Accelerating UTF-8 Decoding Using SIMD Instructions

Hiroshi Inoue
IBM Research – Tokyo
Mar 17, 2008
Background and Goal

- **Background**
  - UTF-8 encoding is commonly used to exchange text data (e.g. HTML, XML)
  - But, Java VM uses UTF-16 as its internal representation
  - Conversion between UTF-8 and UTF-16 matters

- **Goal**
  - Accelerate conversion from UTF-8 to UTF-16
UTF-8 and UTF-16

- What is UTF-8 and UTF-16
  - UTF-8: variable length encoding (from 1 byte to 3 bytes per character)
  - UTF-16: constant length encoding (2 bytes, other than surrogate pair)

- Mapping UTF-8 onto UTF-16

<table>
<thead>
<tr>
<th>range</th>
<th>UTF-8</th>
<th>UTF-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-7F (ASCII char)</td>
<td>0xxxxxxxxx</td>
<td>0000000000xxxxxxxxx</td>
</tr>
<tr>
<td>80-7FF (Greece char etc)</td>
<td>110yyyyy 10xxxxxx</td>
<td>00000yyyyyyxxxxxxxxx</td>
</tr>
<tr>
<td>800-FFFFF (Chinese, Japanese char etc)</td>
<td>1110zzzz 10yyyyyy 10xxxxxx</td>
<td>zzzzyyyyyyyyyyyyyyyyyyyyyyy</td>
</tr>
</tbody>
</table>
Decoding UTF-8 to UTF-16

Naive implementation of decoding process based on *conditional branches*

```plaintext
while (not end of UTF-8 sequence) {
    if first byte of UTF-8 starts with ‘0’ then
        convert 0xxxxxxx \(\rightarrow\) 000000000xxxxxxx
    else if first byte of UTF-8 starts with ‘110’ then
        convert 110yyyyy 10xxxxxx \(\rightarrow\) 00000yyyyyxxxxxx
    else if first byte of UTF-8 starts with ‘1110’ then
        convert 1110zzzz 10yyyyyy 10xxxxxx \(\rightarrow\) zzzzyyyyyyxxxxxx
}
```

Main source of overhead

- Branch mispredictions caused by data-dependent conditional branches that are hard to predict (when having multiple types of characters)
We use a simplified format “**simple encoding**” for ease of explanation.

**simple encoding**
- variable length encoding for integers up to 30-bits long
- 1 value is encoding to 1 to 4 bytes
- Most significant 2 bits of the first byte show the length

<table>
<thead>
<tr>
<th>length</th>
<th>representation in simple encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>00xxxxxxx</td>
</tr>
<tr>
<td>2 bytes</td>
<td>01yyyyyy  xxxxxxxxxx</td>
</tr>
<tr>
<td>3 bytes</td>
<td>10zzzzzz  yyyyyyyy  xxxxxxxxxx</td>
</tr>
<tr>
<td>4 bytes</td>
<td>11aaaaaaaa zzzzzzzzz  yyyyyyyy  xxxxxxxxxx</td>
</tr>
</tbody>
</table>
Our Technique without conditional branches

- Steps for conversion
  - Step 1: Identify data length for next few data (by scalar)
  - Step 2: Load required constants based on the data length identified in Step 1 (by scalar)
  - Step 3: Move data using permute instruction (by SIMD)
  - Step 4: Mask off unused bits (by SIMD)

SIMD Permute (Shuffle) instruction

- reorder input bytes based on a pattern specified by register at runtime (not compile time!)
- many SIMD ISAs have similar instructions (e.g. VMX, SSE)
// Step 1: Identify data length for next few data
int gathered_prefix = 0;
int position = 0;
for (i=0; i<4; i++) {
    prefix = (p[position] >> 6);
    gathered_prefix = (gathered_prefix << 2) | prefix;
    position += prefix+1;
}

// step 2: Load required constants
vector char pattern = pattern_table[gathered_prefix];
vector char mask = mask_table[gathered_prefix];

// step 3: Move data using permute instruction
vector char vin = vector_load(p);
vin = vector_permute(vin, pattern);

// step 4: Mask off unused bits
output = vector_and(vin, mask);
Example

data in simple encoding:

prefix = '00'
prefix = '01'
prefix = '10'
prefix = '11'

load pattern['00011011']

Constant table:

pattern[0] = {...}
pattern[1] = {...}
pattern[2] = {...}
Example

Data in simple encoding: 01 42 34 85 67 89 CA BC DE F0 05 F1

Reorder bytes using a permute instruction:
pattern = {X, X, X, 0, X, X, 1, 2, X, 3, 4, 5, 6, 7, 8, 9}

Mask off unused bits using AND instruction:
mask = {0, 0, 0, 3F, 0, 0, 3F, FF, 0, 3F, FF, FF, 3F, FF, FF, FF}

Decoded data:
00 00 00 01 00 00 02 34 00 05 67 89 0A BC DE F0

As 32-bit integer values:
0x01 0x0234 0x56789 0x0ABCDEF0
Apply our technique for UTF-8 to UTF-16 conversion

- Not that different from the case for simple encoding...
- Some difference:
  - use **two permute** and **two select** instructions instead of just one permute instruction
  - use four constant vector values per iteration
  - convert 8 characters (2 bytes each) in each iteration
UTF-8 to UTF-16 Conversions

// step 1: gather prefix of 8 characters and convert them to length in bytes
for (i=0; i<8; i++) {
    prefix = (p[total_length] >> 3);
    length = prefix_to_length_table[prefix];
    gathered_prefix = (gathered_prefix * 3) + length;
    total_length += length;
}

// step 2: load constants from tables
vpattern1 = pattern1_table[gathered_prefix];
vpattern2 = pattern2_table[gathered_prefix];
vmask1    = mask_for_select_table[gathered_prefix];
vmask2    = mask_for_and_table[gathered_prefix];

// step 3: move data bits using constants
vin1  = vector_load(p);
vout  = vector_load(p+16);
vtmp1 = vector_permute(vin1, vin2, vpattern1);
vtmp2 = vector_permute(vin1, vin2, vpattern2);
vtmp1 = vector_select(vector_shift_left(vtmp1, 4),
                     vector_shift_left(vtmp1, 6),
                     vconstant_0x0FC0);
vtmp3 = vector_select(vtmp1, vtmp2, vmask1);

// step 4: mask off unused bits
vout  = vector_and(vtmp3, vmask2);
Additional optimization for real-world text

- Same type (i.e. same length in UTF-8 representation) of characters tends to appear repeatedly in real-world data
  - If all characters converted in an iteration have same type, we use specialized code for this data type
    - Specialized code for ASCII characters are especially efficient because it simply inserts 0 before each byte
      \[ \text{0xxxxxxx } \rightarrow \text{00000000 0xxxxxxx} \]
    - Specialized code fall backs to the default code if input values include a value of other data type
- Applied for both vectorized version and serial version
Evaluation

- Tested on PowerPC 970MP 2.5 GHz running Linux 2.6.18
- Implemented using VMX intrinsics
- Compiled with gcc-4.0.1
Decode performance for **simple encoding format**

- Scalar processing caused frequent branch mispredictions for random input
- Performance of our technique does not depend on input data
Performance of UTF-8 decoding with artificial text

<table>
<thead>
<tr>
<th>Input Text</th>
<th>ASCII only</th>
<th>3-bytes character only</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles per Character</td>
<td>x12.3</td>
<td>x15.2</td>
<td>x3.1</td>
</tr>
</tbody>
</table>
Performance of UTF-8 decoding with realistic text

![Graph showing performance improvement over scalar processing for different input texts: Japanese, Chinese, Korean, German, Russian, Japanese, Japanese. German shows the highest improvement, while Japanese has the lowest.]

Web (HTML) faster than mail (plain text).
Future work

- To integrated into Java environment
  - Conversion is implemented in Java, where programmer cannot directly write platform-dependent SIMD code
  - JIT compiler must generate SIMD instructions for each platform

- UTF-16 to UTF-8 conversion
  - As important as UTF-8 to UTF-16 conversion
  - Our technique is applicable. However, writing UTF-8 sequence to memory require costly unaligned memory store
Summary

- We developed decoding technique for variable-length encoding format without using hard-to-predict conditional branches

- We observed significant performance boost in UTF-8 to UTF-16 conversion with real-world text data
backup
Comparison with Cameron [PPoPP 2008]

![Bar chart showing performance improvement comparison between different languages and contexts. The chart compares 'ours' and 'u8u16, Cameron [2008]' in English, Germany (Web), Japanese (Web), and Japanese (e-mail). The chart indicates that 'ours' is faster in all cases.]