FESIA: A Fast and SIMD-Efficient Set Intersection Approach on Modern CPUs

Jiyuan Zhang
Carnegie Mellon University
jiyuanz@andrew.cmu.edu

Yi Lu
MIT CSAIL
yilu@csail.mit.edu

Daniele G. Spampinato
Carnegie Mellon University
spampinato@cmu.edu

Franz Franchetti
Carnegie Mellon University
franzf@cmu.edu

***Abstract*—Set intersection is an important operation and widely used in both database and graph analytics applications. However, existing state-of-the-art set intersection methods only consider the size of input sets and fail to optimize for the case in which the intersection size is small. In real-world scenarios, the size of most intersections is usually orders of magnitude smaller than the size of the input sets, e.g., keyword search in databases and common neighbor search in graph analytics. In this paper, we present FESIA, a new set intersection approach on modern CPUs. The time complexity of our approach is $O(n/\sqrt{w}+r)$, in which w is the SIMD width, and n and r are the size of input sets and intersection size, respectively. The key idea behind FESIA is that it first uses bitmaps to filter out unmatched elements from the input sets, and then selects suitable specialized kernels (i.e., small function blocks) at runtime to compute the final intersection on each pair of bitmap segments. In addition, all data structures in FESIA are designed to take advantage of SIMD instructions provided by vector ISAs with various SIMD widths, including SSE, AVX, and the latest AVX512. Our experiments on both real-world and synthetic datasets show that our intersection method achieves more than an order of magnitude better performance than conventional scalar implementations, and up to 4x better performance than state-of-the-art SIMD implementations.**

I. Introduction

Set intersection selects the common elements appearing in all input sets, which is a fundamental operation in database applications. For example, given a search query with multiple keywords, a list of documents containing all input keywords can be computed through a set intersection [5]. In addition, set intersection is becoming a critical building block for a wide range of new applications in graph analytics, such as triangle counting [6], [7], neighborhood discovery [8], subgraph isomorphism [9], and community detection [10], [11]. For example, the common friends of two people on social networks can be computed through a set intersection as well.

Recent work focusing on accelerating set intersection operations in database systems [1]–[3] proposed the use of merge-based approaches, in which the runtime cost usually only depends on the size of the input sets. However, in real-world scenarios, the intersection size is usually dramatically smaller than the sizes of the input sets. For example, 90% of the intersections resulting from search queries in Bing is an order of magnitude smaller than the size of the input sets, and the intersection size of 76% of the queries is even two orders of magnitude smaller than the input [4]. A similar result is also observed in graph analytics [12], in which the size of over 90% of the intersections is smaller than 30% of their input size.

In this paper, we propose FESIA, a fast and efficient set intersection algorithm targeting at modern CPU architectures. Our key insight is that a large number of comparisons required by existing merge-based set intersection approaches are redundant. These redundancies result in significant overheads especially when the intersection size is small. To this end, FESIA accelerates set intersections by avoiding these redundant and unnecessary comparisons. Specifically, we take a two-step approach: it builds an auxiliary bitmap data structure for pruning unnecessary comparisons in the first step. Since the earlier pruning may leave some false positive matches, our approach further compares these elements to produce the final intersection in the second step.

Prior work [4] focusing on new data structures for set intersection achieves lower time complexity but does not take advantage of SIMD instructions available on modern processors. In contrast, other work [1]–[3] focusing on vectorizing set intersections with SIMD instructions has higher time complexity. FESIA is the first approach that considers both aspects at the same time. It achieves fast and efficient set intersections for two reasons: firstly, the new segmented-bitmap data structure and the course-grained filtering step can make the complexity depend on the intersection size instead of the size of input sets. In summary, the time complexity of our approach is $O(n/\sqrt{w}+r)$ as in [4], in which w indicates the SIMD width, and n and r indicate the size of the input set and the intersection size. Secondly, the data structure and intersection algorithm are designed with SIMD in mind, allowing our approach to exploit more data parallelism and be portable to different SIMD widths. This includes an efficient design of the course-grained filtering implementation with the bitwise operations, combined with our specialized SIMD intersection functions for small input sizes, which are more efficient than existing vectorization methods [1], [13].

In summary, this paper makes the following major contributions:

- We present FESIA, a fast and efficient set intersection approach targeting modern CPUs, which leverages the SIMD instructions and achieves better complexity simultaneously.
- We introduce a coarse-grain pruning approach which can be efficiently implemented with the bitwise SIMD operations.
- We design a code specialization mechanism to reduce the cost of fine-grained intersections required after the initial coarse-grained pruning step.
- We describe an implementation of FESIA for two different

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00130

TABLE I
THE SUMMARY OF OUR APPROACH VS. STATE-OF-THE-ART SET INTERSECTION APPROACHES.

Methods	Complexity	Small Intersect	SIMD	Multicore	$n_1 \ll n_2$	k-way intersection	Portable
FESIA	$n/\sqrt{w}+r$	✓	✓	✓	$\min(n_1, n_2)$	$kn/\sqrt{w}+r$	✓
BMiss [1]	n_1+n_2	✓	✓		n_1+n_2	$n_1 \cdots + n_k$	✓
Galloping [2]	$n_1 \log n_2$		✓		$n_1 \log n_2$	$n_1(\log n_2 + \cdots + \log n_k)$	
Hiera [3]	n_1+n_2		✓		n_1+n_2	$n_1 \cdots + n_k$	
Fast [4]	$n/\sqrt{w}+r$	✓			$n/\sqrt{w}+r$	$n/\sqrt{w}+kr$	

Intel platforms with SSE, AVX, and AVX512 instructions. Our experiments on both real-world and synthetic datasets show that our intersection method achieves more than a order of magnitude better performance than conventional scalar implementations, and up to 4x better performance than state-of-the-art SIMD implementations.

II. Background and Related Work

In this section, we describe conventional scalar set intersection approaches and introduce how SIMD instructions are used to accelerate set intersection.

A. Scalar Set Intersection Approaches

Merge-based set intersection is the most common approach to compute the intersection of two (or more) sorted sets, and an example is shown in C in Listing 1. Suppose there are two sets L_1 and L_2 of size n_1 and n_2 respectively, and we use r to denote the number of common elements. The algorithm starts with two pointers that point to the beginning of the two sorted lists. Pointers are iteratively advanced based on the comparison result of their pointed elements. The algorithm finishes when one pointer points to the end of the list. The time complexity of merge-based set intersection is $O(n_1+n_2)$ for two sets and $O(n_1+\cdots+n_k)$ for k-way set intersection. One limitation of merge-based set intersection is that it cannot be easily extended to exploit multicore parallelism. This is because there exists a loop-carried dependency when pointers are advanced.

```
int scalar_merge_intersection(int L1[],
                    int n1, int L2[], int n2) {
  int i = 0, j = 0, r = 0;
  while (i < n1 && j < n2) {
    if (L1[i] < L2[j]) {
      i++;
    } else if (L1[i] > L2[j]) {
      j++;
    } else {
      i++; j++; r++;
    }
  }
  return r;
}
```

Listing 1. A code example of scalar merge-based set intersection

Hash-based set intersection is another popular approach. It builds a hash table from the elements of one set and then probes the hash table with all elements from the other set. The time complexity of hash-based set intersection is $O(\min(n_1, n_2))$, which makes it the best method when one set is dramatically smaller than the other set. As we will discuss in later sections, the time complexity of our approach is the same as hash-based set intersection when two input sets have dramatically different sizes.

In additional, many other data structures have been proposed to compute set intersection, such as treap [14], skiplist [15], bitmap [4], [16], and adaptive data structures [17].

B. Accelerating Set Intersections

Data parallelism using SIMD: SIMD instructions have become the status quo on modern CPUs to exploit data parallelism. For example, Intel CPUs have SSE/AVX2 instructions to support 128-bit and 256-bit vector operations. Similarly, ARM processors have NEON instructions, and IBM Power processors have AltiVec instructions. The prevailing SIMD widths on modern processors are 128-bit and 256-bit. More recently, the Intel Skylake architecture introduced AVX512 instructions. Additionally, Intel SSE4.2 introduces the STTNI instruction for string comparisons and it has been used for all-pair comparisons between two vectors in parallel.

The state-of-the-art set intersection methods: Prior work has proposed different ways to accelerate set intersections, which are summarized and compared with our approach in Table I. We now introduce each method in more detail.

BMiss [1] aims at reducing the number of branch mispredictions in merge-based set intersections and demonstrates this on Intel and IBM Power processors. The complexity of this method is the same as other merge-based methods. Note that BMiss performs better when the intersection size is small, in which mispredictions are more likely to occur.

Galloping [18] and its extension SIMDGalloping [2] are approaches based on binary search. Each element from the smaller set is looked up in the larger set through a binary search. This method usually performs better when the size of two input sets is significantly different. Note that the complexity of k-way intersection with binary search is $n_1(\log n_2 + \cdots + \log n_k)$, in which each element from the smallest set (L_1) is used as an anchor point and looked up in all other sets.

Hiera [3] is an approach that leverages the STTNI instruction to accelerate merge-based set intersections. Since it is a merge-based approach, its time complexity is $O(n_1+n_2)$ on two sets. In addition, a hierarchical data structure is used in Hiera, since STTNI instruction supports only 8-bit and 16-bit data types. One limitation of Hiera is that its effectiveness highly depends on the data distribution. For example, it downgrades to a scalar approach when the elements in input sets are sparse. In addition, it is not portable to processors without the STTNI instruction.

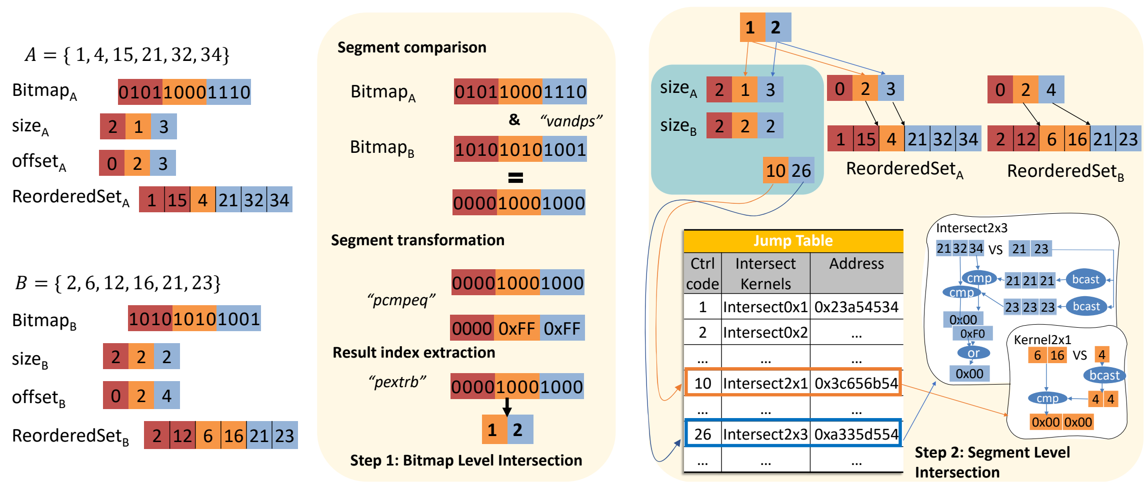


Fig. 1. Illustrating the data structure and the set intersection algorithm in FESIA. There are two steps in the set intersection: (1) the bitmaps are used to filter out unmatched elements, and (2) a segment-by-segment comparison is conducted to compute the final set intersection using specialized SIMD kernels.

Fast [4] is an approach that leverages bitmaps for fast comparisons and it performs better when the intersection size is small. Its time complexity is $O(n/\sqrt{w}+r)$, in which n, r, and w indicate the size of input sets, the intersection size, and the word size, respectively. This method has a better time complexity compared to other merge-based methods, however, it fails to consider SIMD instructions and may not have competitive performance compared to other merge-based methods with SIMD accelerations.

III. The FESIA Approach

In this section, we start with an overview of our FESIA approach. We next explain our data structure and the set intersection algorithm in detail. Finally, we present a theoretical analysis of our approach.

A. Overview

The intersection process is built upon our *segmented-bitmap* data structure. It first compresses and encodes all elements of a set into a *bitmap* in an offline phase. To exploit data-level parallelism, every s bits from the bitmap are further grouped into a *segment*. When performing online intersection, the bitmap serves as a data structure to quickly filter out the unmatched elements between two sets. The online intersection process therefore consists of two steps: (1) bitmap intersection: we compare the segmented bitmaps of two given sets using a `bitwise-AND` operator and output the segments that intersect, and (2) segment intersection: given a list of segments whose bitmap intersects with one another (i.e., the result of `bitwise-AND` is not zero), we go through their corresponding lists and compare the relevant elements from each segment to compute the final set intersection. A larger m can eliminate more false-positive intersections on the bitmap, but leads to more comparison time in step 1. The segment size s affects the intersection sizes at step 2. The m and s are chosen to minimize the total time. A theoretical analysis on the choice of m and s will be presented in Section III-D.

We now introduce our data structure and set intersection algorithm in in more details.

B. Data Structure

FESIA is built on our segmented bitmap data structure. This data structure encodes the elements of a set using a bitmap, and groups the bits as well as the corresponding elements of the bitmap into segments. Specifically, given a set of n elements, its elements are mapped into a bitmap of size m with a universal hash function h. With a properly designed h, all elements can be uniformly distributed in the bitmap. For clarity, we now assume all bitmaps have the same size m and we will relax this assumption with a simple transformation at the end of Section III-C. Every s elements of the bitmap are grouped as a segment. Since the size of the bitmap can be less than the number of elements in the set, more than one element can be mapped to the same location in the bitmap. Therefore, we associate a list to each segment such that elements mapped to this segment are inserted into the list. Note that all elements in a list are always kept in an increasing order.

We now introduce the details of our data structure, as shown in Fig. 1. `Bitmap` is a 0/1 binary bit vector of size m. `Size` is an array of size m/s, storing the size of each segment. `ReorderedSet` is an array having all elements of a set but with a different ordering. Intuitively, it is a concatenation of all segments and the elements are sorted in an increasing order within each segment. `offset` is an array of size m/s, storing the starting index of each segment in `ReorderedSet`. We now use the example in Fig. 1 to illustrate how the data structure works.

Example 1. *Suppose there are two sets:* $A = \{1, 4, 15, 21, 32, 34\}$*, and* $B = \{2, 6, 12, 16, 21, 23\}$*. The size of the bitmap is 12, and we use function* $f(x) = x \bmod 12$ *as our hash function. After mapping all elements into the bitmap, we now have* $\texttt{Bitmap}_A = \{010110001110\}$, $\texttt{size}_A = \{2, 1, 3\}$, $\texttt{offset}_A = \{0, 2, 3\}$, $\texttt{ReorderedSet}_A = \{1, 15, 4, 21, 32, 34\}$, $\texttt{Bitmap}_B = \{101010101001\}$, $\texttt{size}_B = \{2, 2, 2\}$,

$\texttt{offset}_B = \{0, 2, 4\}$, *and* $\texttt{ReorderedSet}_B = \{2, 12, 6, 16, 21, 23\}$.

C. Intersection Algorithm

For the ease of presentation, we discuss the intersection algorithm of two sets in this section. The k-way intersection will be discussed later in Section VI.

Our algorithm first compares the associated bitmaps to quickly eliminate unmatched segments between two sets. It streams through the two bitmaps, compares each bit to find the segment pairs that intersect. Next, it intersects the lists associated with those segments with our specialized intersection kernels. A kernel is a specialized intersection function block for a certain size. We now use Example 1 to demonstrate our intersection algorithm. In the first step, we compare the bitmaps of the two sets with a bitwise AND operation to get a list of segments. In Example 1, we compare $\texttt{Bitmap}_A$ and $\texttt{Bitmap}_B$ with a bitwise AND operation, and the result is $\{000010001000\}$. In the second step, we go through each segment to compute the final intersection result, as shown in Algorithm 1. This is because it's possible to have false positives, i.e., a segment whose bitmap intersects with another one but they do not have common elements. In Example 1, we focus on only non-zero segments, i.e., the second and the third segment. Note that the elements mapped to these two segments are $\{4\}$, $\{6, 16\}$ and $\{21, 32, 34\}$, $\{21, 23\}$. Finally, we use the 1-by-2 kernel to compare $\{4\}$ against $\{6, 16\}$, and the 2-by-3 kernel to compare $\{21, 23\}$ against $\{21, 32, 34\}$ to compute the final result of the intersection. The details of these kernels will be discussed in Section V.

Algorithm 1: The intersection algorithm in FESIA

Input: List L_A, List L_B, $\texttt{Bitmap}_A$, $\texttt{Bitmap}_B$, $\texttt{ReorderedSet}_A$ and $\texttt{ReorderedSet}_B$
Output: the intersection size r

$r = 0$
$N = m/s$ // the number of segments
for $i \in 0 \ldots N-1$ **do**
  if $\texttt{Bitmap}_{A_i}$ & $\texttt{Bitmap}_{B_i} != 0$ **then**
    r = r + Intersect($\texttt{ReorderedSet}_{A_i}$, $\texttt{ReorderedSet}_{B_i}$)
return r

Different bitmap sizes: For any pair of sets, there may exist a pair of sets that have different bitmap sizes. When a bitmap has a different size from the other one, our algorithm requires that the bitmap size m_1 of the larger set can be divided by the bitmap size m_2 of the smaller set. Therefore, the choice of bitmap size in our algorithm is to round the bitmap size to the nearest power of two. When we compare two bitmaps with different sizes, the ith segment from the larger set only needs to compare with the kth segment from the smaller set, in which $k = i \mod m_2$.

D. Theoretical Analysis

We now give an analysis of the time complexity of our intersection algorithm. For clarity, we start our analysis with lists L_1 and L_2 with the same size n. We use w to denote the word size and r to denote the intersection size.

Proposition 1. *The time complexity of Algorithm 1 is* $O(n/\sqrt{w} + r)$.

Proof. There are two steps in the algorithm: (1) bitwise AND on the two bitmaps, and (2) Computing the intersection on the matched segments. The time spent in bitwise AND on the two bitmaps is linearly proportional to m, the size of a bitmap. SIMD allows us to conduct the bitwise operation in parallel. Given the SIMD width w, the time for the bitwise operation is m/w. Let's now analyze the time spent in computing the intersection on the matched segments, which depends on the number of matched segments. There are two types of matches: (1) the true matches, and (2) the false positive matches. The number of true matches equals the intersection size r. A false-positive match happens when there are elements in both lists mapped to the same segment but they are not identical.

More precisely, the number of true matches is

$$\mathrm{E}(I_T) = r$$

while the expected number of false positive matches is:

$$\begin{aligned}\mathrm{E}(I_{\text{FP}}) &= \sum_{\substack{e_1 \in L_1, e_2 \in L_2 \\ e_1 \neq e_2}} P(h(e_1) = h(e_2)) \leq \binom{n}{2} \times \frac{1}{m} \\ &= \frac{n(n-1)}{2m}\end{aligned}$$

Note that $\mathrm{E}(I_{FP})$ depends on the choice of m, which affects the time complexity of Algorithm 1. For example, when $m = 1$, the time spent in the second step is $O(n^2)$. When $m = n^2$, the time spent in the second step is $O(1)$. However, the time spent in the first step now becomes $O(n^2)$. When $m = n\sqrt{w}$, $\mathrm{E}(I_{\text{FP}})$ equals to $n/\sqrt{w}$. Therefore, we have the total number of both true and false positive matches:

$$\mathrm{E}(I) = \mathrm{E}(I_T) + \mathrm{E}(I_{\text{FP}}) = n/\sqrt{w} + r$$

In summary, when $m = n\sqrt{w}$, the time complexity of Algorithm 1 is $O(n/\sqrt{w} + r)$. It achieves the same theoretical bound as in Fast [4]. As we will show in later sections, due to the use of SIMD instructions, our approach also has the best performance in practice. □

IV. Bitmap-Level Intersection

In this section, we discuss how to use SIMD instructions to conduct the bitmap-level intersection, which prunes out the unmatched elements. The discussion of segment-level intersection that computes the final intersection will be discussed in Section V.

Given two sets and the corresponding two bitmaps $\texttt{Bitmap}_A$ and $\texttt{Bitmap}_B$, there are three steps in the bitmap-level intersection to generate a list of segments where one bitmap intersects with the other:

Step 1: Bitwise AND on bitmaps: The key idea is to perform a bitwise AND operation on bitmaps $\texttt{Bitmap}_A$ and $\texttt{Bitmap}_B$ with SIMD instructions. The SIMD instruction that we use is the `vandps` instruction, which operates on w bits at the same time. Note that w is the SIMD width of the processor and it's a multiple of the segment size s. For example, if $w = 256$ and $s = 8$, a single `vandps` instruction can perform the bitwise AND operation on $256/8 = 32$ segments in parallel.

Step 2: Segment transformation: We next summarize the output of bitwise AND operation from the step above by applying a transformation on each segment, i.e., every s bits in the bitwise AND output. A collection of SIMD comparison instructions (e.g., `pcmpeqw`, `pcmpeqq`, etc) are provided on Intel processors such that comparisons can be performed on every 8, 16, 32, or 64 bits at the same time. Similar instructions are also provided on other processors, such as ARM or IBM Power processors.

With the support of these SIMD instructions, we can use a single instruction to compare s bits with integer 0 for multiple segments at the same time. Note that the SIMD instruction we use depends on the value of s, i.e., different values of s lead to the use of different SIMD instructions. The output for each segment also has s bits. Suppose $s = 16$ and if one segment intersects with the other one, the result is 0xFFFF. Otherwise, the result is 0x0000.

Step 3: Non-zero segment index extraction: We now pick the segments with value 0xFFFF. Given a list of segments, we use the `pextrb` instruction to extract the non-zero segments. The instruction uses 1-bit to represent the output for each segment. Suppose the segment size $s = 16$ and there are two segments 0xFFFF0000. We will get 0x2 (i.e., 10 in the binary format) after applying the instruction on the two segments above.

We next use the `tzcnt` instruction to extract the index of bits that are set to one. Given an integer, the `tzcnt` instruction returns the number of trailing zeros (i.e., the index of the least-significant 1-bit). Then we set this bit to zero. We iteratively apply the above process until all the ones are extracted.

V. Segment-Level Intersection

In the previous section, we discussed how to use bitmap intersection to generate a list of pairs of segments that may have common elements. In this section, we focus on using SIMD instructions to accelerate intersection for each pair of segments.

Suppose there are N segments from the output of bitmap intersection, which are indexed by $i_1 \dots i_N$. The task in this step is to find the intersection size with elements from $\texttt{ReorderedSet}_{A,i_1} \dots \texttt{ReorderedSet}_{A,i_N}$ and $\texttt{ReorderedSet}_{B,i_1} \dots \texttt{ReorderedSet}_{B,i_N}$. Prior work has proposed different ways to vectorize set intersections with SIMD instructions, but most of them target the general intersection problem in which the input size is sufficiently large. However, in our segment-level intersection, the number of elements in each segment are usually very small. Therefore, a solution to the general intersection problem may suffer from undesirable performance. As a result, we design specialized SIMD intersection kernels for inputs with small sizes. These specialized kernels are more efficient, since they are able to avoid unnecessary computations. For example, if one set has two elements and the other set has four elements, a general SIMD set intersection approach usually assumes that both sets have four elements, which introduces unnecessary computations.

In this section, we will first discuss how to dispatch each segment to the corresponding specialized intersection kernel in Section V-A. The implementation of each kernel will be described later in Section V-B.

A. Runtime Dispatch

To achieve the best performance, we have implemented and compiled set intersection kernels in advance for all possible scenarios. For example, if there are at most $\mathcal{S}$ elements in a segment, we will have $(\mathcal{S}+1)^2$ different set intersection kernels, i.e., from 0-by-0 up to n-by-n set intersection. Note that some kernels may never be used, e.g., kernel 0-by-i ($0 \le i \le n$). With all set intersection kernels, a runtime dispatch mechanism is needed, which allows us to use the correct set intersection kernel given different input segment sizes (i.e., $\texttt{size}_{A,i_k}$ and $\texttt{size}_{B,i_k}$). We now describe how the runtime dispatch mechanism is implemented with switch statement. Note that the GCC compiler will generate a table in assembly instead of a sequence of if-else statements, which are prohibitively expensive.

```
/* the dispatch control code */
int ctrl = ((Sa << 3) | Sb);
/* jump to a specialized intersection kernel */
switch(ctrl & 0x7F){
/* Each specialized kernel is a macro */
    case 0: break; //kernel0x0
    case 1: break; //kernel0x1
    case 2: break; //kernel0x2
    /* case 3 to 62 are omitted .... */
    case 63: Kernel7x7; break;
    default: GeneralIntersection(); break;
}
```

Listing 2. The jump table for specialized kernels

We implement and place all set intersection kernels in memory. Their starting addresses are referenced by a jump table as shown in Algorithm 1. The entries of the jump table are managed in a way such that a correct entry can be easily located through a control code, which is computed through simple arithmetic computation given a pair of segment sizes. Given the kth pair of segments A and B from a list of segments, we use S_a and S_b to denote their sizes $\texttt{size}_{A,i_k}$ and $\texttt{size}_{B,i_k}$. The control code for the jump table is computed by concatenating the two integers S_a and S_b.

We now use a concrete example to show how we compute the control code. Assume that there are 64 different set intersection kernels (from 0-by-0 up to 7-by-7 set intersection) in Listing 2. Since the maximum segment size is 7, it's sufficient to use three bits to represent S_a or S_b. As a result, we concatenate S_a and S_b into one integer through the following statement: ctrl $\leftarrow (S_a \ll \lceil \log_2^{\mathcal{S}} \rceil)|S_b$.

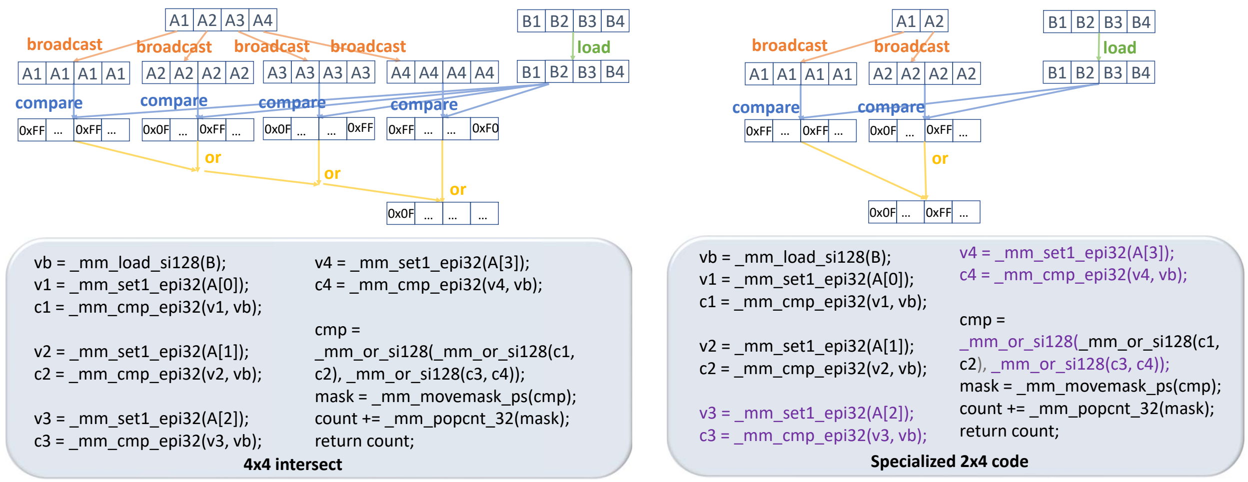


Fig. 2. Illustrating the difference between a general and a specialized SIMD set intersection kernels. A general kernel is shown on the left-hand side, which is implemented with SSE-128 instructions and can be used for any intersection with input size less than 4-by-4. A specialized 2-by-4 kernel is shown on the right-hand side, which reduces unnecessary computation and memory accesses (highlighted in purple).

In summary, bits 3 to 5 of the control code indicate size S_a. Similarly, bits 0 to 2 of the control code indicate size S_b.

B. Specialized vs. General Set Intersection Kernels

The reasons that a specialized set intersection kernel achieves better performance than a general one are twofold: (1) less computations, and (2)more efficient memory access. We illustrate the difference between a general and specialized set intersection kernel in Fig. 2. We show a general 4-by-4 intersection kernel on the left-hand side of the figure. The general kernel is implemented with the SSE-128 instructions and can be used for any pair of sets with input size smaller than 4-by-4 (e.g., 2-by-4). However, it is not as efficient as the specialized 2-by-4 intersection kernel, which is shown on the right-hand side of the figure. The specialized kernel on the right-side of Fig. 2 is also implemented with the SSE-128 instructions but it avoids unnecessary computations and memory accesses (highlighted in purple in the figure).

In addition, specialized kernels can take more advantage of data reuse. For example, when the set size is larger than the vector length, some elements may be used in multiple comparisons. With a specialized kernel, these elements can be put in registers to avoid redundant memory accesses and minimize the cost of comparisons at the same time.

C. Implementing Specialized Intersection Kernels with SIMD

We now describe how we implement the specialized intersection kernels with SIMD for different input sizes. We start with a scenario in which the size of both input sets equals to the vector length (i.e., $S_a = S_b = V$). Note that S_a and S_b denote the sizes of the input sets and $V = w/S_e$, in which w is the SIMD word width and S_e is the size of each element in the set.

V-by-V set intersection ($S_a = S_b = V$): We first describe the key idea of implementing a $V \times V$ intersection kernel, which performs a complete all pair comparisons between all V elements of the two input sets. Our SIMD implementation iteratively picks one element from set A and compares it with with all V elements in set B simultaneously. Note that the comparison results of all V elements of set A are combined together into a single vector (e.g., `cmp` as shown in Fig. 2).

There are four types of SSE instructions on 32-bit integers in our SIMD implementation, namely, (1) load: `_mm_load_si128`, (2) broadcast: `_mm_set1_epi32`, (3) compare: `_mm_cmp_epi32`, and (4) bitwise `OR`: `_mm_or_si128`. We now describe the details as follow. First, the V elements of set B are loaded into one vector register v_b. Second, each element of set A is broadcast to a different vector v_i $(i = 1, 2, 3, 4)$, and is compared with v_b. The comparison result between v_i and set v_b is stored in vector c_i. Finally, the four comparison results $c_1, \ldots, c_4$ are combined together into one mask register with a bitwise `OR` instruction.

We next describe how we implement the specialized intersection kernels for other input sizes. Without loss of generality, we divide all intersection kernels into three categories based their input sizes: (1) small-by-small set intersection ($S_a \leq S_b \leq V$): the size of both input sets is less than the vector length, (2) small-by-larger set intersection ($S_a \leq V < S_b$): the size of one input size is less than the vector length, and the size of the other input size is larger than the vector length, and (3) large-by-large set intersection ($V < S_a \leq S_b$): the size of both input sets is larger than the vector length.

Small-by-small set intersection ($S_a \leq S_b \leq V$): When S_a and S_b are both less than the vector length V, the implementation of our specialized kernels removes unnecessary operations, as shown in Fig. 2. Compared to the general 4-by-4, the specialized 2-by-4 kernel does not need to perform complete 4-by-4 comparisons. Instead, it only compares between the two elements in set A and the four elements in set B, which only takes two broadcasts, two comparisons and one bitwise `OR` instruction. In summary, the number of comparisons and memory access operations in the specialized kernel is only half of that as in the general 4-by-4 kernel.

```
1  Function: Intersect2by7(A, B)
2    v_b = _mm_load_si128(B)
3    v_1 = _mm_set1_epi32(A_1)
4    c_1 = _mm_cmp_epi32(v_1, v_b)
5    v_2 = _mm_set1_epi32(A_2)
6    c_2 = _mm_cmp_epi32(v_2, v_b)
7    cmp = _mm_or_si128(c_1, c_2)
8    mask = _mm_movemask_ps(cmp)
9    count = _mm_popcnt_32(mask)
10   v_b = _mm_load_si128(B_4)
11   c_1 = _mm_cmp_epi32(v_1, v_b)
12   c_2 = _mm_cmp_epi32(v_2, v_b)
13   cmp = _mm_or_si128(c_1, c_2)
14   mask = _mm_movemask_ps(cmp)
15   count += _mm_popcnt_32(mask&7)
16   return count
```

```
1  Function: Intersect4by5(A, B)
2    v_a = _mm_load_si128(A)
3    v_1 = _mm_set1_epi32(B_1)
4    c_1 = _mm_cmp_epi32(v_1, v_a)
5    v_2 = _mm_set1_epi32(B_2)
6    c_2 = _mm_cmp_epi32(v_2, v_a)
7    v_3 = _mm_set1_epi32(B_3)
8    c_3 = _mm_cmp_epi32(v_3, v_a)
9    v_4 = _mm_set1_epi32(B_4)
10   c_4 = _mm_cmp_epi32(v_4, v_a)
11   /* v_5 and c_5 are omitted */
12   t_1 = _mm_or_si128(c_1, c_2)
13   t_2 = _mm_or_si128(c_3, c_4)
14   t_3 = _mm_or_si128(t_2, c_5)
15   cmp = _mm_or_si128(t_1, t_3)
16   mask = _mm_movemask_ps(cmp)
17   count = _mm_popcnt_32(mask)
18   return count
```

```
1  Function: Intersect6by6(A, B)
2    v_b = _mm_load_si128(B)
3    v_1 = _mm_set1_epi32(A_1)
4    c_1 = _mm_cmp_epi32(v_1, v_b)
5    /* v_2 to v_6 and c_2 to c_6 are omitted */
6    t_1 = _mm_or_si128(c_1, c_2)
7    /* t_2 and t_3 are omitted */
8    cmp = _mm_or_si128(t_1, t_2)
9    cmp = _mm_or_si128(cmp, t_3)
10   mask1 = _mm_movemask_ps(cmp)
11   count = _mm_popcnt_32(mask)
12   v_b = _mm_load_si128(B_4)
13   c_7 = _mm_cmp_epi32(v_5, v_b)
14   c_8 = _mm_cmp_epi32(v_6, v_b)
15   cmp = _mm_or_si128(c_7 , c_8)
16   mask2 = _mm_movemask_ps(cmp)
17   count += _mm_popcnt_32(mask2&3)
18   return count
```

Fig. 3. Illustrating the specialized kernels: (1) a 2-by-7 intersection kernel (small-by-large), (2) a 4-by-5 intersection kernel (small-by-large and S_b is slightly larger than V), and (3) a 6-by-6 intersection kernel (large-by-large).

Small-by-large set intersection ($S_a \leq V < S_b$): The kernels for small-by-small intersections can be used to build the specialized kernels for small-by-large intersections, in which the size of one set is less than the vector length V and the size of the other set is larger than the vector length V ($S_a \leq V \leq S_b$). The key idea is that we first compare all elements in set A with the first V elements of set B.

We now use the 2-by-7 set intersection (as shown in the left side of Fig. 3) as an example. We start with comparing the two elements in set A against the first four elements in set B. In particular, two elements A_1 and A_2 are broadcast into vector registers v_1 and v_2. They are next compared with the first 4 elements in set B in vector register v_b. Line 7-9 combines the comparison results together through a bitwise `OR` operation. We next apply the the same process to compare the two elements in set A with the remaining three elements in set B, by loading the three elements into one vector register all together, as shown at line 10-15. Note that two elements A_1 and A_2 can be reused from registers v_1 and v_2.

Compared to the general kernel, this specialized kernel reduces computation, has better data reuse and avoids redundant loads. In summary, the number of broadcasts in the specialized kernel equals to the size of the small set (i.e., $S_a = 2$). The number of loads equals to the size of the large set divided by the SIMD width (i.e., $\lceil S_b/w \rceil = \lceil 7/4 \rceil = 2$). The number of comparisons is $2S_a$, since each element in the small set requires two comparisons with the large set.

However, this specialized kernel may be sub-optimal for some cases due to more comparisons. For example, when the size of set A is 4 and the size of set B is only slightly larger than the vector size V (e.g., $S_b = 5$), the fifth element in set B requires four comparisons against set A. This is because the four elements $A_1, \ldots, A_4$ are in four different vector registers due to broadcast operations. The Intersect4by5 example in the middle of Fig. 3 illustrates a better way to implement such scenario. First, set A is loaded into a vector register. Second, elements in set B are broadcast and compared with this register. In total, the number of comparisons is only five instead of eight.

Large-by-large set intersection ($V < S_a \leq S_b$): The implementation of our specialized kernels is built on top of small-by-large and small-by-large kernels, when both S_a and S_b are larger than the vector length V.

Let's now take the 6-by-6 set intersection (as shown in the right side of Fig. 3) as an example. We first apply a 4-by-4 intersection to compare the first V elements between set A and set B. There are two scenarios in the second step: (1) $A_4 \leq B_4$: To maximize data reuse, it is more efficient to load all elements from set B into one vector and broadcast all elements from set A. Therefore, we apply a 2-by-6 intersection between the fifth and sixth element of set A and all six elements of set B. The example in the right side of Fig. 3 corresponds to this scenario. (2) $A_4 > B_4$: This is the symmetric scenario. Therefore, it is more efficient to load all elements from set A into one vector and broadcast all elements from set B. Note that we have implemented both kernels above, and use the correct one based on the comparison result between A_4 and B_4 in runtime.

VI. Discussion

In this section, we first describe how our approach is used for k-way intersection and its time complexity. We next discuss the case for input with dramatically different sizes and how to extend our approach to exploit multicore parallelism and wider vector width.

k-way intersection: Similar to the process of set intersection between two sets, our approach can also leverage the segmented bitmaps to quickly prune the mismatches among k sets. Given k sets $L_1, L_2, \ldots, L_k$, we now describe the two-step set intersection as follows: (1) for each set L_i, there is a corresponding $\texttt{Bitmap}_i$. In this step, we compare the k bitmaps (i.e., $\texttt{Bitmap}_1 \ldots \texttt{Bitmap}_k$) using a bitwise `AND` operation to quickly filter out segments whose bitmaps do not have intersection with others. As discussed in Section III, the output is a list of non-zero segments; (2) we next apply the specialized k-way intersection kernels to the list of segments to perform the intersections over their associated elements. Since the expensive k-way intersection kernels are only performed on the matched segments (i.e., output from the first step), the complexity of the k-way set intersection with our data structure is proportional to the intersection size, instead of the entire input size. The performance advantage is more significant when the final intersection size is small.

We now give a theoretical analysis of the time complexity of our k-way intersection approach. For clarity, we assume the size of each set is n and we use w to denote the word size and r to denote the intersection size.

Proposition 2. *The time complexity of the k-way intersection algorithm is $O(kn/\sqrt{w}+r)$.*

Proof. There are two steps in the algorithm: (1) bitwise `AND` on k bitmaps, and (2) computing the intersection on the matched segments. The analysis for the time spent in each step is similar to what we discussed in Proposition 1. Therefore, the time for the bitwise operation with SIMD instructions is $O(k * m/w)$. The time spent in the second step depends on the number of false positive matches. As in Proposition 1, we have the expected number of false positive matches:

$$\mathrm{E}(I_{\mathrm{FP}}) = \sum_{e_1 \in L_1,\ldots,e_k \in L_k} P(h(e_1) = \cdots = h(e_k)) = \frac{n^k}{m^{k-1}}$$

In summary, when $m = n\sqrt{w}$, we have the total number of both true and false positive matches:

$$\mathrm{E}(I) = \mathrm{E}(I_T) + \mathrm{E}(I_{\mathrm{FP}}) = n/\sqrt{w}^{k-1} + r$$

In summary, the time complexity of the k-way intersection algorithm is $O(kn/\sqrt{w}+r)$. □

Input with dramatically different sizes: When two sets have dramatically different sizes (i.e., $n_1 \ll n_2$), we adapt our approach to a different strategy such that we can achieve the same time complexity as in a hash-based method, i.e. $O(\min(n_1, n_2)) = O(n_1)$. The key idea is that we go through each element in the smaller set and check the existence of the element in the larger set. Note that if the larger set has the element, then the element must be in segment $v \bmod m_2$, in which m_2 is the size of larger set's bitmap. If the corresponding bit in the bitmap is not set, meaning the element is not in the larger set, all subsequent comparisons are avoided. Otherwise, the element is compared against all the elements mapped to that position.

Multicore parallelism: Our set intersection approach can be easily extended to exploit more parallelism on multicore processors, since there are no cross-iteration dependencies.

The bitmaps of input sets can be partitioned and distributed onto different CPU cores such that each core can independently perform the bitwise `AND` operation on its partition and use specialized kernels to compute the intersection result on the matched segments. Note that when the two input sets have dramatically different sizes, we only partition and distribute the elements in the small set so that each core can next perform our approach agains the bitmap of the large set independently.

Wider vector width: Intel is introducing the AVX512 instructions the Skylake and Cannonlake architecture. As discussed earlier, there is a bitwise `AND` operation in the first step, which can have linear speedups with wider SIMD width. In addition, our specialized kernels are also designed to support arbitrary vector length V. However, directly applying our approach with wider SIMD instructions leads to significant performance degradation. This is because increasing segment size s leads to more elements in a segment on average. As the segment size goes up, more specialized intersection kernels for larger sizes are needed. As a result, the complexity of the jump table goes up as well. When the size of the jump table exceeds the instruction cache size, severe performance degradation will happen.

To reduce the number of specialized intersection kernels in the jump table, some specialized intersection are omitted and not implemented. In particular, instead of enumerating intersection kernels for all size pairs, we only implement intersection kernels at some sampled sizes (e.g., the even sizes). For a segment whose size falls in between those sampled sizes, its size is rounded up to the next larger size, meaning we will use a slightly larger specialized kernel for this segment. Although this causes some redundant computations, it significantly reduces the number of intersection kernels in the jump table.

VII. Experiments

In this section, we study the performance of our approach compared to the state-of-the art set intersection algorithms on both synthetic and real-world datasets.

A. Experimental Setup

Platforms: We implement our algorithms on platforms with SSE/AVX/AVX512 instructions and compare with state-of-the-art set intersection methods. We use two Intel platforms in our evaluation: (1) Intel E5-2695, which is a Haswell architecture and supports SSE(128-bit) and AVX(256-bit) instructions, and (2) Intel i7-7820X, which is a Skylake architecture and has the latest AVX512 instruction support in addition to SSE and AVX instructions.

Datasets: We perform the evaluation on both synthetic and real-world datasets [19], [20], focusing on how the following three key factors: (1) input size (n), (2) selectivity (r/n), and (3) skew in the input sizes (n_1/n_2). Note that if the original input data is unordered, our approach sorts the elements of each segment after hashing all elements into the bitmap.

Methods: We study and compare the performance of our approach with the following state-of-the-art implementations: (1) Scalar: This is an optimized scalar merge implementation similar to the code in Listing 1, but it replaces the expensive if-else branch statements with conditional moves, (2) Shuffling [13]: This is a SIMD implementation for set intersection and it is similar to what is presented in Fig. 2. It uses the SIMD shuffling instruction to perform all pair-wise comparisons between two input vectors by creating all variations of one vector. (3) scalarGalloping [18]: This is a binary-search based

TABLE II
L1 INSTRUCTION CACHE MISS

SIMD Kernels	AVX512	AVX512-stride4	AVX512-stride8
Code size (bytes)	520k	48k	12k
L1 icache miss #	50,009	43,178	34,596

intersection method. (4) SIMDGalloping [2]: A SIMD optimized version of scalarGalloping, and (5) BMiss [1]: A merge-based intersection approach with SIMD and optimizations on reducing branch mispredictions. The experiments on synthetic data are single-threaded. The multi-thread speedup is provided in Fig. 13. Note that the data structure of our approach is built offline and the construction time is not included in all experiments. The time to build the data structure on real world datasets is reported in Section VII-F.

B. Result of Specialized Intersection Kernels

As discussed in Section V, our system adopts different strategies and generates specialized kernels for intersections with different sizes. The specialized kernels can take advantage of SIMD instructions, which perform fewer memory accesses, shuffle and comparison computations.

We now study the performance of our specialized intersection kernels compared to the generalized SIMD implementation. Fig. 4 shows the result of SSE kernels. We generated kernel sizes from 1-by-1 up to 7-by-7, which is twice larger than the SSE SIMD width. We can observe that our specialized kernels are up to 70% faster than the general SIMD intersection implementation. Similarly, Fig. 5 shows the result of AVX kernels. The kernel size goes up to 15-by-15 and the specialized kernels are faster than the general AVX intersection implementation in all scenarios. The performance advantage is even more significant when the size of one set is much larger than the other set. Fig. 6 shows the result for AVX512 kernels, which are up to 6.7x faster than the general SIMD intersection implementation.

Kernel size and cache miss: By allowing sub-sampling of the kernel combinations, we can reduce the total number of kernels and improve the instruction cache hit rate. For example, as shown in Table II, for AVX512 instructions on the Skylake processor, if the kernel combinations are chosen in a stride of 4 (i.e., 4-by-4, 4-by-8, etc), compared to generating all the kernel combinations, the code size can be reduced by 90% and the number of L1 instruction cache misses on synthetic intersection data is reduced by 13%. Note that the total number of instructions remains the same. If the kernels are generated in a stride of 8, the code size can be reduced by 98% and L1 instruction cache misses is reduced by 30%.

C. Effect of Varying the Input Size

We next study the performance of different intersection approaches with a varying input size. In this experiment, we evaluate each intersection method with two synthetic sets. In addition, we make the size of input sets identical and their intersection size be 1% of the input size. We vary the number of elements in input from 400K to 3.2M. Note that our approach is implemented on two different Intel processors with three different SIMD instruction sets: (1) SSE, (2)AVX, and (3)AVX512. The performance of our SSE, AVX implementation is measured on an Intel Haswell architecture, which is reported in Fig. 7(a). The performance of our AVX512 implementation is measured on an Intel Skylake architecture, which is reported in Fig. 7(b).

In Fig. 7, the y-axis shows the CPU time in million cycles (i.e., the lower, the better). We can observe that the relative performance of these methods remains consistent as we increase the input size. On the Haswell architecture, our SSE and AVX implementation can achieve up to 7.6x speedup compared to scalar methods, and 1.4x-3.5x speedup compared to other SIMD methods. On the Skylake architecture, our AVX512 implementation can achieve 2-4x speedup compared to other SIMD methods. On both architectures, Scalar and ScalarGalloping are the slowest, since scalar implementations cannot leverage data-level parallelism. SIMDGalloping performs poorly as well, because it is based on binary search and has higher complexity when two input set sizes are similar.

D. Effect of Varying the Selectivity

Selectivity is the metric that describes how large the intersection size is relative to the input size. It is defined as the ratio of the intersection size divided by the input size (i.e., r/n). We now study the performance of different intersection approaches when varying the selectivity. In this experiment, we fix the size of the two input sets to one million and report the relative speedup of different approaches to the Scalar intersection method in Fig. 8 and Fig. 9.

Fig. 8 shows the performance of each approach on the Haswell architecture using SSE and AVX instructions. We observe that our approach can achieve up to 7.6x speedups compared to state-of-the-art Scalar intersection methods, and 1.8x speedups compared to state-of-the-art SIMD intersection methods. Fig. 9 shows the performance of each approach on the Skylake architecture using AVX512 instructions. We observe that our approach can achieve up to 6x speedups compared to state-of-the-art Scalar intersection methods, and 1.4-3x speedups compared to state-of-the-art SIMD intersection methods.

In addition, we see that the our method's speedup is higher when the selectivity becomes lower. Note that in most real-world scenarios, the intersection size is usually less than 10% of the input set (i.e., selectivity is less than 0.1).

k-way intersection: We also study the performance of each method on three-way intersection. We fix the input size to one million as in the experiment above, and report the result in Fig. 10. The y-axis is the relative speedup to the Scalar method. The x-axis shows the set density, which describes how clustered the elements are distributed among a given range. For example, elements in a dense set are randomly drew from a small range, while a sparse set have elements drew from a larger range. Density can affect the intersection size. For example, two dense sets are more likely to have common

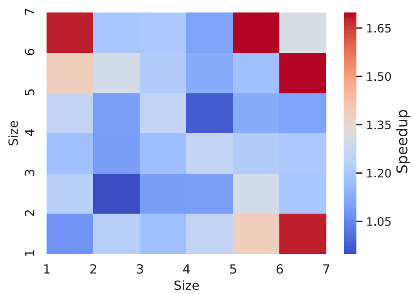


Fig. 4. Speedups of SSE kernels

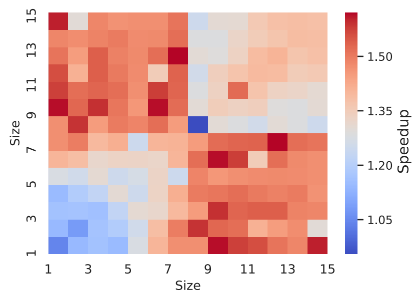


Fig. 5. Speedups of AVX kernels

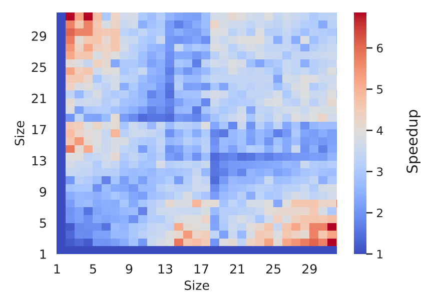
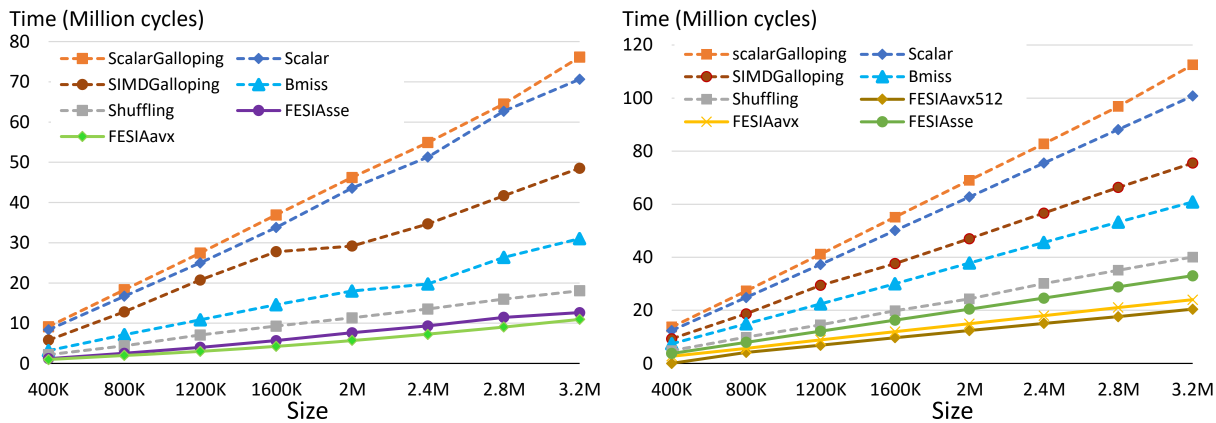


Fig. 6. Speedups of AVX 512 kernels



(a) Performance on varying sizes on Intel Haswell

(b) Performance on varying sizes on Intel Skylake

Fig. 7. Performance comparison with a varying input size

elements. When $k = 3$, the selectivity is proportional to the third power of set density.

We can observe that FESIA can achieve up to 17.8x speedups compared to scalar intersection methods on 3-way intersection, and up to 4.8x speedups compared to state-of-the-art SIMD set intersection approaches. The speedup is higher when the density is lower. When the density is zero, the maximum speedup achieved in 3-way intersection is higher than that in 2-way intersection, which shows the speedup of our approach is more prominent with a larger k. This is because multi-way comparisons are more expensive for larger k. However, our approach can avoid the cost of unnecessary multi-way comparisons through cheap bitmap intersections.

E. Performance of Two Sets with Different Sizes

We now study the performance of each approach when the two input sets have different sizes. In this experiment, we fix the size of the larger input size to one million and the selectivity to 0.1. The x-axis is the skew, i.e., the ratio of the smaller set size to the larger set size (n_1/n_2). The y-axis is the relative speedup to the Scalar method. Note that our approach can adapt to different strategies depending on the value of skew. We use $\text{FESIA}_{\text{merge}}$ to denote the strategy we use when the input has a similar size and $\text{FESIA}_{\text{hash}}$ to denote the strategy we use when the input has dramatically different sizes. We report the performance of both strategies in Fig. 11.

Theoretically, when the skew is small ($n_1 \ll n_2$), the hash-based method has the lowest average complexity and the merge based method has the highest average complexity. The complexity of the binary search method is between the two above. When the skew is large ($n_1 \sim n_2$), the average complexity of the merge method is comparable with the hash-based method, and the binary search method has the highest complexity.

As shown in Fig. 11, the actual performance of the intersection methods can match the theoretical analysis. When the skew is small, our method based on hash (i.e., $\text{FESIA}_{\text{hash}}$) has the best performance, which is 2-3x faster than the binary-search based method SIMDGalloping, while SIMDGalloping is faster than the two SIMD merge-based methods (Shuffling and BMiss). As the skew goes up to more than 1/4, $\text{FESIA}_{\text{merge}}$ starts to outperform $\text{FESIA}_{\text{hash}}$ and it achieves the best performance among all approaches. For other methods, the SIMD merge-based method can outperform methods based on binary search.

In summary, FESIA can adapt to different strategies given different skew in input, and achieves better performance than other approaches. All other approaches only perform well when the skew is either small or large.

F. Performance on Real-World Datasets

We next study the performance of each intersection approach on two real-world tasks: (1) a database query task,

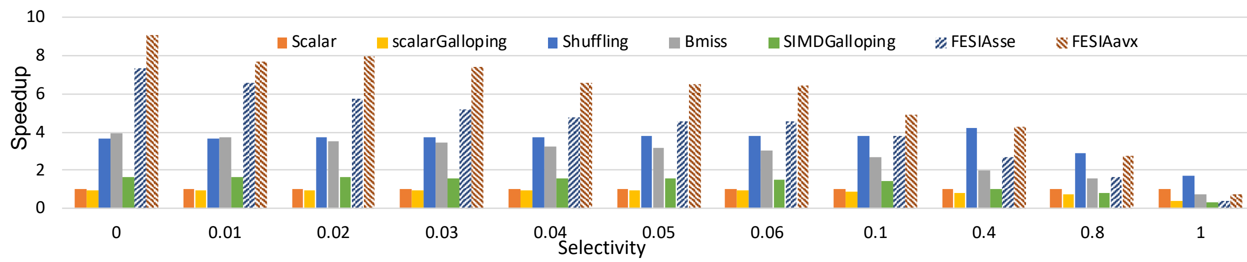


Fig. 8. Performance on different selectivity on Intel Haswell

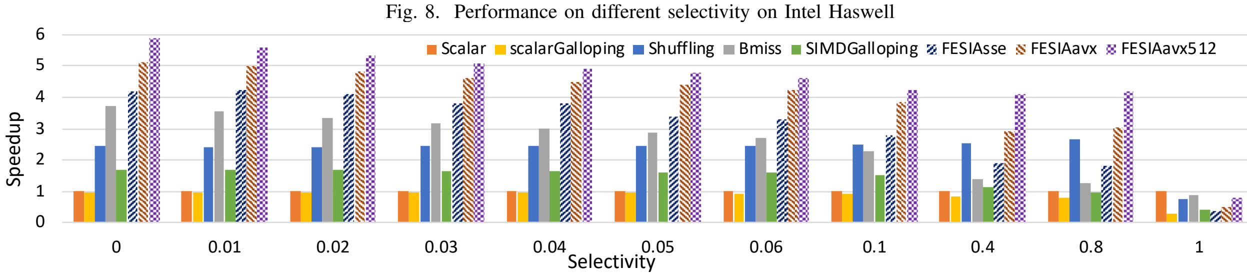


Fig. 9. Performance on different selectivity on Intel Skylake

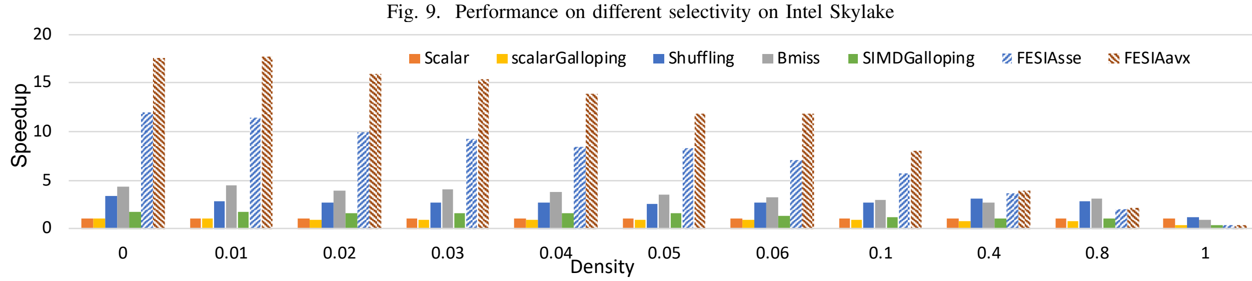


Fig. 10. Performance of three-way intersection

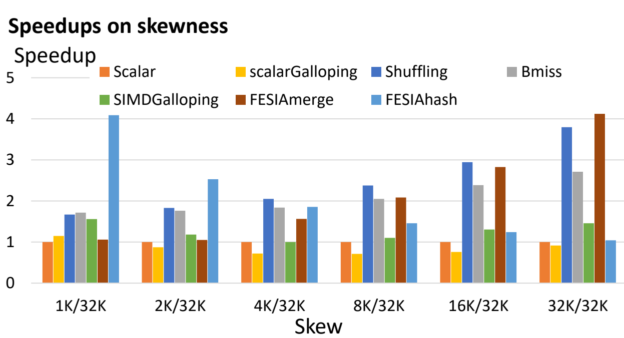


Fig. 11. Performance comparison on varying skew

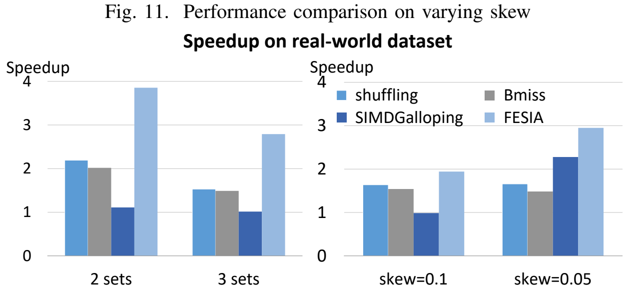


Fig. 12. Results on the database query task

and (2) a triangle counting task in graph analytics [6].

The database query task: In Fig. 12, we report the result of each approach on a real-life dataset called WebDocs from the Frequent Itemset Mining Dataset Repository [19]. The WebDocs dataset is a web crawl dataset built from a collection of web HTML documents. The whole collection has about 1.7 million documents with 5,267,656 distinct items. The data structure's construction time on this dataset is 77.7s.

To simulate the low selectivity of real-world queries, we generate random queries from the dataset and keep the set intersection size below 20% of the input size. We show the result of set intersection with two input sets and three input sets on the top of Fig. 12. We observe that our approach achieves close to 4x speedup compared to the Scalar method, 2x speedup compared to the SIMD Shuffling, and 3.8x speedup compared to SIMDGalloping. The bottom of Fig. 12 shows the performance of each approach when the sizes of input sets are skewed. Overall, we observe that our approach has an up to 3x speedup compared to other approaches.

The triangle counting task: We now study the performance of each approach on the triangle counting task with three graph analytics datasets. The three datasets are from the Stanford Large Network Dataset Collection [20] and we report the details of each dataset in Table III. In Fig. 13, we can observe that our approach can achieve up to 12x speedup compared

TABLE III
THE DETAILS OF EACH GRAPH DATASET

Dataset	# of nodes	# of edges	construction time
Patents	3,774,768	16,518,948	0.25s
HepPh	34,546	421,578	0.004s
LiveJournal	3,997,962	34,681,189	0.38s

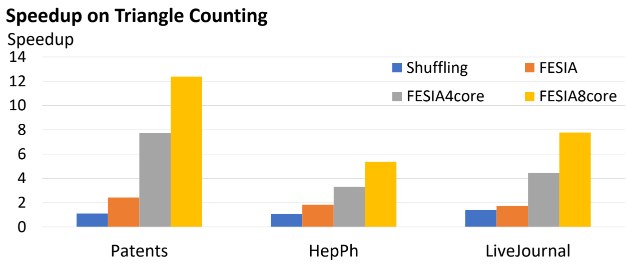


Fig. 13. Results on the triangle counting task

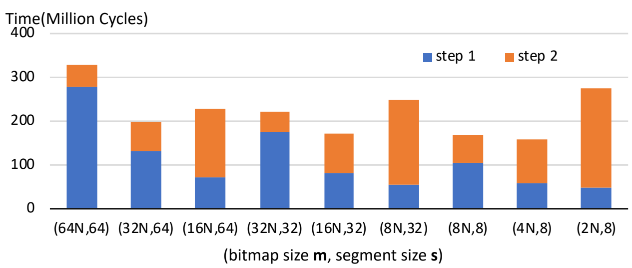


Fig. 14. Performance breakdown on varying bitmap size and segment size

to the Scalar method and up to 1.7x speedup compared to the SIMD Shuffling approach. In addition, the speedup can scale linearly with the number of CPU cores.

G. Performance Breakdown

In this experiment, we study how much time is spent in each step in our approach with different bitmap sizes (m) and segment sizes (s) and report the result in Fig. 14. The input set size is 200kB and the selectivity is zero. As we can observe, when we decrease the segment size (s) with a constant bitmap size (m), the time spent in the bitmap intersection (step 1) increases while the time spent in the segment intersection (step 2) decreases. This is because our approach iterates over the bitmap segment by segment in step 1, and the time is proportional to the number of segments. Smaller s leads to more segments and longer processing time. Meanwhile, smaller s also reduces the intersection time in step 2, since it takes less time for segment intersection.

VIII. CONCLUSION

In this paper, we presented FESIA, an efficient SIMD-vectorized approach for set intersection on modern CPUs. In many real-world tasks such as database queries and graph analytics, the intersection size is usually orders of magnitude smaller than the size of the input sets. FESIA leverages this observation by adopting a bitmap-based approach to efficiently prune necessary comparisons and refine a list of smaller segments with intersecting elements. These segments are later processed by specialized SIMD kernels for final intersection. Experiments on both real-world and synthetic datasets show that our approach can achieve an order of magnitude speedup compared to scalar set intersection methods and be up to 4x faster than state-of-the-art SIMD implementations.

Acknowledgments. We thank the reviewers for their valuable comments. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-0133]. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

REFERENCES

[1] H. Inoue, M. Ohara, and K. Taura, "Faster set intersection with SIMD instructions by reducing branch mispredictions," *PVLDB*, vol. 8, no. 3, pp. 293–304, 2014.
[2] D. Lemire, L. Boytsov, and N. Kurz, "SIMD compression and the intersection of sorted integers," *Softw., Pract. Exper.*, vol. 46, no. 6, pp. 723–749, 2016.
[3] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using SIMD instructions," in *ADMS@VLDB*, 2011, pp. 1–8.
[4] B. Ding and A. C. König, "Fast set intersection in memory," *PVLDB*, vol. 4, no. 4, pp. 255–266, 2011.
[5] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, no. 2, pp. 22–28, 2003.
[6] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *KDD*, 2011, pp. 672–680.
[7] X. Hu, Y. Tao, and C. Chung, "Massive graph triangulation," in *SIGMOD Conference*, 2013, pp. 325–336.
[8] N. Wang, J. Zhang, K. Tan, and A. K. H. Tung, "On triangulation-based dense neighborhood graphs discovery," *PVLDB*, vol. 4, no. 2, pp. 58–68, 2010.
[9] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD Conference*, 2013, pp. 337–348.
[10] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *ISAAC*, 2010, pp. 403–414.
[11] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *SIGMOD Conference*, 2013, pp. 277–288.
[12] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using SIMD instructions," in *SIGMOD Conference*, 2018, pp. 1587–1602.
[13] I. Katsov, "Fast intersection of sorted lists using SSE instructions," https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/, 2012.
[14] G. E. Blelloch and M. Reid-Miller, "Fast set operations using treaps," in *SPAA*, 1998, pp. 16–26.
[15] W. Pugh, "A skip list cookbook," Tech. Rep., 1990.
[16] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. S. Y. Kai, "Roaring bitmaps: Implementation of an optimized software library," *Softw., Pract. Exper.*, vol. 48, no. 4, pp. 867–895, 2018.
[17] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000, pp. 743–752.
[18] J. L. Bentley and A. C.-C. Yao, "An almost optimal algorithm for unbounded searching," *Information processing letters*, vol. 5, no. SLAC-PUB-1679, 1976.
[19] http://fimi.cs.helsinki.fi/data/.
[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.