SIMD-Accelerated Regular Expression Matching

Evangelia Sitaridi[*]
Columbia University
eva@cs.columbia.edu

Orestis Polychroniou
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross[†]
Columbia University
kar@cs.columbia.edu

ABSTRACT

String processing tasks are common in analytical queries powering business intelligence. Besides substring matching, provided in SQL by the `like` operator, popular DBMSs also support regular expressions as selective filters. Substring matching can be optimized by using specialized SIMD instructions on mainstream CPUs, reaching the performance of numeric column scans. However, generic regular expressions are harder to evaluate, being dependent on both the DFA size and the irregularity of the input. Here, we optimize matching string columns against regular expressions using SIMD-vectorized code. Our approach avoids accessing the strings in lockstep without branching, to exploit cases when some strings are accepted or rejected early by looking at the first few characters. On common string lengths, our implementation is up to 2X faster than scalar code on a mainstream CPU and up to 5X faster on the Xeon Phi coprocessor, improving regular expression support in DBMSs.

1. INTRODUCTION

Modern hardware advances have made a fundamental impact on the design and implementation of database systems. The increase in main-memory capacity allows small to medium-scale databases to fit in RAM, shifting the performance bottleneck from the disk to the RAM bandwidth.

In-memory query execution strives to exploit all kinds of parallelism provided by modern CPUs in order to saturate the RAM bandwidth, the most fundamental of which is thread parallelism, driven by the advent of multi-core CPUs.

In the context of databases, scan operators, besides using multiple threads, also utilize SIMD vector instructions to maximize efficiency. When the selective predicates are simple, e.g., `salary > 10000`, multi-threaded scans using SIMD instructions process the data faster than it can be fetched to the CPU, saturating the RAM bandwidth bottleneck [8].

[*]Supported in part by the Onassis Foundation Scholarship.
[†]This research is supported by National Science Foundation grants IIS-1218222, IIS-1422488 and a gift from Oracle Corp.

DaMoN '16, June 26, 2016, San Francisco, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4319-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2933349.2933357

1.1 Substring Matching

Substring matching is a well-studied problem, the most popular algorithms being Knuth-Morris-Pratt [5] and Boyer-Moore [2]. Both methods improve over the worst-case $O(n^2)$ brute-force algorithm, by using pre-computed arrays of offsets for mismatches to achieve $O(n)$ worst-case complexity. The pre-processing step is dependent on the pattern only and is trivial in databases where a single pattern is matched against many tuples. The Boyer-Moore code is shown below. The arrays `pat_jmp` and `sym_jmp` are pre-computed once.

```
bool like(const uint8_t *str, const uint8_t *pat,
          size_t str_len, size_t pat_len, [...]) {
   size_t i = pat_len - 1;
   while (i < str_len) {
      uint8_t b = str[i];
      size_t j = pat_len - 1;
      while (b == pat[j]) {
         if (j-- == 0) return true;
         b = str[--i]; }
      i += max(pat_jmp[j], sym_jmp[b]); }
   return false; }
```

Recent mainstream CPUs offer specialized SIMD instructions for string processing. With suitable parametrization, the instructions can be used to implement substring matching. Specifically, the SSE 4.2 128-bit SIMD instruction set in mainstream CPUs provides the `cmpestr` and `cmpistr` instructions that can match against patterns that fit in a 128-bit SIMD register. The algorithm resembles the brute force approach but runs in worst-case $O(n)$ for patterns up to 16 bytes. We show the implementation below using intrinsics for SIMD instructions. A guide to SIMD intrinsics is available online.[1] The code loads the input string and, when a partial match is found, the string is reloaded from the starting position of the partial match to re-test for a full match.

```
bool like(const uint8_t *string, __m128i pat, [...]) {
   size_t i = 0;
   while (i + 16 < str_len) {
      __m128i str = _mm_loadu_si128(&string[i]);
      size_t j = _mm_cmpistri(pat, str, 12);
      if (j >= 16) i += 16;
      else {
         if (j + pat_len <= 16) return true;
         i += j; } }
   if (i + pat_len <= str_len) {
      __m128i str = _mm_loadu_si128(&string[i]);
      size_t j = _mm_cmpestri(pat, pat_len,
                              str, str_len - i, 12);
      if (j < 16 && j + pat_len <= 16) return true; }
   return false; }
```

[1]software.intel.com/sites/landingpage/IntrinsicsGuide/

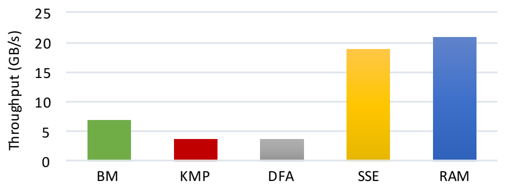

Figure 1: Substring matching for TPC-H Q13

Figure 1 shows the performance of different algorithms for substring matching on TPC-H Q13 (scale factor 300) using multiple threads on a mainstream 4-core CPU. The query has a `like '%special%packages%'` operator that matches patterns `special` and `packages` in that order. By nesting two calls of substring matching that return the match position, we can implement a sequence of pattern matches.

Substring matching without using the specialized hardware instruction is far from the RAM bandwidth, due to branch dependencies for every character of the input string. Knuth-Morris-Pratt (KMP) is very similar to a deterministic finite automaton (DFA) that matches the same pattern but uses an ad-hoc jump table for failed matches. The DFA is implemented using a two-dimensional transition table for each state × all possible values per string character. The number of states is equal to the pattern length. As expected, KMP and the DFA have similar performance since both scan the entire string if there is no match. Boyer-Moore (BM) is much faster than KMP due to skipping a large portion of each input string. Still, we cannot saturate the RAM bandwidth using scalar code, even if we use all hardware threads.

1.2 Regular Expression Matching

While a single instruction is enough to cover most queries with substring matching operators, more advanced predicates such as regular expression matching cannot be optimized as easily. Popular databases offer regular expression matching predicates such as `regexp_like` in Oracle DB, or `rlike/regexp` in MySQL. For example, this MySQL query returns the number of employees with valid e-mail addresses:

```
select count(*) from employees
where email regexp  # or "rlike"
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4}$'
```

To match a string against a regular expression, we typically construct a DFA. DFAs have a number of states that transition to other states based on the next character of the string. Because each character is processed only once, DFAs take worst-case $O(n)$ time, where n is the input string length. The DFAs are represented by an $s \times c$ transition table having s states and c character values. The number of states s depends on the complexity of the regular expression.

We show a DFA that validates e-mail addresses in Figure 2. The DFA has 9 states and `S` is the starting state. The double-circled states `T2`, `T3`, and `T4` are accepting states.

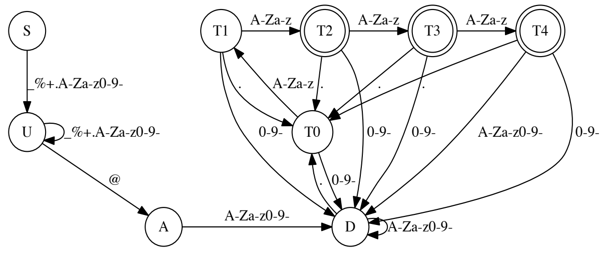

Figure 2: A DFA that validates e-mail addresses

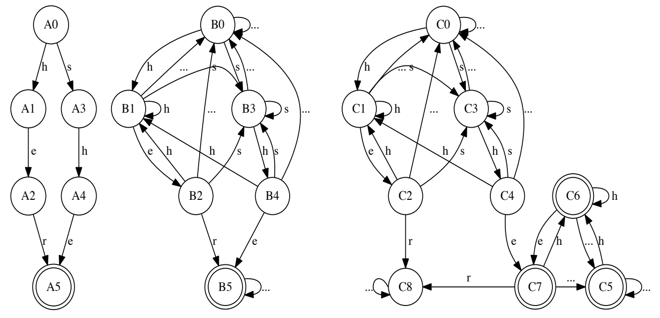

Figure 3: DFAs for combinations of: she, her

All regular expressions have a DFA that matches them. A DFA can be constructed automatically from a regular expression and the number of states can be minimized [4], which is still a pre-processing step in the context of databases. Regular expressions cover all logical combinations of substring matching operators. For example, the selection filter `like '%special%' or like '%packages%'` can either use two calls of substring matching and combine their result or use a DFA to match both words simultaneously. The DFA matches strings in linear time regardless of the number of patterns but its size grows if more words have to be matched. A well-known algorithm for multi-pattern substring matching is Aho-Corasick [1]. Aho-Corasick places all patterns in a trie, traversed as a DFA, and keeps a separate transition table for mismatches, similar to KMP. Still, the extra table can be encoded in the DFA transitions and Aho-Corasick becomes identical to the minimal DFA. However, unlike Aho-Corasick, DFAs can accept all logical combination of positive and negative patterns. Figure 3 shows: (i) the trie that accepts `her` or `she` (`A` states), (ii) a DFA that accepts substrings `her` and `she` (`B` states), and (iii) a DFA that accepts substring `she` but rejects substring `her` (`C` states).

In Figure 2, there is an implicit extra state that works as a reject *sink*. An e-mail is invalid if we encounter an invalid character and can stop the DFA traversal immediately. In Figure 3, `B5` is an accept sink and `C8` is a reject sink. A DFA can have both. These special states allow us to accept or reject a string early, which is crucial for performance in some DFAs, but can also introduce branch mispredictions.

If the DFA has an accept sink state only, such as multi-pattern matching, high selectivity may favor skipping a large portion of each input string. DFAs with a reject sink state are favored by low selectivity. Table 1 shows scenarios of early failures, including example inputs with the character where the transition to the reject sink occurs highlighted.

Regex	Scenario with possible early failures
e-mail	invalid character, e.g., me_@mail.com double @ symbol, e.g., me@@mail.com specific domain: mail.com, e.g. me@meil.com specific username: john, e.g. jim@mail.com
URL	invalid character, e.g., http://site_.com invalid scheme, e.g., htp://site.com specific site: site.com, e.g., http://no.com/a/b/c specific IP: 125.1.*.*, e.g., http://125.2.0.0/a/b/c specific path depth: 2, e.g., http://site.com/a/b/c
address	missing street number before street name non-numeric symbol in postal code specific street number in valid address
name	invalid symbol, e.g., J0hn Smith lowercase first letter, e.g., bob Smith specific surname: Stark, e.g., Peter Parker

Table 1: Early failure scenarios with examples

In this paper, we implement regular expression matching by traversing both the DFA and the input in a data-parallel way. Our implementation traverses both the DFA for multiple strings at a time using non-contiguous loads (gathers) and also accesses different offsets of the strings without assuming lockstep processing, while amortizing the random access cost by buffering multiple bytes per string access. Finally, we use branchless vectorized code to store pointers to matching tuples (rids), replacing old strings that reach a sink state early, in order to maximize SIMD lane utilization.

Our approach works on both recent mainstream CPUs (Intel Haswell) and co-processors (Intel Xeon Phi) and is independent of the SIMD length. Our experimental evaluation shows that compared with scalar code, our implementation achieves a 2X improvement on mainstream CPUs and 5X improvement on Xeon Phi co-processors, providing a crucial tool for supporting efficient regular expression matching.

In Section 2 we present related work. In Section 3 we describe our vectorized implementation, including details such as how to access the input strings, how to traverse the DFA in parallel, and how to replace early failures. In Section 4 we present our evaluation and we conclude in Section 5.

2. RELATED WORK

Regular expression matching has been studied extensively. GPUs were used for fast substring matching where interleaving the strings reduces the cache pressure [13]. Earlier work considered DFA traversal inherently scalar and used NFA representations that can exploit SIMD instructions on the Cell processor [6]. Other techniques to accelerate multi-pattern matching on Cell reduced the alphabet to fit in SIMD registers [12]. Other work suggested breaking dependencies across iterations by enumerating transitions from all possible states per input symbol [7]. Another approach involved two steps, first matching network packet headers using a DFA, and then matching multi-stride pattern segments for the body matches using SIMD instructions [15]. Partitioning a large DFA into cache-resident pieces was evaluated in the Xeon Phi co-processor [14]. Nevertheless, regular expressions used to filter string columns as part of the query, map to DFAs that normally do not exceed the cache size.

Prior work has claimed that DFA traversal has data dependencies that hinder the use of SIMD and propose compacting the DFA to fit in registers or using NFAs, often restricting the optimizations to multi-pattern matching. Earlier work proposed processing multiple input strings in a data-parallel way, either using Cell SPEs [11], or via SIMD gathers in mainstream processors [10], although in the latter case, the hardware did not yet implement gathers to evaluate the actual speedups. In lexical analysis where the leftmost longest match has to be found, the entire string has to be processed and thus processing multiple strings in lockstep [10, 11] is sufficient. In databases, however, regular expressions are used as a boolean filter and the matching can skip large portions of each input string, making lockstep processing wasteful. Vectorization using data-parallel processing of multiple input instances was used to accelerate database operators on CPUs and Xeon Phi co-processors [8, 9]. Our design not only traverses the DFA for multiple strings in parallel, but also accesses the strings at arbitrary offsets rather than in lockstep, buffers multiple bytes per access, and replaces strings as soon as they are accepted or rejected by the DFA, in order to fully utilize the SIMD lanes.

3. IMPLEMENTATION

Each regular expression has a single matching DFA with the minimum number of states. Since the DFA is deterministic, each state has exactly one transition per input character. Thus, we represent the DFA as an $s \times c$ array with s states and c transitions per state, where c is the size of the alphabet. To cover all possible bytes, we use $c = 256$. To avoid storing whether each state is accepting or rejecting the input string, we place the s_{rej} rejecting states in rows $[0, s_{rej})$ and the remaining s_{acc} accepting states in rows $[s_{rej}, s_{acc}+s_{rej})$. Scalar code for matching a single string is shown below:

```
bool regexp(const uint8_t *string, size_t str_len,
            const ssize_t *dfa, [...]) {
   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; }
```

The two-dimensional array of the DFA is accessed as a one-dimensional array using arithmetic. The transition offset is computed using the current state and the next byte of the input string. We stop when we reach the end of the string, unless we transition to one of the two sink states. The snippet shown above is inlined in a loop that scans over the string column and stores the rids of accepted strings.

We simplify the branch tests by setting the transitions to the negative and the positive sink to `-1` and `-2` respectively. The `0<=(ssize_t)s` signed integer comparison tests whether the state is a sink or not. The `s+1>reject_states` unsigned integer comparison tests whether the state is in the range $[-1, s_{rej})$, thus the string should be rejected. By minimizing the branches and using simple arithmetic to access the transition table, we make the scalar code as fast as possible. When storing the rids of matching strings, we can eliminate the branch, using the result of the `s+1>reject_states` comparison to increment the index to the array of rids.

If the number of DFA states is small, we can store the transition table as a byte array (if $s < 255$) and shrink its memory footprint to 1/4. In databases where the regular expression is specified in the query, the DFA is typically small enough to fit in the L1 cache. For instance, to validate URLs we need a sophisticated regular expression with $\approx$100 states, which translates to a 23 KB DFA that still fits in the L1 cache. Writing queries with regular expressions with DFAs that exceed the L1 cache capacity is quite impractical.

When the DFA fits in the cache, the matching throughput is determined by the computation, the read latency when accessing the next DFA state from the cache, and the number of branch mispredictions if strings are determined by the DFA early. Branch mispredictions occur when the strings reach a sink state before reaching the end of the string, exiting the inner loop and skipping the remaining bytes of the string. If the DFA rarely transitions to sink states and the string length is fixed, the inner loop executes a specific number of times and branch mispredictions become negligible.

To facilitate vectorization of the scalar code shown above, we assume that the strings have fixed lengths. String columns of fixed length are often used by main-memory databases to allow fast random access to string values using pointers (rids). If we have a wide range of lengths, we can relax this constraint by re-organizing the column to group strings of similar length together. Thus, we can largely maintain fast random access while avoiding space-inefficient padding.

If we process a different string per vector lane but access the input in lockstep [10, 11], we load W characters from W strings in W vectors, loading data from a single string in each vector. We have an inner loop that (i) packs the first lane from these W vectors into one vector, (ii) computes the transition offset, (iii) gathers the next state per string from the DFA, and (iv) shifts the W vectors by one lane to move the next character to the first lane. Since each string can be larger than a vector, we have an outer loop that is repeated $\lceil L \div W \rceil$ times. The algorithm is shown below. Afterwards, we show a second algorithm and then describe the notation.

Algorithm 1 Lockstep regex matching

```
$\vec{r} \leftarrow \{0, 1, \dots, W-1\}$          ▷ rids of strings (being processed)
$j \leftarrow 0$          ▷ output index for array of accepted string rids)
for $i \leftarrow 0$ to $N$ step $W$ do          ▷ N: # of tuples
    $\vec{s} \leftarrow s_{initial}$          ▷ set to initial state
    for $o \leftarrow 0$ to $L$ step $W$ do          ▷ L: string length
        $\vec{d_1} \leftarrow \text{strings}[i \cdot L + o]$          ▷ load 1st string
        [...]          ▷ W − 2 symmetric lines skipped
        $\vec{d_W} \leftarrow \text{strings}[(i + W - 1) \cdot L + o]$          ▷ load Wth string
        for $l \leftarrow 0$ to $\min(W, L - o)$ do
            $\vec{d} \leftarrow \text{interleave\_first\_lanes}(\vec{d_1}, \dots, \vec{d_W})$
            $\vec{s_i} \leftarrow (\vec{s} << 8) \mid \vec{d}$          ▷ compute offsets in the DFA
            $m \leftarrow \vec{s} \geq 0$          ▷ check if not on sink states
            $\vec{s} \leftarrow_m \text{DFA}[\vec{s_i}]$          ▷ gather next states from the DFA
            $\vec{d_1} \leftarrow \text{shift\_lanes\_right}(\vec{d_1}, 1)$
            [...]          ▷ W − 2 symmetric lines skipped
            $\vec{d_W} \leftarrow \text{shift\_lanes\_right}(\vec{d_W}, 1)$
        end for
    end for
    $m \leftarrow \vec{s} + 1 > s_{reject}$          ▷ find which strings are accepted
    $\text{rids}[j] \leftarrow_m \vec{r}$          ▷ store rids of accepted strings
    $j \leftarrow j + |m|$          ▷ update output index by counting set bits
    $\vec{r} \leftarrow \vec{r} + W$          ▷ update rids to process next W strings
end for
```

Reusing vector lanes dynamically has been shown to work very well on vectorized implementations of other database operations [8, 9]. Here, the input strings can be accessed out-of-order in arbitrary offsets by having old strings replace old strings as soon as they are determined by the DFA. This approach contrasts DFA traversal using multiple consecutive strings in lockstep, which is similar to unrolling the scalar code. A simplified version of the algorithm is outlined below.

Algorithm 2 Short-circuit regex matching (simplified)

```
$\vec{r_0}, \vec{r} \leftarrow \{0, 1, \dots, W-1\}$          ▷ rids of strings (being processed)
$\vec{s} \leftarrow s_{initial}$          ▷ offset in each string (being processed)
$\vec{o} \leftarrow 0$          ▷ state in the DFA of strings (being processed)
$i \leftarrow W$          ▷ input index used to implicitly generate rids
$j \leftarrow 0$          ▷ output index for array of accepted string rids
while $i \leq N$ do          ▷ N: # of tuples, L: string length
    $\vec{d_i} \leftarrow \vec{r} \cdot L + \vec{o}$          ▷ compute (global) offsets in strings
    $\vec{d} \leftarrow \text{strings}[\vec{d_i}]$          ▷ gather bytes of strings from input
    $\vec{s_i} \leftarrow (\vec{s} << 8) \mid \vec{d}$          ▷ compute offsets in the DFA table
    $\vec{s} \leftarrow \text{DFA}[\vec{s_i}]$          ▷ gather next states from the DFA
    $m \leftarrow (\vec{s} + 1 > s_{reject}) \ \& \ (\vec{o} = L)$          ▷ check if accepted
    $\text{rids}[j] \leftarrow_m \vec{r}$          ▷ store rids (if accepted)
    $j \leftarrow j + |m|$          ▷ update output index
    $m \leftarrow (\vec{s} < 0) \mid (\vec{o} = L)$          ▷ check if finished
    $\text{temp}[0] \leftarrow \vec{r_0} + i$          ▷ use input index & lane offsets ...
    $\vec{r} \leftarrow_m \text{temp}[0]$          ▷ ... to replace rids (if finished)
    $i \leftarrow i + |m|$          ▷ update input index
    $\vec{s} \leftarrow m \ ? \ s_{initial} : \vec{s}$          ▷ reset DFA state (if finished)
    $\vec{o} \leftarrow m \ ? \ 0 : (\vec{o} + 1)$          ▷ reset in-string offset (if finished)
end while
```

The notation used in Algorithms 1 and 2 is based on earlier work [8] and is briefly summarized here for clarity. $\vec{x} \leftarrow A[\vec{y}]$ is a gather using $\vec{y}$ for the indexes. $\vec{x} \leftarrow_m A[i]$ is a selective load where only the lanes specified in bitmap m are replaced with data loaded sequentially from memory location A. $A[i] \leftarrow \vec{x}$ is a selective store where the lanes specified in m are stored sequentially to A. $\vec{x} \leftarrow m \ ? \ \vec{y} : \vec{z}$ picks the value of each lane in x from either y or z based on bitmap m. $m \leftarrow \vec{x} < \vec{y}$ generates a bitmap with the boolean result of each comparison. $|m|$ denotes the number of set bits in m. Scalar values in vector operations are implicitly broadcast to all lanes. For example, $\vec{x} \leftarrow \vec{x} + c$ adds c to all lanes of $\vec{x}$.

Since the string column is scanned in order, we implicitly generate rids from 1 to N, where N is the number of tuples. Figure 4 illustrates this functionality. The lanes with rids 14 and 37 refer to accepted strings, while the lanes with rids 16, 32, and 38 refer to rejected strings. The remaining lanes are yet undetermined. We selectively store the rids of accepted strings to an output array and then replace both accepted and rejected strings with new implicitly generated rids by incrementing the input offset. For each string we process, we hold the rid, the current offset in the string, and the current state in the DFA. In the vector lanes with accepted or rejected strings, besides replacing the rids, we also reset the states to the initial state and the string offset to zero.

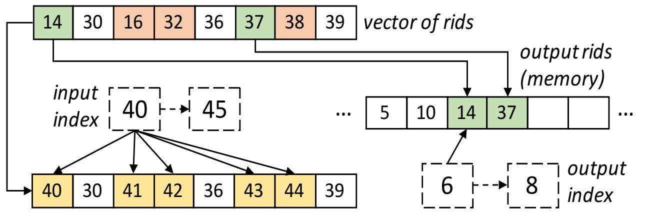

Figure 4: Selective loads & stores of rid vectors

The difference of Algorithm 2 with the baseline scalar code is that it converts all conditional control flow into branchless data flow. However, since the input is no longer accessed in order, we have to use vector gathers to load the bytes from the strings non-contiguously, while the scalar code processes a single string and accesses the string bytes contiguously.

Non-contiguous loads are more expensive than contiguous loads but are necessary if we process multiple strings in parallel. However, executing a new gather to load 1 byte per string instead of a 4-byte word, is wasteful. Also, in practice, we expect to process a non-trivial portion per string to determine if it matches the regular expression. Thus, we buffer more than one byte each time we load data from the strings. Instead of issuing one cache access for each byte of each string, we load multiple consecutive bytes of each string and buffer them in the vector. CPU caches are equally fast whether we access 1 byte, or 8 bytes (aligned). Even aligned 32-byte vector accesses can be equally fast in some CPUs.

When gathering bytes from arbitrary offsets in the strings, the accesses may not be aligned on 4-byte boundaries. For example, if the string length is 15, the second string will start from the 16th byte. Even if scalar loads are allowed to be unaligned, vector gathers may still require aligned pointers. Mainstream CPUs support unaligned gathers in SIMD (AVX 2), thus, we can load 8 bytes per string using a single 64-bit gather. The Xeon Phi, on the other hand, enforces w-byte vector gathers to be aligned to w-byte boundaries, thus unaligned gathers have to be implemented in software.

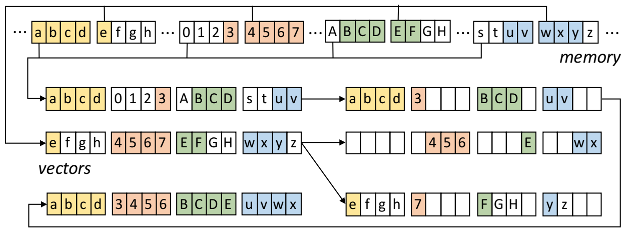

Figure 5: Unaligned vector gathers in Xeon Phi

To implement unaligned vector gathers in software, we use aligned word gathers and variable-stride shifts. First, we align the byte-aligned pointer to a 4-byte-aligned pointer, then we issue two 4-byte gathers to consecutive locations loading 8 consecutive bytes per string, and then we align each vector lane using variable-stride shifts. The process is illustrated in Figure 5. If we issue two 4-byte gathers, which is the minimum possible, the number of usable bytes varies depending on the possible alignments of the strings in the input column. If the (fixed) string length is a multiple of 4, then all strings will be aligned on 4-byte word boundaries and all 8 bytes are valid unless we exceed the string length. If the string length is a multiple of 2, then all strings are aligned on 2-byte boundaries and at least 6 out of 8 bytes are valid. Otherwise, at least 5 are valid. The Xeon Phi code for gathering string bytes from arbitrary offsets is shown below.

```
  // compute index:  rid * length + offset
  __m512i p = _mm512_fmadd_epi32(rid, len, off);
  // align the byte offset to 4-byte boundaries
  __m512i p4 = _mm512_srli_epi32(p, 2);
  // gather 8 bytes per string
  __m512i w1 = _mm512_i32gather_epi32(p4, &str[0], 4);
  __m512i w2 = _mm512_i32gather_epi32(p4, &str[4], 4);
  // compute right shift strides:  s = (p & 3) << 3
  __m512i shr = _mm512_and_epi32(p, m3);
  shr = _mm512_slli_epi32(shr, 3);
  // align 1st word:  w1 = (w1 >> s) | (w2 << (32 - s))
  __m512i shl = _mm512_sub_epi32(m32, shr)
  w1 = _mm512_or_epi32(_mm512_srlv_epi32(w1, shr),
                       _mm512_sllv_epi32(w2, shl));
  // align 2nd word:  w2 >>= shr
  w2 = _mm512_srlv_epi32(w2, shr);
```

To traverse the DFA using all bytes gathered per string, we keep each word of bytes in separate vectors and perform an inner loop for each vector. The loop repeats are only dependent on the string length and are computed once. While we skip the tests to replace finished rids or store accepted rids, we still check for each string if the next loaded byte is valid, i.e., we have not reached a sink state or the end of the string. Xeon Phi code to traverse the DFA is shown below.

```
  // isolate next byte per string
  __m512i b = _mm512_and_epi32(w1, mFF);
  // compute index in transition table
  __m512i p = _mm512_slli_epi32(s, 8);
  p = _mm512_or_epi32(p, b);
  // gather new states (assuming 8-bit DFA array)
  s = _mm512_mask_i32extgather_epi32(s, k, p, dfa,
      _MM_UPCONV_EPI32_SINT8, 1, 0);
  // increment offset for valid lanes using a -1 mask
  off = _mm512_mask_sub_epi32(off, k, off, m1);
  // shift word to get next string byte
  w1 = _mm256_srli_epi32(w1, 8);
  // update valid lanes: check for sink state (s > -1)
  k = _mm512_mask_cmpgt_epi32_mask(k, cur, m1);
  // update valid lanes: check for end of string
  k = _mm512_mask_cmpgt_epi32_mask(k, len, off);
```

In some extreme cases with very short strings or DFAs that reach a sink state very early, we can test whether all vector lanes are invalid on each inner loop iteration and exit. Also, because we perform 5–8 iterations before we reload new strings, some vector lanes remain unutilized during the last inner loop iterations. On average, however, we expect the strings to be larger than 5–8 bytes, and the overhead of a few redundant loops per string after it finishes the DFA traversal, is lower than the overhead of issuing a new gather for each byte per string and check which accepted vector lanes to store and which finished vector lanes to replace.

Finally, we note that the gathers to the DFA transition table cannot be buffered in the same way that gather to the strings were buffered. Even if the DFA is a table of bytes, there is no use for the nearby bytes that would fit in the same processor word. An interesting observation is that if the hardware does not support single byte gathers, the cost of converting (4-byte) int gathers to bytes using shifting is expensive and adds significant overhead to the critical path. Xeon Phi supports this functionality but the latest CPUs (AVX 2) do not. On the CPU, we found that storing the transition table of small DFAs using 4-byte words rather than bytes makes traversal faster, even if the size is quadrupled. Making the DFA resident on the L2 cache rather than the L1 by increasing its footprint, will not affect performance in mainstream CPUs if SIMD gathers are equally fast [3].

Loop unrolling hides latencies among instructions by repeating instructions without data dependencies and boosts performance even in aggressively out-of-order CPUs. Here, we apply 2-way loop unrolling by generating rids from 1 to N and N to 1 until the two rid offsets meet in the middle. The number of variables that hold the state of the two instances is doubled and thus we must ensure that the number of registers suffices to completely avoid register spilling.

4. EXPERIMENTAL EVALUATION

Our evaluation was done on two platforms. The first platform has an Intel Xeon E3-1275v3 CPU with 4 Intel Haswell cores and 2-way SMT running at 3.5 GHz that supports 256-bit SIMD instructions (AVX 2). The platform has 32 GB DDR3 ECC RAM at 1600 MHz with a peak load bandwidth of 21.8 GB/s and runs Linux 4.4. We compile using GCC 6 with `-O3`. The second platform is an Intel Xeon Phi 7120P co-processor with 61 modified P54C cores and 4-way SMT running at 1.238 GHz that supports 512-bit SIMD instructions. The co-processor has 16 GB GDDR5 on-chip RAM with a peak load bandwidth of 212 GB/s and runs embedded Linux 2.6. We compile using ICC 17 with `-O3`. We also tested ICC on the CPU, but GCC was marginally faster.

All figures show the performance of scalar code (`Scalar`), vector code (`Vector (x1)`), which extends Algorithm 2 with an inner loop to process multiple bytes per gather, and vector code with 2-way loop unrolling (`Vector (x2)`). On the CPU platform, we also implement Algoritm 1 that accesses the inlint in lockstep [10, 11] (`Vector (ls)`). The data are synthetically generated for each regular expression to meet specific criteria per experiment. We scan over a fixed-length string column and store the rids of accepted strings. The DFAs are stored as byte arrays if the states are few, except for the vector methods on the CPU where we measured that using 32-bit gathers to access a 4X larger DFA to be faster than emulating 8-bit gathers via 32-bit gathers (AVX 2). Unless otherwise specified, we use all hardware threads.

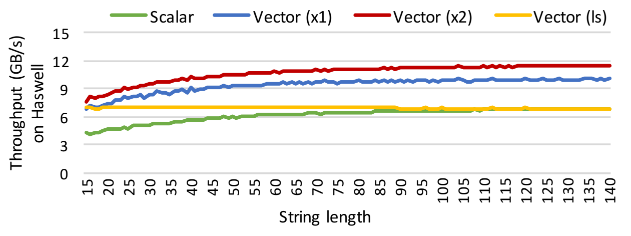

Figure 6: Varying string lengths (URL validation)

Figure 6 shows the throughput of regular expression matching by varying the string length. The gigabytes per second metric measures the total string length, even if some bytes of the string are skipped. The DFA checks whether the string is a valid URL using the regular expression shown below.

```
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})
|((([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]).){3}
([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]))(:[0-9]*)?
(/([._~!$&'()*+,;=A-Za-z0-9:@-]|(%[0-9A-F]{2}))+)*
(?([?._~!$&'()*+,;=A-Za-z0-9:@/-]|(%[0-9A-F]{2}))*)?
(#([?._~!$&'()*+,;=A-Za-z0-9:@/-]|(%[0-9A-F]{2}))*)?$
```

The DFA has 90 states and its footprint is 23 KB if stored as a byte array. The selectivity is set to 1% and we process half of the bytes per string on average before we reach the reject sink state. The speedup is 1.67–1.95X and the average bandwidth usage is increased from 28% to 50%. Loop unrolling boosts the vector code up to 14%. The lockstep method is slower due to processing all the bytes per string.

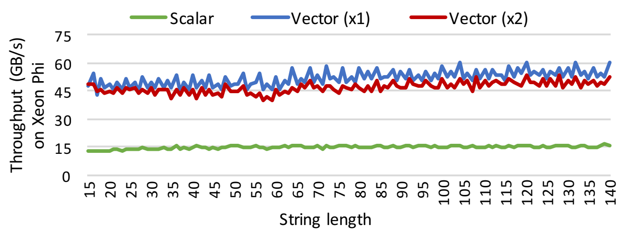

Figure 7: Varying string lengths (URL validation)

Figure 7 shows the throughput on the co-processor. The vectorized code is 2.6–3.7X faster and increases the bandwidth usage from 7% to 26%. Loop unrolling is slower as 4-way SMT already hides instruction latencies effectively. Performance exhibits small spikes due to unaligned gathers.

In Figure 8, we fix the string length to 32 and vary the average failure point. The failure point represents the number of bytes processed per string, or the average number of transitions in the DFA until we reach the reject sink state. The vectorization speedup on the CPU is 1.66–1.92X and 2.57–3.34X on the Xeon Phi by using strings with length equal to 32 and by averaging across all failure points. On the CPU, loop unrolling boosts performance up to 13%. The bandwidth usage is increased from 26% to 47% on average.

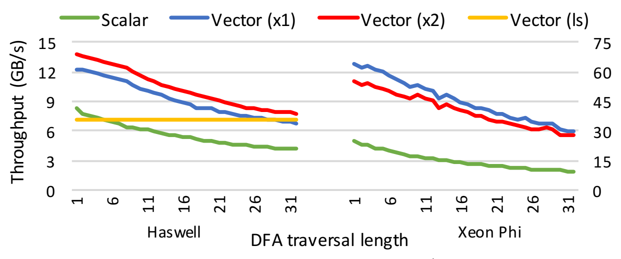

Figure 8: Varying the failure point (URL validation)

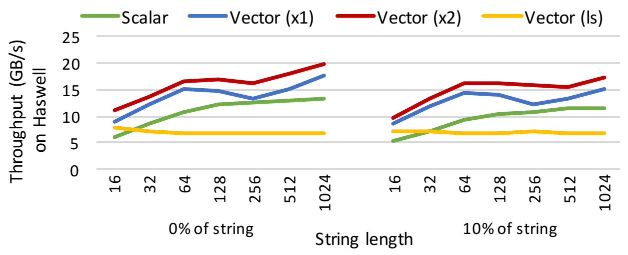

Figure 9: Traversing 0% and 10% of the string and varying the string length (URL validation)

Figures 9 and 10 show the throughput on the Haswell CPU, by varying the string length and by setting the failure point at 0%, 10%, 50%, and 100% of the string length. The selectivity is set to 1%. These results highlight the impact of accessing the input strings in lockstep when the strings are rejected early by the DFA. If 0% or 10% of the string is traversed, the vectorized method that processes the strings in lockstep is slower than even the scalar method, unless the strings are very short. The speedup over the lockstep method is reaching 3X for 1024-byte strings. When the string length exceeds 128 bytes, the performance drops, due to not loading from consecutive cache lines when accessing the input strings. When all strings fail at the first character, the vectorization speedup is 1.3–1.8X and the improvement over the lockstep method is 1.4–2.9X on 1024-byte strings. When the strings fail at the 10% of their length on average, the vectorization speedup is 1.5–1.9X and the improvement over the lockstep method is 2.5X on 1024-byte strings. When half or the whole string is processed, the vectorization speedup is 1.4–1.9X. When the entire string is processed, the lockstep method is only 3–7% faster than our approach.

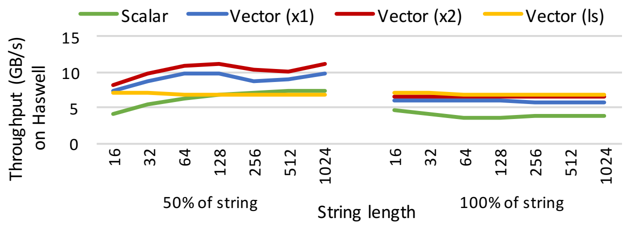

Figure 10: Traversing 50% and 100% of the string and varying the string length (URL validation)

In Figure 11, we set the string length to 1024 bytes and vary the failure point using a logarithmic scale. The selectivity is set to 1%. The performance remains stable regardless of whether we process 1 or 64 characters for each string, saturating the memory bandwidth. Note that even if we access 1 byte for every 16 cache lines and skip 1023 bytes, we are still as fast as fetching from RAM all 1024 bytes per string even if only the first few are used to traversed the DFA.

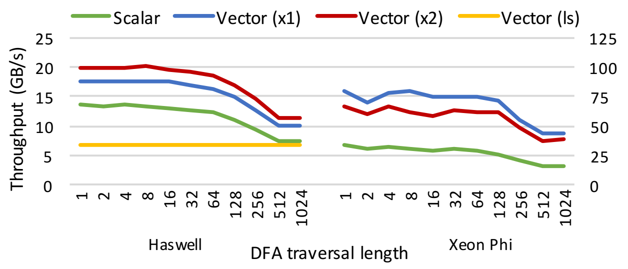

Figure 11: Varying the failure point (log scale) in 1024-byte strings (URL validation)

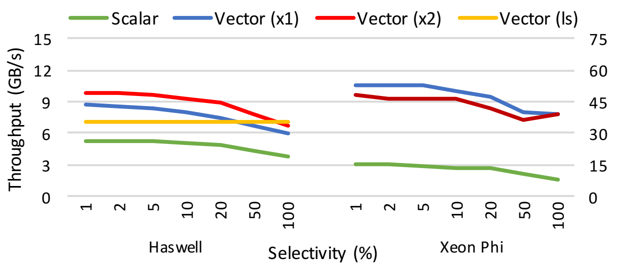

Figure 12: Varying the selectivity (URL validation)

In Figure 12, we set the string length to 32 and we vary the selectivity rate. For rejected strings, we process half their bytes on average until they are rejected. On the CPU, we get 1.8–1.9X vectorization speedup and increase the bandwidth usage from 24% to 45% for low selectivity. The throughput drops by 32% at 100% selectivity and the lockstep method becomes equally fast as the entire string has to be processed. With 1% selectivity, we use up to 45% of the bandwidth. In the co-processor, the vectorization speedup is 3.5–4.7X faster and is maximized at 100% selectivity. Since we are compute-bound, unless the strings are too short, materializing the rids of accepted strings does not affect performance.

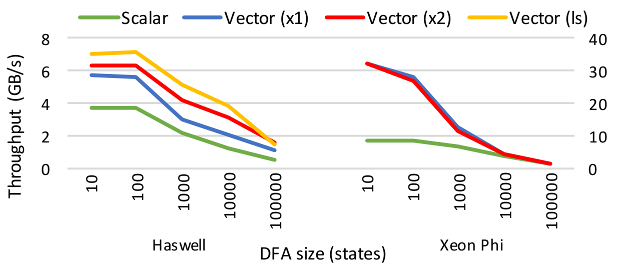

Figure 13: Varying the DFA size (multi-pattern matching using random English dictionary words)

Figure 13 varies the DFA size using multi-pattern substring matching. We vary the number of words in the DFA, creating 10^k states and exceeding the cache size. The selectivity is 1% but the inputs are generated by appending randomly picked dictionary words, to ensure that we traverse long paths in the DFA. On the mainstream CPU, the speedup is 1.72–2.73X and is maximized when the DFA is large. This implies that out-of-cache access latencies are exacerbated when tied with control flow dependencies, which is also supported by the fact that loop unrolling improves performance up to 45% on larger DFAs. In the co-processor, the speedup is 1.05–3.7X and is maximized when the DFA is small enough to be in the L1 cache. Eliminating control flow dependencies is not useful on the in-order cores that expose the latency of cache loads. Processing the input in lockstep is ≈10% faster here because 99% of strings are rejected and we have to process the entire string to search for matches.

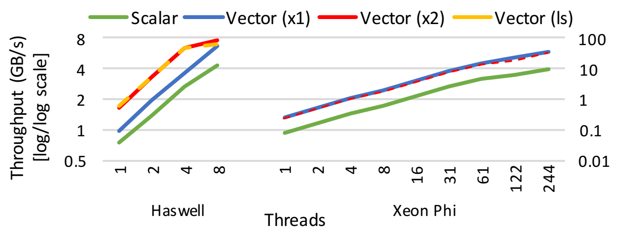

Figure 14: Scalability (multi-pattern matching with positive and negative examples of dictionary words)

Figure 14 shows the scalability using a cache-resident DFA for multi-pattern matching with both positive and negative patterns (see Figure 3 for an example), emphasizing that our approach is more general than disjunctive substring matching. Performance scales linearly with the number of threads. On the Xeon Phi co-processor, we achieve linear speedup, even by using SMT threads, because SMT hides the high latency of vector instructions. On the mainstream CPU, using SMT with loop unrolling gives marginal improvement, thus, our code saturates the performance capacity per core.

5. CONCLUSION

We presented the design and implementation of SIMD-vectorized regular expression matching for filtering string columns. Our approach processes multiple input strings in a data-parallel way without accessing the input in lockstep and achieves up to 2X speedup on a mainstream CPU and 5X speedup on the Xeon Phi co-processor using common string lengths. If a string can be accepted or rejected without looking at all its characters, our approach can achieve significant speedups compared to the previous vectorized approaches that access the strings in lockstep. Our results highlight the impact of vectorization on optimizing compute-bound but minimal scalar code dominated by cache accesses.

6. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *CACM*, 18(6):333–340, June 1975.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, Oct. 1977.

[3] J. Hofmann et al. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips. *CoRR*, arXiv:1401.7494, 2014.

[4] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, 1971.

[5] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[6] F. Kulishov. DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering. In *SIN*, pages 123–127, 2009.

[7] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS*, pages 529–542, 2014.

[8] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[9] O. Polychroniou and K. A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN*, 2014.

[10] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *HPCA*, pages 1–10, 2012.

[11] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In *ICS*, pages 14–25, 2009.

[12] D. P. Scarpazza and O. Villa. Peak-performance DFA-based string matching on the Cell processor. In *SMTPS*, pages 1–8, 2007.

[13] E. A. Sitaridi and K. A. Ross. GPU-accelerated string matching for database applications. *VLDB J.*, pages 1–22, 2015.

[14] N. P. Tran, D. H. Choi, and M. Lee. Optimizing cache locality for irregular data accesses on many-core Intel Xeon Phi accelerator chip. In *HPCC*, pages 153–156, 2014.

[15] Y. H. E. Yang, V. K. Prasanna, and C. Jiang. Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In *IPDPS*, pages 1–11, 2010.