SIMD-Accelerated Regular Expression Matching

Eva Sitaridi, Orestis Polychroniou, Ken Ross
Columbia University

DaMoN 2016, San Francisco, California
In-Memory Query Execution

• Bottleneck shifts from disk-access to RAM access
• Hardware-aware processing to reach RAM bandwidth
  • Mainstream multi-cores (e.g. Intel Haswell, Skylake)
    • Complex cores
    • Aggressively out-of-order
  • Many-core accelerators (e.g. Intel Xeon Phi)
    • Simpler cores
    • In-order execution
    • Wider SIMD width and more cores
Vectorized Database Operators

- Scan filters: e.g. age≥21
  - Saturate RAM bandwidth ✓
    - Use SIMD instructions & thread parallelism
- LIKE operators: e.g.: comment LIKE ‘%packages%’
  - Algorithms
    - Boyer-Moore
    - Knuth-Morris-Pratt
    - DFA
    - SSE instruction
  - Can we reach RAM bandwidth?
Saturate RAM bandwidth by using SSE cmpistri/cmpestri instructions

- For pattern ≤16 chars (common case for DB queries)
Databases & Regular Expressions

• Use more expressive predicates: RLIKE or REGEXP
  • Validate string columns: E-mail address
  • Pattern-matching
    • Her OR She
    • Her AND She
    • Her AND NOT She

Regular expressions mapped to Deterministic Finite Automata (DFA)
Example: E-mail DFA

Stopping conditions
- Invalid string value: e.g. user@@mail.com
- Locate search pattern: Valid addresses in a specific street
- Consume full input string
DFA-Based Expression Matching

- DFA Representation with $N$ states and $c$ alphabet size
  - Array of $N \times c$ characters
  - Special states
    - Reject sink
    - Accept sink
  - Reorder states to simplify termination condition
    - -2: Accept sink
    - -1: Reject sink
    - [0,N)
DFA Transition Array Example

<table>
<thead>
<tr>
<th>Current state</th>
<th>‘a’</th>
<th>‘b’</th>
<th>‘c’</th>
<th>…</th>
<th>‘@’</th>
<th>‘#’</th>
<th>‘%’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
<td>-1</td>
<td>…</td>
<td>-1</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>…</td>
<td>-2</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>-1</td>
<td>…</td>
<td>5</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

DFA footprint: N states x 256 characters x Bytes/state
bool regexp(...){
    size_t i = 0, s = initial_state;
    do {
        s = dfa[(s << 8) | string[i]];
    } while (0 <= (ssize_t) s && ++i != str_len);
    return s + 1 > reject_states; }

Optimize scalar performance
Scalar DFA matching

```cpp
bool regexp(...){
    size_t i = 0, s = initial_state;
    do {
        s = dfa[(s << 8) | string[i]];
    } while (0 <= (ssize_t) s && ++i != str_len);
    return s + 1 > reject_states;
}
```

Optimize scalar performance

- Use integer arithmetic to access the DFA transition table
Scalar DFA matching

```c
bool regexp(...){
    size_t i = 0, s = initial_state;
    do {
        s = dfa[(s << 8) | string[i]];
    } while (0 <= (ssize_t) s && ++i != str_len);
    return s + 1 > reject_states;
}
```

Optimize scalar performance

- Use integer arithmetic to access the DFA transition table
- Minimize branches
Previous Approach: Lock-step Processing

1) Interleave

2) Compute offsets

3) Gather updated state

DFA Transition table (Memory)
Previous Approach: Lock-step Processing

Weakness: Lanes completed underutilized!
Our Approach: Selective Loads & Stores

Short Circuiting: Replace completed strings

Fixed length strings

vector of rids

output rids
(memory)

input index

output index
Our Approach: Selective Loads & Stores

Short Circuiting: Replace completed strings

Input index:

Output index:

Fixed length strings

Selective store

vector of rids

output rids
(memory)
Our Approach: Selective Loads & Stores

[Diagram showing sequence of numbers and operations involving input index, vector of rids, output rids, and output index with labels: Short Circuiting: Replace completed strings, Selective store, Replace completed rids, Fixed length strings, and Selective store.]

1. **Input Index**: 40, 45, and so on.
2. **Vector of Rids**: 14, 30, 16, 32, 36, 37, 38, 39.
3. **Output Rids**: 5, 10, 14, 37, and so on.
4. **Output Index**: 6, 8, and so on.

- Replace completed rids and strings.
- Short Circuiting: Replace completed strings.
Our Approach: Buffer Multiple Bytes

- Executing a gather of 1 byte same cost with gather of a 4-byte word
- A non-trivial portion per string processed to determine if matches regular expression

Buffer multiple characters per string
Our approach: Gather Unaligned Input
Experimental Results Overview

- DFA Matching Implementations
  - Scalar
  - Lock-step (AVX2)
  - Short circuit AVX2 (Unrolling/No unrolling)
  - Short circuit Intel Xeon Phi (Unrolling/No unrolling)

- Experiments for URL validation (90 total states)
  - Vary string length
  - Vary matching failure point
  - Vary matching selectivity
Vary String Length – Haswell

Throughput (GB/s) on Haswell

- Scalar
- Vector (x1)
- Vector (x2)
- Vector (ls)
Vary String Length – Xeon Phi

Throughput (GB/s) on Xeon Phi

- Scalar
- Vector (x1)
- Vector (x2)
Varying Matching Failure Point

![Graph showing throughput vs. DFA traversal length for Haswell and Xeon Phi processors with different vectorization levels.](image-url)
Vary Matching Selectivity

Throughput (GB/s)

- Scalar
- Vector (x1)
- Vector (x2)
- Vector (ls)

Selectivity (%)

- Haswell
- Xeon Phi

0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75

1 2 5 10 20 50 100

1 2 5 10 20 50 100
Conclusions

• Vectorize regular expression matching
  • Prior approach: Lock-step input processing
  • Our approach: Dynamically replace completed input
    • 2X speed-ups on mainstream CPU & 5X speed-up on the Intel Xeon Phi
    • Maximum speed-ups for early failures or matches

• Show impact of vectorization on code bound by cache access
Conclusions

• Vectorize regular expression matching
  • Prior approach: Lock-step input processing
  • Our approach: Dynamically replace completed input
    • 2X speed-ups on mainstream CPU & 5X speed-up on the Intel Xeon Phi
    • Maximum speed-ups for early failures or matches

• Show impact of vectorization on code bound by cache access
Back-up slides
### Early Termination Examples

<table>
<thead>
<tr>
<th><strong>Regex</strong></th>
<th><strong>Scenario with possible early failures</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>invalid character, e.g., me @mail.com</td>
</tr>
<tr>
<td></td>
<td>double @ symbol, e.g., me@@@mail.com</td>
</tr>
<tr>
<td></td>
<td>specific domain: mail.com, e.g. <a href="mailto:me@meil.com">me@meil.com</a></td>
</tr>
<tr>
<td></td>
<td>specific username: john, e.g. <a href="mailto:jim@mail.com">jim@mail.com</a></td>
</tr>
<tr>
<td>URL</td>
<td>invalid character, e.g., <a href="http://site">http://site</a> .com</td>
</tr>
<tr>
<td></td>
<td>invalid scheme, e.g., <a href="http://site.com">http://site.com</a></td>
</tr>
<tr>
<td></td>
<td>specific site: site.com, e.g., <a href="http://no.com/a/b/c">http://no.com/a/b/c</a></td>
</tr>
<tr>
<td></td>
<td>specific IP: 125.1.<em>.</em>, e.g., <a href="http://125.2.0.0/a/b/c">http://125.2.0.0/a/b/c</a></td>
</tr>
<tr>
<td></td>
<td>specific path depth: 2, e.g., <a href="http://site.com/a/b/c">http://site.com/a/b/c</a></td>
</tr>
<tr>
<td>address</td>
<td>missing street number before street name</td>
</tr>
<tr>
<td></td>
<td>non-numeric symbol in postal code</td>
</tr>
<tr>
<td></td>
<td>specific street number in valid address</td>
</tr>
<tr>
<td>name</td>
<td>invalid symbol, e.g., J0hn Smith</td>
</tr>
<tr>
<td></td>
<td>lowercase first letter, e.g., bob Smith</td>
</tr>
<tr>
<td></td>
<td>specic surname: Stark, e.g., Peter Parker</td>
</tr>
</tbody>
</table>
URL Regular Expression

scheme, username, hostname (or IP), port, path, query, fragment

^\(ht|f)tp(s)?://(([!$&'()*+,;=A-Za-z0-9-]+@)?

(((\[~!$&'()*+,;=A-Za-z0-9-] | ([%0-9A-F]{2})+).+) + [a-zA-Z]{2,4})


\( ([._~!$&'()]*+,;=A-Za-z0-9:@-] | (%[0-9A-F]{2})+)\)*

([^._~!$&'()]*+,;=A-Za-z0-9:@/-] | (%[0-9A-F]{2})+)*

#([^._~!$&'()]*+,;=A-Za-z0-9:@/-] | ([%0-9A-F]{2})+)*$
Varying String Length

Throughput (GB/s) on Haswell

String length

0% of string

10% of string

50% of string

100% of string

Scalar

Vector (x1)

Vector (x2)

Vector (ls)
Varying DFA Size

Throughput (GB/s) vs DFA size (states)

Haswell

DFA size (states)

Xeon Phi

Scalar
Vector (x1)
Vector (x2)
Vector (ls)