# APPROXIMATE CHOLESKY DECOMPOSITION WITH OPTIMIZED DATA STRUCTURE, SIMD INTRINSICS, AND FAST BINARY SEARCH

*Daoye Wang*, Fanlin Wang*, Dexin Yang*, Kaifeng Zhao**

Department of Computer Science
ETH Zurich, Switzerland

**ABSTRACT**

A novel preconditioner of Laplacian matrix has recently been proposed and the algorithm of building the preconditioner, Approximate Cholesky Decomposition [1], runs in nearly linear time. In this report, we optimize the implementation of the algorithm from an engineering perspective. We first analyze the spatial locality of original implementation and profile its run-time to identify performance bottlenecks. Based on our analysis and profiling, we propose three improvement: an optimized data structure, using SIMD intrinsics, and a fast binary search. We carry out various experiments to validate our analysis. In the end, we manage to achieve a median speed-up of $19.14\times$ on the performance. Our implementation also significantly reduces the run-time comparing to the original implementation in Julia, up to $8.49\times$.

## 1. INTRODUCTION

Solving large linear systems in weakly diagonally dominant matrices is a key step in many areas, including the finite element method and semi-supervised learning on graph. Constructing a good preconditioner is essential for iterative solvers of such linear systems. With the advent of randomized linear algebra, Spielman & Teng [2] introduce a novel preconditioner. The time complexity of their algorithm is nearly linear, which is a significant improvement over previous methods with polynomial time complexity. Given a graph of $n$ vertices and $m$ edges, the time complexity of Spielman & Teng's algorithm is only $\mathcal{O}(m \log^c(n))$.

While Spielman & Teng's method is a breakthrough in the field, they have not optimized for $c$, the power of $\log(n)$. In particular, $c = 39$ in their original paper [2]. Based on Spielman & Teng's work, Kyng & Sachdeva [1] proposed an algorithm of time complexity $\mathcal{O}(m \log^3(n))$. Other than being fast, Kyng & Sachdeva's algorithm is surprisingly simple, which makes it of high practical value and theoretical elegance. We refer to the original implementation in Julia [3].

* In alphebetical order of the authors' last name

While Kyng & Sachdeva's method is algorithmically simple and fast, from an engineering perspective, there is still room for improvement. Due to the random nature of the algorithm, the implementation in Julia has a bad spatial locality. Iteration over the entire matrix is needed when one only tries to access a column. Another performance bottleneck is the binary search which happens in the inner loop of the algorithms and is repeatedly carried out. Last but not least, computation flow can be further optimized with vectorization and control flow can be optimized by eliminating `if-else` statement.

In this report, we describe how we tackle these issues. We propose to use efficient data structures to achieve better spatial locality. We use SIMD intrinsics in C++ to accelerate the computation. We also implement a fast binary search method [4], which further simplifies the control flow and makes vectorized binary search possible.

The contribution of this report is three-fold:

(1) We translate the algorithm from Julia into C++ and build a code base both for validating the correctness of implementations and for measuring the performance.

(2) We optimize the implementations of the algorithms from an engineering perspective and gain an average speed-up of roughly 20 times in CPU cycles.

(3) We profile the run-time and analyze the performance gain of various optimization approaches.

## 2. ALGORITHM OVERVIEW

In this section, we first introduce the background of the algorithm. Next, a pseudo-code of the algorithm is presented. We then talk about the validation. Finally, we introduce the cost measure used in this report.

### 2.1. Background

A matrix $A$ is symmetric if $A_{i,j} = A_{j,i}$. A matrix $A$ is weakly diagonally dominant if $A_{i,i} > \sum_{j \neq i} |A_{i,j}|$. A Laplacian matrix is a symmetric weakly diagonally dominant matrix such that $A_{i,j} \leq 0, \forall j \neq i$ and $A_{i,i} = \sum_{j \neq i} |A_{i,j}|$. A Laplacian matrix is naturally associated with a graph. Given a weighted graph $G$, denote its adjacency matrix as $A$, where $A_{i,j} \neq 0$ means there exists an edge between vertex $i$ and vertex $j$, with weight $A_{i,j}$. Denote $D$ as the diagonal matrix with $D_{i,i}$ equal the degree of vertex $i$. Let $L = A - D$. $L$ can be shown to be a Laplacian matrix as defined above.

Given a Laplacian matrix $L$, our goal is to find a matrix decomposition

$$L \approx M = \mathcal{L} D \mathcal{L}^T$$

where $\mathcal{L}$ is a lower-triangular matrix with diagonal element 1 and $D$ is a diagonal matrix. If $M$ approximate $L$ well, it naturally leads to a solver of the linear system $Lx = b$, i.e. $x = M^\dagger b$, where $M^\dagger$ is the Moore–Penrose pseudo-inverse of matrix $M$. In particular,

$$M^\dagger = \Pi_M (\mathcal{L}^T)^{-1} D^\dagger \mathcal{L}^{-1} \Pi_{M^T}$$

where $\Pi_M$ is the projection matrix to the column space of matrix $M$. Notice that for a triangular matrix, matrix inverse can be easily calculated via forward and backward substitution. For a Laplacian matrix $M$, $\Pi_M = \Pi_{M^T}$ is exactly the operator which subtracts the mean of elements in a vector.

After constructing the preconditioner $M$, iterative methods like Preconditioned Iterative Refinement (PIR) or Preconditioned Conjugate Gradient (PCG) will give a solution of the linear system $Lx = b$.

**Algorithm 1:** Approx. Cholesky Decomp.

```
1  Let $A$ be the given sparse Laplacian matrix with n columns
2  Let ldli be the output
3  for $i = 1 \cdots n$ do
4      // Iterate over columns
5      col ← get, compress, and sort column $i$
6      cum ← compute partial sum of col
7      wdeg ← last term of cum
8      colScale ← 1
9      for $j = 1 \cdots size(col)$ do
10         // Iterate over elements of a column
11         k ← random sample from $\{j, .., m_{col}\}$ according to edge weights
12         w ← $A_{i,j} *$ colScale
13         f ← w/wdeg
14         Save f into ldli at $(i, j)$
15         create new edge in A at $(j, k)$ with weight $w * (1 - f)$
16         colScale ← colScale * (1 - f)
17         wdeg ← wdeg - 2w - $w^2$/wdeg
18     end
19     save w into ldli at diagonal
20 end
```

## 2.2. Algorithm overview

Here is a pseudo-code of the approximate cholesky decomposition algorithm. For more details, we refer readers to [1].

We roughly divide the algorithm into three parts. First, get a column from the matrix and process it (*Alg. 1, Line 5*). Second, randomly sample $k$ from $\{j, .., \text{size(col)}\}$ for each element $j$ in the column and compute weights of the new edge $(j, k)$ (*Alg. 1, Line 6-13,16,17*). Third, add new edges $(j, k)$ with computed weights into the current Laplacian matrix (*Alg. 1, Line 15*).

In this report, we will first discuss how to optimize the first and third part of the algorithm by changing the underlying data structure. The spatial locality will be improved in the optimized data structure; in particular, procedures in the first part will be significantly simplified. Then we will talk about how SIMD intrinsics help accelerate the second part of the algorithm. Last but not least, a faster binary search will be implemented to improve the bottleneck in the second part.

Notice that there are several versions of the algorithm in [3]. We pick the fixed-order version. Optimization of other versions is left for future work.

## 2.3. Validation

We validate our implementation of the algorithm by computing the eigenvalues of $LM^\dagger$. We compute the ratio of the largest eigenvalue and the second smallest eigenvalue. The closer to 1 the ratio is, the better the approximation is. The eigenvalues are computed via the power method and the inverse power method. $M^\dagger$ is treated as an operator to facilitate implementation. Note that when applying the operator $M^\dagger$, we use PCG solver instead of the straightforward solver consisting of forward and backward substitution. This deviates from the original approach (*Line 967-1027* in `approxChol.jl` [3]).

## 2.4. Cost Analysis

As shown in Alg. 1, we found that integer operations account for a large part (column compression, sorting, etc). Therefore, we decide to count not only floating point operations but also integer operations.

Specifically, we use the following cost measure,

$$\mathcal{C} = \mathcal{C}(\text{int ops}) + \mathcal{C}(\text{float ops}).$$

Since random numbers and sorting operations are introduced in the algorithm, it is difficult to determine the exact number of operations it has before it runs. So we implement a manual counter in the code for both integer and floating point operations. Besides, we also use two command-line performance tools, namely `perf` and `valgrind`, for validation. These two performance tools also give more detailed results for data movement and cache misses.

The time complexity of Alg. 1 is $\mathcal{O}(m \log^3(n))$, as stated in Sec.1.

### 2.5. Profiling

We implement the algorithm in C++ based on [3] and this serves as our baseline for performance comparison. We use `perf` to detect the potential bottleneck of the algorithm. Fig. 1 shows the percentage of CPU time (in cycles) spent on different parts of the algorithm in our baseline implementation. We can see that part 1 and part 3 of the algorithm as illustrated in section 2.2 occupies a large part of run-time. Note that Fig. 1 only roughly divides the algorithm into different parts, with each part including the time for data movement. A further investigation into the assembly code suggests that the CPU time spent on data movement is significant. Fig. 1 also shows that as the problem size grows, the bottleneck of the algorithm will change as well. For example, for a larger graph, it spends 20% more time on adding new edges than it does on a small graph. The profiling guides our later efforts to optimize the implementation.

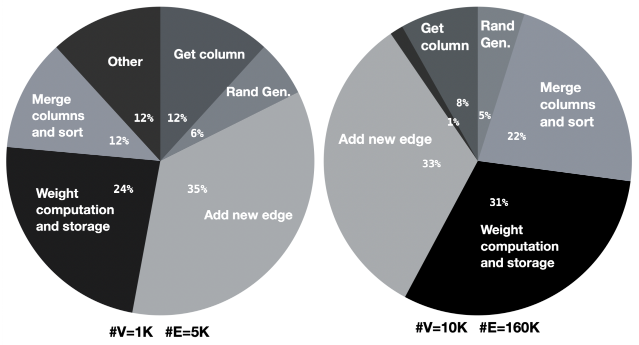


**Fig. 1**: CPU time breakdowns.

## 3. PROPOSED METHOD

We first investigate Alg. 1 to see possible optimizations. The outer loop iterates over columns and applies column-wise elimination. Unfortunately, operations on one column may change all subsequent columns, which leads to dependencies between columns and makes it impossible to unroll the outer loop to achieve column level parallelism. Therefore, the majority of proposed optimizations are limited to be within a column.

The decomposition can be basically divided into three parts for each iteration: (1)merge duplicate edges and sort them; (2) generate new edges from random sampling; (3) insert new edges. From cost analysis, we can see that most time is spent on accessing and storing edges. The profiling result prioritizes proposing an efficient data structure of the sparse matrix, which should support dynamic insertion. Another significant part of run-time is spent on random sampling, more specifically, the binary search. The rest consists of computation of weights and values, most of which are executed sequentially and can be easily vectorized.

### 3.1. Data structure

The nature of approximate Cholesky decomposition requires a sparse matrix supporting dynamic edge insertion. We try two data structures: one based on linked list and the other one based on array. Additionally, We experiment whether to store each edge as a whole in memory.

**Linked list with prefetch.** This approach follows strictly the original implementation in [3]. Initially we construct the sparse matrix as a modified version of compressed sparse column representation like in figure 2. A `next` attribute is added to enable insertion like a linked list.

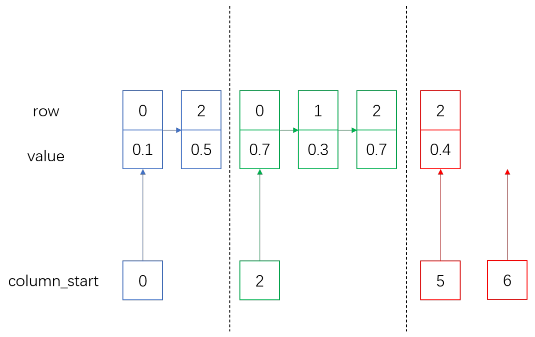


**Fig. 2**: Illustration of the initial representation of a Laplacian matrix with 3 columns. Each edge consists of row number, edge value and a next pointer. Each column has a head pointer. In the graph, columns are color-coded.

Note that each edge in a column will generate exactly one new edge, which means a sparse matrix initially with $m$ edges always has $m$ edges (counting duplicate edges) throughout the decomposition. When inserting new edges, we store the new edges in place and modify the pointers, as shown in figure 3. The advantage of this method is its memory-efficiency.

However, data access from linked list suffers from the bad spatial locality and we alleviate this problem by prefetching. Before processing one column, we get all the elements of this column out from the linked list and store them consecutively. This improves performance and makes further optimization like SIMD possible.

**Array with dynamic memory management (V1Mg).** The performance of linked list is still not satisfactory even with prefetch. We propose an array-based data structure, which gives a better performance at the cost of higher memory usage. We carefully deal with the trade-off between performance and memory. Our core idea is to represent each column as a consecutive array of edges and the sparse matrix

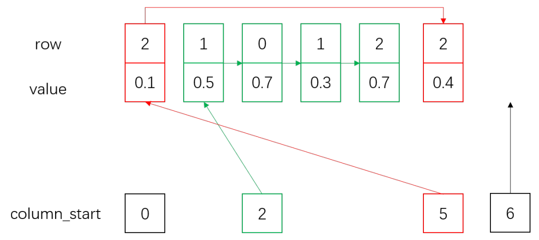


**Fig. 3**: Illustration of edge insertion after processing first column. Edges of first column will never be used again and generated new edges are stored inplace. Note the head pointers of subsequent columns change accordingly.

as an array of columns. One problem of this naive method is being unacceptably memory demanding. The theoretical upper bound of the total number of edges generated during the decomposition is $\mathcal{O}(mn)$, given a sparse Laplacian matrix of a graph with $m$ vertices and $n$ edges. However, a good attribute of the algorithm is that all the elements will be used only once; therefore, we can free the memory after finishing processing a column to achieve reasonable memory usage. We rely on `STL` vector as infrastructure for dynamic memory management. Initially we construct the sparse Laplacian matrix as a vector of vectors, which represents columns with pre-allocated memory with respect to the sparsity of the Laplacian. At the end of the decomposition of each column, we simply push new edges to the end of the corresponding vectors. We also free the memory of the current column's edges. The memory recycling leads to some performance improvement in practice.

Implicit merge (V1) It is worth mentioning that we also tried another method requiring no prefetch and explicit merge. Edges of a column will be kept sorted and unique after dynamic insertion by using a map or a sorted vector. However, experiments show that the performance is inferior.

**Edge as a whole (V2Mg).** Each edge has two attributes in a compressed sparse column representation, row and value. From the perspective of computation optimization like SIMD vectorization, it is normally better to store row and value separately which gives a struct of arrays. However, sorting and merging edges is a dominant part in our context, where edges need to be processed as a whole. This makes array of structs advantageous. After thinking carefully about the trade-off, we choose array of structs for better global performance.

Here we show some profiling results to illustrate how the proposed methods improve locality and performance. Table. 1 and Table. 2 summarize the D1 cache miss rate (D1mr), D1 cache miss write (D1mw), data reads (Dr), data writes (Dw), LL miss rate (LLmr), conditional branch executed (Bc), and branch miss rate (Bmr) for a middle-sized and a large-sized graph respectively. We refer to the linked list variant as baseline, the array with dynamic memory management variant as V1Mg, the variant of a sorted array with implicit merge as V1, variant of V1Mg plus locality optimization of storing edge as a whole as V2Mg.

We can see that storing the row and value of each edge consecutively in memory optimizes the locality and contributes to considerable performance improvement, particularly when we have big Laplacian matrices. Note that the LL miss rate and branch miss rate are not always a good indicator of memory performance. For example, in Table. 1, the total data access for baseline is so large that the rate is not suggestive.

**Table 1**: $V = 10\mathrm{k}, E = 50\mathrm{k}$.

| Function | D1mr | D1mw | Dr | Dw | Bc | LLmr | Bmr | Run-time[s] |
|---|---|---|---|---|---|---|---|---|
| Baseline | 1.26E+08 | 1.26E+08 | 5.11E+08 | 5.07E+08 | 1.01E+09 | **0.001** | **0.097** | 1.2676 |
| V1 | 9.68E+05 | 1.37E+05 | 1.72E+07 | 8.32E+06 | 1.14E+07 | 0.040 | 18.958 | 0.0831 |
| V1Mg | **3.30E+05** | 3.82E+05 | 1.66E+07 | 8.98E+06 | 8.71E+06 | 0.004 | 14.874 | 0.0688 |
| V2Mg | 4.03E+03 | **7.13E+04** | **1.46E+07** | **7.75E+06** | **8.40E+06** | 0.056 | 17.163 | **0.0661** |

**Table 2**: $V = 50\mathrm{k}, E = 250\mathrm{k}$

| function | D1mr | D1mw | Dr | Dw | Bc | LLmr | Bmr | Run-time[s] |
|---|---|---|---|---|---|---|---|---|
| Baseline | 3.13E+09 | 3.13E+09 | 1.26E+10 | 1.25E+10 | 2.50E+10 | **0.000** | **0.026** | 2502.720 |
| V1 | 7.59E+06 | 9.86E+05 | 1.09E+08 | 5.24E+07 | 7.52E+07 | 0.045 | 19.228 | 9.726 |
| V1Mg | 2.45E+06 | 2.81E+06 | 1.18E+08 | 5.73E+07 | 6.62E+07 | 0.046 | 12.959 | 8.390 |
| V2Mg | **7.63E+04** | **5.85E+05** | **9.49E+07** | **4.71E+07** | **5.82E+07** | 0.054 | 16.196 | **7.983** |

## 3.2. Faster binary search

Binary search is another hot spot for the decomposition. We improved it using leading bit binary search [4, 5] which simplifies the computation and reduces branch prediction misses. Besides, the branchless nature of leading bit binary search makes further vectorization possible.

**Algorithm 2:** Leading bit binary search.

```
1 Input: $z, X_{i=0}^{N}, p = 2^{\lceil \log_2 N \rceil}$
2 Output: $i$
3 $i \leftarrow p - 1$
4 $k \leftarrow p/2$
5 repeat
6     $r \leftarrow i \; xor \; k$
7     $i \leftarrow (z \leq X_r)? \quad r : \quad i$
8     $k \leftarrow k/2$
9 until $k$ = 0
```

**Leading bit binary search (BS).** Leading bit, as shown in Alg. 2 is a modification of standard binary search. There are two aspects of improvement: branchless and bit operation only. In standard binary search, e.g, `std::lower_bound`, we need to query various indices according to the comparison of $z$ and $X_r$. The control flow for different $z$ can be quite different. Leading bit unifies the control flow and makes it branchless by using some bit tricks and ternary operators instead of `if-else` statement. Every input $z$ will go through

$\lceil \log_2 N \rceil$ iterations and the index is determined using conditional assignment. This optimization removed the jump instruction in assembly compared to the `if-else` statement in standard binary search. Leading bit, however, introduces the overhead of padding the array to be of a length of $2^p$ to avoid index out of range check. Another advantage is that leading bit only uses bit-wise operators if we implement division by two as right shifting, which is computationally cheaper than addition and division.

**SIMD leading bit binary search (BSSIMD).** The branchless property of leading bit makes it SIMD and GPU friendly. We can easily vectorize leading bit by making input $z$ a vector instead of a scalar. There is no need to change the operations shown above and we can replace the ternary operator with masked assignment according to vector comparison of $z$ and $X_r$.

Here we show profiling results of standard binary search, leading bit and leading bit SIMD in Table. 3 and Fig. 4 and find the main source of performance improvement.

**Table 3**: $V = 10\text{k}, E = 50\text{k}$. (Bc: Branch executed; Bmr: Branch prediction miss rate)

| Function | Bc | Bmr | Run-time[s] |
|---|---|---|---|
| V1 | 1.14E+07 | 18.958 | 0.0831 |
| V1MgBS | 5.55E+06 | 9.291 | 0.0634 |
| V1MgBSSIMD | **4.65E+06** | **3.777** | **0.0608** |

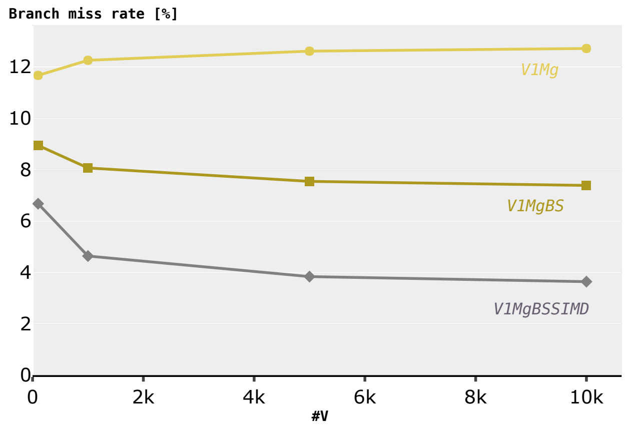


**Fig. 4**: Branch miss rate comparison. (BS: leading bit binary search; BSSIMD: vectorized leading bit binary search)

We can see leading bit brings a boost in run-time and leading bit SIMD offers further improvements. Here we only achieved minor performance improvement through SIMD although in a stand-alone binary search test SIMD can give a 4x speed-up. This is because binary search only takes a small percentage of instructions executed as shown in profiling.

On the other hand, Table. 3 and Fig. 4 show that both the branch prediction misses and miss rate reduce significantly with leading bit. Modern CPU architectures have branch predictors that speculate the result of conditional operations and precompute based on the guess. If the branch prediction is correct, this can significantly improve the performance. Due to the property of being branchless, leading bit results in much less branch prediction misses. Therefore, we contribute the majority of improvement to the optimized branch prediction.

## 3.3. SIMD

Due to the complexity of the original algorithm, most parts are not suitable for vectorization.

Specifically, we only apply SIMD optimization for the following two parts: weighted sampling and conditional swapping of two lists when adding a new edge.

**Vectorization for weighted sampling.** The weighted sampling in the algorithm consists of two parts: scaling the random number according to the weights and doing the binary search based on it.

For the scaling part, we can directly do SIMD optimization since every iteration is independent of each other.

Since we are using the leading bit binary search algorithm illustrated in Sec. 3.2, in which every query has the same number of iteration and is independent of other queries, we can also easily vectorize it using SIMD intrinsics.

**Vectorization for swapping $j$ and $k$.** Before adding a new edge between node $j$ and $k$, we need to determine which of them is larger to keep the adjacency matrix lower-triangular.

In the baseline implementation, we use `if-else` statements to determine which is larger. However, we can use SIMD instructions to swap $j$'s and $k$'s in advance so that $j$'s is always larger. Therefore, we prevent using `if-else` statements when adding a new edge.

## 3.4. Others

We also tried other optimization methods but they only have minor effects on performance improvement.

**Loop unrolling and scalar replacement.** Since the computation between iterations is heavily dependent, we can hardly achieve any performance gain by performing loop unrolling and scalar replacement.

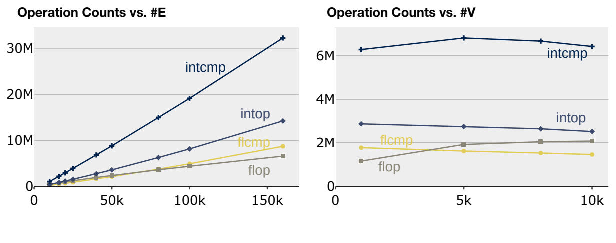
**Better temporal locality.** We tried computing the sum of all edge weights when merging multi-edges, and computing partial sums when computing new edge values, but no significant performance gain is observed.

**Optimization on random numbers.** We tried using better random number generator `PCG_32`, and pre-generating random numbers before the main loop, hoping to get better locality. However, no significant performance gain is observed. Random numbers generated from `PCG_32` perform well in randomness, but require a longer time and thus result in worse performance.

## 4. EXPERIMENTAL RESULTS

In this section, we evaluate our proposed methods in terms of `int` and `flop` counts per cycle. We also present a roofline plot and compare actual run-time (in seconds) between original implementation in Julia and our implementations.

**Experiment setup.** We conduct our experiments on a machine with 4-core Intel Xeon E5-2673 at 2.60GHz. The L1d/L1i, L2, L3 cache size are 32K, 256K, 30720K respectively. Unless otherwise stated, we use GNU 7.5.0 compiler with major optimisation flags (`-O3`, `-ffast-math`, `-march=native`) turned on. Our measurements are done with Turbo boost disabled in BIOS.



(a) $V = 5000$ (b) $E = 40000$

**Fig. 5**: Counts for integer comparisons (intcmp), integer operations (intop), floating point comparisons (flcmp) and floating point operations (flop) for fixed number of V and E.

Figure 5 shows that the number of integer comparisons, integer operations, floating point comparisons and floating point operations grows linearly with respect to the number of edges fixing vertex size. On the other hand, these operation counts roughly remain constant varying the size of vertex with fixed number of edges. As integer operations take the major part, we will consider both integer and floating point arithmetic operations in our following measurements. We will only show selected plots with varying average degree because it spans different magnitudes and is proportional to the total operation counts.

### 4.1. Performance plot

Our main result is shown in Fig. 6. Fixing the number of vertices at 10000, we change the average degree of vertices (i.e. how sparse the graph is). We thus show the performance of various implementation.

We only plot the implementations with major performance gain for clearer illustration. The first major performance gain comes from the optimized data structure (V1Mg, V1 vs. Baseline). We can see that the vector of vector data structure indeed performs better than that of linked list. What's more, by comparing V1Mg and V1, we can also verify our analysis in section 3.1 that always keeping a column sorted is inferior to only merging and sorting a column before processing it. The next major performance gain comes from the leading bit binary search. By comparing V1Mg and V1MgBS, we can clearly see the advantage of the leading bit binary search. We then further change struct of vector into vector of struct in V2MgBS. We can see that when the average degree is small, vector of struct has clear advantage, which also agrees with our analysis in section 3.1. Thus, we justify our analysis and our optimization approach in the previous session.

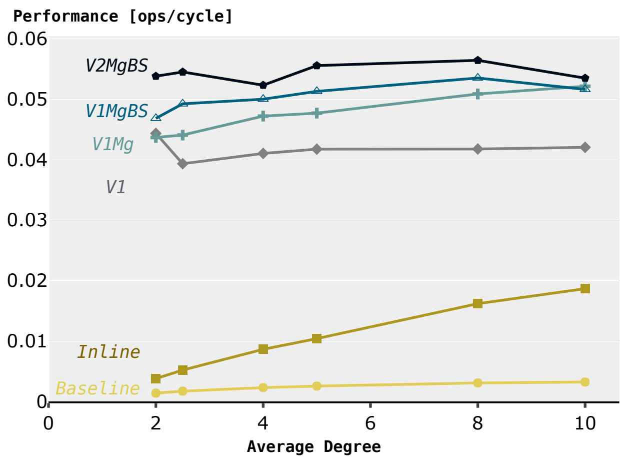


**Fig. 6**: The performance plot for optimizations that have significant improvement. The graph fixes $V = 10000$ and changes the average degree. In the graph, "V" stands for vector, which means the implementation uses vector of vectors as the data structure. The difference between "V1" and "V2" is that "V1" uses struct of vector, while "V2" use vector of struct. "Mg" stands for merge, which means the implementation merge and sort elements of a vector right before its being processed. "BS" stands for Binary search, which means the leading bit binary search is implemented. The fastest implementation *V2MgBS* gains a median speed-up of $18.67\times$ and a max speed-up of $41.38\times$.

We present a more detailed graph to illustrate the effect of using SIMD intrinsics. As shown in Fig. 7, the performance improvement of using SIMD (except for binary search) is not as large as optimizations stated above. We believe this is because there are not many parts of the algorithm are suitable for SIMD optimization. Those parts where SIMD is used account for only a little share in run-time.

However, SIMD optimization on binary search can bring visible improvement, as shown in Fig. 8, which might result from less branch misses, as stated in Sec. 3.2.

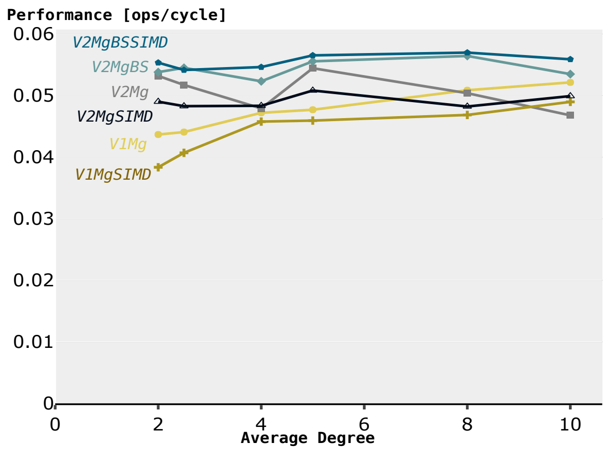


**Fig. 7**: Performance comparison on SIMD implementations (Minor).

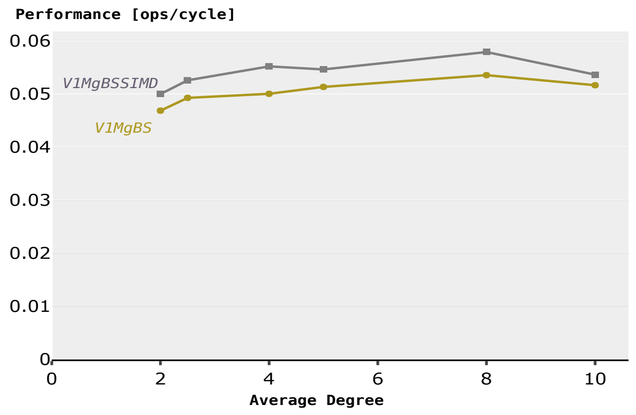


**Fig. 8**: Performance comparison on SIMD implementations (Major).

## 4.2. Roofline model

Roofline model plots performance against operational intensity, which helps to distinguish memory- and compute-bound applications and illustrates the change of intensity across sizes. The operation intensity for an input of size $n$ is defined as $I(n) = W(n)/Q(n)$, where $W(n)$ is the operation counts and $Q(n)$ is bytes transferred in memory. In our case, we consider both integer and floating point operations. For the memory movement, we use `Cachegrind` to detect memory reads and writes. We consider cold cache in our measurement by flushing up to the LLC by filling a large buffer with random numbers. We measure the theoretical peak memory bandwidth by using STREAM benchmark, a tool developed in [6]. Our test machine has a sustainable memory bandwidth of 4.16 bytes/cycle. The theoretical peak performance $\pi$ is $4 \times 4 = 16$ ops/cycle with instruction-level parallelism enabled.

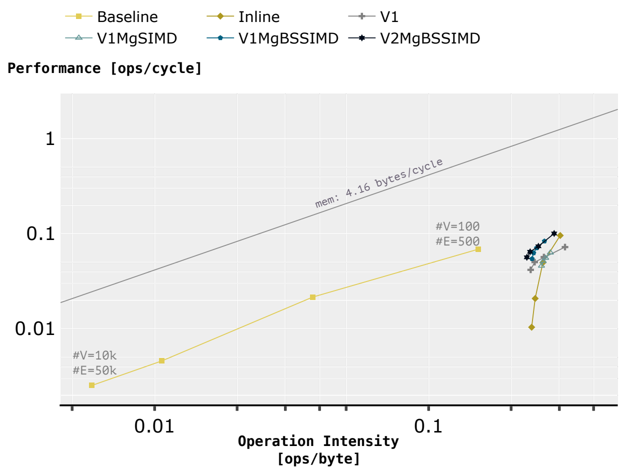


**Fig. 9**: Roofline measurement.

As shown in Figure 9, baseline implementation is a memory-bound task and the operational intensity drops quickly with growing problem size. Our implementation of data structure significantly reduces the amount of data moving in the algorithm, resulting in a large increase in operational intensity. Furthermore, optimizations like SIMD and leading bit binary search increase the performance under roughly the same operational intensity. However, due to the random access pattern of the algorithm, we argue that the theoretical bound cannot be approached and there is limited spatial locality even with our optimizations, and therefore the effective memory bandwidth will be much smaller than the theoretical one.

## 4.3. Run-time

We further compare our implementation with the reference implementation in Julia [3]. Since it is not convenient to get the CPU cycles of Julia programs, we directly compare the run-time in seconds. Notice that Julia is a programming language known for its fast performance. In Fig. 10, we compare our baseline implementation in C++ with the reference implementation in Julia. We can see that Julia is faster than our baseline, especially when the average degree is large. Next, we compare the Julia code with our optimized implementation in Fig. 11. We can see that after optimization, our implementation consistently performs better than the Julia implementation. This result agrees with the performance plot we showed in Sec. 4.1. Notice that on a medium-sized graph as illustrated in Fig. 11, the performance for V1MgVSSIMD and V2MgBSSIMD are similar. On large graphs with edges in order of millions, V2MgBSSIMD is preferred compared to V1MgVSSIMD as shown in 12.

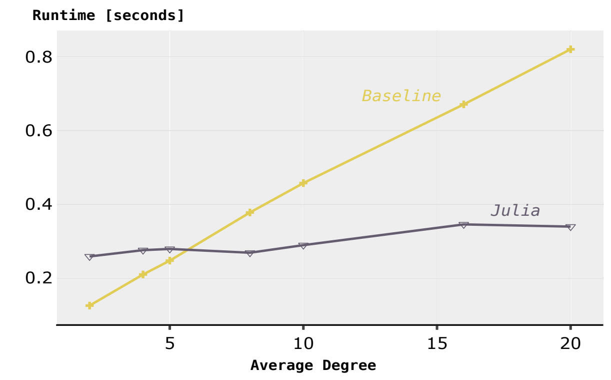


**Fig. 10**: Run-time comparison between baseline C++ implementation and Julia implementation.

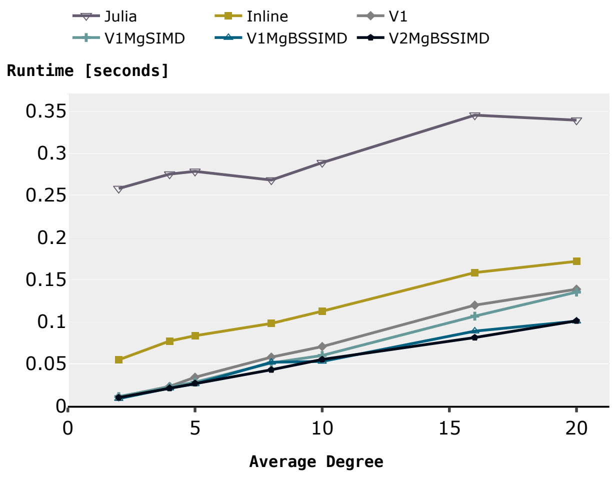


**Fig. 11**: $V = 5K$

## 5. CONCLUSIONS

We first analyze the original implementation of the algorithm and profile the run-time. We state that the original implementation has bad spatial locality and we identify that the binary search is a bottleneck of the algorithm. We thus propose a new data structure, which uses array with dynamic memory management to replace the linked list. In this way, we achieve better spatial locality and already achieve a roughly $5\times$ speed-up. We also talk about various designing concerns, including whether or not to save an edge as a whole, and when to merge and sort the edges. We carry out the experiment to justify our choice. We also implement the leading bit binary search to solve the bottleneck. The leading bit binary search uses only bit operations and makes the algorithm branchless and make further vectorization possible. We also apply SIMD intrinsics to accelerate both the computation flow and the optimization flow. Finally, we show the experiment results in various graphs to validate our analysis. We draw a performance plot which shows all the major improvements we achieved. We also draw a roofline model. What's more, to compare with the original implementation in Julia, we directly compare ours and the Julia code in terms of run-time in seconds. The experiment shows that our implementation achieves a significant performance boost.

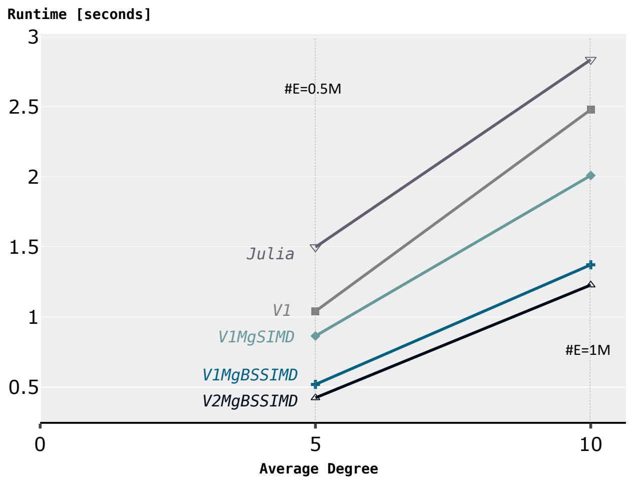


**Fig. 12**: Large graph with $V = 100K$

## 6. FURTHER COMMENTS

### 6.1. Why no blocking

We discuss a lot about blocking in the lecture of sparse matrix vector multiplication. However, blocking might not help in our case. Dependency between columns makes blocking columns together unreasonable. What's more, the algorithm does not reuse much. Based on all these considerations, we decide not to use blocking in our implementation.

### 6.2. Is sorting the columns necessary

We validate our algorithm by feeding the result of the decomposition into the PCG iterative solver and we observe that the convergence speed is the same as the Julia code. Decomposition without sorting the column reduces the runtime by roughly 30% at the expense of slower convergence.

### 6.3. Future works

We observe that the PCG random number generator does not give us much performance gain and it is possible to further investigate how to generate a vector of random numbers to speed up precomputation. Inserting a new edge is still a bottleneck in our array-based data structure. We look forward to improving it by optimizing random memory write or insert new edges in consecutive memory.

## 7. CONTRIBUTIONS OF TEAM MEMBERS

Here are the contributions of team members.

**Daoye.** helped analyze the bottleneck, helped Kaifeng with data structure implementation, verified that unordered_map does not work. helped dexin with computation flow acceleration with SIMD, implemented the validation.

**Fanlin.** Mainly focused on code and performance analysis: tried different profiling tools like perf, VTune, gperftool and valgrind and helped to find the bottlenecks; generated plots for cache analysis, performance comparisons, roofline model and run-time comparisons; also helped with validation and non-SIMD optimization like precomputation using PCG random number generator.

**Dexin.** Implemented baseline code with help of Kaifeng and Daoye; tried several non-SIMD optimizations like inlining, loop unrolling and scalar replacement; focused on SIMD optimization on computation flow acceleration and weighted random sampling including binary search.

**Kaifeng.** Mainly focused on data structure design for sparse Laplacian matrix decomposition, including some locality optimization. I also proposed to use leading bit binary search to accelerate random sampling and implemented a modified version from reference paper for both scalar and SIMD vector inputs. In addition, I devoted some efforts in profiling using perf and valgringd cooperating with Fanlin.

## 8. REFERENCES

[1] Rasmus Kyng and Sushant Sachdeva, "Approximate gaussian elimination for laplacians-fast, sparse, and simple," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2016, pp. 573–582.

[2] Daniel A Spielman and Shang-Hua Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 81–90.

[3] Danial A. Spielman, "Laplacians.jl," https://danspielman.github.io/Laplacians.jl/v0.1/.

[4] Fabio Cannizzo, "A fast and vectorizable alternative to binary search in o (1) with wide applicability to arrays of floating point numbers," *Journal of Parallel and Distributed Computing*, vol. 113, pp. 37–54, 2018.

[5] Matt Pulver, "Binary search revisited," .

[6] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.