# OPTIMIZING VARIOUS TRIANGLE LISTING ALGORITHMS

*Nils Blach, Matthias Bungeroth, Zikai Liu, Jingyi Zhu*

Department of Computer Science
ETH Zurich, Switzerland

## ABSTRACT

We present optimized versions of three common triangle listing algorithms. These algorithms are used in various applications, but many of them only use straightforward implementations. With increasing interest in analyzing large graphs like the Internet, the performance of these algorithms becomes even more important. We optimize the algorithms in depth through improving memory layouts, reducing branch mispredictions, and applying SIMD instructions. Evaluations are performed on both generated graphs with various densities and real-world graphs, achieving significant speedups of up to 15x in comparison to the straightforward implementations. Furthermore, we reason about the achieved performance against the theoretical upper bounds and discuss a few limiting factors in the algorithms.

## 1. INTRODUCTION

**Motivation.** Listing triangles in graphs is a fundamental problem for many applications, such as network analysis of the Internet or social networks. For example, [1] shows the potential of detecting spam websites by listing triangles in a web graph, while [2] argues that triangles are good indicators of social roles in online forums. While the problem has been intensively studied from theoretical perspectives, the increasing interest in analyzing large networks like the WWW introduces demand for high-performance triangle listing implementations. However, optimizing those algorithms is a non-trivial work, due to dynamics in the data structures, dependencies between operations, etc.

**Contributions.** In this work, we present implementations and evaluations of several well-known triangle listing algorithms on the x86_64 single-core platform. Our algorithms show considerable speedup compared with the straightforward implementations. As the adjacency list is used as the basic data structure, those algorithms are capable to run on large, realistic graphs even on machines with limited memory. We comprehensively analyze them on various graphs, including generated graphs with different densities as well as real-world graphs.

**Related works.** [3] presents the triangle listing algorithms that this work focuses on. We also use the same or similar graphs in this work for our evaluations. However, the paper [3] only evaluates a straightforward C++ implementation for the algorithms. There are a few existing Python [4] and Java [5, 6] implementations of the triangle counting versions of the algorithms, but they are also straightforward and use off-the-shelf data structures for adjacency list intersection. Our work brings the algorithms further to in-depth optimizations with inspirations from works on fast set intersection. [7] proposes a method for vectorized balanced set intersection, which leverages SIMD instructions to reduce branch mispredictions. We extract the key ideas and extend them to work with larger vectors and smaller elements, as well as combine them with the insights gained from [8] to also cover unbalanced set intersection, which is often required in real-world scenarios. Aside from set intersection, sorting is another key operation for two out of the three algorithms, which we optimize through some ideas from [9].

## 2. BACKGROUND ON THE ALGORITHMS

The problem we try to solve is triangle listing. Given an undirected, simple graph $G = (V, E)$, we want to list all triangles in $G$. A triangle is a set of nodes $\{u, v, w\} \subseteq V$ such that all three edges $\{u, v\}$, $\{u, w\}$, $\{v, w\} \in E$.

### 2.1. Algorithms and Baseline Implementations

We focus on three triangle listing algorithms—Edge Iterator, Forward and Forward Hashed—proposed in [3]. We use an adjacency list representation of the graph because it allows us to store much larger graphs than the adjacency matrix representation. We use 32-bit integers to uniquely index Nodes (starting from 0).

**Edge Iterator.** It iterates over all edges $\{s, t\} \in E$ with two-level loops. We only consider pairs $(s, t)$ such that $s < t$ to avoid duplicated triangle. We use a while loop to intersect the sorted adjacency lists of $s$ and $t$.

**Forward.** It is the same as Edge Iterator except that it maintains a dynamic growing neighbor list for each node. Intersections are performed on these lists instead of the original graph. The lists are built dynamically, such that after

intersecting the lists of $s$ and $t$, $s$ is added to the list of $t$. In this way, Forward dynamically converts all edges into directed edges pointing from larger ID to smaller ID and only performs set intersections on the directed graph.

**Forward Hashed.** It is the same as Forward except that it uses a hash table for the dynamic neighbor list. The hash container uses separate chaining. It is implemented as an array of pointers to linked lists of nodes.

**Sorting.** Both Edge Iterator and Forward perform set intersections by iterating over two sorted arrays, so they need a preprocessing step to sort all the adjacency lists. We use the std::sort algorithm from the C++ standard for the baseline versions.

**Listing triangles.** When a common node $r$ such that $r > s$ and $r > t$ is found in the adjacency list intersection, we find a triangle and list it. The algorithms list triangles as $\{s, t, r\}$ in an array of `structs` of three nodes.

### 2.2. Cost Analysis

We consider the preprocessing step and the main algorithms separately. Quick sort takes $O(\sum_{v \in V} d(v) \log d(v))$. According to [3], the asymptotic runtime of Edge Iterator is $\sum_{v \in V} d(v)^2$, where $d(v)$ is the degree of node $v$. For Forward, it is $\sum_{\{u,v\} \in E} d_{in}(u) + d_{in}(v)$, where $d_{in}$ is the in-degree. As for Forward Hashed, the runtime is $\sum_{\{u,v\} \in E} \min\{d_{in}(u), d_{in}(v)\}$, assuming the hash operations have $O(1)$ complexity. These asymptotic complexities are captured by the number of *triangle operations* [3]. Triangle operations check whether three nodes construct a triangle. We define integer comparison operations as our cost measure because triangle operations are in essence comparison operations, while the integer operations allow qualitative comparisons with the theoretical upper bounds of processors. We instrumented our code to count the comparison operations, which depend on the topology of the graph and number of triangles.

## 3. OPTIMIZATIONS FOR EDGE ITERATOR AND FORWARD

Edge Iterator and Forward have a lot of similarities. Many optimizations apply to both algorithms, which we discuss in unison here in this section. Our optimizations focus on the set intersection which is the bottleneck. Despite all presented optimizations targeting triangle listing, we also explore triangle counting since it typically benefits more from the optimizations with fewer dependencies and branching.

### 3.1. Preprocessing

**Shorten Adjacency Lists.** The algorithms are given an undirected graph as the input. However, they only need an underlying directed graph to produce a correct listing without duplication. The baseline implementation sorts the entire adjacency lists in the preprocessing step and compares the node IDs ( `s < t` and `t < t.neighbor` ) on the fly, which essentially transforms the undirected graph into a directed graph. That introduces a lot of redundant branching and a greater possibility of branch mispredictions. We optimize the code by transforming the graph in place during preprocessing, we refer to this process as **cutting**. Before sorting, we perform a quick sort partition using $s$ as the pivot to move all `s.neighbor > s` to the front of the adjacency list and update the neighbor count accordingly to ignore all `s.neighbor < s`. This optimization shortens the adjacency lists, removes unnecessary branching in the code, and makes it easier to apply further optimizations.

**Sorting.** Since sorting takes a significant amount of time for graphs with long edge lists (up to 45% for measured graphs), we decided to optimize the sorting as well. We implement merge sort and sort buckets of size 8 with min, max, and shuffling AVX2 instructions. We use the approach described in [9] that sorts a single 512bit vector and port it to AVX2 256-bit.

### 3.2. Unbalanced Set Intersections

For some real-world graphs, we observe significant differences in the size of adjacency lists, whose degrees vary up to 4 orders of magnitude. Some previous works ([7], [8]) state that sorted set intersection of unbalanced lists (at least one order of magnitude difference) can be accelerated by binary-search-based techniques. At that scale, the benefit of the reduction in operation counts outweighs the degradation caused by non-continuous memory accesses, in comparison to linearly iterating through both lists. We perform exponential search on the larger of two lists and use the next-greater element of the previous iteration as the start point of the next. In case the list lengths do not differ by at least one magnitude, we utilize the standard linear approach, which we optimize in the following sections.

### 3.3. Block Set Intersections

In order to increase ILP and eliminate branches for the linear intersection of two sorted lists, we unroll two elements at a time from each of the two lists and perform a crosswise comparison, as depicted in Fig. 1. This reduces the dependencies on the two pointers of the lists and allows for the elimination of some branching, at the expense of an increased operation count. However, this trade-off is generally beneficial as can be seen in Sec. 5.

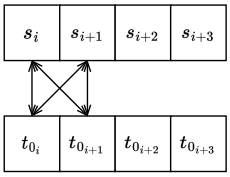


**Fig. 1**. Example 2-by-2 block intersection of two sorted lists.

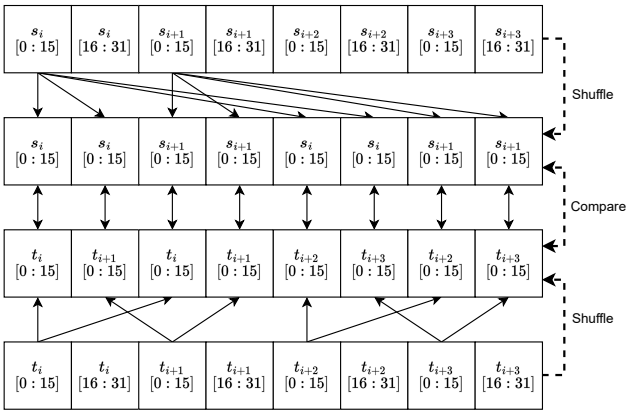


**Fig. 2**. Example showcasing the vectorized block set intersection of the lower 16 bits. Note that the shuffled vectors only depict the first 128 bits and not the full 256-bit vector. The complete shuffled $s$ vector is the depicted vector concatenated with the same vector but every element advanced by 2, while the full shuffled $t$ vector is twice the depicted version concatenated. This ensures each element of $s$ is compared with every other element in $t$.

### 3.4. Vectorized Block Set Intersections

We further unroll the intersection to perform 4-by-4 block intersections using AVX2 vector intrinsics. We do so in a hierarchical fashion similar to [7] by first comparing the lower 16 bits of the elements, checking for equivalence, and only comparing the upper 16 bits in case of matches. Fig. 2 shows how this comparison is performed. The final result is a bitmask of matches in a 32-bit integer, extracted using the `movemask` intrinsic and several bitwise operations.

In order to collect the triangles, we need to retrieve the index of the 1's in the bit-mask and the corresponding element. This requires 4 individual checks, one for each potential match. However, as a comparison, if only triangle counting is needed, it can be achieved by counting the number of set bits using the built-in function `popcount` .

Overall, this optimization allows us to exploit the reduction in branching and solves the issue of increased operation counts through vectorization, but does require significant shuffling operations that do not allow high ILP due to their sequential dependencies.

### 3.5. Interleaved Set Intersections

As another attempt focusing on improving ILP, we implement versions of Edge Iterator and Forward that perform multiple set intersections interleavingly. For example, given a node $s$ and its neighbors $t_0, t_1 \ldots, t_n$, we try to perform multiple intersections like $(s, t_0), (s, t_1), \ldots$ within one loop. In order to do this, it is essential to track the indices of both sets of what elements to be compared next. As those intersections can finish at different times, we iterate till the last one is done and stop the progress for the finished ones (this naturally happens since the indices are not smaller than the list length anymore). We unroll by a factor of 4 and apply this optimization to achieve the best possible ILP.

One problem with this approach is that, if the discrepancy in list lengths is large, we do a lot of additional work since we can not exit the loop early.

### 3.6. Vectorized Interleaved Set Intersections

In order to vectorize the interleaved set intersections, we first remove all the branches in the inner loop. It is achieved by storing "if conditions" in boolean values and incrementing indices with these boolean values. In order to stop the intersections that are finished already, those boolean variables are anded with the boolean indication if the intersection is done or not.

We then vectorize the implementation by replacing all operations with AVX2 operations. The most tricky part is to load the correct neighbors of $t$ and of $s_0, \ldots, s_8$. To load neighbors of $t$, `_mm256_i32gather_epi32` is used with the index vector. To load different nodes from sets $s_0, \ldots, s_8$, we use the following approach. When allocating the neighbor container, we ensure that all of the adjacency lists are allocated in a continuous memory region such that there is a common base address for lists of all $s_i$. We can then compute the offset to the base address from the nodes indices of $s_i$, enabling us to use `_mm256_i32gather_epi32` to load the 8 different elements using the indices vector. This only works if the neighbor containers have a size smaller than $2^{36}$ bytes, since we can only use 32-bit offsets multiplied by `sizeof(uint32_t)= 4`.

In order to collect the triangles, we use `move_mask` to get a bitmask of what elements we need to collect. We then store the relevant vector that is needed to reconstruct the triangle in memory and add the triangle to the array. As a comparison, if only triangle counting is needed, the operation is much more trivial: just add the boolean condition vector to the result count vector.

To increase the ILP and speed up the version even more, we use 4 instead of 1 vector, which effectively unrolls the loop by a factor of 32.

### 3.7. Scalar Replacement and Compiler Optimizations

Our graph is laid out as one array of adjacency structures and one memory region of contiguous adjacency lists. To access one adjacency list, we at least need to follow two levels of indirection. Since the adjacency list of $s$ is shared by all of its neighbors for set intersection, scalar replacement of the adjacency structure of $s$ would reduce memory accesses and improve the performance. We expected the compiler to perform scalar replacement. However, the assembly of Edge Iterator shows that instead of storing the address of the adjacency list of $s$ in a register, the compiled code dereferences its pointer in every inner loop. We, therefore, manually store the adjacency structure of $s$ and other common expressions in temporary variables in the code.

The reason why the compiler does not apply scalar replacement is largely related to it being conservative about memory aliasing, so we tried compiling with the `-fno-strict-aliasing` flag. However, the manual scalar replacement version compiled without the flag still outperforms it by a small margin and using the flag results in a worse performance for the vectorized versions.

## 4. OPTIMIZATIONS FOR FORWARD HASHED

Our optimizations for Forward Hashed mainly focus on the hash table. In the baseline version, we implement a hash table that uses separate chaining. Each slot of the hash table holds a pointer to NULL or the first entry of a linked list. Each hash entry consists of a 32-bit node ID and a pointer to the next entry. The hash table has a fixed number of slots that are dynamically allocated. The number of slots is a power of two and we use a single modulo as the hash function, which is transformed into a AND operation. Iterating through a hash table involves iterating every slot. In this section, we discuss the optimizations that are effective, while the failed attempts are briefly discussed in Section 7.2.

### 4.1. Remove Indirections

The baseline implementation is inefficient since accessing the first entry of a non-empty slot already involves one pointer access, even if there is no conflict. We address this problem by embedding one entry in every slot. In other words, the hash table is pre-allocating one entry per slot. Figure 3 shows an example of 4-slot hash table. This optimization eliminates one level of Indirection when accessing entries.

As pre-allocated entries take space even if the slots are not used, an empty table becomes twice as large (pointers are aligned to 64 bits so there is 32-bit padding after `Node`, but it is no longer a problem after applying the optimization discussed next). To indicate a slot is empty, we use `UINT32_MAX` for a non-existing node (NULL node).

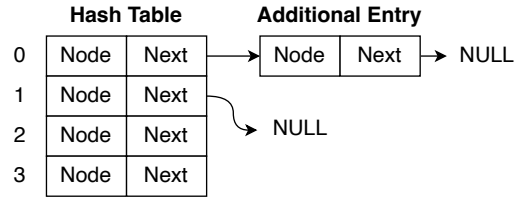


**Fig. 3**. Example hash table with 4 slots and one entry pre-allocated for each slot.

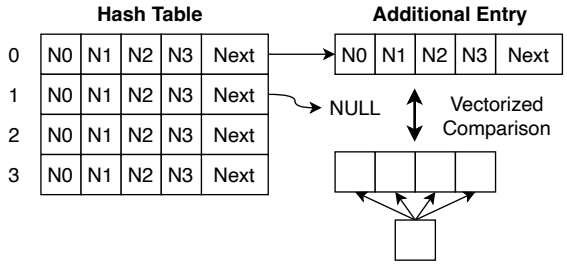


**Fig. 4**. Example hash table with 4 slots and 4 nodes per entry. The 4 nodes can be compared with the target node in a single SIMD instruction.

In addition, instead of dynamically allocating a fixed number of slots, we use fixed-size arrays, which eliminates another level of indirections and allows hash tables of all nodes in the graph to be compactly packed into continuous memory. Computing the address of a slot becomes straightforward arithmetics without any pointer access. Only conflicting entries that are inserted later are accessed by chasing down the linked list.

### 4.2. Multiple Nodes per Each Entry

Pre-allocation removes one level of indirection. However, pointer chasing is still a costly operation when there are conflicts. To amortize the cost of pointer access and improve ILP, we put multiple nodes into each entry. Figure 4 shows an example where 4 nodes are packed into one entry. All nodes are initialized to the NULL node. When inserting a node, it is replacing the first NULL node. If the entry is full, a new entry is created and chained. As each entry holds more nodes, we reduce the number of slots in the hash table to introduce conflicts intentionally, so that entries can be better utilized. Initially, we pack 4 nodes per entry, and later we change to 8 since it is more performant.

This optimization further enlarges the pre-allocating size, but generally it is not a problem given a reasonable pre-allocated node count that is close to or less than the average degree of the input graph. Furthermore, since packed entries save several pointer fields in long linked lists, the memory footprint can be even lower for dense graphs.

### 4.3. Vectorized Lookup

With packed entries, we further optimize the hash table by applying vectorization. Only the lookup operation turns out to be effective with vectorization, while the other operations are not (we will discuss them in Section 7.2. For brevity, we discuss vectorization on 4-node entries using SSE. 8-node entries can be vectorized in a similar way using AVX.

In the original version, nodes in each entry are compared against the target node with a loop. In the vectorized version, the target node is propagated to a `__m128i` using `_mm_set1_epi32`. 4 nodes in the entry is loaded with `_mm_loadu_si128`. Unaligned load is used since the 4 nodes do not align to 16 bytes (or there need to be additional paddings). Two vectors are compared for equality and the result is extracted with `_mm_movemask_epi8`. If the resulting integer is not zero, there is a match. We do not care about NULL nodes since the target node to look up is never a NULL node.

## 5. EXPERIMENTAL RESULTS

In this section, evaluation results are presented. As mentioned in Sec. 2.2, we define operations as integer comparisons. Since the optimizations frequently change operation counts, we mainly present the speedup plots.

### 5.1. Experimental Setup

We used an Intel core i9-9900K, Coffee Lake processor. The base frequency is 3.60 GHz. The actual memory bandwidth is 22 GB/s as measured using Intel Advisor. It has a 32 KB, 8-way set associative, per-core L1 cache, a 256 KB, 8-way set associative, per-core L2 cache, and a 16 MB, 16-way set associative, shared L3 cache. The actual data movement in the roofline plots is measured as the data movement between L3 cache and memory using Intel Advisor.

We compiled the code with the GCC compiler version 11 with `-O3 -march=native` flags. We instrumented our code to count the number of comparison operations (for performance measurements). When we include sorting in the runtime, we create copies of the graph for each warmup and actual run in advance, since sorting changes the graph. Our program then returns the average runtime of all runs for each phase. We benchmark our optimisations on two kinds of randomly generated graphs and large real-world graphs.

**Sparse Graphs.** Graphs with constant average degree equal to 32 and number of edges ranging from 600 to 21000 edges. For any scenario where we only evaluate a single sparse graph, we use the graph defined for 20000 edges, as for this size saturation has already been reached.

**Dense Graphs.** Graphs with average degree proportional to the number of nodes (10%) and number of nodes ranging from 500 to 6000 nodes. For any scenario where we only evaluate a single dense graph, we use the graph defined for 6000 nodes, as for this size saturation has already been reached.

**Real World Graphs (RWGs).** These consists of a graph modeling the German road network [10], which consists of roughly $10^7$ nodes and $10^7$ edges (exact numbers reported in the Appendix), a citation graph for papers in hepth [11] ($10^4$ nodes, $10^5$ edges) and four additional graphs from Stanford's SNAP research group [12], which are all fairly sparse. However, some have very high maximum to average degree ratios, such as the WWW Notre Dame graph ($\frac{d_{max}}{d_{avg}} \approx 1600$) that links between web pages within the nd.edu domain, which results in very unbalanced set intersections. Node and edge counts can be found in table 4.

### 5.2. Preprocessing

We evaluate the speedup gained through preprocessing (as described in Sec. 3.1) the input graphs for the different algorithms. The results are presented in Table 1, note that we do not perform sorting for the Forward Hashed algorithm. We observe significant speedup for Edge Iterator for both sorting and cutting, but not forward because the creation of the dynamic structure already transforms the graph into a directed version. Similarly Forward Hashed also does not benefit. In the following evaluations, we exclude preprocessing and only measure the main algorithms.

| | Edge Iterator | | | Forward | | | Forward Hashed |
|---|---|---|---|---|---|---|---|
| | S | C | C+S | S | C | C+S | C |
| **Dense** | 1.19 | 2.69 | 3.40 | 1.17 | 1.17 | 1.22 | 0.99 |
| **Sparse** | 1.20 | 2.56 | 3.16 | 1.19 | 1.21 | 1.33 | 0.96 |
| **WWW ND** | 1.26 | 1.14 | 1.68 | 1.17 | 1.17 | 1.26 | 0.97 |
| **US Patents** | 1.14 | 2.14 | 2.32 | 1.11 | 1.11 | 1.11 | 0.99 |
| **GRN** | 1.00 | 1.26 | 1.24 | 1.02 | 0.95 | 0.94 | 0.89 |
| **CS Authors** | 1.17 | 1.81 | 2.10 | 1.19 | 1.07 | 1.19 | 0.96 |
| **Google Contest** | 1.17 | 1.37 | 3.70 | 1.15 | 1.14 | 1.22 | 0.96 |
| **HEP Literature** | 1.15 | 1.88 | 2.42 | 1.14 | 1.17 | 1.24 | 0.98 |

**Table 1**. Speedup gained by optimizing preprocessing (S:sorting, C:cutting, C+S:cutting+sorting combined) compared with the baseline version.

### 5.3. Edge Iterator and Forward

Since Edge Iterator and Forward have similar runtime and performance patterns, we show plots for Forward in this section and attach plots for Edge Iterator in the appendix.

As is shown in figure 5, interleaving 4 set intersections (Sec. 3.5) gives a speedup of up to $1.75\times$ on dense graphs by utilizing ILP. Vectorizing the interleaved set intersections achieves a more than $2.75\times$ speedup (Sec. 3.6) compared with the baseline. Vectorized interleaved intersections does not give another $4\times$ speedup because of the overhead to pack and unpack the vectors for index computations and node id comparisons. The vectorized block intersection only achieves a speedup of around $1.8\times$ because it uses

many shuffle and permute operations that depend on each other. Note that the exponential search for unbalanced set intersections is not utilized on generated sparse/dense graph and thus does not contribute to the observed speedup of vectorized block intersection. However, it gains significant improvement for unbalanced real-world graphs, as can be seen in Sec. 5.5.

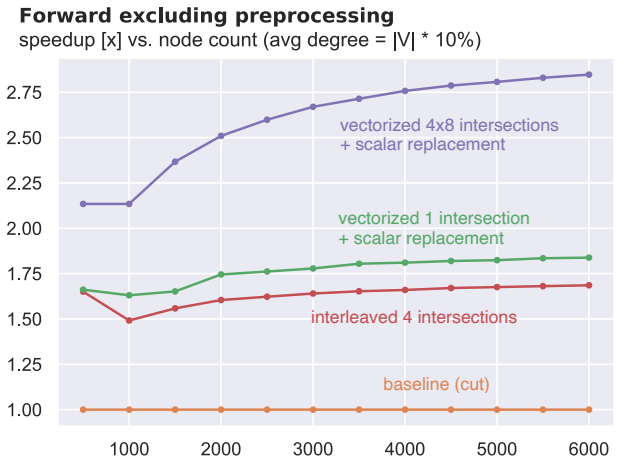


**Fig. 5**. Speedup of optimized Forward versions against the baseline version on dense graphs.

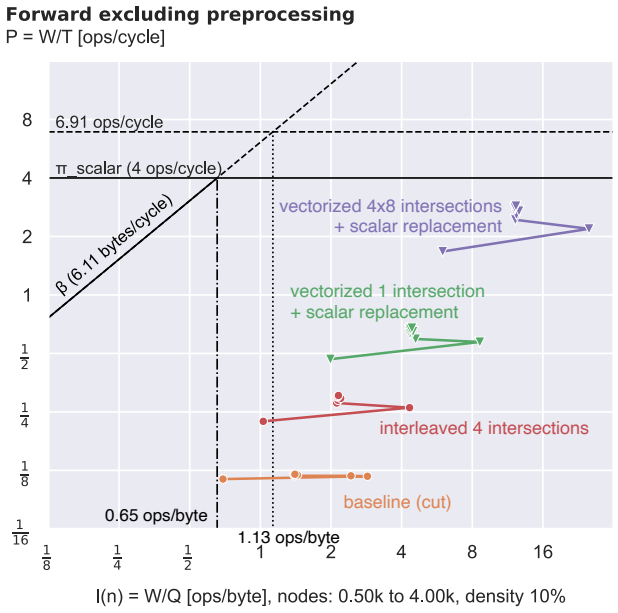


**Fig. 6**. Roofline plot of Forward on dense graphs.

The roofline plot of Forward (Fig. 6) shows that both the scalar and the vectorized versions are compute bound even when we only consider integer comparison operations. The performance of the scalar version is only $6\%$ of the peak

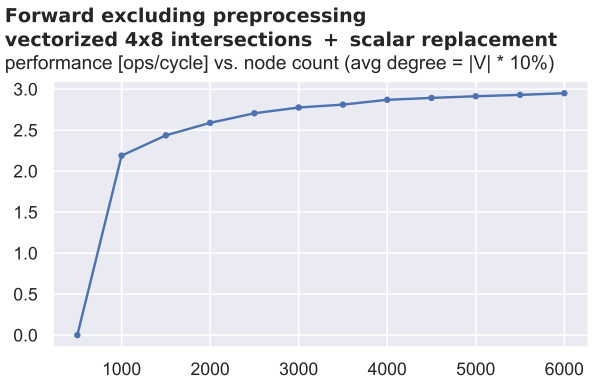


**Fig. 7**. Performance of the best version of Forward on dense graphs.

performance because we do not count index operations and listing triangles to an array is blocking. The performance of our best performing version of Forward plateaus at 3 comparisons/cycle as is shown in figure 7. It reaches $45\%$ of the theoretical peak performance based on instruction mix. We do not see the effect of the memory hierarchy here because our optimized versions are still compute bound.

| Comparison ops/Cycle | Collecting | Counting |
|---|---|---|
| Theoretical Peak | 6.91 | 7.91 |
| Measured | 3.14 | 6.77 |

**Table 2**. Theoretical peak performance based on instruction mix and measured performance of the innermost loop.

**Listing vs. Counting.** Table 2 shows the performance of the best-performing Forward with the vectorized interleaved set intersections, measuring the innermost loop. The peak performance when doing triangle listing is lower than the theoretical peak based on the instruction mix compared to triangle counting. As the only difference in the code of the two is the counting vs. collecting, it shows that we lose a lot of performance in the collecting part. Listing can not be vectorized nor can achieve high ILP since triangles are added sequentially to a list (and in each iteration we do not know how much and which elements to add). In contrast, the measured performance for counting is very close to the theoretical peak, since all operations can be vectorized.

### 5.4. Forward Hashed

Fig. 8 shows the speedup of Forward Hashed with optimizations applied one after another, evaluated on dense graphs. All optimizations, removing indirection (Sec. 4.1), multiple nodes per entry (Sec. 4.2), and vectorization (Sec. 4.3), give significant performance boost compared to the baseline. The best-performing version achieves more than $14\times$

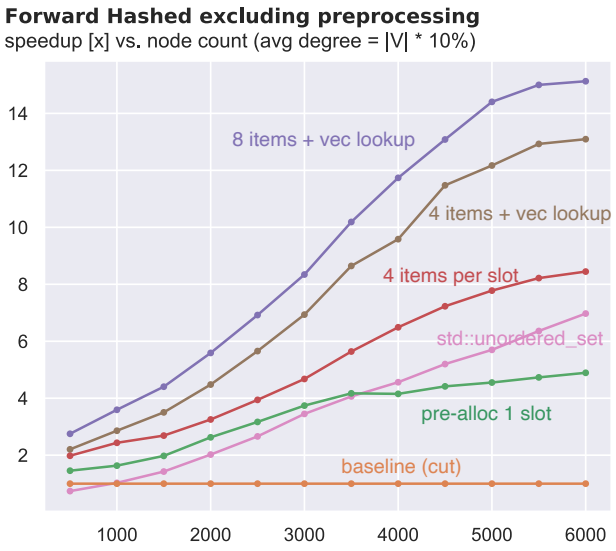


**Fig. 8**. Speedup of optimized Forward Hashed on dense graphs compared with the baseline version.

speedup on the 6000-node graph. The speedup increases as the graph becomes larger. Other dense graphs with different degree fraction show the same trend.

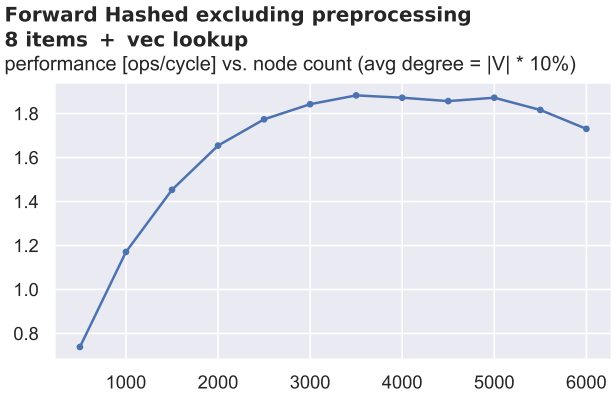


**Fig. 9**. Performance of the best-performing Forward Hashed (8 items per slot + vectorized lookup) on dense graphs.

However, although the speedup is significant, the performance of Forward Hashed is still far from the theoretical peak. Fig. 9 shows the performance of the best-performing version with different dense graphs. Here we only count integer comparison operations, while Forward Hashed has a lot of memory accesses including a large portion of random ones. The performance decreases as the number of node increases, since the average neighbor count increases and there are more conflicts. However, our best-performing version still outperforms `std::unordered_set` by more than $2\times$ on dense graphs, $4.5\times$ on sparse graphs.

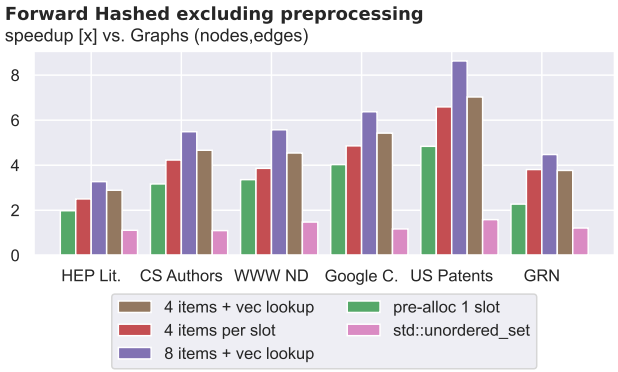


**Fig. 10**. Speedup of optimized Forward Hashed versions on real-world graphs compared with the baseline version.

The optimizations show a similar result on sparse graphs, except that the speedup is scaled by about $1/3$. The best-performing version achieves a speedup of about $3.5\times$ on the 6000-node graph. Fig. 10 shows the speedups of Forward Hashed on the real graphs. The best-performing version achieves about 3 to $8\times$ speedups, which is more like the sparse graphs as real-world graphs are mostly sparse.

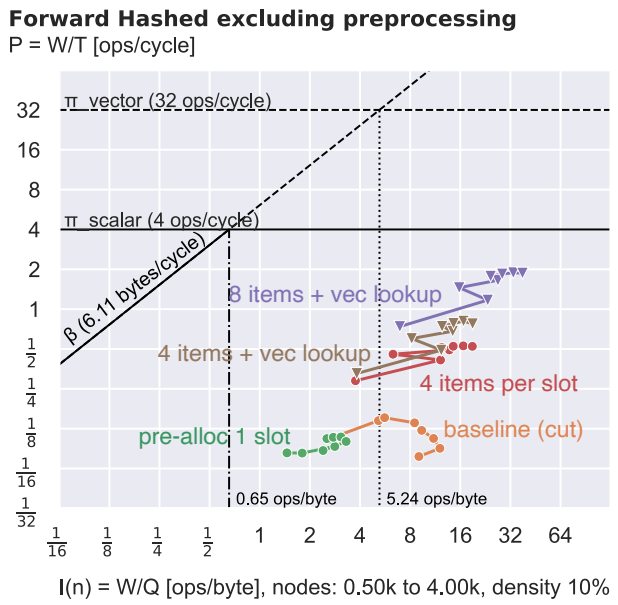


**Fig. 11**. Roofline plot of Forward Hashed on dense graphs, excluding preprocessing. $\pi$'s show theoretical upper bounds while the achievable performance is much lower due to complex branching and dependencies.

Fig. 11 shows the roofline models of Forward Hashed on dense graphs. The performance increases with the optimizations, but overall they are yet compute bound.

### 5.5. End-to-End Evaluation

We now show the end-to-end runtime and speedup we were able to achieve in Table 3. The measurements include cutting and sorting for Edge Iterator and Forward, and no preprocessing (as this gives best results) for Forward Hashed. Generally, we achieve greater speedups for dense graphs in comparison to sparse graphs, due to our focus on list intersections. We only gain marginal performance improvements on most real-world graphs due to their very small average degree (we only show GRN for brevity, while the others are similar). WWW ND is the exception, for which VBSI can effectively leverage exponential search for its large degree variance. For the other real-world graphs. VISI generally performs better than VBSI.

| | Edge Iterator | | | Forward | | | Forward Hashed | | |
|---|---|---|---|---|---|---|---|---|---|
| | base | VBSI | VISI | base | VBSI | VISI | base | std | best |
| **Dense** | $40.5 \cdot 10^9$ | 5.6 | 8.49 | $13.4 \cdot 10^9$ | 2.14 | 3.24 | $469 \cdot 10^9$ | 6.7 | 14.46 |
| **Sparse** | $26.8 \cdot 10^6$ | 4.25 | 6.31 | $10.2 \cdot 10^6$ | 1.95 | 2.14 | $19.5 \cdot 10^6$ | 0.88 | 3.87 |
| **WWW ND** | $1.66 \cdot 10^9$ | 2.82 | 1.36 | $0.287 \cdot 10^9$ | 1.13 | 1.16 | $1.23 \cdot 10^9$ | 1.17 | 4.87 |
| **GRN** | $2.46 \cdot 10^9$ | 1.06 | 1.13 | $2.46 \cdot 10^9$ | 1.02 | 1.02 | $7.75 \cdot 10^9$ | 1.18 | 4.56 |

**Table 3**. Overall runtime [cycles] of the base version and speedup [$\times$] of the best vectorized versions (VBSI: Vectorized Block Set Intersection, VISI: Vectorized Interleaved Set Intersection, best: best-performing Forward Hashed, 8 Nodes per entry vectorized) compared with the baseline versions, including preprocessing.

## 6. CONCLUSIONS

We performed a series of optimizations and achieved significant speedups for all three algorithms. Edge Iterator showed more than $8\times$ speedups for dense graphs. Forward, although showing the least speedups with respect to baselines, was actually one of the fastest algorithms on all graphs. Forward Hashed showed the most significant speedups (up to $15\times$) with effective optimizations on memory hierarchy and SIMD, but it was yet slower than the other two due to frequent memory access. The comparison of counting vs. listing showed that we were reaching the performance limit for triangle listing given the instruction dependencies. We observed some variance in the performance improvements achieved by the different optimization techniques across the various graphs, so automatic algorithm selection and parameter tuning (such as the hash table size of Forward Hashed) based on graph properties and the machine is possible, which can be explored in future work.

## 7. FURTHER COMMENTS

We achieve significant speedups with various optimizations. However, there are also many more attempts that are not effective. We discuss a few representative ones in this section.

### 7.1. Failed Attempts for Cache Optimization

We explored cache optimization opportunities for the adjacency list representation.

**Blocking for cache.** We tried cache blocking to improve temporal locality for adjacency list accesses. There are two ways of blocking: 1. Read in $K$ adjacency lists at once and perform set intersections on that block. It did not improve the performance or runtime as we need to check whether one node is in the adjacency list of another node, which results in a higher asymptotic runtime. 2. Process the adjacency lists in groups of $K$ items. For one set intersection, every group of $K$ items needs to be intersected with all the other groups. It did not improve the performance or runtime as the intersection now has a higher runtime complexity $O((\frac{n}{K})^2 \cdot K)$ and more boundary checking.

The random access pattern to the nodes and their adjacency lists and the non-uniform adjacency list lengths require a lot of additional branching, so blocking for cache introduces more overhead than speedup.

### 7.2. Failed Attempts for Forward Hashed

Most effective optimizations for Forward Hashed are about the memory layout. We present a few failed attempts here.

**Linked list for existing items.** This attempt constructs a linked list on top of the hash table that chains existing items, which is then used to iterate through the table. To construct the linked list, each node holds an additional pointer. It does not improve the performance compared with the current implementation—iterating through all slots. We think the main reason is that the additional pointers reduce the cache efficiency more than the gain from faster iteration. This optimization may be effective for sparse hash tables, but in that case, our optimizations of node packing and vectorization are no longer effective.

**Compute the next empty node with SIMD.** Instead of finding the first NULL node sequentially, we tried to compute its index using SIMD. The vector of the entry is compared with a vector of NULL nodes. The masks are gathered into an integer and the index of the first NULL node is computed by counting the leading zeros with the GNU build-in `__builtin_clz`. Unlike the lookup operation, this approach turns out to be slower as additional arithmetics for index and address computation make it slower than sequential comparison and insertion.

**Lookup multiple target nodes at the same time.** In our best-performing versions, the algorithm looks up one target node among multiple nodes in the hash entry at once. Looking up multiple target nodes does not work since pairwise comparisons are still needed since the nodes are unordered. Furthermore, packing multiple target nodes into a vector goes against the triangle listing. Those nodes need to be unpacked in order to exactly list the found triangles.

## 8. CONTRIBUTIONS OF TEAM MEMBERS

**Nils Blach.** Implemented unbalanced set intersections (Sec. 3.2), block set intersections (Sec. 3.3) and vectorized block set intersections (Sec. 3.4). Performed real-world graph analysis.
**Matthias Bungeroth.** Implemented sorting optimisations (Sec. 3.1), interleaved set intersections including vectorization for forward (Sec. 3.5 and Sec. 3.6), as well as scalar replacement (forward_{v1,v2,v3,v4,v5,v6}.hpp). Analysis of this shown in table 2. Further, scalar replacement for "vectorized 1 intersection" (egde_iterator_v4.hpp based on egde_iterator_v3.hpp (provided by Nils)) and then porting the implementation to forward. Performed peak performance analysis and data movement analysis using Intel Advisor for all algorithms.
**Zikai Liu.** Implemented the baseline version of Forward. Optimized everthing about Forward Hashed. Some worked (Sec. 4), and more failed (Sec. 7.2).
**Jingyi Zhu.** Implemented shortening adjacency lists (Sec. 3.1). Analyzed compiler behavior on scalar replacement and experimented with compiler flags (Sec. 3.7). Analyzed access patterns and experimented with cache blocking (Sec. 7.1). Ported vectorized interleaved intersections (provided by Matthias) to Edge Iterator. Worked on roofline plots with Matthias.

## 9. REFERENCES


[1] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2008, KDD '08, p. 16–24, Association for Computing Machinery.

[2] Howard Welser, Eric Gleave, D. Fisher, and Marc Smith, "Visualizing the signatures of social roles in online discussion groups," *Journal of Social Structure*, vol. 8, pp. 1–31, 01 2007.

[3] Thomas Schank and Dorothea Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, Berlin, Heidelberg, 2005, WEA'05, p. 606–609, Springer-Verlag.

[4] Colin Goldberg, "triangle-counting," https://github.com/colingo1/triangle-counting, 2019.

[5] Alexander Hetzer, "Lemming," https://github.com/BlackHawkLex/Lemming, 2021.

[6] Data Science Group at UPB, "Lemming," https://github.com/dice-group/Lemming, 2021.

[7] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura, "Faster set intersection with simd instructions by reducing branch mispredictions," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 293–304, nov 2014.

[8] Jon Louis Bentley and Andrew Chi-Chih Yao, "An Almost Optimal Algorithm for Unbounded Searching," *Inform. Proc. Lett.*, vol. 5, pp. 82–87, 1976.

[9] Berenger Bramas, "A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake," 2017, International Journal of Advanced Computer Science and Applications.

[10] Ryan A. Rossi and Nesreen K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[11] "Citation network," https://www.cs.cornell.edu/projects/kddcup/datasets.html, Accessed: 2022-03-30.

[12] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

## A. INFORMATION ABOUT THE REAL-WORLD GRAPHS

Table 4 shows the statistics of the real-world graphs used in the evaluation. We use the same graphs as [3]. Some of them are the same, while the other only have updated versions available. Please refer to [3] for details of these graphs.

| Name | Node Count | Edge Count |
|---|---|---|
| HEP Literature | 27770 | 352285 |
| CS Authors | 317080 | 1049866 |
| WWW ND | 325729 | 1090108 |
| Google Contest | 875713 | 4322051 |
| US Patents | 3774768 | 16518947 |
| GRN | 11548845 | 1236918 |

**Table 4**. Real World Graphs node and edge counts.

## B. MORE PLOTS

In this appendix section, we present additional plots of the algorithms for completeness.

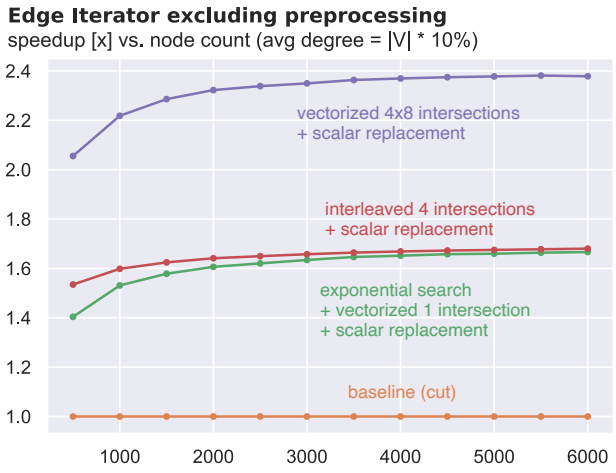


**Fig. 12**. Speedup of optimized Edge Iterator versions against the baseline version on dense graphs.

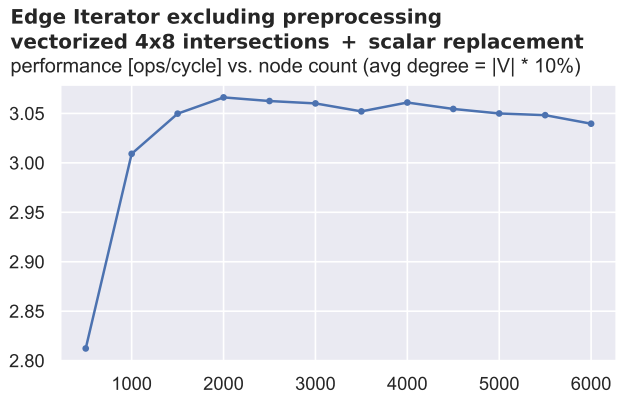


**Fig. 13**. Performance of the best version of Edge Iterator on dense graphs.

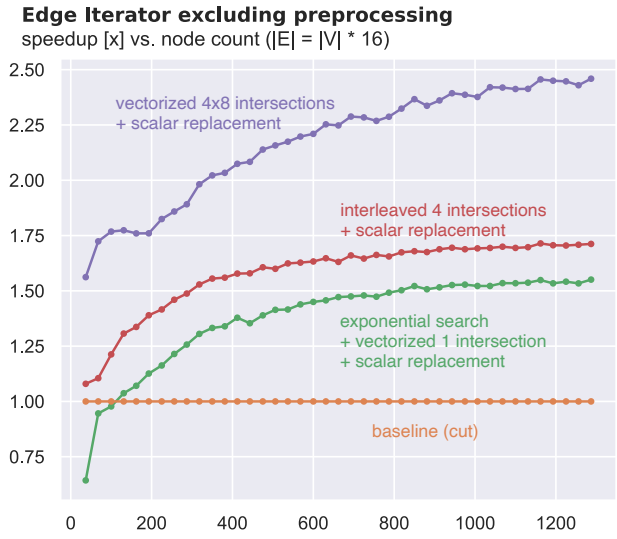


**Fig. 14**. Speedup of optimized Edge Iterator versions against the baseline version on sparse graphs. The graph becomes sparser as the node count increases.

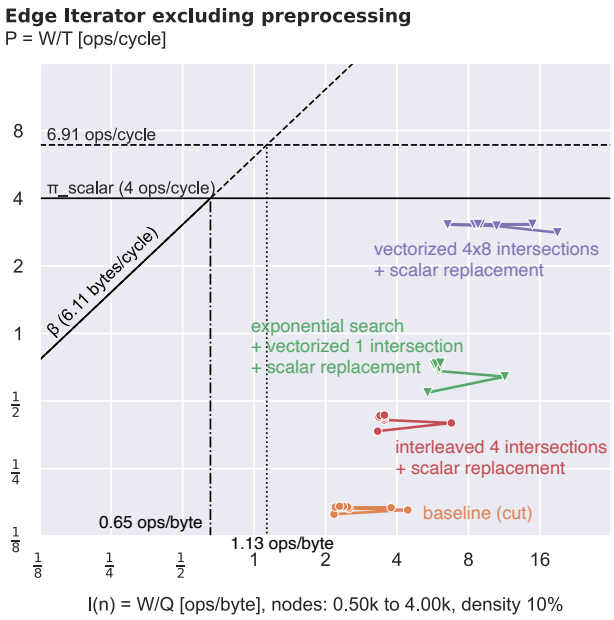


**Fig. 15**. Roofline plot of Edge Iterator on dense graphs.

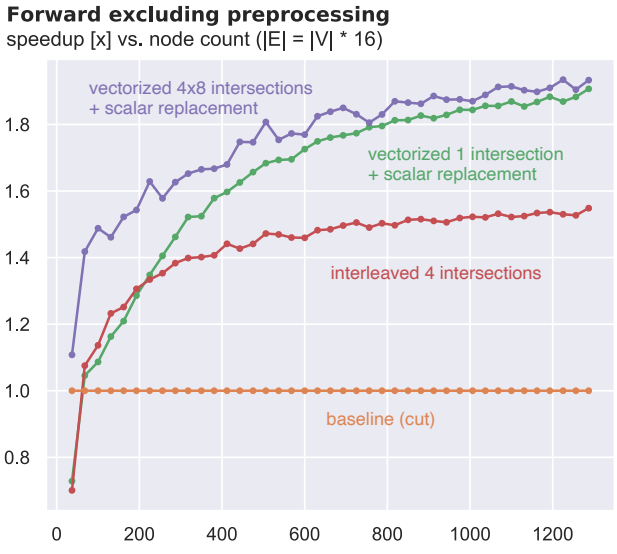


**Fig. 17**. Speedup of optimized Forward versions against the baseline version on sparse graphs. The graph becomes sparser as the node count increases.

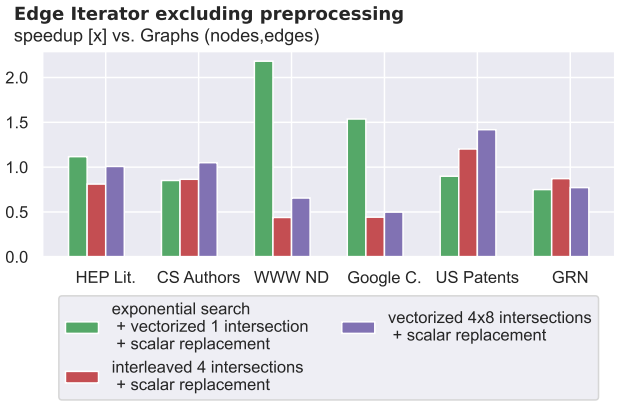


**Fig. 16**. Speedup of optimized Edge Iterator versions on real-world graphs compared with the baseline version.

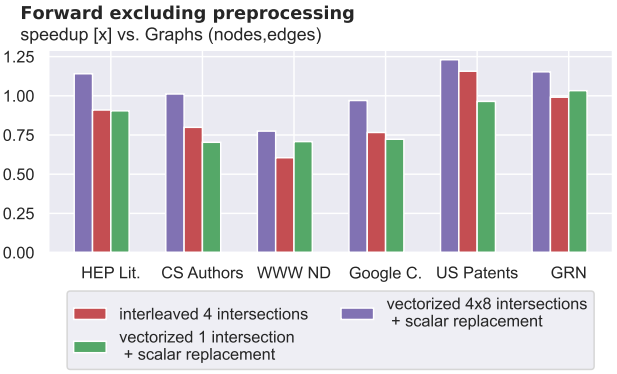


**Fig. 18**. Speedup of optimized Forward versions on real-world graphs compared with the baseline version.