = c0 + (c0 & [-+])
001000000000001000000010001000 (c1+[0-9])&~[0-9] &~c1
Regular Expression Matching

Parallel Matching of \([-+]?[0-9]+\) Regular Expression

;5.796953 - 6++ 4+ gnorw 17- 421

character stream

\([-+]?\) character class

[0-9] character class

c0, initial cursor

c1 = c0 + (c0 & \([-+\)])

end_mask

(c1+[0-9])&~[0-9] &~c1

end_mask & c2

Three complete matches found.
Conclusions

- Parallel bit stream technology:
  - delivers benefits of parallel programming to the desktop.
  - has the potential to transform the way the world does text.
  - will have an increasing advantage over byte-at-a-time methods as hardware advances.
  - needs support by new compiler/language technology.
  - could enable use of high-level grammar-based text processing languages.