Fast Sorted-Set Intersection using SIMD Instructions

Benjamin Schlegel
TU Dresden
Dresden, Germany
benjamin.schlegel@tu-dresden.de

Thomas Willhalm
Intel GmbH
Munich, Germany
thomas.willhalm@intel.com

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

In this paper, we focus on sorted-set intersection which is an important part in many algorithms, e.g., RID-list intersection, inverted indexes, and others. In contrast to traditional scalar sorted-set intersection algorithms that try to reduce the number of comparisons, we propose a parallel algorithm that relies on speculative execution of comparisons. In general, our algorithm requires more comparisons but less instructions than scalar algorithms that translates into a better overall speed. We achieve this by utilizing efficient single-instruction-multiple-data (SIMD) instructions that are available in many processors. We provide different sorted-set intersection algorithms for different integer data types. We propose versions that use uncompressed integer values as input and output as well as a version that uses a tailor-made data layout for even faster intersections. In our experiments, we achieve speedups up to 5.3x compared to popular fast scalar algorithms.

1. INTRODUCTION

Sorted-set intersection is a fundamental operation in query processing in the area of databases and information retrieval. It is part of many algorithms and often accounts for a large fraction of the overall runtime. Examples are inverted indexes in information retrieval [25], lists intersection in frequent-itemset mining [1, 28], and merging of RID-lists in database query processing [19]. In many of these application areas low latencies are a key concern so that reducing the execution time of set intersection is very important.

There has been done a lot of research with the goal of improving sorted-set intersection. Many approaches focus on speed up sequential intersection [4, 5, 9, 11, 13, 20] by using efficient data structures or improved processing techniques. Other approaches focus on utilizing modern hardware like graphic processors (GPUs) [1, 2, 12, 26, 27] and multi-core CPUs [23, 24] to utilize the parallelism offered by these processors. However, the main focus of these approaches is only on thread-level parallelism. Data-level parallelism is so far not considered although it is available via SIMD instruction sets in almost all modern CPUs. For this reason, our goal in this paper is to speed up the intersection of sorted sets using SIMD capabilities.

Utilization of SIMD capabilities in algorithms is ideally achieved through automatic vectorizing by compilers or by inserting SIMD instructions manually. However, many compilers detect vectorization opportunities only for simple loop constructs with few or without any data dependencies. In all other cases, hand-tuned assembly or intrinsics must be used. Unfortunately, all sorted-set intersection algorithms have complex data dependencies so that automatic vectorization cannot be applied.

In this paper, we introduce parallel sorted-set intersection algorithms that rely on *speculative execution*. Our main intention is to speculatively execute more than the necessary comparisons as done by the scalar algorithms. To do this efficiently, we use the *string and text processing new instructions* (STTNI) that are part of the Intel® Streaming SIMD Extensions 4.2 (Intel® SSE 4.2).[1] These instructions allow a fast *full comparison* of either eight 16-bit values (= 64 comparisons) or sixteen 8-bit values (= 256 comparisons) with only one instruction. Many of these comparisons are useless and would not been exectuted by scalar algorithms. However, since these instructions itself require only 8 cycles to complete [14] we achieve significant speedups.

In summary, our contributions are as follows:

- We propose fast parallel sorted-set intersection algorithms for 8-bit, 16-bit and 32-bit integer values based on STTNI of Intel SSE 4.2. The algorithms use uncompressed integer values as input and output. We explain in detail the necessary SIMD instructions and steps of the parallel intersection.
- We present a hierarchical data layout that is tailor-made for a fast parallel intersection of integer values with a precision higher than 16 bits.
- We compare our parallel algorithms with two highly efficient scalar versions on synthetic datasets.

The scope of our paper is as follows. We focus solely on sorted-set intersection of integer values without duplicates. Usually, this is not a limitation since both of our example scenarios—information retrieval and frequent-itemset

[1]Intel and Core are trademarks of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others.

mining—deal only with integer values without any duplicates. Nonetheless, our parallel algorithms could be extended to allow duplicates by storing the frequencies of the integer values. We further assume that all sets being intersected are already available in main memory. This holds true for all of our application areas since the data is either loaded or generated in main memory during pre-processing or kept in memory all the time, e.g., in in-memory databases.

2. BACKGROUND AND RELATED WORK

In this section, we survey related work and explain a typical scalar sorted-set intersection algorithm. In the remainder, we present necessary SIMD instructions used for our parallel algorithms.

2.1 Related Work

We focus on set intersection where both sorted sets have a similar cardinality. If the difference of the cardinality of the two sets is very large, search techniques (e.g., binary or interpolation search) are used to find corresponding items in the larger set [13]. For sets with similar cardinality the well-known merge algorithm—similar to the merge step of merge-sort [18]—is used. Prior research provided many optimizations for this algorithm that can be divided into three classes: (1) *adaptive intersection*, (2) *hierarchical intersection*, and (3) *parallel intersection*.

Adaptive algorithms [3, 4, 5, 9] focus on reducing the number of comparisons during the intersection; this is achieved by combining the merge algorithm with various search techniques. This works very well when applied on skewed data. However, adaptivity sometimes introduces a large overhead that offsets the benefit of the fewer comparisons [10, 11]. Our main idea to execute even more than the necessary comparisons seems contrary to the adaptive approaches. Nevertheless, a combination of our approach with adaptive algorithms would be interesting to reduce the number of full comparisons.

Besides the pure merge approaches there are also hierarchical approaches that utilize different—often tree-like—data structures to speed up the intersection. The main advantage is a faster insertion of new integer values into the sets. Examples for such data structures are AVL trees [7] and treaps [6]. We will show that many of these hierarchical approaches can be combined with our parallel algorithms. Other approaches represent sets using two levels, an upper and a lower level. Sanders [20] provides a two-level representation in which an upper set contains entry points for a number of lower sets. The goal is to use compression for the lower-level sets and allow binary search on the upper level set. Tsirogiannis [24] partitions a set into subsets until these subsets consist of only uniformly distributed values. This increases the effect of interpolation search within the subsets. A small partition table is used for upper-level intersection. This approach is orthogonal to our work; intersection of the small subsets could be executed using our data-parallel algorithms. In recent work, Ding [11] provides a two-level approach based on hashing. The values of a set are mapped to bitmaps tailor-made for the word-length of a processor. The intersection is then executed by a bit-wise ANDing of the bitmaps. The approach works very well if the cardinality of the result set is small. Otherwise, the reconstruction of the bitmaps to actual integer values usually offsets any benefit of the fast bitmap-merging.

```
int intersect(short *A, short *B,
              int l_a, int l_b, short* C) {
 int count = 0;
 int i_a = 0, i_b = 0;

 while(i_a < l_a && i_b < l_a) {
  if(A[i_a] == B[i_b]) {
   C[count++]=A[i_a];
   i_a++;
   i_b++;
  }
  else if (A[i_a] > B[i_b])
   i_b++;
  else
   i_a++;
 }
 return count;
}
```

Figure 1: Code snippet for scalar sorted-set intersection with branches for 16-bit integer values.

Parallel set intersection algorithms [23, 24] focus on utilizing multi-core processors to speed up the intersection process. Tsirogiannis [24] proposes a set partitioning that allows an efficient load balancing for multi-core processors. Furthermore, there is a large number of algorithms tailor-made for GPUs [1, 2, 12, 26, 27]; all these works focus on massive thread-level parallelism and efficient data layouts for set intersection on GPUs. In summary, all of these parallel algorithms focus solely on thread-level parallelism. In this paper, we provide the first intersection algorithm that utilizes data-level parallelism offered by modern CPUs. To the best of our knowledge, there exists no such work so far.

Apart from the set intersection algorithms, there are several algorithms that benefit from STTNI. Clearly, the most obvious use-cases are string and text processing algorithms; a large number of code samples that were optimized using STTNI can be found in Chapter 11 of Intel's Optimization Reference Manual [14]. This includes amongst others null character identification, string comparison, string token extraction, word counting, sub-string searching, and string-to-integer conversion. More application-specific algorithms like accelerating XML parsing and XML schema validation [17] are further typical candidates since they heavily rely on string processing. However, also other algorithms that have, at first sight, nothing in common with string processing benefit of STTNI. Shi [22] proposes optimized search algorithms for searching in arrays, trees, and hash-tables; these algorithms achieve high speedups compared to their scalar counterparts. Nevertheless, none of these works considers to accelerate sorted-set intersection using STTNI.

2.2 Scalar sorted-set intersection

Sorted-set intersection requires two sorted sets as input and outputs a sorted set that contains common values of both input sets. Let A and B be two sorted-sets of length l_a and l_b, respectively, and C the result set. The scalar algorithm iteratively compares two values taken from each set; it starts with the first value of both sets. Whenever the two compared values are equal, the value is written into the result set and a counter—that counts the number of common values—is incremented. Then, the next value of each set is loaded for the next comparison. If the compared values are

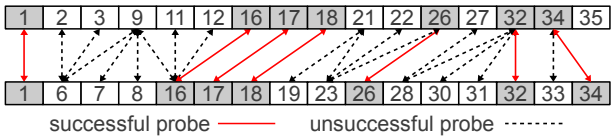

Figure 2: Scalar intersection of two sorted sets with 24 comparisons; the resulting intersection set consists of 7 elements (gray shaded).

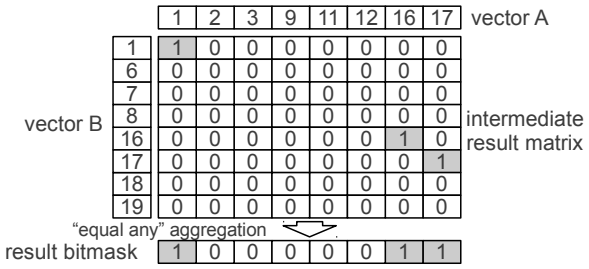

Figure 3: Parallel intersection of two vectors using the PCMPESTRM instruction. The intermediate result matrix holds the results of the full-comparison and is aggregated using the "equal any" aggregation method. The result bitmask indicates that the first, seventh, and eighth value of vector A are also found in vector B.

unequal, then only the next value of the sorted set with the smaller value is loaded and compared with the current value of the other set. The algorithm finishes when there is no next value in one the two sets. Furthermore, indices i_a and i_b are used for maintaining the position of the next value of each set. Figure 1 shows the C-code for an implementation of this algorithm.

The worst-case number of comparisons for intersecting two sorted-sets of length l_a and l_b is given by $l_a + l_b - 1$; in this case, the sets have no common value. The best-case requires only $\min(l_a, l_b)$ comparisons. Figure 2 illustrates the scalar sorted-set intersection with an example. Here, 24 comparisons are necessary to find the 7 common values.

With a data-parallel version of the scalar algorithm in mind, the following characteristics of this algorithm are important. The scalar code consists only of comparison, arithmetic, and load/store instructions. Furthermore, there is no defined pattern for increasing the indices i_a and i_b because whether the next value is loaded from set A or set B depends on the values being intersected. Consider the example in Figure 1. Here, the indices (0-based) are increased for the first five comparisons as follows: $(i_a = 0, i_b = 0)$, $(i_a = 1, i_b = 1)$, $(i_a = 3, i_b = 2)$, $(i_a = 4, i_b = 2)$, and $(i_a = 4, i_b = 3)$. This behaviour plays an important role for a data-parallel version of the merge algorithm.

2.3 SIMD capabilities

Initially introduced with MMX for the Intel Pentium processor, there exist a large number of SIMD instruction sets like 3DNow, different revisions of SSE, and AVX for modern x86 processors. Compared to traditional scalar instructions that process only a single data element at once, a SIMD instruction can process k elements at once; k itself depends on the capabilities of the instruction set as well as on the data type. For example, most SSE instructions can process four 32-bit or eight 16-bit values at once, while AVX instructions allow to process eight 32-bit values. SIMD instructions can be classified into load and store, element-wise, and horizontal instructions. In this paper, we require instructions of all three classes.

Load and store instructions copy data from main memory into the SIMD registers and vice versa. For our algorithms, we require aligned as well as unaligned load and store instructions. For current processors, there is no performance difference whether aligned or unaligned memory load/store instructions are used; however, unaligned loads can cause cacheline splits. Note that loading and storing of vector registers is restricted to continuous memory chunks, i.e., if four 32-bit values should be loaded into a vector register with one instruction, all four values must reside in a continuous block of 128 bit. Scatter and gather operations—that allow to load/store data for each of the k elements independently—are not supported by current and near future processors.[2]

Element-wise instructions execute an operation element-wise on one or more input vectors, i.e., the n-th element of the output vector only depends on the n-th element of each input vector. There exists a large number of element-wise instructions including arithmetic, comparison, shift, conversion, and logical instructions. For the algorithms in this work, we only need element-wise arithmetic and conversion instructions. More in detail, we require the conversion instructions `PUNPCKHWD` and `PUNPCKLWD` that take two vectors as input and mix the eight upper (`PUNPCKHWD`) or eight lower (`PUNPCKLWD`) 16-bit elements of both vectors and write them into an output vector.

Horizontal instructions act across the elements of a SIMD register. Here, the n-th element of the output vector can possibly depend on all elements of the input vectors. In this paper, we make extensive use of one of the *four* instructions of STTNI. Originally intended for string comparison, the four instructions differ only in the output format (index or bitmask) and whether the length specification of the processed strings is given explicitly or implicitly (using a '0'-termination symbol). The execution procedure of all four instructions consist of three stages: (1) full comparison of two input vectors both consisting of either sixteen 8-bit values or eight 16-bit values. (2) aggregation of the intermediate comparison result using one of four different aggregation functions. For our algorithms, we use the "equal any" aggregation function; this function ORs the comparison results per element for the second input vector. In other words, if there is at least one element in the first input vector equal to the n-th element of the second input vector, then the n-th bit in the aggregated bitmask is set to one. (3) the final output is returned. Two instructions of STTNI return the aggregated bitmask while the other two instructions return the position of the first or last "one" bit in the aggregated bitmap. In our algorithms, we use the `PCMPESTRM` instruction that requires an explicit length specification and returns an aggregated mask. Figure 3 illustrates the intersection of two sets—each represented as a vector with 8 elements—using the `PCMPESTRM` instruction.

Besides of the `PCMPESTRM` instruction, we make use of the byte-shuffle instruction `PSHUFB` that is part of the SSSE3

[2]Gather instructions have just been announced for Haswell.

instruction set and performs arbitrary in-place shuffles of the bytes in a SIMD register. The byte permutation itself is controlled by a shuffle control mask.

All required instructions are supported by Intel processors starting with the Nehalem architecture [16] and forthcoming AMD processors starting with the Bulldozer architecture [8]. More details about the used instructions are available in the Intel Software Developer's Manual [15].

3. PARALLEL SET INTERSECTION

In general, exploiting SIMD capabilities for sorted-set intersection could be done by either executing k sorted set intersections in parallel or speed up the execution of a single sorted-set intersection. We will first discuss why SIMD capabilities cannot be efficiently utilized for the first option and will then propose our solution for the second option.

3.1 Multiple intersections

The standard approach for exploiting SIMD instructions would be to execute k independent set intersections in parallel instead of a single scalar set intersection. This could be achieved by exchanging scalar instructions with SIMD instructions. Scalar comparison instructions as well as scalar arithmetic instructions could be exchanged since SSE provides SIMD counterparts for these instructions (see Section 2.3). However, the main problem of this approach is that available load and store instructions are restricted to loading and storing continuous memory chunks.

In each step, the scalar merge requires to load one value from each set; for k parallel merges this implies loading k values from the k upper sets and loading k values from the k lower sets. Since as mentioned before, increasing of the set's indices for a scalar merge depends on the values being intersected, the indices of the k upper sets as well as the indices of the k lower sets are increased independently of each other. For this reason, it is not possible to load the k values from k sets from a continuous chunk of memory using the aforementioned load instructions. Scatter and gather instructions would be a solution to this problem, but these instructions are not supported by the current and next-generation of processors. So the only way is to fill up the vector registers with $2 \cdot k$ scalar load instructions, introducing a high sequential fraction to the algorithm.

Another problem is that the sorted sets being intersected should have similar lengths because the achievable speedup is reduced if some of the k merges finish earlier. For these two reasons, we focus on the second option: exploiting SIMD instructions to speed up the intersection of two sorted sets.

3.2 Basic intersection

The parallel merge algorithm is based on *speculative execution* of comparisons. While the scalar merge compares in each iteration only the two values indexed by i_a and i_b of both input sets, the main idea of the parallel merge is to compare values that are beyond the current index values. Usually, such an approach would not be beneficial because of the additional overhead for the large percentage of unnecessary comparisons. Nevertheless, because of the fact that the `PCMPESTRM` instructions allow a large number of comparisons in almost the same amount of time of a single comparison, we can achieve a high speedup despite the many unnecessary comparisons.

Since all the instructions of STTNI are restricted to 8-bit and 16-bit integer values, the basic parallel sorted-set intersection can only process integer values of such a precision. In the following, we focus on the parallel merge of two sets with 16-bit values; the version for 8-bit values works similarly.

```
int intersect(short *A, short *B,
              int l_a, int l_b, short* C) {
 int count = 0;
 short i_a = 0, i_b = 0;

 while(i_a < l_a && i_b < l_b) {
  // 1. Load the vectors
  __m128i v_a = _mm_load_si128((__m128i*)&A[i_a]);
  __m128i v_b = _mm_load_si128((__m128i*)&B[i_b]);

  // 2. Full comparison
  __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8,
     _SIDD_UWORD_OPS|_SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
  int r = _mm_extract_epi32(res_v, 0);
  unsigned short a7 = _mm_extract_epi32(v_a, 7);
  unsigned short b7 = _mm_extract_epi32(v_b, 7);
  A += ( a7 <= b7 ) * 8;
  B += ( a7 >= b7 ) * 8;

  // 3. Write back common values
  __m128i p = _mm_shuffle_epi8(v_a, sh_mask[r]);
  _mm_storeu_si128((__m128i*)&C[count], p);
  count += _mm_popcnt_u32(r);
 }
 return count;
}
```

Figure 4: Code snippet for parallel sorted-set intersection of 16-bit integer values using SIMD instructions.

Similar to the scalar version, the input of the parallel algorithm are two sorted arrays A and B with lengths l_a and l_b, respectively. Both arrays are 16-byte aligned and, for the ease of explanation, we assume that the length of each vector is a multiple of 8. Furthermore, two variables $i_a = 0$ and $i_b = 0$ are required to indicate already processed values in the arrays A and B, respectively. Finally, the output consisting of the common values of A and B is written back to an array C. The number of common values is stored in *count*. The major steps of the algorithm are: (1) load the data into the vector registers, (2) do a full-comparison of both vectors using the `PCMPESTRM` instruction, and (3) write back and count the common values. We will now explain the algorithm in more detail.

The main loop is executed as long as there are unprocessed values in both sets ($i_a < l_a$ and $i_b < l_b$).

1. **Load both vectors** Eight consecutive values of the array A starting at position i_a are loaded into a vector register v_a. Similar, the vector register v_b is filled with eight consecutive values of B starting at i_b. Both loads are executed using two aligned load instructions.

2. **Fully compare both vectors** The full comparison of v_a and v_b is executed using the `PCMPESTRM` instruction; the comparison result—a bitmap consisting of 8 bits—is stored in a variable r. Each "one" bit in r indicates a common value in v_a and v_b. More specifically, a bit at position n in r is set to one if there is a value in v_b equal to the n-th value in v_a.

   Furthermore, a scalar comparison of the respective last

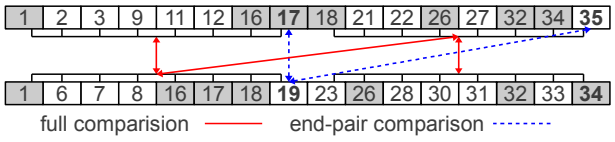

Figure 5: Parallel linear merge of two sorted sets with only three full comparisons and two end-pair comparisons. These comparisons on the particular last value of each vector (bold) are used to decide which vectors are compared next.

values of v_a and v_b is used to select the values of A and B for the next full comparison. If the last value of v_a is smaller than the last value of v_b, then i_a is increased by 8. Otherwise, v_b is increased by 8. In case of the last values of v_a and v_b are equal, both values i_a and i_b are increased by 8. This ensures that all common values are found once and only once.

3. **Write back common values** Finally, the common values stored in v_a are packed together and are stored in C. This packing is performed using the byte permutation (shuffle) instruction. The required permutation mask is obtained by using the bitmask r as an index in a permutation mask lookup-table. Finally, a unaligned store instruction is used to write back the packed common values. The number of common values is calculated using a population count instruction on the bitmask r.

The C-code of the algorithm[3] is shown in Figure 4 while Figure 5 illustrates the parallel merge on the same sorted sets as used before (c.f. Figure 2). Each set consists of 16 values. The first step is the full comparison of the first eight values of both vectors. After this comparison, the common values 1, 16, and 17 are found and written back to the result vector (not shown). The comparison of the last value of each vector, 17 for the upper vector and 19 for the lower vector, indicates that all values smaller or equal than 17 are already merged. However, since the values 18 and 19 from the lower set may be also in the upper set, only the index of the upper set is increased by 8. After the second full comparison, the value 18 is found. The last full comparison (second 8 values of both sets) reveals the values 26, 32, and 34. After only three full comparisons and two scalar comparisons all common values are found.

The parallel merge requires more comparisons than the scalar merge. For the 16-bit parallel merge, the best-case occurs if both sets are identical and the length of both sets is a multiple of 8. Given that, the number of comparisons is given by $8 \cdot l_a$ because after each full comparison the algorithm proceeds in both sets by 8 values. The worst-case number of comparisons is given by $16 \cdot l_a - 64$. In this case, both sets have the same cardinality but the last values of all fully compared vectors are always unequal so that the algorithm proceeds after a full comparison only in one set by 8 values. Under the *unrealistic* assumption that the `PCMPESTRM` instruction can process the 64 comparisons in the same time like a single scalar comparison instruction, the speedup for the parallel merge would be in a range of 4x to 8x. Similar considerations lead to a potential speedup from 8x to 16x for the parallel merge of 8-bit values.

[3]The code snippet has been adapted for better readability and differs from the source code used for the experiments.

3.3 Hierarchical intersection

So far, we presented an parallel sorted-set intersection algorithm for 8-bit and 16-bit integer values. Since STTNI can only be applied on such integer values, we propose a hierarchical intersection approach that allows processing integers of higher precision (> 16-bit). We make use of the fact that each integer value for a given domain $\mathcal{D}$ could be divided into h upper bits and 16 lower bits where $h + 16$ is the precision of the integer; e.g., $h = 16$ for 32-bit integer values. Using this, we can partition the set A into disjoint subsets $A_1, A_2, \ldots, A_m$ where each subset A_i consists of integer values that share the same upper h bits. The set B is partitioned into $B_1, B_2, \ldots, B_n$ in the same way. Based on that, hierarchical intersection works as follows:

Upper-level intersection is done by merging the subsets $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$ based on the shared h bits of each subset. For each pair A_i and B_j that share the same h bits, the lower-level intersection is executed.

Lower-level intersection merges two subsets A_i and B_j based on the lower 16-bits of each value. This is done using our parallel 16-bit intersection algorithm.

In summary, we have a single upper-level intersection and multiple lower-level intersections. The upper-level intersection could be executed based on the scalar merge (c.f. Section 2.2) or also using an intersection based on STTNI. However, an important observation is that in most cases the upper-level intersection is responsible only for a small fraction of the execution time of the complete intersection process. In general, the fraction of upper-level and lower-level execution time depends on the average cardinality of the subsets. Roughly speaking, the higher the cardinality of the subsets, the more of the execution time is spend for the intersection of the subsets. For this reason, we use a scalar algorithm for the upper-level intersection and our 16-bit parallel algorithm for the lower-level intersections.

We provide two different hierarchical parallel merge algorithms. The first algorithm processes uncompressed integer values and requires a *pre- and post-processing* to partition these values into subsets during processing. The second algorithm uses a tailor-made data structure to avoid the overhead of such processing during the intersection. We will explain both algorithms for 32-bit values only; the steps are similar for integers of higher precision.

In the first algorithm, we execute pre- and post-processing during the intersection to avoid large intermediate results that would be necessary if the phases are separated from the complete intersection process. Instead, we maintain only two buffers each with 65,536 elements. Whenever a lower-level intersection is executed, we copy the lower 16-bit values of both subsets being intersected into the buffers. We achieve this be using the `PSHUFB` instruction to move the lower 16-bit of all values to the lower part of a SIMD register. We copy the lower part of two such "shuffled" SIMD registers into the respective buffer using a `PUNPCKHWD` and a store instruction. After all values of both subsets are copied into their assigned buffers, the pre-processing for this lower-level intersection is finished and the actual parallel 16-bit

intersection is executed.[4] The separation of the complete copying of the 16-bit values and their intersection afterwards avoids complex code that would require many branches and would probably be slow. However, post-processing is executed without buffering. It is used to concatenate the upper 16 bits with each 16-bit value of the lower-level intersection result. We utilize the instructions `PUNPCKHWD` and `PUNPCKLWD` for this concatenation. The upper 16-bit are replicated stored in the first register while the second register contains the intersection results. After each intersection iteration, we execute both conversion instructions followed by two unaligned store instructions.

The second algorithm avoids the pre- and post-processing overhead by using a tailor-made data layout for the input sets. The main idea is to store the values of a set already partitioned using two levels; all data of a set is stored in a continuous memory area consisting of an array of 16-bit values. The subsets $A_1, A_2, \ldots, A_m$ of A are then stored as follows: starting at the array, the shared 16 bits of the values in A_1 are stored once using a single cell. The next cell contains the cardinality of A_1. A single cell for the cardinality is sufficient since the maximum number of values in a subset never exceeds 65536. The next $|A_1|$ cells comprise the lower 16-bit of the values of A_1. After this, the next subsets A_2 to A_m are stored in a similar way.

The two-level data layout has often lower memory requirements than the original representation of the uncompressed integer values it consists of. This is because the upper bits of all values of a subset are stored only once. Only in the worst-case—each subset contains only one element—we have additional overhead of 16 bit for the subset's length field per integer value. However, this case is very unusual for realistic datasets. For example, document ids in inverted indexes have normally a sequential numbering. Furthermore, already for an average cardinality of 2 the memory requirements are the same as for storing uncompressed 32-bit values. For higher average cardinality the memory requirements approach 16 bit per value.

Conversion of the two-level data layout into 32-bit values can be done with almost no overhead during intersection using the post-processing of the first hierarchical algorithm. As indicated by experiments, there is only a small difference in the execution times whether the intersection result is written back in the two-level data layout or as uncompressed 32-bit integer values.

4. EXPERIMENTAL EVALUATION

This section contains the results of our experimental evaluation. We first give an overview of the used setup and will then show the results of our experiments.

4.1 Setup

We implemented several algorithms using C as programming language. More specifically, we implemented two scalar versions. The first version (*branch*) uses branches in the main loop for increasing the indices (c.f. Figure 1) while the second version (*branchless*) is based on predication[5] and has no branches in the main loop. We optimized both versions by hand. As we will show, the usual goal to avoid branches leads not always to the best results since under certain circumstances (perfect branch prediction) code with branches could be very fast. Each of the scalar versions is instrumented with C preprocessor directives to obtain versions tailor-made for different integer precisions (i.e., 8-bit, 16-bit, and 32-bit). Furthermore, we implemented *four* parallel algorithms using STTNI. The first three algorithms—*SIMD8*, *SIMD16*, and *SIMD32*—differ only in the precision of the integers they process. All three algorithms use the same interface as the scalar algorithms; they process and output uncompressed integer values. Finally, the fourth parallel version (*SIMD32-H*) is based on the proposed hierarchical data layout, which is used as input and output of the algorithm. We used C-intrinsics to insert SIMD instructions. All algorithms were compiled using Intel Parallel Composer 2011. We also tried the GCC compiler but the generated code was in all experiments slower.

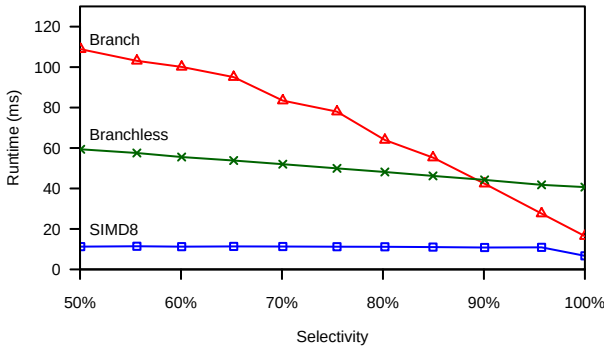

Figure 6: Varying the selectivity for 78,125 sorted-set intersections on sets with 128 elements and 8-bit integer values.

We evaluated all algorithms on a system comprised of an Intel® Core™ i7-920 processor with a core frequency of 2.67GHz and 6GB of main memory. We used Linux (2.6.34) as operating system. In all experiments, we measured the wall-clock time for the intersection process only. All input datasets were main memory resident. Furthermore, the output is written on preallocated memory so that there is no time required for retrieving the memory (e.g., malloc calls).

We generated synthetic datasets consisting of uniformly distributed values. We varied the *selectivity*—denoted as the fraction of the number of common values of both sets and the cardinality of the smaller set—of the datasets by changing the domain of the values. A greater domain leads to a smaller selectivity and vice versa. In each experiment we intersected in total 10 million values with another 10 million values. Since the maximum number of integers of the 8-bit and 16-bit versions is restricted to 256 and 65,536, respectively, we executed multiple merges for these versions; 78,125 merges of sets each with 128 elements for the 8-bit versions and 5,000 merges of sets each with 2,000 elements for the 16-bit versions.

4.2 Results

In the first experiment, we compared the execution time for intersecting 8-bit and 16-bit sorted sets. For that, we compared both scalar versions, branch and branchless, with

[4]Each 8 consecutive values in the buffers are not ordered anymore. The full comparison still works but we have to adapt the shuffle control masks that are required for packing the result in each iteration of the intersection.

[5]Transforming branches into data dependencies to avoid mispredicted branches.

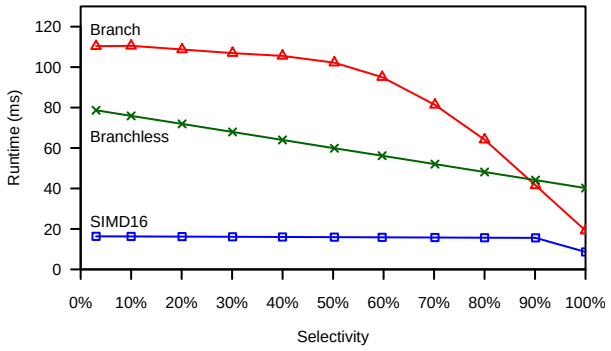

Figure 7: Varying the selectivity for 5,000 sorted-set intersections on sets with 2,000 elements and 16-bit integer values.

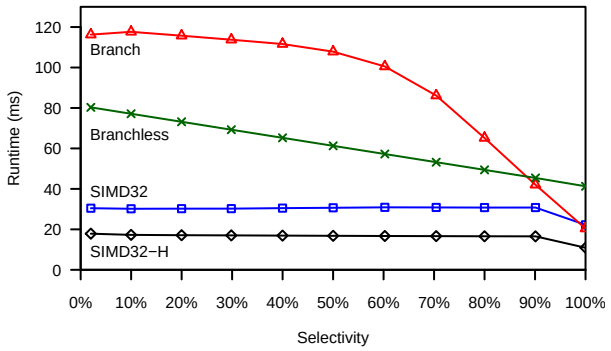

Figure 8: Varying the selectivity for a single sorted-set intersection of two sets with 10 million elements and 32-bit integer values.

SIMD8 and SIMD16. We varied the selectivity from 50% to 100% for 8-bit intersection algorithms and from 0% to 100% for the 16-bit algorithms.

Figure 6 illustrates the results for 8-bit intersection. Independently of the selectivity, SIMD8 outperforms both scalar versions; it has a nearly constant execution time of only about 11ms because it proceeds in almost each iteration by 16 elements in one of both sets. However, as both sets fully overlap (100% selectivity) it iteratively proceeds in each set by 16 elements so the runtime further decreases to 7ms. Of both scalar algorithms, branchless performs better up to a selectivity of 90%. At this point branch gets faster because of the nearly perfect branch prediction. Nevertheless, branchless as well as branch benefit from a increasing selectivity because for each found pair the index of both sets is increased. We achieve speedups from 2.4x to 5.3x compared to the *best performing* of the two considered scalar implementations at each measuring point.

We obtain similar results for the 16-bit sorted set intersection shown in Figure 7. Again, our parallel merge algorithm is always faster than both scalar algorithms. The average execution time of SIMD16 was 16ms and only 9ms for 100% selectivity. Since SIMD16 proceeds in almost each iteration by 8 elements, the speedups are lower than achieved by SIMD8. Nevertheless, SIMD16 is 2.2x up to 4.8x faster than the scalar algorithms (always considering the better runtime

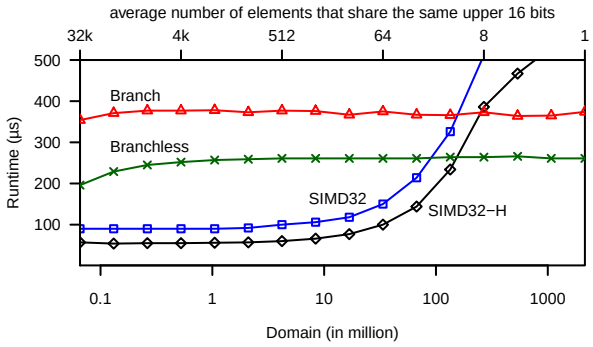

Figure 9: Varying the domain for sorted-set intersection of sets with 32,768 elements and 32-bit integer values. The higher the domain of the integer values is, the fewer values share the same upper 16 bits.

of both scalar algorithms). In summary, the speedups of both parallel algorithms differ from the theoretical speedups (c.f. Section 3.2) because of the additional overhead for the result permutation and higher latency of the `PCMPESTRM` instruction compared to a traditional compare instruction.

In the next experiment, we compared the two scalar 32-bit algorithms with our two parallel 32-bit algorithms, SIMD32 and SIMD32-H. We varied the selectivity from 0% to 100%. The results—shown in Figure 8—are similar to the results of the previous experiments. Also here, our parallel algorithms outperform the two scalar algorithms. However, Branch has nearly the same execution time as SIMD32 for 95% selectivity and is even slightly faster at 100% selectivity. Because of the pre- and post-processing overhead of SIMD32, the execution time of the parallel algorithms differs in average by 13ms. In other words, SIMD32-H is 1.8x faster than SIMD32. Furthermore, SIMD32-H is only about 1.5ms slower than SIMD16 indicating that the overhead for the two-level data layout is negligible.

In the last experiment, we varied the domain of the integer values being intersected. While in the previous experiments we used a relatively small domain for sets with a large number of elements, we will now proceed the other way around. Figure 9 illustrates the results for the intersection of 32,768 values for a domain ranging from 64k values up to 2 billion values. The increasing domain has nearly no effect on the scalar intersection algorithms. The slight increase of their execution times from 64k values to about 256k values (note the log scale) results from the higher selectivity for this domain. As shown by the other experiments, the scalar versions benefit from a higher selectivity. For a domain larger than 256k values, the selectivity gets smaller than 1%. The execution time of the parallel algorithms increases for a growing domain because the average number of values in a subset decreases. This again leads to a smaller positive effect of the full comparison. Nevertheless, the execution time deteriorates only for very large domains that are probably not typical for realistic datasets. SIMD32 is faster than the scalar algorithms as long there are 18 values per subset. SIMD32-H is even better; only for subsets with fewer than 12 values per subset it is slower than the scalar versions.

5. CONCLUSION

In this paper, we presented parallel sorted-set intersection algorithms that are based on STTNI of Intel SSE 4.2. Three of the algorithms use sorted sets consisting of uncompressed integer values as input and output; they differ only in the precision (8-bit, 16-bit, and 32-bit) of the integers they process. Since the 32-bit variant requires pre- and post-processing of the integer values, we provide a hierarchical data layout that avoids such overhead during the intersection process. Our experiments indicate high speedups up to 5.3x of our parallel sorted-set intersection algorithms over highly efficient scalar counterparts. We believe that many algorithms in query processing, frequent-itemset mining, or information retrieval—in which set intersection accounts for a large part of the overall execution time—could benefit from our proposed approach. In future work, we want to investigate the combination with adaptive intersection algorithms and parallel compression techniques [21].

6. REFERENCES

[1] R. R. Amossen and R. Pagh. A new data layout for set intersection on gpus. *CoRR*, abs/1102.1003, 2011.

[2] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. In *VLDB*, 2011.

[3] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, pages 400–408. 2004.

[4] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *WEA*, pages 146–157. Springer, 2006.

[5] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:7:3.7–7:3.24, January 2010.

[6] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallel algorithms and architectures*, SPAA '98, pages 16–26, New York, NY, USA, 1998. ACM.

[7] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26:211–226, April 1979.

[8] M. Butler. Amd "bulldozer" core - a new approach to multithreaded compute performance for maximum efficiency and throughput. *Hot Chips 22*, August 2010.

[9] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[10] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, London, UK, 2001. Springer-Verlag.

[11] B. Ding and A. C. König. Fast set intersection in memory. *Proc. VLDB Endow.*, 4:255–266, January 2011.

[12] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high-performance ir query processing. In *WWW*, pages 1213–1214, New York, NY, USA, 2008. ACM.

[13] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *Siam Journal on Computing*, 1:31–39, 1972.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2011.

[15] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A and 2B: Instruction Set Reference, A-Z*, May 2011.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, May 2011.

[17] Intel Inc. *Accelerating XML Processing with Intel SSE4.2 to Improve Business Solutions.*

[18] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[19] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, SIGMOD '07, pages 773–784, New York, NY, USA, 2007. ACM.

[20] P. Sanders and F. Transier. Intersection in integer inverted indices. In *ALENEX*. SIAM, 2007.

[21] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using simd instructions. In *DaMoN*, 2010.

[22] G. Shi, M. Li, and M. Lipasti. Accelerating search and recognition workloads with sse 4.2 string and text processing instructions. In *ISPASS*, pages 145 –153, april 2011.

[23] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *SIGIR*, SIGIR '09, pages 738–739, New York, NY, USA, 2009. ACM.

[24] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endow.*, 2:838–849, August 2009.

[25] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[26] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang. A batched gpu algorithm for set intersection. In *Symposium on Pervasive Systems, Algorithms, and Networks*, ISPAN '09, pages 752–756, Washington, DC, USA, 2009. IEEE Computer Society.

[27] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[28] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.