Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions

Hiroshi Inoue†‡, Moriyoshi Ohara†, Kenjiro Taura‡
† IBM Research – Tokyo ‡ University of Tokyo
What is Set Intersection?

- The operation to find common elements from two sets
- We think intersecting two sorted integer arrays (e.g. `std::set_intersection` in STL of C++)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Output array:

| 5    | 17     | 41     | .....  |
Does it matter?

- Heavily used in DBMS (join operator) and information retrieval systems (multiword AND query)

**Multiword query**

→ find documents including all keywords
→ “set intersection” of posting lists!

List of document IDs for keywordA

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>

List of document IDs for keywordB

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
How can we implement this?

1. check the equality of two elements
2. advance a pointer by 1

Merge-based approach

```c
while (pA < pAend && pB < pBend) {
    if      (*pA == *pB) { *pOut++ = *pA++; pB++; }
    else if (*pA  < *pB) { pA++; }
    else                 { pB++; }
}
```
Many techniques have been proposed for intersecting two arrays of very different sizes (10x ~)
- based on binary search (e.g. galloping)
- based on additional data structures (e.g. skip list, hash etc)
They focus on reducing the number of comparisons
For arrays with similar sizes, the merge-based algorithm is faster than these advanced algorithms ➔ our focus
Key observation

while (pA < pAend && pB < pBend) {
    if (*pA == *pB) { *pOut++ = *pA++; pB++; }
    else if (*pA < *pB) { pA++; }
    else { pB++; }
}

- The comparison to select an input array for the next block is hard to predict for branch prediction hardware
  - It will be taken in arbitrary order
- The comparison to check equality is much easier to predict
  - It is not taken frequently for many applications

⇒ We reduce the hard-to-predict conditional branches
Our approach for reducing branch mispredictions

- Reduce the number of the hard-to-predict conditional branches to 1/S
- Increase other (easy-to-predict) conditional branches by S times

Based on a simple cost model, the block size of 3 is the best when misprediction penalty is 10~22 cycles
Pseudo code of our approach (with block size $S = 2$)

```c
while (pA < pAend-1 && pB < pBend-1) {
    A0 = *pA; A1 = *(pA+1); B0 = *pB; B1 = *(pB+1);
    if (A0 == B0) { *pOut++ = A0; }
    else if (A0 == B1) { *pOut++ = A0; Bpos+ = 2; continue; }
    else if (A1 == B0) { *pOut++ = A1; Apos+ = 2; continue; }
    if (A1 == B1) { *pOut++ = A1; Apos+ = 2; Bpos+ = 2; }
    else if (A1 < B1) { Apos+ = 2; }
    else { Bpos+ = 2; }
}
```

$S^2$ easy-to-predict branches per $S$ elements $\Rightarrow$ $S$ times more

only one while processing $S$ elements $\Rightarrow$ reduced to $1/S$
Determining the best block size

- A simple cost model of branches for block size $S$

<table>
<thead>
<tr>
<th></th>
<th>execution per element</th>
<th>misprediction rate</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_equal branches</td>
<td>$S$</td>
<td>0%</td>
<td>$S \cdot \text{cost}_{\text{exec}}$</td>
</tr>
<tr>
<td>if_greater branches</td>
<td>$1/S$</td>
<td>50%</td>
<td>$(\text{cost}<em>{\text{exec}} + \text{cost}</em>{\text{misp}} \times 0.5) / S$</td>
</tr>
</tbody>
</table>

- Best block size is determined by $r = \frac{\text{cost}_{\text{misp}}}{\text{cost}_{\text{exec}}}$
  
  - $S = 1$ when $r \leq 2$
  - $S = 2$ when $2 \leq r \leq 10$
  - $S = 3$ when $10 \leq r \leq 22$
  - $S = 4$ when $22 \leq r \leq 38$

the case for many of recent processors

with SIMD, we use $S = 4$ to fully exploit vector register size
### Our approach for exploiting SIMD instructions

- **Existing approach:** *full* comparison by SIMD to find matching pairs [Lemire *et al.* 2015, Schlegel *et al.* 2011]
  - limited data parallelism
  - limited element size

- **Our approach:** *partial* comparison by SIMD to filter out redundant comparisons
  - We can enjoy higher data parallelism
  - We can support larger elements (e.g. 32-bit or 64-bit integers)
  - Optimized for the common case
Partial comparison by SIMD

- We introduce partial comparison by SIMD before the scalar comparison to reduce redundant comparisons.

We can skip the all-pairs comparison by scalar if the no matching pair found in the partial comparison by SIMD.

compare only a part of each element to increase parallelism.
Performance Evaluations

- Systems
  - 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors
    • using SSE instructions (128-bit SIMD)
    • Redhat Enterprise Linux 6.4, gcc-4.8.2
  - 4.1-GHz POWER7+ processors
    • using VSX instructions (128-bit SIMD)
    • Redhat Enterprise Linux 6.4, gcc-4.8.3
Performance improvements by our scalar algorithm

up to 2.1x and 1.8x gain over STL (with block size of 3)

Intersecting two 256k random 32-bit integers, output / input = 0%
Performance improvements with SIMD instructions

- Further 2x gain over our scalar algorithm (about 5x over STL)
- Lower gain with existing SIMD algorithm (V1 SIMD algorithm, Lemire et al.)
- Intersecting two 256k random 32-bit integers, output / input = 0%
Numbers of branch mispredictions and instructions

![Bar chart showing branch mispredictions and instructions](chart_image)

- Branch mispredictions per input element
  - 7x reduction
  - Better performance with SIMD

- Instructions executed per input element
  - 1.54x reduction
  - Better performance with SIMD
Performance for arrays with different sizes

intersecting two random 32-bit integer arrays
Adaptive fallback to avoid pathological degradations

Our SIMD algorithm is the best with low selectivity (common case)

Our scalar algorithm is the best until ~65% selectivity

intersects two 256k random 32-bit integers
Adaptive fallback to avoid pathological degradations

We adaptively select the best algorithm at runtime based on output/input ratio.

on Xeon

Our SIMD algorithm ➔ Our non-SIMD algorithm ➔ Naive algorithm
Adaptive algorithm overview

- **Start of SIMD algorithm**:
  - < 1:2
  - 1:2~1:32
  - > 1:32

  - SIMD algorithm (block size 4x4)
  - SIMD algorithm (block size 4x8)
  - SIMD galloping [9]

- **Start of scalar algorithm**:
  - < 1:2
  - 1:2~1:32
  - > 1:32

  - scalar algorithm (block size 3x3)
  - scalar algorithm (block size 2x4)
  - galloping [10]

- **Adaptive fallback** with a runtime check of selectivity

- **Select algorithm** based on the difference in the sizes of the two input arrays.

- Our adaptive scalar algorithm
- Our adaptive SIMD algorithm

- Selectivity
  - > 65%
  - > 35%
  - > 15%
  - < 1:2
  - 1:2~1:32
  - > 1:32
Performance with realistic dataset (multiword queries in Wikipedia)

Our SIMD algorithm + SIMD galloping (binary-search-based)

lower gain with existing SIMD algorithm
V1 SIMD algorithm + SIMD galloping
(Lemire et al. [3])
Summary

- We proposed a new set intersection algorithm which is efficient on today’s processors
  - by reducing branch mispredictions
  - by avoiding redundant comparisons using SIMD

- Our new algorithm accelerates set intersection for artificial dataset compared to STL
  - by up to 2x without SIMD
  - by up to 5x using SIMD

- It also achieves better performance in an emulated query serving system
  - by up to 2.3x with SIMD over STL
  - by up to 1.5x over existing SIMD algorithms [Lemire et al. ’15]