IMPLEMENTING FAST VECTORIZED SINGLE-CORE SHAPLEY VALUE CALCULATION

Jonas Althaus [*], *Patrick Eigensatz* [*], *Christopher Meier* [*], *Nicolas Muntwyler* [*]

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

With recent interest in Shapley values within the Machine Learning community, its computation has become increasingly important. Since their calculation is computationally expensive, having a high-performance implementation is beneficial. In this work, we present highly-optimized single-core implementations of recent algorithmic efforts addressing Shapley value calculations in the K-nearest neighbour setting. We examine two algorithms, provide their baseline implementations and apply a multitude of optimization techniques. We then study their effect on the runtime and identify their limitations. Finally, a fully optimized implementation is proposed for both algorithms.

1. INTRODUCTION

Motivation. Data commoditization has been increasingly witnessed in recent years. Since the quality of data makes or breaks the performance of many Machine Learning applications, there has been a lot of interest in creating data marketplaces. On such marketplaces, data consumers may access the data provided by multiple individuals with ease and use it to train their individual models. This, however, brings up the critical challenge of rewarding each data contributor with an adequate revenue. One possible solution is to use Shapley values (SV) as a revenue allocation scheme. Each data point receives a SV indicating its helpfulness for the overall model. Unfortunately, SV are known to be computationally heavy [1]. Therefore, an efficient implementation to calculate SV even for larger data sets is important.

Contribution. Algorithmic improvements with a significant runtime complexity reduction for SV calculation in the case of K-nearest neighbour (KNN) models were introduced by Jia et al. [2]. We implement their algorithms in C in order to identify various performance bottlenecks. We then present highly optimized and vectorized single-core implementations. Additionally, we show through various theoretical analyses why certain optimizations lead to better results and where limitations exist.

Related work. While using Shapley values for revenue sharing has been used in various applications [3, 4], scaling to larger data sets has been a standing problem. Only recently Jia et al. [2] have proposed an efficient algorithm for SV calculation in the case of K-nearest neighbour [5] models. There exist a plethora of methods for fast KNN algorithms [6, 7], which also include GPU support. However, the proposed algorithms from Jia et al. [2] differ from a simple KNN calculation. To be more precise, one particular subproblem is to calculate a full KNN (i.e where all neighbours are calculated, hence $K = N$). For this case, the prior mentioned KNN implementations that often use a variation of a KD-tree don't perform particularly well.

2. BACKGROUND ON THE ALGORITHMS

In this section, we provide a brief overview of what Shapley values are, why they're important and introduce two algorithms that calculate the Shapley values in different ways.

2.1. Shapley values

Shapley values provide a unique allocation scheme for revenue sharing in various applications. Given the data point of interest, the corresponding Shapley value provides a unique measure of the marginal improvement of utility over all possible subsets of data points. For a given utility function $v(\cdot)$, the SV for the i-th data point $z_i \in D$ is defined as:

$$s_i := \frac{1}{N} \sum_{S \subseteq D \setminus z_i} \frac{1}{\binom{N-1}{|S|}} \left[v(S \cup \{z_i\}) - v(S) \right] \tag{1}$$

or equivalently:

$$s_i := \frac{1}{N!} \sum_{\pi \in \Pi(D)} \left[v(P_i^\pi \cup i) - v(P_i^\pi) \right] \tag{2}$$

where $\Pi(D)$ is the set of all possible permutations of data points and P_i^π is the set of data points which precede the i-th point in π.

[*] The authors share equal contribution

2.2. Exact SV Algorithm

In order to understand the SV computation, we will first introduce unweighted KNN classifiers. Given a data set consisting of the training data D and the test data D_{test}, unweighted KNN works as follows:
For each single testing point x_{test} and its corresponding label y_{test}, output the probability $P[x_{test} \rightarrow y_{test}]$, by finding the K-nearest training points $(x_{\alpha 1}, ..., x_{\alpha_K})$ regarding x_{test} in the feature space. The probability then is given by $P[x_{test} \rightarrow y_{test}] = \frac{1}{K}\sum_{k=1}^{K} \mathbb{1}[y_{\alpha_k} = y_{test}]$, where α_k is the index of the k-th nearest neighbor.
The utility of a KNN classifier on a subset S can then be defined as the likelihood of the right label:

$$v(S) = \frac{1}{K} \sum_{k=1}^{min(K,|S|} \mathbb{1}[y_{\alpha_k(S)} = y_{test}] \tag{3}$$

and may be easily extended for multiple test points by using the average of the SV for every single test point. Jia et al. [2] show that the SV for unweighted KNN classifiers can be calculated recursively as follows:

$$s_{\alpha_N} = \frac{\mathbb{1}[y_{a_N} = y_{test}]}{N}$$

$$s_{\alpha_i} = s_{\alpha_{i+1}} + \frac{\mathbb{1}[y_{\alpha_i} = y_{test}] - \mathbb{1}[y_{\alpha_{i+1}} = y_{test}]}{K} \frac{\min\{K, i\}}{i}$$

As a direct product of the recursion, Jia et al. [2] proposes an efficient way to compute the exact SV for unweighted KNN in Algorithm 1.

Algorithm 1 Exact SV for an unweighted KNN classifier

```
Input: Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$, Test data
  $D_{test} = \{(\bar{x}_i, \bar{y}_i)\}_{i=1}^{N_{test}}$
Output: The SV $\{s_{j,i}\}_{j=1}^{N_{test}}{}_{,i=1}^{N}$
for $j = 1 \dots N_{test}$ do
    $(\alpha_1, \dots, \alpha_N) \leftarrow$ Indices of sorted KNN distances
    $s_{j,\alpha_N} \leftarrow \frac{\mathbb{1}[y_{a_N} = \bar{y}_j]}{N}$
    for $i = N - 1 \dots 1$ do
        $s_{j,\alpha_i} \leftarrow \frac{\mathbb{1}[y_{\alpha_i} = \bar{y}_j] - \mathbb{1}[y_{\alpha_{i+1}} = \bar{y}_j]}{K} \frac{\min\{K,i\}}{i}$
            $+ s_{j,\alpha_{i+1}}$
    end for
end for
```

2.3. Approximate SV Algorithm

For general KNN classifiers, currently the only feasible way to calculate the SV is a Monte Carlo (MC) sampling algorithm [8]. The idea behind this approach is that the SV s_i can be regarded as the expected value of the random variable $\phi_i = v(P_i^{\pi} \cup i) - v(P_i^{\pi})$. Thus,

$$\hat{s_i} = \frac{1}{T} \sum_{t=1}^{T} \underbrace{[v(P_i^{\pi_t} \cup i) - v(P_i^{\pi_t})]}_{\phi_i^t} \tag{4}$$

is a consistent approximation of the real SV s_i, where π^t is the t-th random permutation of the training data. In fact, Jia et al. [2] demonstrate that $P[\max_i |\hat{s}_i - s_i| \leq \epsilon] \geq 1 - \delta$ can be achieved for unweighted KNN when choosing the number of random permutations T, such that:

$$T \geq \frac{1}{K^2 \epsilon^2} \log \frac{2K}{\delta}$$

Jia et al. [2] propose Algorithm 2 to compute the approximate SV for general KNN classifiers using (4).

Algorithm 2 Approximate SV for KNN classifier

```
Input: Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$, test data
  $D_{test} = \{(\bar{x}_i, \bar{y}_i)\}_{i=1}^{N_{test}}$, utility functions $\{v_i(\cdot)\}_{i=1}^{N_{test}}$
Output: The approximate SV $\{\hat{s}_{j,i}\}_{j=1}^{N_{test}}{}_{,i=1}^{N}$
for $j = 1 \dots N_{test}$ do
    for $t = 1 \dots T$ do
        $\pi^t \leftarrow \texttt{GenerateRandomPermutation}(D)$
        Initialize a length-K max-heap $H_t$ for the KNN
        for $i = 1 \dots N$ do
            Insert $\pi_i^t$ into $H_t$
            if $H_t$ changes then
                $\phi_{\pi_i^t}^t \leftarrow v_j(\pi_{1:i}^t) - v_j(\pi_{1:i-1}^t)$
            else
                $\phi_{\pi_i^t}^t \leftarrow 0$
            end if
        end for
    end for
    $\hat{s}_{j,i} = \frac{1}{T}\sum_{t=1}^{T} \phi_i^t$ for i = 1,..., $N$
end for
```

3. METHOD

In this section, we show the optimizations applied to the exact SV algorithm 1 and approximate SV algorithm 2. For both algorithms, we briefly discuss the baseline implementation in Section 3.1. Notice that both algorithms rely on the result of the sorted KNN algorithm 3. An initial performance analysis shows that up to 95% of the runtime in algorithm 1 and 2 is used for the sorted KNN calculation. Runtime improvements of the SV calculations would therefore be overshadowed by the precalculation of the KNN indices. Hence, the sorted KNN algorithm and the SV calculation are studied in isolation and optimized independently.

3.1. Baseline implementations

Here, we introduce the baseline implementations which will be used to calculate the speedups of the optimizations.

Sorted KNN Baseline. The baseline of the sorted KNN algorithm is implemented straightforwardly with a triple loop as seen in algorithm 3. It computes the indices of the KNN for the exact and approximated Shapley algorithms. The indices are sorted in ascending order by the euclidean distance between the corresponding x_{train} and x_{test} data points in the feature space.

Algorithm 3 Sorted KNN

```
   Input: Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$,
     Test data $D_{test} = \{(\bar{x}_i, \bar{y}_i)\}_{i=1}^{N_{test}}$
   Output: Sorted indices $\alpha = \{\alpha_{i,j}\}_{i=1,}^{N}{}_{j=1}^{N_{test}}$
1: for $i = 0 \dots N_{test}$ do
2:     for $j = 0 \dots N$ do
3:         for $k = 0 \dots L$ do
4:             $distance[j]$ += $(x_j[k] - \bar{x}_i[k])^2$
5:         end for
6:         $distance[j] = \sqrt{distance[j]}$
7:     end for
8:     $\alpha_{i,:} \leftarrow argsort(distance)$
9: end for
```

Exact SV Baseline. The baseline implementation follows directly from algorithm 1. Jia et al. [2] provide a python implementation, which is used as the reference for the C implementation.

Approximate SV Baseline. We implement algorithm 2 for unweighted KNN. To generate a uniform random permutation of the training data, we randomly shuffle the indices [0, ..., N-1] using the Fisher-Yates shuffle [9] below. The Fisher-Yates shuffle is an efficient random shuffling algorithm with time complexity $\mathcal{O}(N)$.

```c
void fisher_yates_shuffle(int* arr, int N){
    for (int i = N-1; i >= 0; i--) {
        int j = rand() % (i+1);
        swap(arr[i], arr[j]);
    }
}
```

We use a max-heap [9] array to keep track of the KNN distances. In the first K iterations of the innermost loop, the distance of the respective training point π_i^t is inserted into the heap. In the following N-K iterations, the heap is updated only if the new distance is smaller than the root. In that case, the root is replaced with this new distance and the heap properties are rebuild by calling `heapify`. Finally, the utility functions in line 8 are evaluated using the values in the max-heap:

$$v_j(\pi_{t,1:i}) = \frac{1}{K}\sum_{k=0}^{K-1} \mathbb{1}[y_{\alpha_{H_t[k]}} = \bar{y}_j]$$

where $\alpha_{H_t[k]}$ represents the index of the training sample that corresponds to the k$^{\text{th}}$ element in the heap H$_\text{t}$.

3.2. Sorted KNN optimization

ILP improvement. We observe that the base implementation lacks instruction-level parallelism (ILP). Unrolling the innermost loop by a factor of K allows us to use K accumulators, since the instructions are independent. Depending on the concrete CPU microarchitecture, different choices of K are required for optimal speedup. The goal is to saturate the pipeline at all time. Our instruction mix for the accumulators consists of two adds and one multiplication. Assuming that multiplications and additions have the same latency and throughput, the optimal choice of $K = 8$ is given by the formula:

$$K = \lceil \text{Latency} \cdot \text{Throughput} \rceil$$

Cache Locality. Assume that $\gamma \ll N$, where γ represents the cache size and N the number of train or test samples. Let L be the number of features. A cache miss analysis for the triple loop with a cache block size of eight doubles results in $N^2 \cdot 2\frac{L}{8} = \frac{N^2 L}{4}$ misses. This holds, since both the train and test matrix are accessed row-wise, which is the contiguous dimension of size L. We therefore have $\frac{L}{8}$ misses for the train matrix and $\frac{L}{8}$ misses for the test matrix, as only every 8th access results in a cache miss. Since we have to calculate all N^2 elements of the dist matrix, we get the final result of $\frac{N^2 L}{4}$ misses. Although the cache locality of this loop is already high, we can do better by introducing blocking. Figure 1 depicts the accessed elements in order to calculate the first block in the distance matrix. Each block is of size b, where b is a multiple of eight.

The cache miss analysis for blocks is as follows: In order to calculate one block in the distance matrix, we need to access $\frac{l}{b}$ blocks in both the train and test matrix. As each accessed row results in $\frac{b}{8}$ cache misses, the calculation of one block in the distance matrix generates $2\frac{l}{b} \cdot b \cdot \frac{b}{8} = \frac{lb}{4}$ cache misses. Since we have $(\frac{n}{b})^2$ many blocks in the distance matrix, the total amount of cache misses is $(\frac{n}{b})^2 \cdot \frac{l \cdot b}{4} = \frac{n^2 l}{4b}$.

Note that the cache miss rate decreases with increasing b. We therefore want to choose b as large as possible, while ensuring that the working set does not exceed the cache capacity. To calculate a single result block, three blocks need to be kept in cache. Since a cache block holds b^2 doubles, the cache size needs to be $3 \cdot b^2 \cdot 8$ Bytes. Reformulating gives us the final formula for the ideal block size:

$$b \leq \sqrt{\frac{\gamma}{8 \cdot 3}}$$

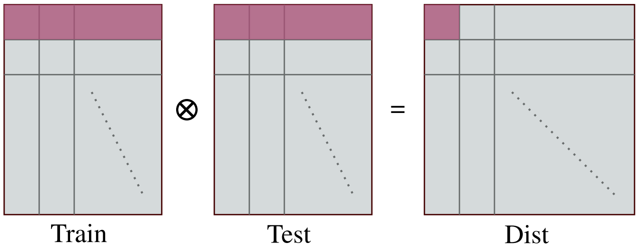

Fig. 1. Blocked version of the sorted KNN algorithm

Roofline. As a preliminary analysis for vectorization, we want to determine whether the algorithm is compute or memory bounded, as vectorization primarily increases performance for compute bound algorithms. Given the peak performance π and the memory bandwidth β, an algorithm is memory bounded, if its operational intensity is lower than $\frac{\pi}{\beta}$.
The operational intensity of algorithm 3 is:

$$I(N, L) = \frac{W(N, L)}{Q(N, L)}$$

Line 4 of algorithm 3 has two additions, one multiplication and is executed N^2L times, resulting in $3N^2L$ flops. The computation of the square root in line 6 is performed N^2 times. Finally, the sorting operation in line 8 has an expected comparison count of $\mathcal{O}(Nlog(N))$ and is called N times. Therefore, the overall work is:

$$W(N, L) = 3N^2L + N^2 + N\mathcal{O}(N\log(N))$$

We assume $log(N) \leq L$, as this is generally the case for large data sets. Therefore, we can approximate the work to:

$$W(N, L) \approx 3N^2L$$

To estimate $Q(N, L)$, we only consider reads and compulsory misses. We need to read the entire train matrix ($N \times L$), the test matrix ($N \times L$) and the distance matrix ($N \times N$). Since each feature is stored as a double, the total number of bytes transferred between memory and CPU is at least $8(2NL + N^2)$. Thus, the operational intensity may be bounded as:

$$I(N, L) \leq \frac{3N^2L}{8(2NL + N^2)} = \mathcal{O}(N + L)$$

Hence, algorithm 3 is most likely compute bounded.

Vectorization. We use one vector to hold four accumulators. One challenge is to sum the accumulators since each vector needs to be reduced to one element. We therefore use the following helper function [10]:

```
double vec_sum(__m256d vec) {
    __m128d vlow  = _mm256_castpd256_pd128(vec)
    __m128d vhigh = _mm256_extractf128_pd(vec,1)
    vlow  = _mm_add_pd(vlow,vhigh)
    __m128d high64 = _mm_unpackhi_pd(vlow,vlow)
    return _mm_cvtsd_f64(_mm_add_sd(vlow,high64))
}
```

In order to fully utilize the stores, we unroll the second innermost loop by a factor of four. This allows us to calculate four of the accumulated sums concurrently, which can then be stored using a single AVX store instruction.
To get rid of unaligned loads and stores, the data in memory is 32-byte aligned.
Lastly, we replace multiplications followed by additions with FMA vector instructions and restructure the code to follow a strict load - compute - store format, such that the compiler may identify dependencies more easily.

3.3. Exact SV optimization

Constant Precomputation. In line 5 of algorithm 1, the only data dependencies are $s_{j,\alpha_{i+1}}$ and the indicator variables. The other variables are data oblivious and depend solely on the current loop iteration. Thus, line 5 can be rewritten to:

$$s_{j,\alpha_{i+1}} + \left(\mathbb{1}[y_{\alpha_i} = \bar{y}_j] - \mathbb{1}[y_{\alpha_{i+1}} = \bar{y}_j]\right) \cdot \underbrace{\frac{\min\{K, i\}}{K \cdot i}}_{const}$$

The constant part may be precomputed and is of the form:

$$\left[\frac{1}{N-1}, \frac{1}{N-2}, \ldots, \frac{1}{K+1}, \underbrace{\frac{1}{K}, \ldots, \frac{1}{K}}_{K-times}\right]$$

Roofline. We calculate the operational intensity $I(N)$ as follows. The work $W(N)$ is $3N^2$, because we only have two additions and one multiplications in the innermost loop after the previous optimizations. Regarding data movement $Q(N)$, only reads and compulsory misses are counted. Over the course of the algorithm, we read all entries from the sorted KNN result matrix (N^2), the training labels (N), the test labels (N) and all temporary SV (N^2). This results in:

$$Q(N) \geq 8(2N^2 + 2N)$$

We can bound the operational intensity to:

$$I(N) \leq \frac{W(N)}{Q(N)} = \frac{3N^2}{(8(2N^2 + 2N)} \approx \frac{3}{16} = 0.1875$$

Based on the instruction mix, the achievable peak performance is 3 flops/cycle, as one FMA and one add can be issued per cycle. For a memory bandwidth β, algorithm 1 is therefore memory bound if $I(N) \leq \frac{3}{\beta}$. This is the case if β is smaller than 16 Bytes/cycle. For most computing platforms, it is thus likely that the computation becomes memory bound.

Reduce Data Accesses. As the computation is memory bound, we focus on reducing the number of memory accesses. Here, blocking is not applicable, since the computations always depend on the entire row, which is accessed in a data dependent order.

First, in line 5, the program accesses both y_{α_i} and $y_{\alpha_{i+1}}$ in each iteration. Thus, each y_{α_i} gets accessed twice in total (except for $i = 1$ and $i = N$). By precomputing the indicator variables, we can remove half of the accesses to y.

Second, again in line 5, the explicit load of $s_{j,\alpha_{i+1}}$ can be removed by scalar replacement, as the value is already held in the variable s_{j,α_i} from the previous iteration. The compiler may not optimize the explicit load away, since the possibility of aliasing exists.

Loop unrolling. We apply loop unrolling at two locations: First, we aim towards speeding up the precomputation step introduced earlier. By unrolling the indicator array computation by a factor of 8, different local variables can be used to save the results of the indicator array values, potentially increasing ILP. Next, we intend to unroll the main computation loop. The computations in the innermost loop depend on the results of previous iteration, it is thus not possible to increase ILP for this part. However, unrolling the outermost loop results in the same training point being compared to different test points. Hence, there is no dependency between loop iterations. The loop is unrolled by a factor of 4, as the optimization is aimed as a pre-step to SIMD vectorization.

SIMD. Examining the compiled assembly instructions of the previously optimized implementation shows that the compiler generates multiplications, followed by single additions. There, Fused multiply-add (FMA) operations would increase performance. The goal of this optimization step is to enforce FMA usage using intrinsics. However, loading the y_{train} data from the training into AVX registers as well as storing it into the shapley matrix is tedious, since the memory locations are dependent on the previously calculated indices. Therefore, the access pattern for each of the four register values is unpredictable, which prevents the use of efficient vectorized loads and stores.

Further optimizations regarding memory accesses are not possible, since the memory accesses of each iteration are dependent on the entire data. Neither can the data be efficiently placed out in memory, as the overhead of copying data around in order to properly align it would outweigh the later benefits of efficient SIMD computations.

3.4. Approximate SV optimization

Fast random shuffling. By profiling the base implementation using `perf`, we find that over 20% of the CPU cycles are spent on the Fisher-Yates shuffle. In each of its N iterations, there is one `rand()` call and one `modulo` operation. Both slow down the loop tremendously. Loop unrolling to increase ILP is not possible, as all succeeding instructions depend on the result of those two operations. Therefore, the only viable option is to use a faster random bounded number generator. Lemire [11] shows an efficient way of generating 32-bit random numbers in range [0, R]:

```
uint32_t random_bounded(uint32_t R) {
  uint64_t random32bit =  pcg_random32();
  uint64_t multiresult = random32bit * R;
  return multiresult >> 32;
}
```

To generate the intermediate random 32 bit number, we use `pcg_random32()`, as PCG [12] offers fast and statistically good algorithms for random number generation.

Basic block optimizations. First, precomputing the constant factors $\frac{1}{T}$ and $\frac{1}{K}$ removes most of the costly division operations. Additionally, we do scalar replacement for repeated array accesses. Then, to compute each SV in line 14, we unroll the inner loop by a factor of 8. This allows us to use separate accumulators to increase ILP, as seen in chapter 3.2 with the sorted KNN algorithm. Finally, in line 8, the operation count of the utility difference computations can be reduced from over $2K$ down to 2 operations. In the baseline implementation, the computation is:

$$\phi^t_{\pi^t_i} = \underbrace{\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{1}[y_{\alpha_{H_t[k]}} = \bar{y}_j]}_{v_j(\pi^t_{1:i})} - \underbrace{\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{1}[y_{\alpha_{H'_t[k]}} = \bar{y}_j]}_{v_j(\pi^t_{1:i-1})}$$

where H'_t represents the heap H_t prior to inserting the new training sample π^t_i. Thus, both heaps share the same values except for one (the root of H'_t got replaced by π^t_i in H_t). Therefore, we can simplify the computation to

$$\phi^t_{\pi^t_i} = \frac{1}{K}(\mathbb{1}[y_{\pi^t_i} = \bar{y}_j] - \mathbb{1}[y_{\alpha_{H'_t[0]}} = \bar{y}_j])$$

Furthermore, we can precompute $\mathbb{1}[y_i = \bar{y}_j]$ for all training labels y_i and test labels $\bar{y}_j$, as each result is used T times in the algorithm.

Increase spatial locality. After unrolling in line 14, we have the following loop structure.

```c
for (int i = 0; i < N; i++) {
    for (int t = 0; t < T; t+=8) {
        acc0 += phi[t*N+i];
        acc1 += phi[(t+1)*N+i];
        ...
        acc7 += phi[(t+7)*N+i];
    }
    ...
}
```

We note that the accesses to ϕ in the inner loop are not sequential. To improve spatial locality, we can change the dimensions of matrix ϕ from $T \times N$ to $N \times T$. Thus, the loop structure changes to:

```c
for (int i = 0; i < N; i++) {
    for (int t = 0; t < T; t+=8) {
        acc0 += phi[i*T+t];
        acc1 += phi[i*T+(t+1)];
        ...
        acc7 += phi[i*T+(t+7)];
    }
    ...
}
```

This results in a sequential access pattern which potentially increases the cache hit rate and thus improves performance.

SIMD vectorization. It's difficult to vectorize the innermost loop of algorithm 2, as the computations in each iteration depend on the results from the previous iteration. In addition, the random indices hinder us from efficiently using AVX loads and stores. It is possible to shuffle all the data arrays instead of the indices to allow for sequential access. However, this doesn't remove the problem of having mostly sequentially dependent instructions. On the other hand, it is possible to further speed up the random shuffling by generating 8 random bounded 32-bit integers at once using vectorized multiplication and shift operations [13]:

```c
__m256i avx_bound(__m256i randvals, __m256i R) {
    __m256i t1 = _mm256_mul_epu32(randvals, R);
    __m256i t2 = _mm256_srli_epi64(randvals, 32);
    __m256i t3 = _mm256_srli_epi64(R, 32);
    __m256i x = _mm256_srli_epi64(t1, 32);
    __m256i y = _mm256_mul_epu32(t2, t3);
    return _mm256_blend_epi32(x, y, 0b10101010);
}
```

Finally, the computations in line 14 can be vectorized as shown with the sorted KNN algorithm in chapter 3.2.

Reduce memory accesses. In the baseline, the temporary SV for each permutation are stored in a matrix ϕ. Then, in line 14, the actual SV are calculated by averaging all temporary results, resulting in $N * T$ accesses to ϕ. This process can be simplified by directly adding the temporary values to the SV array, as shown in algorithm 4. Then, in line 12, the shapley values only need to be multiplied by the factor $\frac{1}{T}$, which reduces the number of memory accesses down to N accesses.

Algorithm 4 Optimized Approximate SV

```
    Input: Training data $D = \{(x_i, y_i)\}_{i=1}^{N}$, Test data $D_{test} = \{(\bar{x}_i, \bar{y}_i)\}_{i=1}^{N_{test}}$, utility functions $\{v_i(\cdot)\}_{i=1}^{N_{test}}$
    Output: The approximate SV $\{\hat{s}_{j,i}\}_{j=1}^{N_{test}}{}_{,i=1}^{N}$
 1: for $j = 1 \dots N_{test}$ do
 2:     for $t = 1 \dots T$ do
 3:         $\pi^t \leftarrow$ GenerateRandomPermutation($D$)
 4:         Initialize a length-K max-heap $H_t$ for the KNN
 5:         for $i = 1 \dots N$ do
 6:             Insert $\pi_i^t$ into $H_t$
 7:             if $H_t$ changes then
 8:                 $\hat{s}_{j,\pi_i^t} \mathrel{+}= v_j(\pi_{1:i}^t) - v_j(\pi_{1:i-1}^t)$
 9:             end if
10:         end for
11:     end for
12:     $\hat{s}_{j,i}$ *= $\frac{1}{T}$ for i = 1,..., $N$
13: end for
```

Heap. The heap is an important data structure in the approximate SV algorithm that has not been optimized yet. It keeps track of the K nearest neighbors while iterating through each permutation of the data. In each iteration, a value may be inserted into the heap. This is generally slow, as heap operations such as `heapify` have terrible ILP [14] and are data dependent, making it very hard to optimize. However, we can try gaining a little speedup by inlining the `heapify()` function and using an iterative approach instead of recursive calls [15]. This can potentially allow for better compiler optimizations.

4. EXPERIMENTAL RESULTS

In this section, we demonstrate the runtime and performance of our optimized algorithms. We show the speedup gained with every single optimization step compared to the respective baseline implementation, while fixing $K = \sqrt{N}$.

Experimental setup. In the experiments, the input size corresponds to the number of training and test samples of the CIFAR-10 data set [16]. Each sample has 2048 features stored as doubles. Therefore, each sample is of size 16 KB. For runtime measurements, we use `tsc_x86.h` [17], which counts the number of cycles using the CPU's time stamp counter. We always report the median runtime of multiple test runs. To determine the number of floating point operations, we use `perf` to read the CPU's performance counter registers.

To get a fair baseline, we used `gcc 12.1` and allowed the compiler to fully utilise the microarchitectural features on the measurement hosts. Specifically, the baseline is compiled with flags `-std=c11 -O3 -march=native`. We also tested `clang 13.01` and `Intel C 2022.1` as compilers. However, the runtimes of the fully optimized binaries only differ from each other by a diminishable percentage. Measurements were conducted on a Intel Core i7-1065G7 (Ice Lake microarchitecture), with a fixed CPU base frequency of 1.3GHz and the respective cache sizes: L1: 192 KiB, L2: 2 MiB, L3: 8 MiB. For measurement consistency, the Intel Turbo Boost feature was disabled for all experiments.

To nullify the effects of input dependent runtimes, such as different page access patterns, execution paths or cycles per instruction, we reuse the same input data for the same input size. Measurements start after the memory allocation and end at the end of the computations, sorting is therefore included.

Sorted KNN results. Since no optimization on the KNN algorithm influences the operations count, we use a performance plot as seen in Figure 2.

The baseline has a performance of 0.69 flops/cycle. The first optimization step (blocking) improves cache locality and almost doubles the performance to 1.61 flops/cycle. As the cache locality of the baseline is already decent, the performance gain from blocking is not as high as it could be for other numerical algorithms.

The next optimization introduces separate accumulators, increasing ILP and giving us a performance of 3.85 flops/cycle. Note that on this particular microarchitecture, the peak performance without SIMD is 6 flops/cycle, since FMA instructions may be scheduled on three ports. With all scalar optimizations, we reach up to 64% of the maximal achievable performance. 100% utilization is not possible given our current instruction mix. Lastly, we vectorize our code and gain an additional performance boost visualized in red. The final vectorized performance is 5.05 flops/cycle, which is far from the theoretically best possible speedup of factor 4x. By profiling, we notice that the helper function `vec_sum` takes up a long time. More specifically, we see that the native instruction resulting from the `_mm256_unpackhi_pd` intrinsic is the bottleneck for the vector sum computation.

Exact SV results. Figure 3 shows the runtime of the baseline implementation, as well as the runtime of each added optimization step. The input size n is incremented with a step size of 524, thus 1028 additional doubles are added in each step. By precomputing the constant parts of the exact SV algorithm 1, the runtime is improved by 1.39x. This is due to the fact that strength reduction reduces the critical path of the computation, which included two divisions in the base implementation. The next optimization step, concerning the reduction of memory accesses, results in the best overall runtime speedup of 1.87x. This is in accordance with the result of the theoretical roofline analysis, where the computation is found to be memory bound. Applying

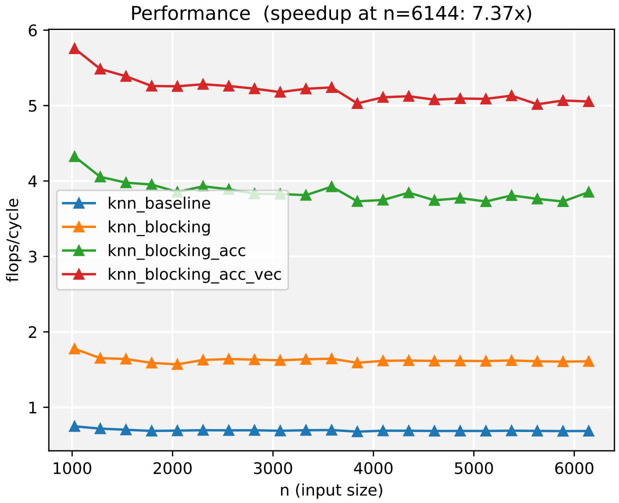

Fig. 2. Performance for sorted KNN.

the two unrolling optimizations on the precomputation, as well as the outer loop of the exact SV computation, worsens the performance. Now, the overall runtime speedup is merely 1.63x. By investigating the executable using `gdb` and `perf`, the issue for the performance decrease was identified as follows: In the previous optimization step, the compiler managed to completely optimize out the indicator array and perform the indicator array computations on the fly. With explicit unrolling, however, the compiler no longer optimized away the array. We assume that this additional array in the working set degrades cache performance, due to an increased number of conflict misses. Thus, more memory movement is imposed on the already memory bound computation. The final, vectorized version is implemented to force FMA usage using intrinsics. If the memory bandwidth has not been reached with the scalar implementation, a performance gain could now be observed. Figure 3 shows that the vectorized runtime is identical to the scalar version with improved data locality. Therefore, we conclude that our final implementation is nearly optimal and that the bottleneck is the memory bandwidth.

Approximate SV results. In the experiments for algorithm 2, we choose the number of permutations T such that

$$T \geq \frac{1}{K^2\epsilon^2}\log\frac{2K}{\delta}$$

with $\epsilon = \delta = 0.01$. Figure 4 shows the runtime of the baseline implementation and each individual optimization discussed in chapter 3.4. Using the optimized random number generator improves the runtime by a factor of 1.56x, as this strength reduction reduces the number of cycles along the critical path of the Fisher-Yates shuffle. The various block optimizations result in an additional speedup of 1.89x.

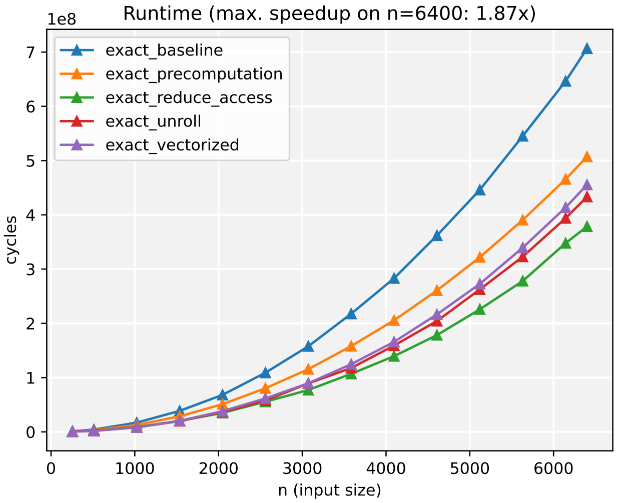

Fig. 3. Runtime of the exact Shapley algorithm 1

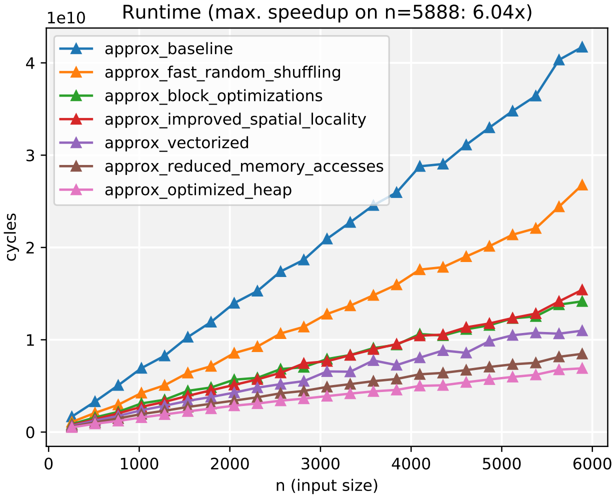

Fig. 4. Runtime of the improved MC algorithm 2.

Increasing the spatial locality in line 14 does not reduce the runtime. A possible explanation for this is that before switching the dimensions of ϕ, T blocks need to be kept in cache to benefit from spatial locality. For $N \geq 256$, that is $T \leq 316$. Thus, the program most likely already benefits from spatial locality even without the sequential access pattern, as 316 blocks fit into the 192KB L1 cache. Next, as discussed in chapter 3.4, not many parts of algorithm 2 can be efficiently vectorized. Therefore, only a small speedup of 1.29x is achieved through vectorization of loop 14 and the vectorized random number generator. Removing all accesses to matrix ϕ by directly adding the temporary results into the SV array reduces the runtime as expected. Overall, including the optimized `heapify`, we achieve a total speedup of 6.04x compared to the baseline implementation.

5. CONCLUSION AND FUTURE WORK

We have presented two highly optimized single-core implementations for the calculation of exact and approximate Shapley values in the case of K-nearest neighbour models. In both algorithms, the bottleneck is the sorted KNN subproblem. There, by increasing ILP, optimizing cache locality and utilizing AVX2 vectorization, we achieve 7.37x speedup compared to the baseline C implementation. By regarding the actual Shapley algorithms in isolation we achieve a speedup of 1.87x for the exact Shapley and a speedup of 6.04x for the approximate algorithm. In both Shapley algorithms, the data dependent access patterns hinder further optimizations on the computation, which is found to be memory bound.

Future work could expand on the possibility of using single precision floating point numbers instead of double precision floating point number. We believe that the runtime could be greatly improved, with a manageable precision loss. Since bookkeeping in the heap data structure is costly, improvements would directly carry over to our use case. Finally, since matrix multiplication shows similarities to our problem, employing similar specific optimizations may be beneficial and are worth exploring.

6. CONTRIBUTIONS OF TEAM MEMBERS

Jonas. Optimized the improved MC algorithm (algorithm 2). Implemented its baseline, studied various random number generators and random shuffling techniques. performed basic block optimizations and SIMD optimizations for algorithm 2, tried various ways to increase cache performance, implemented the faster iterative heapify function.

Christopher. Wrote the baseline implementation of the sorted KNN algorithm 3, as well as the baseline implementation of algorithm 1. Initial bottleneck identification of both base implementations. Focused on the exact Shapley optimization algorithm 1. Data locality optimization, unrolling and SIMD. Worked with Nicolas and Patrick on the SIMD optimization of the sorted KNN algorithm 3. Ran benchmarks regarding the exact Shapley optimizations.

Patrick. Worked mostly together with Christopher and Nicolas on the performance improvements of the sorted KNN algorithm by tackling the practical challenges that came up while improving cache locality using blocking and AVX2, as well as consistent memory alignment. Was also responsible for the performance and microarchitectural analyses using the Intel VTune profiler [18] and the theoretical and (later) practical flop analysis as a foundation of our performance and rooftop analyses.

Nicolas. Created all data sets with feature extraction from ResNet. Did the theoretical analysis for the roofline and cache miss rate of both the sorted KNN and exact Shapley algorithm. Implemented Blocking for sorted KNN and exact Shapley. Together with Patrick performed various optimizations (accumulators, strength reduction,..) and basic SIMD optimizations on the sorted KNN algorithm. Implemented more advanced SIMD optimizations with Chris and Patrick by combining the sorted KNN and exact Shapley algorithm. Added some additional precomputation optimizations to the exact Shapley algorithm.

7. REFERENCES

[1] Sasan Maleki, Long Tran-Thanh, Greg Hines, Talal Rahwan, and Alex Rogers, "Bounding the estimation error of sampling-based shapley value approximation," 2013.

[2] 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," *arXiv preprint arXiv:1908.08619*, 2019.

[3] John J Bartholdi and Eda Kemahlioğlu-Ziya, "Using shapley value to allocate savings in a supply chain," in *Supply chain optimization*, pp. 169–208. Springer, 2005.

[4] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu, "How to price shared optimizations in the cloud," *arXiv preprint arXiv:1203.0059*, 2012.

[5] Sahibsingh A Dudani, "The distance-weighted k-nearest-neighbor rule," *IEEE Transactions on Systems, Man, and Cybernetics*, , no. 4, pp. 325–327, 1976.

[6] Jeff Johnson, Matthijs Douze, and Hervé Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[7] Vincent Garcia, Eric Debreuve, and Michel Barlaud, "Fast k nearest neighbor search using gpu," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–6.

[8] Sasan Maleki, *Addressing the computational issues of the Shapley value with applications in the smart grid*, Ph.D. thesis, 08 2015.

[9] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Boston, third edition, 1997.

[10] Peter Cordes, "Get sum of values stored in __m256d with sseavx," https://stackoverflow.com/questions/49941645/get-sum-of-values-stored-in-m256d-with-sse-avx.

[11] Prof. Daniel Lemire, "Fast random shuffling," https://lemire.me/blog/2016/06/30/fast-random-shuffling/, Accessed: 2022-06-19.

[12] Melissa E. O'Neill, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. 2014.

[13] Daniel Lemire, "Simdxorshift," https://github.com/lemire/SIMDxorshift.

[14] Malte Skarupke, "On modern hardware the min-max heap beats a binary heap," https://probablydance.com/2020/08/31/on-modern-hardware-the-min-max-heap-beats-a-binary-heap/.

[15] Ashis Kr. Das, "Generic binary heap data structure and associated algorithms," https://github.com/AKD92/libbh-binaryheap.

[16] Alex Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[17] Mikhail Kurnosov, "tsc_x86.h: Simple benchmark based on time-stamp counter," https://github.com/mkurnosov/tscbench/blob/master/tsc_x86.h, 2014.

[18] Intel, "Intel vtune profiler 2022.3.0," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.htmlgs.i6xhgk.