A FAST IMPLEMENTATION OF DATA VALUATION WITH SHAPLEY VALUE AND KNN

Ziqiao Kong, Yuening Yang, Junzhen Lou, Yijun Ma

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

This paper presents several optimizations on two algorithms that calculate the Shapley Values for KNN-based ML models. We exploited ILP by unrolling and vectorization, optimized for cache locality by blocking and attempted several other mechanisms. Compared to the baseline, a total speedup of $7.2\times$ and $8.1\times$ for these two algorithms has been achieved.

1. INTRODUCTION

This section introduces the motivation and contribution of our work. It also provides a brief overview of related work.

Motivation. Large-scale databases constitute an essential part in ML training, which are usually contributed by several entities. To decide how much revenue each contributor should gain, we try to figure out how each point in the training sets is valued for a specific ML model training on the whole dataset. A widely adopted means is to calculate its Shapley Value, which serves as its unique measure in profit allocation.

Though the shares are expected to be calculated and returned to the user in a reasonable response time, datasets are usually large and the original algorithm takes long by going though each pair of points in the test and training sets. Apart from optimizing the original algorithm, we focus on an improved algorithm and try to optimize its implementation to achieve reasonable performance.

Contribution. This paper presents several optimizations of two algorithms that calculate the Shapley Values for the KNN-based family of ML models. The optimizations will be detailed for each sub-part of the algorithms.

2. BACKGROUND ON THE ALGORITHMS

This section describes the theoretical background on the two algorithms we implemented.

2.1. Exact Computation (Algorithm 1)

K-nearest neighbors (KNN) is widely used in machine learning models adopting the nearest neighbor algorithm. Given the number of the training set data points N and the number of the test set data points N_{test}, the exact Shapley Value (SV) of a KNN classifier can be computed recursively in $\mathcal{O}(N \log N N_{test})$ time [1].

2.2. Improved MC Approach (Algorithm 2)

The aforementioned algorithm is not scalable for the models based on weighted KNN and multiple-data-per-seller KNN due to the exponential time complexity. Therefore, a modified version of the MC algorithm is proposed. It limits the number of permutations and is implemented using a length-K heap to trade off between the approximation errors and the efficiency, and has a time complexity of $\mathcal{O}(\frac{N}{\epsilon^2} \log K \log \frac{K}{\delta})$ where ϵ is the error bound and $1-\delta$ is the confidence [1].

It is clear that this approach is mainly an algorithmic improvement on the SV calculation part of the exact computation with a smaller permutation size and a limited-length heap.

2.3. Cost Analysis

Assume the input data is a $N \times (d+1)$ training matrix and a $M \times d$ test matrix, the cost analysis of the two algorithms above can be divided into three parts.

Point Distance. Both the exact computation and the improved approximation share the common code path to calculate the point distance as part of the KNN algorithm. The cost of the point distance C_{pd} can be defined as:

$$C_{pd} = ((2C_{add} + C_{mult}) \times d + C_{sqrt}) \times N \times M \quad (1)$$

Exact Computation. For the exact computation of the Shapley Value shown in Algorithm 1, the actual cost is hard to obtain as it needs several array sorts to get indices α_i, where each comparison is counted as 1 unit cost. Our implementation choose `std::sort`, which is backed by quick

Algorithm 1: Exact computation

input : Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$, test data $D_{\text{test}} = \{(x_{\text{test},i}, y_{\text{test},i})\}_{i=1}^{N_{\text{test}}}$
output: The SV $\{s_i\}_{i=1}^{N}$

for $j \leftarrow 1$ **to** N_{test} **do**
  $(\alpha_1, ..., \alpha_N) \leftarrow$ Indices of training data in an ascending order using $d(\cdot, x_{\text{test}})$;
  $s_{j,\alpha_N} \leftarrow \frac{1[y_{\alpha_N} = y_{\text{test}}]}{N}$;
  for $i \leftarrow N-1$ **to** 1 **do**
    $s_{j,\alpha_i} \leftarrow s_{j,\alpha_{i+1}} + \frac{1[y_{\alpha_i} = y_{\text{test},j}] - 1[y_{\alpha_{i+1}} = y_{\text{test},j}]}{K} \frac{\min\{K,i\}}{i}$;
  end
end
for $i \leftarrow 1$ **to** N **do**
  $s_i \leftarrow \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} s_{j,i}$;
end

sort with an average of $1.39NlgN$ comparisons[2] in our case. The cost of the sort C_{sort} can thus be defined as:

$$C_{sort} \approx 1.39 \times M \times N \log N \tag{2}$$

The cost of the Shapyley Value calculation C_{sp} can be defined as:

$$C_{sp} = N \times (C_{div} + C_{cmp} + (N-1) \times (2C_{cmp} + 2C_{div} + C_{mult})) \tag{3}$$

Therefore, the total cost C_{ec} of the exact computation can be defined the sum of the cost:

$$C_{ec} = C_{pd} + C_{sort} + C_{sp} \tag{4}$$

Improved MC Approach. This algorithm refers to the following functions: permutation-matrix generation, point-distance calculation, and SV calculation.

Since the permutation is only on the index and does not involve any floating-point operation, the cost is only related to the cost of point-distance calculation that are explained before and the cost of SV calculation which is input-value dependent and is calculated during runtime.

$$C_{util} = 3 \times K + 1 \tag{5}$$

$$N \times T \times K \times (C_{util} + 1) \leq C_{SV} \leq N \times T \times (N-1) \times (C_{util} + 1) \tag{6}$$

The total value of the improved MC approximation algorithm can be calculated as:

$$C_{appr} = C_{pd} + C_{SV} \tag{7}$$

Algorithm 2: Improved MC Approach

input : Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$, test data $D_{\text{test}} = \{(x_{\text{test},i}, y_{\text{test},i})\}_{i=1}^{N_{\text{test}}}$, utility function $v(.)$, the number of permutation T
output: The SV $\{s_i\}_{i=1}^{N}$

for $j \leftarrow 1$ **to** N_{test} **do**
  for $t \leftarrow 1$ **to** T **do**
    $\pi_t \leftarrow$ GenerateUniformRandomPermutation(D);
    Initialize a length-K max-heap H to maintain the KNN;
    for $i \leftarrow 1$ **to** N **do**
      Insert $\pi_{t,i}$ to H;
      if H *changes* **then**
        $\phi^t_{\pi_{t,i}} \leftarrow v(\pi_{t,1:i} - v(\pi_{t,1:i-1}$
      else
        $\phi^t_{\pi_{t,i}} \leftarrow \phi^t_{\pi_{t,i-1}}$
      end
    end
  end
  $\hat{s}_{j,i} = \frac{1}{T} \sum_{t=1}^{T} \phi_{t,i}$
end
for $i \leftarrow 1$ **to** N **do**
  $s_i \leftarrow \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} s_{j,i}$;
end

Actual Cost. As previously discussed, the cost of both algorithms varies with the input. In practice, we perform instrumentation at runtime to determine the actual cost.

3. OPTIMIZATION METHODS

3.1. Baseline Implementation

The baseline implementation mostly is a plain and trivial translation of the reference Python implementation [3].

Exact Computation. The exact computation contains two phases: KNN calculation, which calculates the distances between each training point and test point pair and sorts the training data based on these distances; SV calculation, which takes the sorted training data as input and outputs the SV values.

Improved MC Approach. The improved MC approach is mainly based on three functions: permutation-matrix generation, which permutes on the row index of the training data T times; point-distance calculation, which calculates the distances between each training point and test point pair; SV calculation, which takes the permutation-matrix and distance-matrix as input, uses a K- length heap to store the nearest points, and applies the utility function to calculate SV values.

3.2. Profiling

Profiling consists of both execution time and cache miss analysis.

Execution Time Analysis. As discussed in section 2.3, the actual cost is retrieved by instrumentation. The table 1 shows the actual cost of both algorithms' baseline implementations given $M = 32$, $N = 8192$ and $d = 512$.

Cost	Flops
C_{pd}	402915328
C_{ec}	408307571
C_{appr}	403441116

Table 1. The cost of algorithms

The cost of point distance almost dominates the whole computation of both algorithms by 98.68% and 99.87% and thus the optimization of the calculation of the point distance will be our main focus.

Cache Miss Analysis. We calculated the cache miss rate according to the memory access pattern of the algorithms. Depending on the input size, the analysis is divided into 2 cases: when the input size is large enough, we assume that a single row of elements will not fit into the cache, and thus there should be compulsory capacity, and conflict misses; when the input size is small, we assume that all matrices in concern will fit into the cache, so there are only compulsory misses.

In detail, for Algorithm 1, the KNN function benefits from both spatial and temporal locality with small inputs, but only spatial locality with large input; the computation of Shapley value has a bad access pattern caused by random reading, so the theoretical cache miss rate is only a rough estimate of the upper limit. For Algorithm 2, there are 3 functions in concern: calculation of point distances is nearly the same as KNN function in Algorithm 1; permutation on index only causes a small amount of cache misses; and calculation of Shapley value also has a bad and nearly random access pattern with a heap involved, which results in an approximation.

Above all, the result of the cache miss analysis is shown in table 2 and 3.

	Large Input Size	Small Input Size
KNN	$M \times (15N/8 + Nd/4)$	$(Nd+NM+d+N)/8$
SV	$5.125NM$	$(2NM + 2d)/8$

Table 2. Cache Miss Analysis of Algorithm 1

	Large Input Size	Small Input Size
Point Distances	$M\times(Nd+d+4N)/8$	$(Nd + NM + d + N)/8$
Permutation	$3N/8$	$N/4$
SV	$53MN/8 + M/8$	$(2NM + 2d)/8$

Table 3. Cache Miss Analysis of Algorithm 2

The loop order of the functions in baseline implementation is already optimal, providing maximized unit-stride. Blocking can reduce conflict misses, thus further optimizing cache locality and achieving better performance.

3.3. Unroll and Scalar Replacement

Point-distance Calculation. We would like to apply scalar replacement to eliminate memory aliasing and precompute common subexpressions, and loop unrolling and accumulators to exploit instruction-level parallelism (ILP).

The previous analysis shows that the point-distance calculation function is the main bottleneck of both algorithms. Inside this function, the computations in the main loop are independent of the previous iterations. Therefore, all three methods could be applied. We tested the algorithm with different levels of unrolling, including unrolling only the inner loop and unrolling both the outer and inner loop different times.

SV Calculation in Algorithm 1. Though the SV calculation is not the main bottleneck, we still applied the unroll the loops to achieve a slight improvement in the overall performance. Since there are 2 divisions per loop but only 1 port for division is available on our machine, scalar replacement is applied to compute the inverse of divisors in advance for better unrolling.

SV Calculation in Algorithm 2. A slight improvement in the overall performance by unrolling and accumulators were also applied here when summing the results at the end of the algorithm. However, since the branching is execution order dependent, it is hard to achieve great ILP in the main part of this function.

3.4. Blocking

Point-distance Calculation. Considering the loop order of i-j-k has already optimized the access pattern with unit strides, there is no need to change the loop order. By nature, there is no dependency on reading or writing in the inner loop, so we introduced blocking to the computation of point distances to achieve better cache locality.

We chose the inner double-loop of j and k to apply blocking. Through loop-tiling and loop exchange, we obtained four-fold loops with different blocking sizes, and then unrolled the inner-most loop and applied scalar replacement. The experiment result shows that blocking leads to fewer cache misses and thus better performance.

SV Calculation in Algorithm 1 and Algorithm 2. As we can expect, SV calculation in Algorithm 1 suffers from the poor locality. For all writing and most reading, the index is determined by an unknown value rather than loop accumulators, so the access pattern is not in sequence and is nearly random. Moreover, the mix of single-layer loop and double-layer loop leads to a consequence that the results of the outer loop decide the actual code path in the inner loop. This uncertainty and "randomness" also implies serious read-after-write dependency, which makes changing execution order impractical. The unpredictability hinders accurate theoretical cache miss analysis as well. With heap computation involved, the SV calculation in Algorithm 2 is even worse. As a result, applying blocking optimization to SV calculation in both Algorithm 1 and 2 is considered unfeasible.

3.5. Vectorization with AVX2 Intrinsics

Point-distance Calculation. Vectorization allows us to do the square-and-add in the point-distance calculations in parallel, and this is done based on the previous unrolling trials. First, set the accumulator vectors to all zeros. In each inner loop, we load the point coordinates into 256-bit vectors and subtract it from the target point vector. Then the result is squared and added to the accumulators using $\_mm256\_fmadd\_pd$. Finally, in the outer loop, the accumulator is then summed and the square roots are stored back in the matrix.

SV Calculation in Algorithm 1. By examining the code of SV calculation, we found that the element access pattern is highly dependent on the output of the previous KNN calculation, i.e. is not in sequence and is input-value dependent. Therefore, we believe it is hard to do SIMD here.

SV Calculation in Algorithm 2. In contrast to the SV calculation in Algorithm 1, the SV calculation in Algorithm 2 consists of two phases: The first phase is to calculate the differences between the utility values using max-heap and branching, and the second phase is to sum the values in different permutation into a single value. Though the first phase has a high execution order dependency, we can do vectorization on the second phase, which is also the bottleneck of this function.

3.6. Function Inlining

General. The same as what we have done in the homework, we tried to inline the matrix element access functions. No obvious improvement is shown, and we found by reading the assembly code that GCC had already done it.

Algorithm 2 Specialized. By inspecting the function calls of the utility function in SV calculation, we found that the output values are highly reused in the inner loop. In this case, we used a register to store that result to eliminate the duplicate function calls.

A slight improvement was also shown for inlining the utility functions, probably because this function is only in one if-branch in the SV calculation and is called not that often.

3.7. Sorting Algorithm Optimization

This is an algorithm-level optimization specialized in Algorithm 1 that sorts the training points using the result of KNN point distances. Different sort algorithms behave differently[4, 2] on comparisons, and runtime varies with the actual input. The reference Python implementation uses a stable sort algorithm which might be slow in most cases and thus could be optimized. We tried to write our own iterative sorting methods and applied different sorting algorithms in C++ standard library to find the one with the best performance.

3.8. Sparse Matrix Optimization

This optimization is specialized in Algorithm 2. Since the intermediate values in utility value computation are just zero or one, we considered doing some pre-computing and using a sparse matrix to store the result.

However, this attempt failed as we examined our randomly generated input and found that the result matrix is in most cases not sparse, and the pre-computing will lead to more work than directly computing the needed one in the if-branch in the SV calculation.

4. EXPERIMENTAL RESULTS

In this section, we describe the hardware and software specifications and then explore how different compilers and flags, and prefetching influence the performance. In the end, we evaluate the results of the optimizations introduced in the previous section.

4.1. Experimental Setup

All benchmarks and experiments are conducted on an i7-8700[5] platform, with specifications listed in table 4.

4.2. Input Sizes

To facilitate the optimizations and analysis, the d of both matrices is fixed to 512 and the M of the test matrix is fixed

CPU Name	i7-8700H
CPU Microarchitecture	Coffeelake
Base Frequency	3.2 GHz
Turbo Boost	Disabled
Prefetch	Enabled
L1 Cache	64 KB per core
L2 Cache	256 KB per core
L3 Cache	12 MB shared
Memory Bandwidth	30733.74 MiB/s
System	Ubuntu 22.04 LTS
GCC	11.2.0-19ubuntu1
Clang	14.0.0-1ubuntu1
ICC	2021.6.0 20220226

Table 4. The benchmark platform

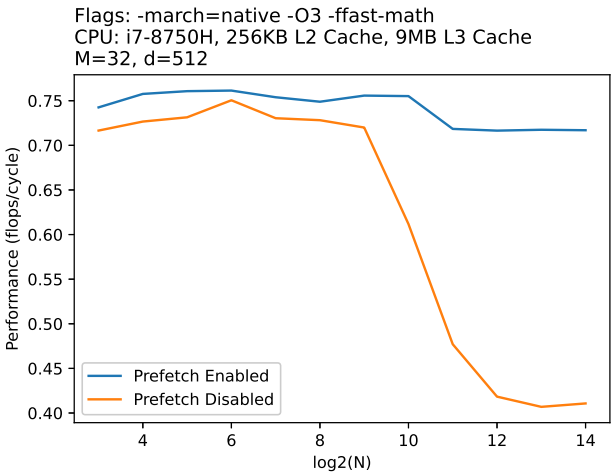

Fig. 1. Performance of point distance

to 32. The N of the training matrix will increase from 8 to 16384 during our experiment.

4.3. Compilers and Flags

The very first experiment is to identify the optimal compiler flags for our experiment. We pick the baseline computation of point distance to test the flags because it has a perfectly continuous memory access pattern and accounts for most of the cost as analyzed in section 3.2. The results are shown in table 5, and all compilers share the common flags `-march=native`, `-O3`, `-ffast-math` (omitted in table).

Compiler and Flags	Performance
gcc -ftree-vectorize -funroll-all-loops	0.680
gcc -fno-tree-vectorize -fno-unroll-loops	0.680
clang -fno-tree-vectorize -funroll-loops	0.674
clang -fno-tree-vectorize -fno-unroll-loops	0.676
icc -ax=COFFEELAKE -funroll-loops	0.322
icc -no-vec -fno-unroll-loops	0.321

Table 5. Different compilers and flags combination

Surprisingly, `icc` behaves the worst while `gcc` and `clang` achieve similar performance. In addition, the compilers' auto-vectorization and unroll doesn't make a difference in our case. Therefore, in the following experiments, if not specified, the default compiler is `gcc` and the flags are `-march=native`, `-O3` and `-ffast-math`.

4.4. Prefetch

Another interesting factor involved is prefetching. As the memory access pattern of KNN calculation is rather regular and predictable, prefetching could have more than 90% accuracy in this case[6], which largely improves the performance. The figure 1 shows that the performance doesn't drop much as the input size increases when prefetching is enabled[1]. To achieve better performance in the following experiments, the prefecthing is enabled by default.

Cost	Flops	Cycles
C_{pd} with `std::sort`	128177	54405157
C_{pd} with `std::stable_sort`	104853	62658323

Table 6. The cost of sort algorithms

4.5. Sorting Algorithms

Though the original Python reference code uses a stable sort algorithm, we have tested and confirmed that both `std::sort` and `std::stable_sort` work fine on all of our tests.

Table 6 shows the cost and actual runtime of both algorithms[2] given $N = 8192$. As expected, the `std::stable_sort`, backed by merge sort, performs approximately $N * lgN$ comparisons[4], while the `std::sort`, backed by quick sort, performs about $1.39NlgN$ comparisons[4, 2]. However, the overhead of moves and memory allocation in merge sort outweighs that of more flops in quick sort. What's more, we do not need a stable sort here to ensure correctness. Therefore, we replace the original `std::stable_sort` with `std::sort` to achieve better performance.

[1]The platform mentioned above is not capable of disabling prefetching and thus this experiment is conducted on a different machine as stated.

[2]We also tried to implement the sort algorithm by ourselves, but its flops and cycles are worse than std::stable_sort, and it's thus omitted.

		Outer Loop			
Inner Loop		2	4	6	8
	1	0.620	1.712	1.421	1.436
	2	1.583	1.872	1.821	1.411
	4	1.700	1.636	1.660	1.474

Table 7. Unroll performance of point distance

4.6. Unroll

The second experiment tries to figure out the best unroll factors. Table 7 shows the performance of the different unroll factors on the KNN point distance computation when N equals 8192. When we unroll the inner loop by 2 and the outer loop by 4, the computation achieves the best performance. Besides, based on the similar steps, we could also decide on the best unroll factors of the exact Shapley Value and improved MC approach computation.

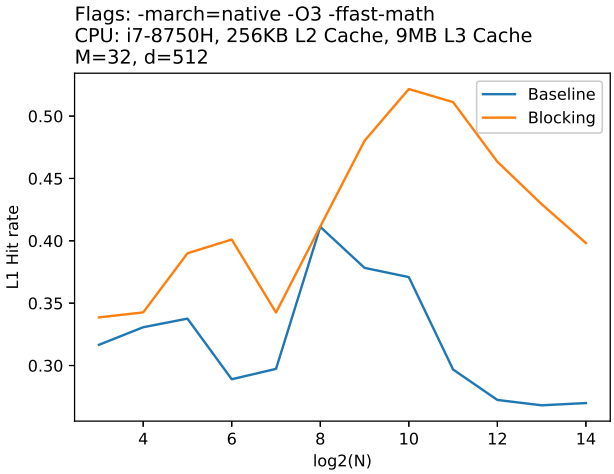

Fig. 2. L1 Hit rate of point distance

4.7. Blocking

Previous analysis in section 3.2 indicates that when N increases, there will be much more memory transfer as the cache can no longer hold all the data. Therefore, we used `perf` to test with different data sizes and plotted the figure 2 that shows how much the blocking optimization helps improve the L1 hit rate.

The experiment result shows that the blocking optimization not only largely improves L1 hit rate but also postpones the beginning of the drop to $N = 2^{10}$ compared to the baseline. Also, it's interesting that the L1 hit rate of the baseline even increases as the matrix becomes larger when $N \leq 2^8$. The reason is probably that more continuous cache line fetch leads to better CPU prefetching. However, when the whole data cannot fit into the cache, the penalty of cache line eviction gradually outweighs the improvement of prefetching and thus the L1 hit rate starts to drop for both cases.

4.8. Overall Performance

By Combining the best optimizations of different parts of algorithms, we plotted the overall performance and tried to make comparison of different optimization methods.

In the plot legend, "optimization1 + optimization2" means that the first optimization is done for point-distance calculation and the second optimization is done for SV calculation.

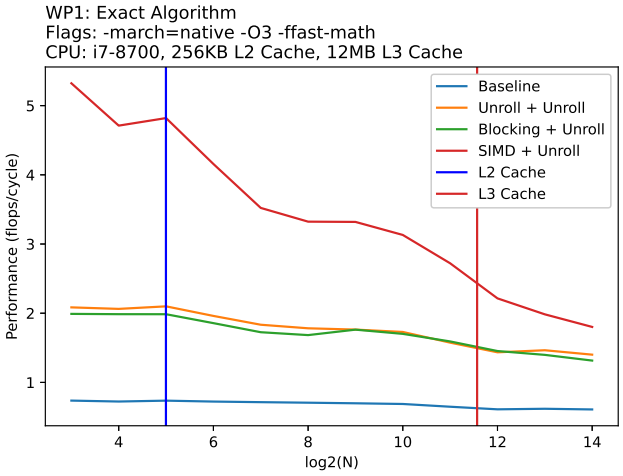

Fig. 3. Overall performance of Algorithm 1

Algorithm 1: Exact Computation. We tested with different data sizes and plotted Figure 3 that shows how different optimization methods help to improve the overall performance. As a baseline, the straightforward implementation reaches a flat performance of around 0.65 flops/cycle for all input sizes.

In all optimizations, SV calculation is optimized by unrolling consistently since it is not the bottleneck and only counts for negligible execution time. As expected, the best unroll version of point distances computation combined with SV calculation improves the performance by around 3 times compared to the baseline and reaches 2.0 flops/cycle as the peak. Since the inner loop of point distances computation has a clear structure that does not contain any kind of dependencies, the obvious speedup is produced by a high level of ILP given by unrolling and scalar replacement. Other basic block optimizations such as precomputation also contribute a little to the speedup.

The optimal blocking version of point distances computation combined with SV calculation surprisingly improves the performance by around 2.8 times on average. Considering the code structure, the calculation of point distances has a good access pattern with unit-stride and no dependency involved. This leads to a noticeable speedup. Moreover, as Figure 3 shows, the performance curve is slightly lower than the unroll version at the beginning but rises even a little bit higher than the unroll version with input size ranging from 9 to 12 approximately, then drops down again at the end. This tendency perfectly reflects the difference in L1 hit rate in Figure 2.

We now discuss the result of the optimal SIMD version of point distances with SV calculation. Note that the optimal SIMD optimization is actually achieved by implementing AVX2 intrinsics in the inner-most loop and unrolling the second inner loop to apply more AVX2 commands. As expected, the computation of point distances can benefit a lot from vectorization because of its clear structure for pack processing and lack of dependency. As the most optimized implementation of all versions, it achieves a $7.2\times$ speedup and reaches about 5.3 flops/cycle at best.

Prefetching helps to explain why the curve of SIMD optimization drops significantly with the input size growing larger than L2 and L3 Cache capacity but the others don't. As we can see in Figure 1, without prefetching the performance decreases dramatically as the input size grows larger. During the experiments, we discovered that both the unroll and blocking versions of point distances computation benefit from prefetching, but the SIMD version does not. This is the reason why only the SIMD performance curve drops while the curves of unroll and blocking remain comparatively stable.

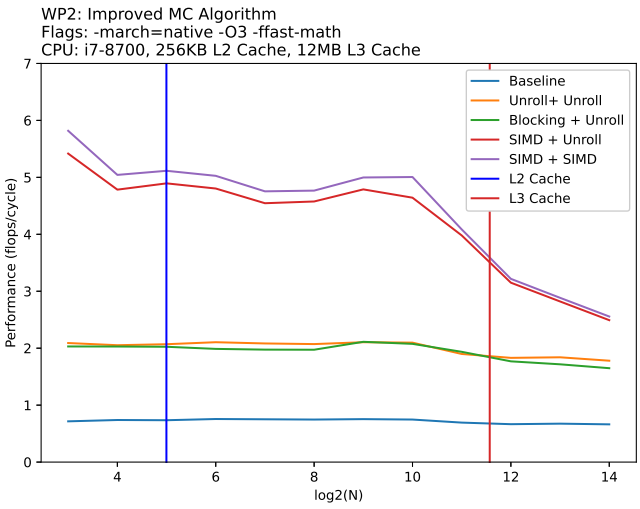

Fig. 4. Overall performance of Algorithm 2

Algorithm 2: Improved MC Approach. The same testing approach is performed on the improved MC approach. The basic implementation is used as the baseline which reaches a flat performance of around 0.72 flops/cycle for all input sized as shown in Figure 4.

Similar to the result in the Algorithm 1, unroll and blocking coincidentally achieve almost the same speedup result. They are almost stable around 2.09 flops/cycle which is a $2.9\times$ speedup from the baseline. In addition, the curve of the blocking version also exhibits a slight rise with the input size around 2^{10}, which corresponds to the analysis of cache hit rate in Figure 2.

The SIMD optimization in the calculation of point distances brings about a significant improvement on the performance compared to that of the baseline. Combined with the unroll version of the SV calculation, it reaches the peak performance of 5.42 flops/cycle, i.e. $7.5\times$ speedup when the input size is 8 and slowly drops as the input size increases. The performance drops significantly when the data is unable to fit into L3 caches.

Different from Algorithm 1, we also implemented a SIMD optimization for the SV calculation in Algorithm 2. As discussed before, the majority of the computation in SV calculation has a bad, unpredictable access pattern thus infeasible to be vectorized, but we still managed to apply some AVX2 intrinsics in a minor part. Though the SV calculation is not the bottleneck, vectorization also helps when the data could fit into L2 caches. With the help of SIMD on both functions, we reaches the highest performance of 5.82 flops/cycle and improves the performance by around 8.1 times.

Compared to Algorithm 1, Algorithm 2 shows a higher performance in general. Furthermore, Algorithm 2 notably enhances the performance as input size growing larger, which makes all curves even flatter than in Algorithm 1 and produces less performance loss. Taking into account that Algorithm 2 is an algorithmic improvement of Algorithm 1, we believe that this is a tight verification of the real-world design purpose.

4.9. Roofline

The roofline plots of both algorithms are quite similar. Here we focus on the roofline of Algorithm 1 as an example in Figure 5 and 6. In addition to the common roofline based on memory bandwidth and theoretical float point performance, we also derive two tighter rooflines considering the instruction mix in KNN computation as shown in the figure.

Figure 5 is calculated with the theoretical cache miss analysis. For all optimizations except the SIMD, if the cache could hold all data input, they are computation bound. However, when the input matrix doesn't fit into the cache, they become memory bound. For vectorization optimization, it initially is memory bound and then becomes compute bound once the size of input data begins to increase. Meanwhile,

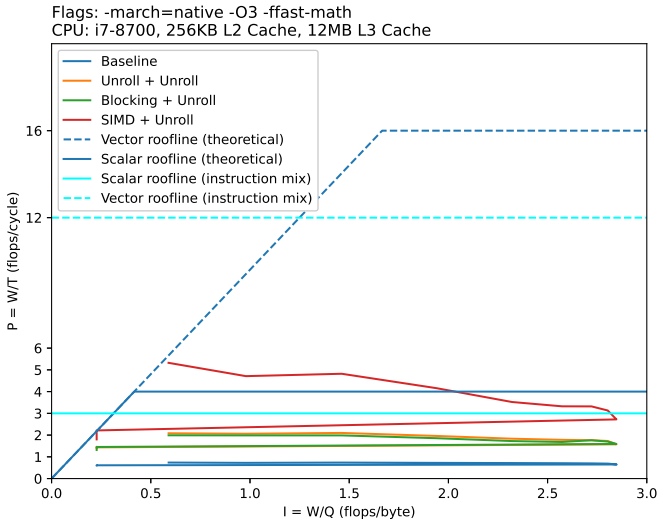

Fig. 5. Roofline of Algorithm 1 Based on Cache Miss Analysis

like other optimizations, once the cache is full, it becomes memory bound. However, it is worth mentioning that, the vectorization optimization almost reaches the theoretical performance at its operational intensity when the input matrix is large or smaller enough.

Figure 6 is calculated with the tested cache hit rate. As we can see, these two figures share a huge difference. This is because the theory is only divided into 2 cases such that it will produce inaccuracy. The tested cache miss rate shows that all versions are almost compute bound.

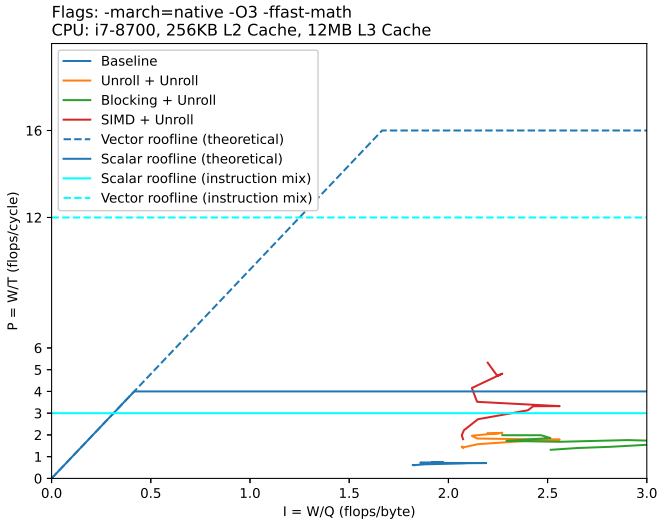

Fig. 6. Roofline of Algorithm 1 Based on Tested Cache Hit Rate

5. CONCLUSIONS

We have presented several optimized implementations for two algorithm in KNN-based data evaluation: Exact computation and Improved MC Approach.

Our optimization achieved $7.2\times$ speedup in total for Exact Computation, and the best performance gave a $8.1\times$ speedup in Improved MC Approach.

Profiling shows that the calculation of point distances makes up the vast majority of the total cost, and that the expected numbers of cache misses for all functions are high and contain many compulsory misses, implying the performance bottleneck and chances of optimizing memory hierarchy.

Exploiting ILP by loop unrolling and vectorization brought $2\times$ and $8\times$ speedup respectively. Optimizations on cache locality, or specifically blocking, increased the cache hit rate by half, and we gained $2\times$ speedup. We have also tuned compiler flags and prefetching settings to attain optimal performance, and attempted non-recursive sorting algorithm and sparse matrix optimizations.

6. CONTRIBUTIONS OF TEAM MEMBERS

6.1. Ziqiao Kong

Implemented and profiled the baseline, part of unroll, scalar replacement and SIMD optimizations of Algorithm 1. Cost analysis of the Algorithm 1. Revised the SIMD implementation of Shapley Value calculation of Algorithm 2. Instrumented the code for further analysis. Collected and analyzed the data of the compiler flags, sorting algorithms, unroll factors analysis, prefetch analysis, early performance plot and early roofline[3]. Plotted the figure 1, 2 and table 1, 4, 5, 6, 7.

6.2. Yuening Yang

Implemented the inital baseline of Algorithm 2 and did the cost analysis of it. Implemented the optimizations of sorting algorithm in Algorithm 1 and unroll, scalar replacement and SIMD in Algorithm 2. Helped Junzhen to do blocking. Collected data of the unroll and SIMD performance and did the early performance plot and early roofline plot. Plotted Figure 5

6.3. Junzhen Lou

In charge of conducting the cost analysis and cache miss analysis of Algorithm 1. Implemented scalar optimizations including double-loop unroll, scalar replacement, and basic

[3]To make the report easier to follow and understand, we divided the algorithms into smaller units and re-did a few experiments.

block optimization. Worked with Yuening on SIMD optimizations of the bottlenecks (calculation of point distances and KNN) in Algorithm 1 and 2. In particular focused on blocking related optimizations and analysis. Created and analyzed performance plots (Figure 3 and 4). Evaluated the tighter roofline based on instruction mix.

6.4. Yijun Ma

Wrote shell scripts for teammates to run tests automatically. Worked with Junzhen on block optimizations; in particular, focused on using `perf` to measure hit and miss counts, and plotted 2. Calculated the cache miss analysis part of Algorithm 2, and plotted 3. Early implementation of SIMD of Shapley function in Algo 2. Attempted sparse matrix optimization.

7. REFERENCES

[1] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nezihe Merve Gurel, Bo Li, Ce Zhang, Costas J. Spanos, and Dawn Song, "Efficient task-specific data valuation for nearest neighbor algorithms," 2019.

[2] Jon L. Bentley and M. Douglas Mcilroy, *Engineering a Sort Function*, 1993.

[3] AI-secure, "Knn-pvldb," `https://github.com/AI-secure/KNN-PVLDB`, 2020.

[4] Jeffrey S. Leon, "Performance of quicksort," `http://homepages.math.uic.edu/~leon/cs-mcs401-r07/handouts/quicksort-continued.pdf`, 2007.

[5] "Intel® core™ i7-8700 processor," `https://web.archive.org/web/20220317101816/https://www.intel.com/content/www/us/en/products/sku/126686/intel-core-i78700-processor-12m-cache-up-to-4-60-ghz/specifications.html`, 03 2022.

[6] Mohammad Alaul haque monil, Seyong Lee, Jeffrey S. Vetter, and Allen Malony, "Understanding the impact of memory access patterns in intel processors," 11 2020.