# FAST COMPUTATION OF SHAPLEY VALUES FOR NEAREST NEIGHBOR ALGORITHMS

*Stefan Jokić, Lucas Cosier, Theresa Wakonig, Olivier Becker*

Department of Computer Science
ETH Zurich, Switzerland

## ABSTRACT

$K$-nearest-neighbor algorithms are commonly used classifiers for machine learning tasks. Their performance highly depends on the data set on which they were trained. Using Shapley values we can determine the usefulness of a new data point w.r.t. a trained model. Yet evaluating the Shapley values for models proves to be computationally expensive.

In this paper we present new implementations for existing Shapley value algorithms over KNN classifiers, which are optimized to achieve minimal runtimes. Using various optimization techniques such as strength reduction, inlining, code motion, separate accumulators, blocking and vector intrinsics, we achieve a speedup of up to 78× for the exact computation and 25× for the MC approximation.

## 1. INTRODUCTION

**Motivation.** Collecting qualitative data for training machine learning models is a time consuming and expensive undertaking. Over the past years, this has lead to an increased interest in the development of data marketplaces where individuals may sell their personal data. To fairly compensate data providers, the value of their contribution relative to the performance of a trained model has to be determined. Jia et al. propose the use of *Shapley values* of the individual data points as a measure of their value [1]. Shapley values have seen wide applications in the domain of fair revenue determination, ranging from power grids [2] to cloud computing [3].

On this basis, the main objective of our work is to achieve minimal runtimes for algorithms that calculate Shapley values for *K-nearest neighbor* (KNN) classifiers. To this end, we apply various optimizations presented during the Advanced Systems Lab course [4].

**Contribution.** In this paper we present fast, novel implementations for calculating Shapley values based on *Algorithm 1* (exact) and *Algorithm 2* (Monte Carlo approx.) from [1]. First, based on the Python implementations [5] provided by the original paper, we implement both algorithms in C, which serve as baselines. Subsequently, we optimize our baseline implementations using various optimization techniques such as precomputation, blocking, strength reduction and vectorization.

**Related work.** Jia et al. introduced three algorithms for the efficient computation of Shapley values in the context of KNN classification and regression [1]. In our work, we focus on two particular variants: one used for the exact computation and one for the Monte Carlo (MC) approximation.

Shapley Values are useful in a wide range of applications. Upadhyaya et al. employed Shapley values to calculate a fair revenue distribution for cloud computing [3]. Bremer et al. applied revenue sharing for power grids using Shapley values [2]. Koutris et al. describe a prototype data marketplace [6]. Arbitrage-free pricing functions for data marketplaces are explored by Lin et al. [7].

**Layout.** Section 2 introduces the background of the algorithms discussed in this paper. In Section 3 we present our proposed optimizations. We discuss our experimental results in Section 4. Ultimately, in Section 5 we draw our conclusions and discuss future work. Section 6 lists the contributions of the individual members.

## 2. BACKGROUND

This chapter introduces the concept of the Shapley value and provides relevant mathematical definitions. Subsequently, the two main algorithms of this project are presented and described. Lastly, a suitable cost measure for analyzing the program implementation is discussed.

**Shapley value.** The Shapley value (SV) is a concept originating from economic game theory, where it is used to describe the average contribution of each individual player to the outcome of a cooperative game. This idea of relative value can readily be transformed to a machine learning (ML) setting. Motivated by the application of data marketplaces, Jia et al. [1] use the SV to capture the relative marginal contribution of each data point to training a performant ML model. In the following, we consider a training data set $\mathcal{D}$ with data points $\{z_i\} \in \mathbb{R}^d$, $i = 1, ..., N$. The Shapley value $s_i$ of a given training point $z_i$ is defined as follows:

$$s_i = \frac{1}{N} \sum_{S \subseteq \mathcal{D} \setminus z_i} \frac{1}{\binom{N-1}{|S|}} [\nu(S \cup \{z_i\}) - \nu(S)] \quad (1)$$

where $\nu(\cdot)$ represents the utility function, which is a performance measure of the model considered. Hence, equation (1) iterates over all subsets $S$ of $\mathcal{D}$, excluding data point $z_i$, sums up the marginal utility of $z_i$ to the respective subsets and averages over the contributions.

An alternative but mathematically equivalent formulation of (1) is given by:

$$s_i = \frac{1}{N!} \sum_{\pi \in \Pi(\mathcal{D})} [\nu(P_i^\pi \cup \{z_i\}) - \nu(P_i^\pi)] \quad (2)$$

where $\pi$ is a permutation from the set of all possible permutations $\Pi(\mathcal{D})$ of the training data set. Moreover, $P_i^\pi$ is the set of data points that precede $z_i$ in permutation $\pi$.

**KNN classification.** The focus of this work lies explicitly on data valuation via the use of Shapley values over *K-nearest neighbor (KNN) classifiers*. The KNN algorithm predicts the class label of a query point according to the label most common among its $K$ nearest neighbors. The utility function $\nu(\cdot)$ used in the scope of data valuation for KNN classifiers is defined as follows:

$$\nu(S) = \frac{1}{K} \sum_{k=1}^{\{K,|S|\}} \mathbb{1}[y_{\alpha_k(S)} = y_{tst}] \quad (3)$$

Equation (3) is also referred to as *KNN utility*. Here, $S$ is a subset of the training data and $y_{\alpha_k}$ refers to the label of the $k^{th}$ nearest neighbor of the query point $x_{tst}$. Moreover, $\mathbb{1}$ is the *indicator function* which evaluates to either 1 or 0 depending on whether the function argument is true or false respectively. Intuitively, the KNN utiltiy describes the likelihood of assigning the correct label to $x_{tst}$. It is important to note that (3) satisfies the *piecewise difference* property. Hence, the expression $\nu(S \cup \{z_i\}) - \nu(S \cup \{z_j\})$ describes the difference in contribution to the model of data points $z_i$ and $z_j$.

**Baseline Algorithm.** A straightforward implementation of (1) would require to iterate over all possible subsets of the training data, leading to $\mathcal{O}(2^N)$ many evaluations of the utility function $\nu(\cdot)$. For the use case of ML models, where training data sets may comprise millions of data points, this exponential complexity is simply not feasible. Prior to the work of Jia et al. [1], the only practical approach to using SVs for data valuation was to compute approximations of the exact values via Monte Carlo (MC) sampling in $\mathcal{O}(N_{tst} \cdot \frac{N^2}{\varepsilon^2} logN log\frac{N}{\delta})$.

In the following subsections we introduce two algorithms presented in [1] which are designed for the specific setting of KNN classifiers. While *Algorithm 1* enables the exact computation of Shapley values in quasi-linear time, *Algorithm 2* yields approximate values and can be seen as an enhanced version of the baseline MC algorithm.

## 2.1. Algorithm 1 - Exact computation of SV

**Input:** Labelled data sets for training and testing, $K$
$X_{trn} \in \mathbb{R}^{N \times d}$, $X_{tst} \in \mathbb{R}^{N_{tst} \times d}$
$Y_{trn} \in \mathbb{R}^N$, $Y_{tst} \in \mathbb{R}^{N_{tst}}$, $K \in \mathbb{N}$
**Output:** Shapley values $s_i$ of all training points
$(i = 1, 2, ..., N)$
**Computational complexity:** $\mathcal{O}(N_{tst} \cdot NlogN)$

For each test point $x_{tst}$, the exact algorithm computes the Euclidean distances to all $N$ training points and sorts them in ascending order. Next, the Shapley values of the training points, which represent the marginal contribution to the classification of the query point $x_{tst}$, are computed recursively according to (4) and (5):

$$s_{\alpha_N} = \frac{\mathbb{1}[y_{\alpha_N} = y_{tst}]}{N} \quad (4)$$

$$s_{\alpha_i} = s_{\alpha_{i+1}} \frac{\mathbb{1}[y_{\alpha_i} = y_{tst}] - \mathbb{1}[y_{\alpha_{i+1}} = y_{tst}]}{K} \frac{min\{K, i\}}{i} \quad (5)$$

where $s_{\alpha_i}$ refers to the SV of the $i^{th}$ nearest neighbor of $x_{tst}$ and $y_{\alpha_i}$ to the corresponding label. The recursive definition can be derived from (1) and follows from the additivity property of the Shapley value and the piecewise difference property of the KNN utility (3). For the interested reader we refer to the proof of the recursive formulation ("Theorem 1") in [1]. Finally, the Shapley value of each training point is found by averaging over the $N_{tst}$ intermediate Shapley values corresponding to the training points' contribution to each of the test points.

Since the algorithm is dominated by the sorting routine, it exhibits a total asymptotic complexity of $\mathcal{O}(N_{tst} \cdot NlogN)$.

## 2.2. Algorithm 2 - Monte Carlo approximation of SV

**Input:** Labelled data sets for training and testing, $K, \widetilde{T}$
$X_{trn} \in \mathbb{R}^{N \times d}$, $X_{tst} \in \mathbb{R}^{N_{tst} \times d}$,
$Y_{trn} \in \mathbb{R}^N$, $Y_{tst} \in \mathbb{R}^{N_{tst}}$, $K, \widetilde{T} \in \mathbb{N}$
**Output:** Shapley value estimators $\hat{s}_i$ of all training points
$(i = 1, 2, ..., N)$
**Computational complexity:** $\mathcal{O}(N_{tst} \cdot \frac{N}{\varepsilon^2} logK log\frac{K}{\delta})$

The MC algorithm computes (2) via random sampling and hence only considers a subset of all possible permutations $\Pi(\mathcal{D})$. It computes an $(\varepsilon, \delta)$- approximation of the exact Shapley values with error bound $\varepsilon$ and confidence $1 - \delta$. In contrast to the baseline algorithm, this improved MC algorithm uses a much tighter bound on the number of permutations $\widetilde{T}$, which greatly improves the runtime. For the special case of unweighted KNN classifiers, $\widetilde{T} \geq \frac{1}{K^2\varepsilon^2} log \frac{2K}{\delta}$ holds. Moreover, the algorithm makes use of a *maxheap* to maintain the $K$ nearest neighbors to the query point.

Per test point $x_{tst}$, $\widetilde{T}$ permutations of the training data set are considered by shuffling the indices uniformly at random. For every permutation, a new *maxheap* of length $K$ is created. The root node always holds the currently closest neighbor to $x_{tst}$. Iterating over all neighbors $i = 1, ..., N$, the distance to the query point is computed and compared to the root element. A data point is only ever inserted into the heap if the heap is not yet full, or a new root element (nearest neighbor) has been found. In this case, the current value of the utility difference $\phi_i^{\pi_t}$, defined as

$$\phi_i^{\pi_t} = \nu(P_i^{\pi_t} \cup \{z_i\}) - \nu(P_i^{\pi_t}) \tag{6}$$

is updated. Finally, the SV approximations are obtained by averaging over all permutations and test points.

For a heap of length $K$, insertion costs $\mathcal{O}(logK)$. Together with the bound on $\widetilde{T}$, the total asymptotic time complexity is $\mathcal{O}(N_{tst} \cdot \frac{N}{\varepsilon^2} logK \, log\frac{K}{\delta})$ which depends mostly on the number of permutations $\widetilde{T}$.

**Cost Measure.** The cost measure used for performance analysis comprises the number of arithmetic and logical floating point operations performed. More precisely, floating point addition, subtraction, multiplication, division and comparison are considered as single flops and FMA (fused multiply-add) instructions are counted as two flops.

$$\begin{aligned} C(N) = C_{add} \cdot N_{add} + C_{sub} \cdot N_{sub} + C_{mul} \cdot N_{mul} + \\ C_{div} \cdot N_{div} + C_{fma} \cdot N_{fma} + C_{cmp} \cdot N_{cmp} \end{aligned} \tag{7}$$

For the analysis of our program implementations we determine the cost *empirically* using the Intel® Advisor tool [8], cf. Section 3 Profiling.

## 3. METHODOLOGY

In the following, we describe the rigorous methodology involved in implementing, testing, benchmarking and refining our optimizations for both aforementioned algorithms.

**Assumptions.** We consider that the input matrices ($X_{trn} \in \mathbb{R}^{N \times d}$ and $X_{tst} \in \mathbb{R}^{N_{tst} \times d}$) are square, of the same size, i.e. $N = N_{tst} = d$, and *dense*. We assume a single data point per seller setting. Since the authors of the provided algorithms [1] employ single-precision floating-point values as input, our optimizations were also implemented using `floats`. Finally, we assume that any of the dimensions of the matrices are divisible by 32, which is a requirement for our $l_2$-norm SIMD implementations. In the following sections, we refer to *Algorithm 1* (cf. Section 2.1) and *Algorithm 2* (cf. Section 2.2) as $A1$ and $A2$, respectively.

**Correctness.** All optimizations were tested for correctness against their Python counterparts. The Python implementations of $A1$ and $A2$ were taken from [5]. We deem the output of any of our C implementations (including optimized ones) to be correct if each value in the output $c_{out}$ satisfies the following inequality:

| Algorithm | Subroutine | Description |
|---|---|---|
| $A1$ & $A2$ | $l_2$-norm | Euclidean distance |
| $A1$ | argsort | sorts array and returns indices |
| $A1$ | get_true_knn | computes the $N_{tst} \times N$ KNN matrix |
| $A1$ | compute_shapley | computes the Shapley values based on the KNN matrix |
| $A2$ | max-heap | maintains the K nearest neighbors |
| $A2$ | col_mean | column-wise mean of a matrix |
| $A2$ | knn_utility | evaluates the utility function |

**Table 1**. Overview of identified subroutines in $A1$ & $A2$

$$|p_{out} - c_{out}| \leq (t_{abs} + t_{rel} \cdot |c_{out}|) \tag{8}$$

where $p_{out}$ is the corresponding output value of the Python reference implementation, $t_{abs} = 10^{-6}$ is the absolute tolerance and $t_{rel} = 10^{-7}$ is the relative tolerance. Additionally, to make the comparison between Python and C outputs easier, we only considered *stable* sorting algorithms for $A1$. For $A2$, the random seed was set to the same value for both the Python and C implementations.

**Profiling.** Program analysis was performed using Intel® Advisor [8], a design and analysis tool for developing performant code. This tool enabled us to measure the work $W$ in terms of single-precision flops, the number of bytes $Q$ transferred between last level cache (LLC) and memory (DRAM) during loads and stores, as well as the runtime $T$. From these quantities, the performance was computed as $P = \frac{W}{T}$ and the operational intensity as $I = \frac{W}{Q}$, which were then used to create roofline plots. This came in handy especially for $A2$, as the MC approach is per definition non-deterministic and does not enable us to retrieve a formula for the flop count in advance. However, approximate pen and paper flop counts were used to perform sanity checks on the numbers provided by the tool (while fixing the random seed in $A2$). For the runtime plots however, we made our own runtime measurements in cycles using the `RDTSC` instruction.

**Optimization.** After implementing the C baseline for both $A1$ and $A2$, the runtime of all subroutines was measured in order to detect the bottlenecks of our implementations. Subroutines identified in $A1$ and $A2$ are shown in Table 1. Next, each subroutine was optimized with respect to the following criteria: 1) *work* and 2) *memory traffic*, where work refers to the number of flops and memory traffic quantifies the number of bytes transferred between LLC and DRAM. In the following, we delve deeper into these two categories, explaining the considerations that went into optimizing the aforementioned subroutines. Based on this, we generated four optimizations for $A1$ and two for $A2$.

**Work.** By inspection of $A1$ and $A2$, a noticeable part of the algorithms reduces to performing comparisons, or logical flops. This is a direct consequence of the KNN classification algorithm and the respective utility function (3) used. Specifically, the sorting subroutine in $A1$ performs comparison based sort on $N_{tst}$ arrays of length $N$ to find the nearest neighbors. Likewise, the maxheap in $A2$ makes extensive use of comparison operations to maintain the heap property at insertion of a new element. However, besides logical comparisons, we argue that a large part of the runtime will be spent computing the Euclidean distance between test and training observations. We assume this to be a crucial bottleneck for both algorithms, as it is the subroutine performing the majority of arithmetic flops.

**Euclidean distance.** Our baseline of the Euclidean distance is a straightforward implementation of the mathematical definition and includes calls to the `pow()` and `sqrt()` math library functions. In a first optimization step, the call to `pow()` is replaced by a `mul` operation and basic rewriting as well as scalar replacement is performed. Realizing that the distances between two observations are only ever needed in relative terms, leads us to omitting the call to `sqrt()` and using the value of the squared distance instead. Since $\sqrt{a} > \sqrt{b} \implies a > b$ for $a, b \in \mathbb{R}_{\geq 0}$, this transformation is valid. Using one `sub`, one `add` and one `mul` per loop iteration, this implementation yields a cost of $3N$ flops for input arrays of length $N$. Next, loop unrolling is performed. The empirically determined, best performing *unrolling factor* of 8 coincides with the theoretical result according to $\#accumulators = \lceil lat \cdot tp \rceil = \lceil 4 \cdot 2 \rceil = 8$. Through unrolling we are able to use separate accumulators and create independent strands of operation. We expect this optimization to yield an increase in performance as more instruction level parallelism (ILP) improves the throughput of the computation. Finally, we proceed to implement an analogous SIMD variant using AVX2 intrinsics, thereby explicitly mapping to `fmadd` instructions. We perform unrolling by a factor of 32, processing 4 vectors of 8 elements at a time, for each of the two arrays. Moreover, another important matter in optimizing the squared distance was to preserve a certain order of computation, despite separate accumulators, as associativity does in general not hold for floating-point arithmetic.

## 3.1. Algorithm 1: Analysis and Optimizations

**Memory traffic.** In $A1$, data movement can be broken down into three parts: **1)** First, row-wise accesses to data points from the test and train matrices (representing $d$-dimensional feature vectors) are performed for computing the $l_2$-norm between each observation in the test and training set. **2)** Then, once the Euclidean distances have been computed, they are sorted. **3)** Lastly, the indices of these sorted distances are copied into a matrix to form the KNN matrix ($N_{tst} \times N$).

Parts **1)** and **3)** correspond to the subroutine `get_true_knn`, and part **2)** corresponds to `argsort`. Out of these three, we expect the first part to take up the majority of the runtime.

**Row-wise matrix accesses.** Since $A1$ is *exact*, the Shapley value computation is *exhaustive*. For each test data point, *all* training data points are considered. Since for one iteration of the outer $N_{tst}$-loop the test data point is reused to compute its $l_2$-distance to each train data point, the naive C implementation already benefits from temporal locality on the test row. However, since the computation is exhaustive, one can consider reusing the test row for the computation over multiple training rows, or vice-versa. In this case we achieve temporal and spatial locality on the test row, since the same values are accessed multiple times. One can then consider multiple train and test rows, such that values in both are revisited multiple times while computing the $l_2$-norm.

With those observations in mind, we note that naive unrolling can be beneficial. Through naive unrolling, accessing the train and test rows would result in both temporal and spatial locality. Additionally, we can reduce data movement by passing rows directly by reference to the $l_2$-norm function instead of copying them into memory first. While copying in memory isn't detrimental for small input sizes, it quickly becomes sub-optimal once the input does not fit in cache. Therefore, for large $N$ referencing is desirable.

**Sorting and forming the KNN matrix.** For `argsort`, we tested multiple sorting variants on the sparse datasets provided in [5]. Stability was one of our main considerations. Another was runtime complexity: since we optimize for performance and are agnostic to the inputs, we require worst case complexity of $\mathcal{O}(N \log N)$. Therefore, we focused our attention on mergesort [9], as it fulfills both of our requirements.

For argsort, we sort two arrays: one for the indices and one for the values. Therefore, one can create two separate arrays or one array of structures (AoS), with the indices and Euclidean distances interleaved. This has the advantage that we can use sorting algorithms with a function signature similar to *qsort* without the need to modify the function body to be suitable for `argsort`, which can be burdensome for optimized implementations. Additionally, the penalty of using a *stride of two* to access values and copy the indices into the KNN matrix is minimal, as documented in Section 4.

**Scalar blocking.** Through naive unrolling, the size of the working set grows as a function of the unrolling factor and the length of the rows of each matrix. This limits the amount of improvement one can achieve through unrolling, as locality will worsen for unrolling factors that are too large. Instead, we can do better by changing the description of part **1)**: instead of *row-wise* accesses to compute the

**Code Snippet 1** Scalar Blocking

```
// Nb: block size
for $bi = 0;\ bi < N_{tst};\ bi \mathrel{+}= Nb$ :
    for $bj = 0;\ bj < N;\ bj \mathrel{+}= Nb$ :
        for $bk = 0;\ bk < d;\ bk \mathrel{+}= Nb$ :
            for $i = bi;\ i < Nb + bi;\ i \mathrel{+}= 1$ :
                for $j = bj;\ j < Nb + bj;\ j \mathrel{+}= 1$ :
                    for $k = bk;\ k < Nb + bk;\ k \mathrel{+}= 1$ :
                        $C[i][j] \mathrel{+}= (A[i][k] - B[j][k])^2$
```

$l_2$-norm, the computation can be *blocked* to achieve better locality. We can delay parts **2)** and **3)** and consider **1)** as the *precomputation* stage. We can accumulate the individual squared difference sums in the expression of the $l_2$-norm.

In fact, scalar blocking strongly resembles matrix-matrix multiplication (MMM) and can be adapted with minor modifications. First, we need to change the performed operation. In each iteration of MMM, we sum up the product of one element from the first matrix with an element from the second matrix. In our adapted version, we sum up the squared difference between the two elements. The second adaption consists in changing the order in which elements are accessed. In MMM, we multiply one row from the first matrix with a column form the second matrix. To correctly calculate the $l_2$-norm, we need one row from the first matrix (test) and one row from the second matrix (train). Both modifications are shown in Code Snippet 1.

**SIMD blocking.** Next, we implement a SIMD variant of scalar blocking, which further increases ILP and reduces memory traffic. To use SIMD instructions for blocking, further considerations have to be taken into account. First, we increase ILP by unrolling and using separate accumulators. Unrolling in this case refers to both the (outer) $N_{tst}$ and (inner) $N$-loops. Then, similar to what we did in scalar blocking, we compute the partial Euclidean distances in each block of a given size using AVX2 intrinsics and accumulate. In SIMD instructions, this can be expressed on two vectors as one `vsubps` to compute the element-wise differences, one `fmadd` to compute the square of the differences, and finally one horizontal add of the accumulator vector (partial reduction of a *single* vector). Since there is no intrinsic equivalent to the horizontal add of one vector, it is actually a collection of intrinsics. Further, we trial two approaches for parts **2)** and **3)**: we either separate **1)** from **2)** and **3)** by precomputing the distances like in scalar blocking, or we sort and store on the fly, keeping all parts interleaved.

### 3.2. Algorithm 2: Analysis and Optimizations

**Memory traffic.** Similarly to $A1$, there is a large amount of data movement in $A2$ as a result of having multiple accesses to the rows of the train and test matrices. As in $A1$, the rows of the test matrix are accessed in sequential order. In contrast, however, in $A2$ the row-wise access pattern within the train matrix is non-deterministic, as new permutations of training set indices are generated as part of the MC-approximation. As a result, it is difficult to optimize $A2$ with regard to the data movement involved in accessing training observations in an unpredictable fashion. Therefore, we shifted our attention in optimizing the main data structure employed in maintaining the KNN: the *max-heap*.

**Heap.** In our baseline implementation, the heap is implemented as a `struct` comprising an integer array that represents the values of the heap itself, an array holding a copy of the values of the current test observation (with which the $l2$-norm is computed), a pointer to the train matrix and an integer counter to maintain the current size of the heap. Next, the heap consists of the functions `up` and `down` which are called recursively when an element is inserted using the function `insert`, so as to restore the heap property. To optimize the heap, we first replaced its `struct` by just a single array. In this way, we no longer have this complicated data structure that holds various arrays, pointers and counters. The next step was to inline the heap, which enabled further optimizations: elements are inserted into the heap iteratively and the loop variable itself can now be used in place of the heap's own counter. This also enabled the removal of several `if`-branches that depend on the counter's value. Lastly, we replaced the recursive approach of restoring the heap property by an *iterative* one using while loops. This completely removes the overhead of having to use the call stack to allocate space for the local variables of the large number of recursive `up` and `down` function calls.

**Column mean.** The elements needed to calculate the mean of a column are in different cache blocks as matrices are stored in row-major order, assuming the matrices are large enough. As a result, each access results in a cache miss. To reduce the number of cache misses, we calculate the mean of multiple columns at the same time by unrolling the loop and using separate accumulators. Not only does this increase the *spatial locality* of the `col_mean` computation, but it also increases its ILP. We expand on this idea of simultaneous calculation and further optimize the performance of `col_mean` by using AVX2 intrinsics. We load the values of 8 subsequent columns from various rows into multiple vectors and add them up. Once this has been repeated for all rows, we divide each element of the vector containing the sums by the column size.

## 4. EXPERIMENTAL RESULTS

In this chapter the effects of the aforementioned optimizations on our baseline implementation are presented.

**Experimental setup.** All measurements were made using the Intel® Core™ i7-11700K 8-core processor which boasts a base frequency of 3.6 GHz (up to 5 GHz with Turbo

| Algorithm | Name | Subroutines Optimized |
|---|---|---|
| $A1$ | opt 1 | $l_2$-norm |
| $A1$ | opt 2 | opt 1 + argsort |
| $A1$ | opt 3 | opt 2 + get_true_knn (unrolling + referencing) + compute_ shapley |
| $A1$ | opt 4 | get_true_knn (blocking) |
| $A2$ | opt 1 | $l_2$-norm |
| $A2$ | opt 2 | opt 1 + heap + col_mean + knn_util. |
| $A1$ flags: | -O3 -mfma -fno-ipa-cp-clone | |
| $A2$ flags: | -O3 -ffast-math -march=native -mfma | |

**Table 2**. Modified subroutines per *opt* and compiler flags

Boost). It comprises three cache levels: L1 (48 KB, 12-way set assoc.), L2 (512 KB, 8-way set assoc.), and L3 (16 MB, 16-way set assoc.). Compilation was carried out using GCC 7.5.0 on the Ubuntu 18.04 OS. Turbo Boost was disabled. In our experiments we vary $N$, fix $K = 3$ and consider $\widetilde{T} = 32$ permutations in the MC algorithm.

**Experiments.** We follow an iterative optimization approach and collect runtime and performance measurements for every major stage of the process. Table 2 provides an overview of the individual optimization steps and lists the compiler flags used. Additionally, all scalar implementations were compiled with `-fno-tree-vectorize` and the SIMD versions with `-mavx2`. The `-fno-ipa-cp-clone` flag was used for quadsort to remain stable.

**Bottlenecks.** The identification of bottlenecks is crucial in every stage of the optimization process. The bottlenecks of the SIMD implementations (identical for scalar variants) of $A1$ and $A2$ are depicted in Figure 1. First, we confirm our hypothesis that the $l_2$-norm, which is shared between $A1$ and $A2$, represents the largest bottleneck in terms of runtime (90.5% and 47.1% of total runtime of $A1$ and $A2$ baselines, respectively). Note that the baseline implementations are not displayed in Figure 1. Thus, our $l_2$-norm optimization is the first major optimization in both algorithms, namely $A1$ *opt 1* and $A2$ *opt 1*.

***A1*&*A2* opt 1.** The optimizations performed on the $l2$-norm subroutine break the sequential dependency of the computation and enable ILP. The increase in throughput can be observed in Figure 2, as the performance w.r.t. the baseline improves by a factor of 2.3 for scalar and 18 for SIMD implementations. Both implementations reach $1/8$ of their respective peak performance. Since the SIMD instructions process vectors of 8 `floats`, theoretically the vectorized version should perform 8 times more flops per cycle than the scalar implementation. Our measurements in Figure 2 support this theoretical result. It is worth noting that, for big $N$ the operational intensity of *opt1* starts to decrease and there is a visible performance drop for $N > 2048$. This behavior hints at the implementation becoming memory bound as the input's size reaches more than 33 MB. As this observation is coherent with the LLC size of the processor (16 MB),

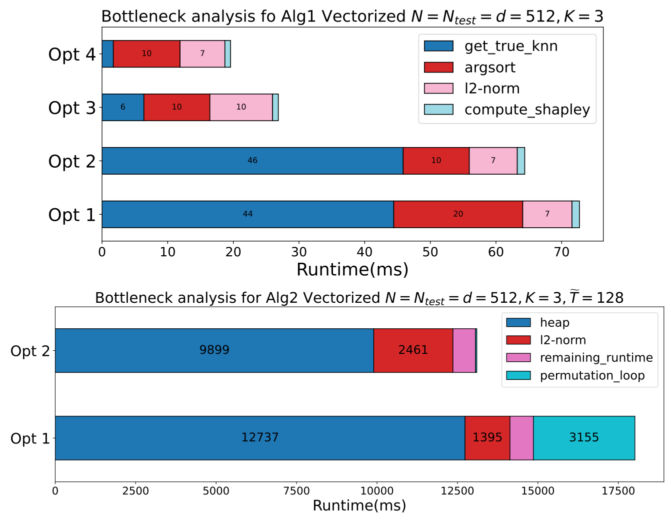

**Fig. 1**. Runtimes of subroutines in $A1$ & $A2$ (SIMD)

the performance drop can be attributed to exceeding the L3 capacity (more time spent transferring data from DRAM to processor). Just through optimizing the $l2$-norm we achieve a maximum speedup of up to 6.5x w.r.t. the C baseline.

## 4.1. Algorithm 1

**Roofline model.** The roofline plot in Figure 2 shows that the baseline and our optimized implementations are compute bound. *opt 2 (SIMD)* however, becomes memory bound for input sizes of $\geq 2^{11}$ ($\approx$ 33.57 MB), as the input no longer fits in LLC. The fact that *opt 3* and *opt 4* do not become memory bound at larger input sizes highlights their improved locality. The baseline has an average performance of $0.2$ flops/cycle across all input sizes, reaching 5% of peak scalar performance. The best scalar and vectorized optimizations (*opt 4*, blocking) achieve 46.3% of peak scalar performance with $P \approx 1.85$ flops/cycle and 40.6% of vector SP speak with $P \approx 13$ flops/cycle, respectively.

***A1* opt 2.** The second largest bottleneck we target is `argsort` (cf. Figure 1). In total two sorting algorithms were benchmarked: *klib's mergesort* [10] and *quadsort* [11], which are fast mergesort implementations. *Klib's mergesort* was on average 10-20% slower than quadsort. As can be seen in Figure 1, for input sizes which fit in cache we measure up to $2\times$ speedup with quadsort over our baseline mergesort implementation (in *opt 1*). However, the benefit of quadsort diminishes with increasing $N$ (cf. Figure 3). Populating the KNN matrix directly by turning it into an AoS also proved suboptimal. Although there would be $N \cdot N_{tst}$ less copies of indices, the working set is increased by $2N \cdot N_{tst} - N \cdot N_{tst} - 2 \cdot N = N \cdot (N_{tst} - 2)$. As the distance values are no longer needed after sorting the indices, an additional step for improvement is to discard them.

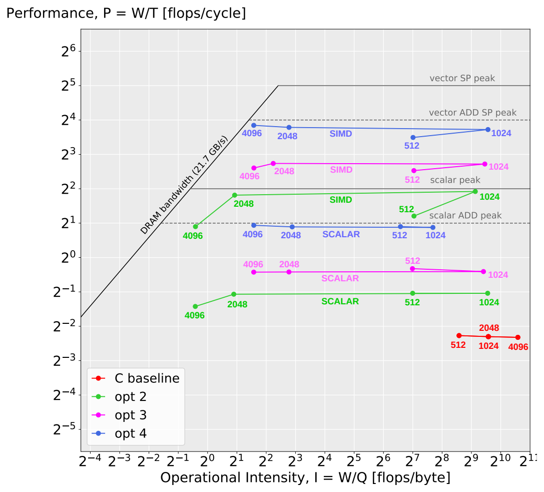

**Fig. 2**. Roofline plot for $A1$ optimizations

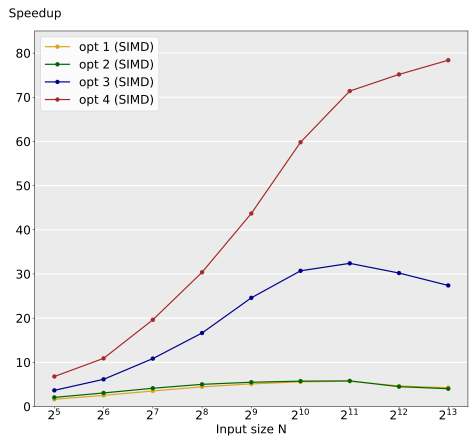

**Fig. 3**. Speedup plot for $A1$ optimizations (SIMD)

***A*1 opt 3.** The largest remaining bottleneck is the `get_true_knn` routine (cf. Figure 1). For this optimization, our hypothesis that naive unrolling would improve locality was invalidated. Unrolling the outer and inner loops ($N_{tst}$-loop and $N$-loop) by a factor of *Nb* and *Mb* respectively, increases the working set by *Nb* $\times$ *Mb* arrays of length $N$ that need to be sorted, plus the *Nb* + *Mb* test and train rows that need to be kept in memory. Hence, the input size at which the working set no longer fits in L3 cache diminishes quickly and any potential gains vanish. Furthermore, referencing rows by pointers was detrimental for small input sizes compared to copying into memory. However, as soon as the working set no longer fits in cache ($N \geq 2^{11}$), copying becomes suboptimal as arrays are evicted at every iteration and need to be brought back to memory. Our testing revealed that combining both of these ideas is superior. By referencing, the size of the working set now decreases by *Nb* $\times$ *Mb*, and we can achieve the desired spatial and temporal locality through unrolling. Moreover, standard C optimizations in the `compute_shapley` routine, such as loop unrolling and strength reduction were performed. As this subroutine is the smallest bottleneck (cf. Figure 1), the optimizations have only very little impact on runtime. For testing purposes we tried unrolling the $N_{tst}$ and $N$-loops by a multitude of factors. Overall, using $Nb = 4$, $Mb = 1$ achieves the best performance. In total, *opt 3* reaches approx. 32$\times$ speedup for an input size of $2^{11}$. Further, we observe in Figure 3 that the speedup of *opt 3* drops after an input size of $2^{11}$, as larger inputs no longer fit into L3 cache.

***A*1 opt 4.** Finally, we apply a different approach to optimizing `get_true_knn`: blocking. We tried various block sizes, ranging from $2 \times 2$ to $16 \times 16$, non-squares included. Our experiments reveal that computing the distances between all rows in a given block and directly accumulating its results provides inferior improvements compared to *opt 3*. Instead, via code motion we postpone the summation reduction until after the block computation is finished. Further, allocating temporary arrays and immediately sorting after the block $l_2$-norm computation is finished is preferred over precomputing the distances and then sorting and copying, especially for large $N$, when the Euclidean distances cannot be kept in cache anymore. We observed the best results with a block size of $4 \times 8$. As seen in Figure 3, *opt 4* yields up to 78$\times$ speedup over the C baseline and improves the performance w.r.t. *opt 3* by a factor of ~2.3, as displayed in the roofline plot in Figure 2. Increasing the block size further leads to suboptimal results, arguably because the working set no longer fits in LLC. At this point the majority of the runtime is spent sorting the distances, as reflected in Figure 1.

## 4.2. Algorithm 2

**Roofline model.** The roofline plot for $A2$ in Figure 4 shows that the baseline, scalar and vectorized implementations are compute bound. The iterative approach to the heap in *opt 2* achieves slightly higher performance than the recursive heap. With the $l_2$-norm optimization (*opt 1*) and using SIMD instructions, an increase of $\sim 0.15$ flops/cycle is achieved on average. The baseline has an average performance of 0.22 flops/cycle across input sizes, reaching 5.5% of peak scalar performance. The best scalar and vectorized optimizations (*opt 2*, heap) achieve 49.5% of peak scalar performance with $P \approx 1.98$ flops/cycle and 14.7% of vector SP speak with $P \approx 4.7$ ($N \geq 256$) flops/cycle, respectively. The smaller increase in performance from scalar to SIMD implementations in comparison to $A1$ is due to less subroutines implemented in AVX2 intrinsics (only $l2$-norm and column mean).

***A*2 opt 2.** We identified the heap as the largest bottleneck after applying the $l_2$-norm optimization (cf. Figure 1).

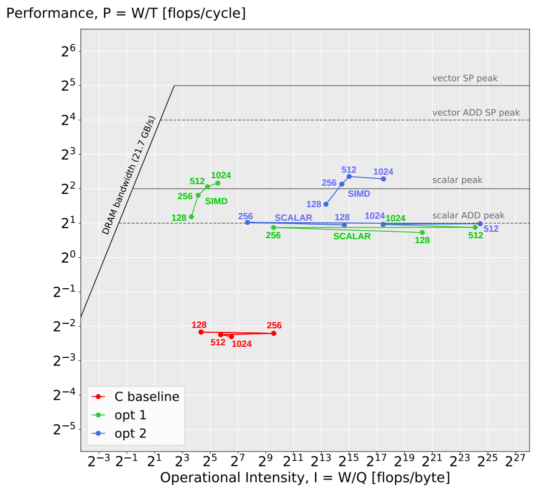

**Fig. 4**. Roofline plot for $A2$ optimizations

We combine the iterative approach to the heap with our optimizations for computing the column mean (cf. Section 3.2). Additionally, we inline the KNN utility function which enables minor optimizations such as removing the need to copy data into a temporary array to be passed to the function. However, the KNN utility and column mean constitute an insignificant ($\ll\sim$1%) portion of the runtime. Hence, the reduction in runtime from *opt 1* to *opt 2* is primarily due to the heap optimization. The iterative implementation has resulted in better memory usage due to not requiring the call stack anymore. Furthermore, in combination with the inlining, analysis of the assembly code reveals that the compiler is capable of performing further optimizations when `-ffast-math` is used, which allows to reorder independent instructions. At an input size of $2^{10}$, *opt 2* achieves a speedup of up to $\sim$27$\times$ over the baseline. We can see the effect of cache sizes in Figure 6 as the speedup drops twice. First, for an input size of $2^8$ at which point the input does not fit into L2 cache anymore. Then, a second time, for an input size of $2^{10}$ at which point the input does not fit into L3 cache anymore.

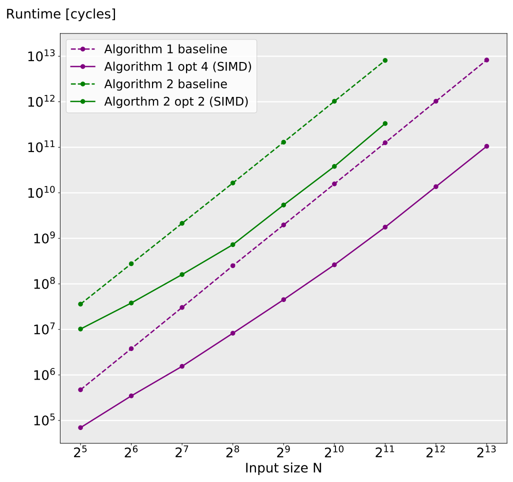

**Fig. 5**. Runtime plot for baselines and best optimizations

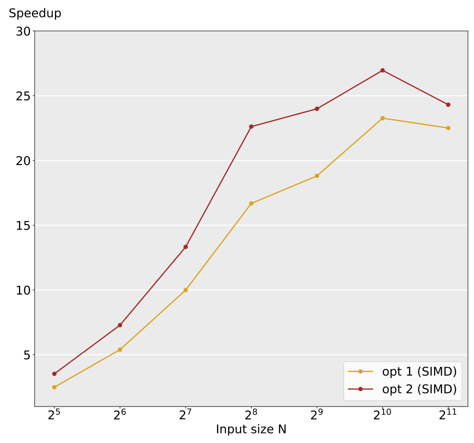

**Fig. 6**. Speedup plot for $A2$ optimizations (SIMD)

**Runtime.** Figure 5 depicts the runtime of our baseline implementations together with the best performing SIMD optimizations for both algorithms. For $N \geq 2^8$, where the advantages of L2 cache locality vanish, the relationship between runtime and input size converges to a constant factor: as $N$ doubles, $t_{cycles}$ increases eightfold. Overall, Figure 5 displays that our optimized versions reduce the runtime of the C baseline by two orders of magnitude.

## 5. CONCLUSION

Our experiments reaffirm the results of the paper by Jia et al. [1]. Not only is the proposed exact approach faster, it is also easier to implement and optimize compared to the MC variant. For our best optimizations, we measure speedups of up to 78$\times$ for $A1$ ($N = 2^{13}$) and 27$\times$ ($N = 2^{10}$) for $A2$ over the baselines. For $A1$ we exploit temporal and spatial locality together with vectorization to reach 40.6% of vector FP peak and become bounded by sorting. Because of this, one natural area of future work would be to exploit SIMD sorting. $A2$'s performance, reaching 14.7% of vector FP peak, is limited due to the choice of data structure, which impedes further optimizations. However, the combination of inlining and an iterative approach now enables the vectorization of the while loops.

Additionally, improvements can be achieved with further compiler flag refinement, which is an area we did not explore in this paper. For compiler flag auto-tuning, one could employ a modern approach such as Bayesian Optimization as described in [12], with a search space for flags similar to that in matrix-matrix multiplication.

## 6. CONTRIBUTIONS OF TEAM MEMBERS

**Stefan.** Implemented C baseline version for $A1$ with Lucas. Focused on optimizing the heap for $A2$. Worked with Lucas on SIMD blocking. Created the roofline, speedup and runtime plots with Theresa and Olivier.
**Lucas.** Implemented C baseline version for $A1$ with Stefan. Focused on optimizing the sorting algorithm in $A1$. Optimized the get_true_knn function for scalar and SIMD implementations in $A1$. Worked with Olivier on scalar blocking for $A1$. Worked on SIMD blocking with Stefan. Created the bottleneck plots.
**Theresa.** Implemented C baseline version for $A2$ with Olivier. Focused on optimizing the L2-Norm for scalar and SIMD implementations. Optimized the compute_shapley routine. Created the roofline plots with Stefan.
**Olivier.** Implemented C baseline version for $A2$ with Theresa. Focused on optimizing the col-mean in $A2$ for scalar and SIMD implementations. Worked with Lucas on scalar blocking for $A1$. Created speedup plots with Stefan.

## 7. REFERENCES

[1] R. Jia *et al.*, “Efficient task-specific data valuation for nearest neighbor algorithms,” *CoRR*, vol. abs/1908.08619, 2019. [Online]. Available: http://arxiv.org/abs/1908.08619

[2] J. Bremer and M. Sonnenschein, “Estimating shapley values for fair profit distribution in power planning smart grid coalitions,” in *Multiagent System Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–221.

[3] P. Upadhyaya, M. Balazinska, and D. Suciu, “How to price shared optimizations in the cloud,” *CoRR*, vol. abs/1203.0059, 2012. [Online]. Available: http://arxiv.org/abs/1203.0059

[4] M. Püschel and C. Zhang, “Advanced Systems Lab,” Lecture at ETH Zurich, 2022. [Online]. Available: https://acl.inf.ethz.ch/teaching/fastcode/2022/

[5] R. Jia *et al.*, “Python implementations of Algorithm 1 and Algorithm 2.” [Online]. Available: https://github.com/sunblaze-ucb/data-valuation/

[6] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu, “Querymarket demonstration: Pricing for online data markets,” *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1962–1965, aug 2012. [Online]. Available: https://doi.org/10.14778/2367502.2367548

[7] B.-R. Lin and D. Kifer, “On arbitrage-free pricing for general data queries,” *Proc. VLDB Endow.*, vol. 7, no. 9, p. 757–768, may 2014. [Online]. Available: https://doi.org/10.14778/2732939.2732948

[8] “Intel® Advisor: Design code for efficient vectorization, threading, memory usage, and accelerator offloading.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

[9] D. Knuth, *The Art of Computer Programming*, 2nd ed. Addison-Wesley, 1998, vol. 3.

[10] “Klib: a generic library in c.” [Online]. Available: https://github.com/attractivechaos/klib

[11] “Quadsort: stable bottom-up adaptive branchless merge sort.” [Online]. Available: https://github.com/scandum/quadsort

[12] J. Chen, N. Xu, P. Chen, and H. Zhang, “Efficient compiler autotuning via bayesian optimization,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1198–1209. [Online]. Available: https://ieeexplore.ieee.org/document/9401979