OPTIMIZING XGBOOST

Team 39: Amory Hoste, Fizza Zafar, Noman Sheikh, Rishabh Singh

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

XGBoost is the state-of-the-art framework for gradient boosting used in a variety of machine learning applications. A number of techniques have been proposed to optimize and scale XGBoost for larger data sets, most of them relying on multi-core or distributed infrastructure. In this work, we present a sequence of optimizations which eventually result in superior performance on a single core. The primary focus is to improve the split finding algorithm which contains the majority of floating point operations (flops) in the XGBoost framework. Our experimental results show that one can achieve: 1) 4.8x speed up in run time of split finding; 2) 3.8x improvement in performance of split finding; and 3) 2.1x improvement in overall run time with respect to baseline implementation.

1. INTRODUCTION

Motivation. Gradient Boosting is a powerful tool with applications in a vast range of supervised machine learning problems such as classification, regression and ranking. XGBoost, the state-of-the-art gradient boosting framework, captures non-linearity in the data and provides high accuracy whilst being scalable. However, with increasing size of data nowadays, processing requirements are blowing, thereby increasing run time.

Related work. A lot of research work has previously discussed gradient boosting trees and their scalability, focusing on both algorithmic and hardware level optimizations. LightGBM [1] and Planet [2] suggest considering selective split candidates rather than every data point to make it more scalable. XGBoost [3] proposes three different algorithms for learning trees. Two of these are applicable to dense datasets while the third leverages sparsity of datasets. Similar to [1] and [2], the second algorithm also narrows down the split search to some candidates. Parallelizing the gradient boosting algorithm has also been a popular idea. [2] uses the Map-Reduce [4] framework to parallelize XGBoost over commodity hardware. [5] suggests parallelizing the construction of individual trees whereas [6] proposes a new algorithm for scalable parallel training, with minimum inter-node communication. On the other hand, the use of special hardware has also been suggested to gain speedup. [7] suggests using GPUs for expensive computation involved in gradient boosting. Further works explored specialized techniques by which XGBoost, specifically, can be made more scalable. [8] and [9] propose accelerating XGBoost tree formation by using specialized hardware such as GPUs and FPGAs.

Contribution. Library implementations of XGBoost leverage, either multi-core and distributed architectures, or sparsity of datasets, to provide scalability. However, our work focuses on improving the performance of XGBoost, for dense data processed over a single core. We investigate the performance bottlenecks faced by the algorithm in a single core setup and explore different ways of eliminating them. To this end, we explore the benefits and drawbacks of various different memory layouts and compute patterns. We focus on algorithmic changes as well as processor and cache level optimizations.

2. BACKGROUND

In this section we give a brief overview of gradient boosting, introduce the XGBoost algorithm and perform a cost analysis for both the exact greedy and the approximate greedy split finding algorithm.

Gradient boosting is a machine learning method based on boosting, used to create tree ensemble models. It creates multiple decision trees that are used together to make predictions. This is known to perform better in terms of accuracy, compared to using only a single decision tree. Using the first and second order derivative of a given loss function, gradient boosting tries to optimize this loss function in a way similar to gradient descent. But instead of changing the parameters of a function, gradient boosting adds new decision trees to traverse the loss function in the direction of steepest descent. In the end, an ensemble of trees is obtained which can then be used to make predictions for new input data.

XGBoost is a gradient boosting framework that can be used for both regression and classification. It uses different optimizations and tricks to make it much more efficient compared to other available gradient boosting frameworks. Since tree models are very prone to over-fitting, it also introduces regularization techniques to prevent that.

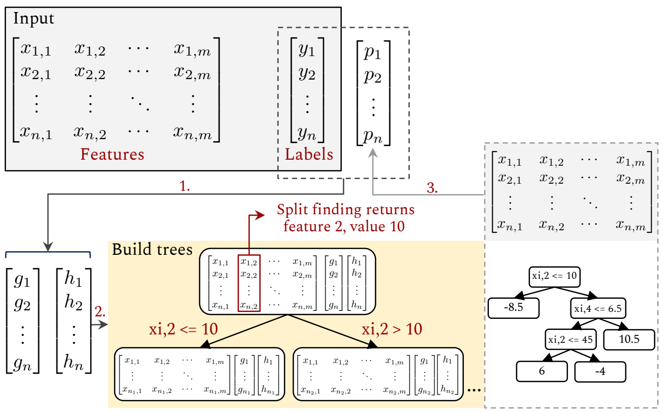

Fig. 1: Visualisation tree building algorithm

In subsequent sections, we discuss the two main parts of XGBoost: the tree building and split finding. As a cost function, we will count the amount of floating point operations (flops) that are performed within the algorithm. Since the split finding routine is the only part containing a substantial amount of floating point calculations, the cost will only be discussed for the different split finding algorithms. For the whole tree building algorithm, the asymptotic runtime will be discussed instead.

2.1. Tree Building Algorithm

The algorithm takes a feature matrix X, label vector Y and a loss function as input. The loss function depends on the type of problem being solved; we used binary cross-entropy as we were focusing on a classification problem. The pseudocode of the tree building algorithm is given in algorithm 1. The algorithm starts with an initial prediction and a root node containing all features and the corresponding values of the first and second order derivatives of the loss function. In order to keep track of which rows were splitted to which node, we define the instance set I_s, of a node s as all rows that ended up in node s after splitting the parent node of s. Using the score of a node $s = \sum_{i \in I_s} \frac{g_i^2}{h_i+\lambda}$, the tree is created by recursively splitting the current node s into child nodes l and r. The optimal feature and value to split on is found by a routine called split finding which finds the split feature and value that results in the highest score $gain = Score_l + Score_r - Score_s$. This gain represents how much the score increased by splitting a node into two separate nodes. Figure 1 gives an overview of this entire procedure. In order to get a tree ensemble model, this process is repeated t times, creating a new tree in each iteration that is fitted to the residuals of the prediction obtained from the previous iterations.

2.2. Exact Greedy Splitfinding

The first algorithm to calculate the best splits takes a greedy approach by calculating the gain for every possible valid split of each feature. A split index i for a sorted column of features c is valid if $c_i \neq c_{i+1}$. This means that split indices are only valid if the next feature in the sorted column has a different value than the value we are splitting on. Without this check, it would be possible to split in the middle of a sequence of which all values are equal which would give inconsistent results. The best split is obtained by taking the feature and feature value of the valid split with the highest gain. The complete algorithm is given in algorithm 2.

Algorithm 1 *Tree building algorithm*

```
    Input: Features $X$, labels $y$, loss function $l$
 1: Let $p \leftarrow average(y)$ be the initial prediction
 2: repeat
 3:     $g_i = \frac{d}{dp_i} l(y_i, p_i)$, for $i = 1 \dots n$
 4:     $h_i = \frac{d^2}{d^2 p_i} l(y_i, p_i)$, for $i = 1 \dots n$
 5:     repeat
 6:         $s \leftarrow$ next node to split
 7:         $gain, feat, val \leftarrow splitfind(s, g \in I_s, h \in I_s)$
 8:         if $gain >= \gamma$ then
 9:             create child nodes $l$ and $r$ of $s$
10:             $x_j, g_j, h_j \in I_s$ where $x_{j,feat} \leq val$ go to $l$
11:             $x_j, g_j, h_j \in I_s$ where $x_{j,feat} > val$ go to $r$
12:         end if
13:         $s.output \leftarrow \sum_{i \in I_s} \frac{-g_i}{h_i+\lambda}$
14:     until no nodes left to split or max tree depth reached
15:     $p_i$ += $\epsilon \cdot tree.predict(X_i)$, for $i = 1 \dots n$
16: until $t$ trees are build
```

Algorithm 2 *Exact Greedy Algorithm for Split Finding*

```
 1: $gain \leftarrow 0$
 2: $G \leftarrow \sum g_i,\ H \leftarrow \sum h_i$
 3: for $k = 1 \dots m$ do
 4:     $G_L \leftarrow 0,\ H_L \leftarrow 0$
 5:     for $j$ in $sorted(x_{jk})$ do
 6:         $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
 7:         $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
 8:         if $j$ is a valid split index then
 9:             $gain \leftarrow \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
10:         end if
11:     end for
12: end for
    Output: Split with max gain
```

Cost The cost of the exact greedy split finding algorithm is $15mn + 2n$ flops for a dataset with n instances and m features. More specifically, we have $9mn + 2n$ additions and subtractions, $3mn$ multiplications and $3mn$ divisions. $3mn$ multiplications, $3mn$ divisions and $5mn$ additions/-subtractions come from line 9 where we have 3 divisions, 3 multiplications, 4 additions and one subtraction that are executed for all m features and all n rows. Lines 6 and 7 contain an additional $4mn$ additions and subtractions. The final $2n$ additions are executed on line 2 where we compute the total sum of the first and second order derivatives.

2.3. Approximate Splitfinding

The key difference between the exact and approximate algorithms, is in the candidate split points. The exact greedy algorithm considers every unique feature value as a split candidate. For this reason, it is good at finding the best possible split. However, iterating over all instances doesn't scale well for large datasets. On the other hand, the approximate algorithm uses a method called Weighted Quantile Sketch [3] and proposes possible splitting points considering percentiles of the feature distribution. We now explain that method.

Let multi-set $D_k = \{(x_{1k}, h_1), (x_{2k}, h_2)...(x_{nk}, h_n)\}$ denote feature values and second-order derivatives for k-th feature value. The rank r_k is thus defined as:

$$r_k(z) = \frac{1}{\sum_{(x,h)\in D_k} h} \sum_{(x,h)\in D_k, x<z} h$$

The rank r_k represents the fraction of instances whose feature value k is less than z. We find split points $\{s_{k1}.s_{k2}, ...s_{kl}\}$ such that

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon$$

where ϵ is an approximation factor. This essentially means we weigh each data point i by h_i and assign it into different buckets based on its rank r_k. Once the candidate splits are calculated, the feature values are mapped into buckets. Algorithm 3 then proceeds in a fashion similar to Algorithm 2. The best split is determined based on the gain aggregated over each bucket.

Algorithm 3 *Approximate Algorithm for Split Finding*

```
1: for $k = 1 \ldots m$ do
2:     Propose $S_k = \{s_{k1}, s_{k2}, ..., s_{kl}\}$ by percentile on feature $k$.
3:     Proposal can be done per tree (global), or per split (local).
4: end for
5: for $k = 1 \ldots m$ do
6:     $G_{kv} \longleftarrow= \sum_{j\in\{j|s_{k,v}\geq x_{jk}>s_{k,v-1}\}} g_j$
7:     $H_{kv} \longleftarrow= \sum_{j\in\{j|s_{k,v}\geq x_{jk}>s_{k,v-1}\}} h_j$
8: end for
9: Follow same step as in algorithm 2 to find max score only among proposed splits.
```

Cost The cost of the approximate split finding algorithm is $2n + 6mn + 10m(b - 1)$, where n,m and b represent the number of rows, features and candidate split points respectively. This consists of $2n + 4mn + 6m(b - 1)$ additions/subtractions, $mn + 2m(b - 1)$ multiplications and $mn + 2m(b - 1)$ divisions.

2.4. Asymptotic complexity of Splitfinding Algorithms

Assuming no pruning, building t trees has an asymptotic runtime of $O(t \cdot d \cdot m \cdot n \cdot \log n)$ where d represents the depth of the tree that is built. Both splitfinding routines have an asymptotic complexity of $O(m \cdot n \cdot \log n)$ when executed on an input with n rows and m features. For the both algorithms, we iterate over all m features and for each feature, we do a sorting step of complexity $O(n \log n)$. Since no pruning is assumed and a single row is either split to the left or the right child node, the total amount of rows of all nodes on the same depth always equals n. This means that in total, we get a complexity of $O(d \cdot m \cdot n \cdot \log n)$ which gives the final result of $O(t \cdot d \cdot m \cdot n \cdot \log n)$ when repeating this for t trees.

3. PROPOSED METHOD

The optimization workflow can be understood cogently by classifying the proposed methods into three paradigms depending on the organization of data in the memory. We name these paradigms: 1) Row Major; 2) Column Major; and 3) Blocking. All implementations consist of two major functions: 1) Splitfinding and 2) Node splitting (where the instances are split based on the feature and value returned by the splitfinding function). We assume that the baseline implementation proposed in [3] follows row major layout of data. Hereon, we refer to the gradients as *g* and second-order derivatives as *h*.

3.1. Baseline

The Baseline implementation was done as stated in Algorithm 2. We used *mergesort* to sort the feature values. It can be observed in the pseudocode that the algorithm performs sorting on each feature and the inner loop after the sorting step is doing constant-time operations. Therefore, the asymptotic runtime of this algorithm is $O(m.n \log(n))$ where n is the number of samples in dataset. Our empirical observations also confirm that sorting is indeed the bottleneck. We tried two feasible ways to eliminate sorting as the bottleneck: 1) Making an assumption on the dataset that all the values are integers and replacing *mergesort* with *bucketsort.* 2) Presorting the data as a one-time preprocessing step. We now explain these steps in detail.

Bucketsort Switching from *mergesort* to *bucketsort* makes Algorithm 2 linear in n with asymptotic runtime of $O(m.n)$. While the improvement is significant, we pay the extreme cost of making a strong assumption on the dataset that all the values are integers. Such an assumption can possibly render our algorithm useless in many applications. As a result we do not build any further on this idea.

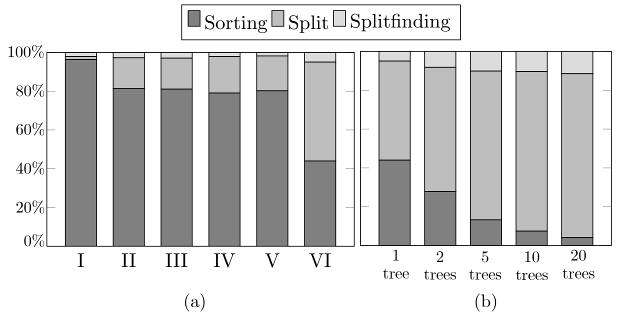

Fig. 2: (a): Runtime contribution each function when building one tree for different implementations. I: baseline, II: Presort + no pointer chasing, III: Cumulative sum of g/h, IV: Blocking, V: Blocking with SIMD + mergesort, VI: Blocking with simd + quicksort, (b): Time taken by functions for multiple trees.

Presorting As was proposed in [3], one can perform sorting only once on the entire dataset where each feature column is sorted independently and then re-use the sorted columns in the *splitfinding* function. Using this optimization, we no longer need to sort every time the splitfinding function is invoked. As a result a large amount of computation is taken out of the *splitfinding* function. The benefit of presorting becomes even more evident when more than one tree is created. However, this also makes the splitting of nodes non trivial. Since each feature column is sorted independently, a significant amount of I/O will start occurring in the *split* function. This is because in order to split a node, we first need to reconstruct the original non-sorted matrix of this nodes instance set which essentially requires reads from arbitrary locations in the presorted matrix. This effect can also be seen in figure 2(a), where we clearly observe that the contribution of splitting increases substantially when presorting compared to the non presorted implementation.

Also note that presorting the data significantly changes the flop count of *splitfinding* routine. Therefore, we consider "presorting" as the new baseline to compare the performance of further optimizations mentioned in this report.

3.2. Row Major

Once we get sorting out of the *splitfinding* function, it is easy to convince that the vectorization of this function is a straightforward task. In Algorithm 2, one can unroll outer loop taking four features at a time. Since the data is arranged in row major fashion, first and second order derivatives for adjacent features are also adjacent in memory layout. Therefore, this optimization also allows us to exploit the spatial locality in the data. It can be seen later in the next section that vectorizing gives a significant reduction in runtime for row major implementation.

Further Optimizations. We implemented other basic optimizations wherever possible. These included unrolling loops that performed array summation and using accumulators, applying code motion etc. In order to reduce the memory footprint of our code, we also free the data memory of a node after it has been split.

3.3. Column Major

The next direction we explored was storing data in column major order i.e. storing feature columns contiguously in memory. This stemmed from the observation that the splitfinding function iterates over all rows of a column before moving on to the next. With row major data, we bring in four features at a time but use only one. By the time the other features are needed, they have been evicted from cache due to capacity/conflict misses. Therefore, in order to get better spatial locality, we tried column major data.

Eliminate Pointer Chasing. Storing presorted columns required us to keep pointers to the corresponding g and h statistics for every instance. This resulted in non-contiguous memory accesses for their corresponding array. In order to eliminate the pointer chasing that resulted, we replicated theses statistics and stored them in sorted order for each feature(or column). This allowed contiguous accesses at a cost of approximately 2x memory compared to the presorting version (replicating derivatives for each column). At this point, the only cache misses that occurred in the splitfinding function were compulsory ones.

Cumulative Sum of Derivatives. The splitfinding function evaluates every candidate split based on the sum of g and h of instances on either side of the split. Since we already store g and h separately for each column, we instead store the cumulative sum of all preceding entries. However, this means that doing the actual splitting of nodes requires recalculating them from the cumulative sums. Since the node splitting function contains a lot of memory I/O, we did not expect this computational overhead to be significant. The function has to stall a lot for memory I/O so these cycles can possibly be used for recalculating the g and h.

Column Major SIMD. We tried to implement both the splitfinding and node splitting functions using SIMD. We did not observe much benefit here due to branching and dependencies in the code. We only provide results for the best performing configuration in the next section.

Splitfinding. Column major format is particularly not well-suited to the use of vector intrinsics for splitfinding because we need to compare the gain from a split-candidate to the gain from the previous candidate. Therefore, there is an inherent dependency when processing data row by row. Processing multiple columns together in one vector would require non-contiguous memory accesses and destroy the locality benefits we get from column major format. Therefore, using SIMD here did not yield any significant improvement, sometimes even making the performance worse off.

Node Splitting. This function is heavily dominated by memory I/O as instances are divided into two arrays depending on which side of the split they correspond to. This means that memory is read/written based on a conditional instruction and requires non-contiguous memory writes. For this, we had to use many expensive operations which required frequently crossing the 128-bit vector boundary. Moreover, to control the program's memory footprint, we used a bit vector for maintaining which side of the split each instance belongs. This was non-trivial to manipulate alongside double precision floating point numbers.

3.4. Blocking

In order to resolve the dependency issues of the column major implementation while still maintaining the locality benefits of the column major format, we decided to store the input data in a blocking based format. In this format the data is stored in blocks of 4 features, within which the data is stored in a row major order. In order to preserve good locality for the initial presorting step, the data is read in column major format, presorted and then converted to the blocking format.

Blocking without SIMD. The blocked data format allows us to easily unroll the loops in algorithm 2 while still maintaining good locality. To compute the total G and H sum, we used 4 accumulators in order to avoid the latency penalty of the additions. To compute the best splits (Algo 2, line 3-10), we applied loop unrolling to the outer loop by calculating the best split for 4 features per iteration.

Blocking with SIMD. For the SIMD implementation, we also compute the best splits for 4 features at a time. Because of the blocking format, we can now easily load 4 features of a single row at a time into a AVX register and do all computations using SIMD intrinsics. When doing this naively, the if condition that checks if we are at a valid split index reduced the performance a lot. In order to also vectorize this if condition, we first created a mask using `_mm256_cmp_pd` and `_mm256_and_pd`. This mask is a `_mm256d` vector where the ith double in this vector is 1 if the current row is a valid split index for feature i and the new gain for feature i is larger than the old gain. We also created an inverted mask where the 0s and 1s have been switched. Using these masks, we can then update the gain vector for all 4 columns in parallel by adding the product of the newly calculated gain vector and the regular mask to the product of the old gain vector and the inverted mask. This process is also illustrated in table 1.

Additional Optimizations. Instead of using mergesort, we also tried using the `qsort` function out of the c standard library for the presorting step. Since this is a C standard library function that has been highly optimized, this

Mask calculation				
$newgain > gain$	0	1	0	1
$validsplit$	1	1	0	0
$mask$	0	1	0	0

Update gain				
$newgain$	3.5	8.0	0.5	5.5
$mask$	0	1	0	0
$gain$	4.5	6.5	1.5	4.5
$invertedmask$	1	0	1	1
result	4.5	8.0	1.5	4.5

Table 1: Gain vector update illustration

outperformed our own mergesort implementation we initially used. We also tried storing the data as floats instead of double. However, we were not able to move much calculations from double to float format since 4 bytes didn't provide enough precision for the calculations.

4. EXPERIMENTAL RESULTS

In this section, we evaluate each of the optimizations outlined in the previous section. We provide results and reason about the scope for further improvement.

Experimental setup. We used an Intel(R) Core (TM) i5-8257U CPU @ 1.40 GHz (Coffee Lake) processor, with turbo-boost disabled for our experiments. The machine had a 32KB L1, 256 KB L2 and 6 MB (shared) L3 cache. The operating system used was macOS Catalina (10.15.3). The gcc 4.2.1 (linked to clang 11.0.0) compiler was used with flags `-O3`, `-march=native` and `-fno-tree-vectorize` (unless stated otherwise). We used the RDTSC counters to measure the runtime in cycles, for individual functions as well as the overall runtime. We manually calculated flops in every section of the code to convert runtime to performance where needed.

Dataset and Input Sizes. We used the Criteo Terabyte Click Log Dataset[1] which, after preprocessing, contained around 7M samples with 10 features and 1 binary label column each. In order to take measurements over different input sizes, we randomly sampled from the rows while keeping all 10 columns. We varied the sample size from 100K to 1M in steps of 10K, and report measurements for fitting a single tree of depth 4.

4.1. Results

We performed optimizations on both the exact greedy and approximate splitfinding algorithms. We report runtime of the splitfinding function as well as entire tree creation for all optimizations. Since the number of flops vary significantly over different implementations, we include performance plots only for optimizations where the flop count is

[1]http://labs.criteo.com/downloads/download-terabyte-click-logs/

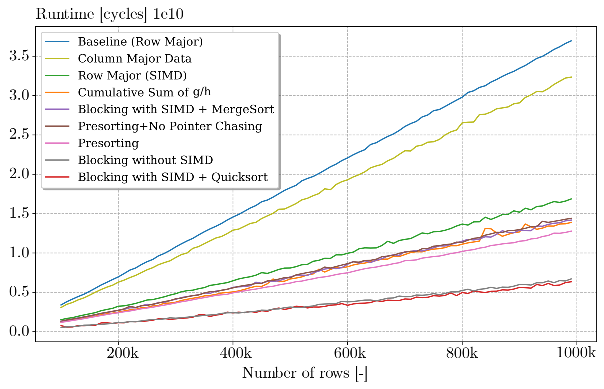

Fig. 3: Runtime of single tree creation

Data Format	Mean	Max
Row Major	1.23	1.61
Column Major	4.30	5.66
Blocked	3.41	4.40

Table 2: Performance Gain for Exact Splitfinding Algorithms

asymptotically the same. Moreover, the bulk of flops lies in the splitfinding algorithm with the remaining code being dominated by memory I/O. Therefore, we quote performance values only for splitfinding.

Runtime to Create a Single Tree. Fig. 3 shows how the runtime changes as the input dataset becomes larger. Blocking with SIMD gave us the best improvement in runtime for the entire tree creation. This implementation runs, on average, 6.2x faster than the original baseline and 2.1x faster than the presorted baseline. Depending on input size, we achieve a maximum improvement of 6.9x and 2.4x over the baseline and presorted version respectively.

Exact Greedy Splitfinding. Fig. 4 shows how runtime of the splitfinding function varies with input size. Eliminating pointer chasing and storing derivative sums gave a 4.4x improvement over the presorted baseline. Storing data in blocked format gave a 4.8x reduction in runtime over the presorted version. Using SIMD in row major gave a minor improvement of 1.23x with respect to the presorted baseline. However this improvement is not seen while using SIMD on blocking implementations due to the fact that blocking implementations are already optimized and very close to the roof in roofline plot Figure 6, thereby leaving less room for improvement by SIMD.

Fig. 5 shows how the performance changes with input size as we attempt different optimizations. Column major optimizations improved splitfinding performance by the most, with blocking giving a comparable performance. However, as shown in Table 3 column major format increased the overall tree's runtime whereas blocking format led to a reduction there too.

Another observation is that vectorized row major implementation performs slightly worse than the baseline for smaller input sizes until 250k rows. When we have more

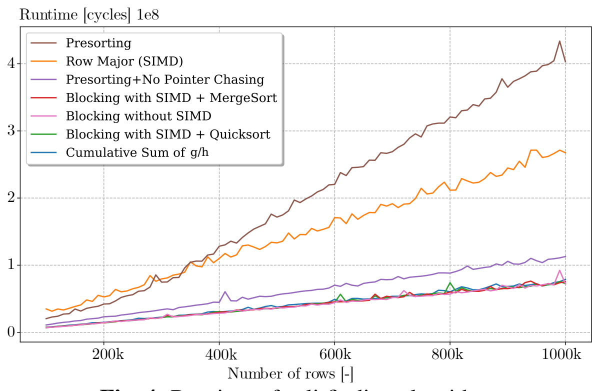

Fig. 4: Runtime of splitfinding algorithm

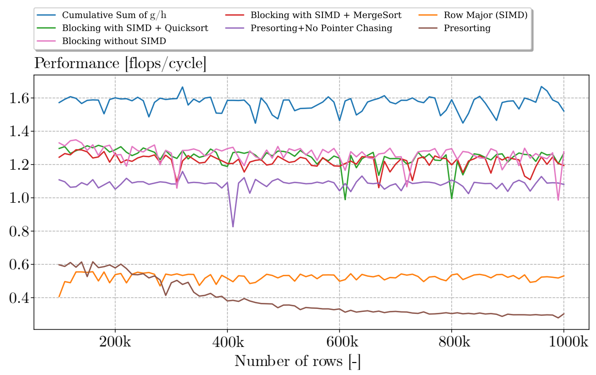

Fig. 5: Performance of splitfinding function

than 250k rows, the total memory footprint is larger than $250,000 \cdot 8 \approx 6MB$ which doesn't fit into L3 cache anymore. In this case, the non contiguous accesses in the baseline algorithm start incurring a higher penalty since the data is not cached, hence the performance drop. For the row major SIMD, in which the pointer chasing has been eliminated, this does not matter since all memory accesses are sequential. We only incur compulsory misses which explains why the performance stays roughly constant. The performance improvements (over presorted version) of best performing optimizations of each format are summarized in Table 2.

Roofline Analysis. In order to determine the extent to which this program can be further optimized, we make a roofline plot. We used the stream [10] and nova [2] benchmarks to experimentally determine the reference machine's read/write memory bandwidth. Fig. 6 shows the compute and memory bounds of our machine and the points represent various optimizations for the exact greedy splitfinding. Each optimization is represented by a single point as the operational intensity remains approximately constant across input sizes. The initial baseline implementation which performs sorting repeatedly is compute bound. However, all subsequent optimizations tend to be memory bound. The blocking and column major optimizations are very close to the roof. The vectorized implementations achieve around

[2]https://novabench.com

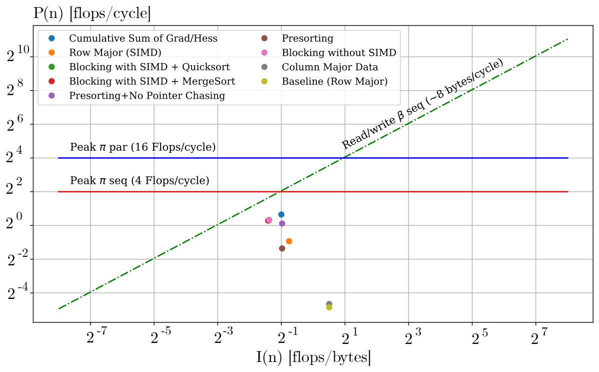

Fig. 6: Roofline Plot for Splitfinding

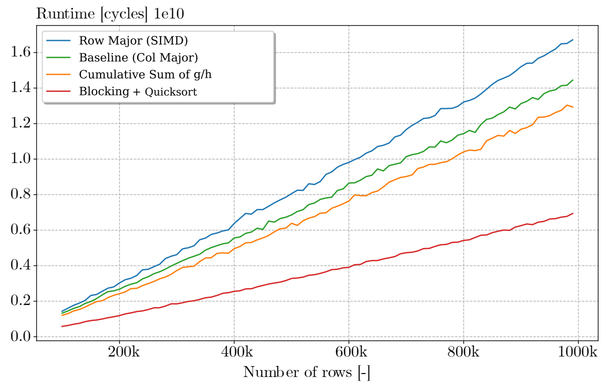

Fig. 7: Runtime for overall tree using Approx. Splitfinding

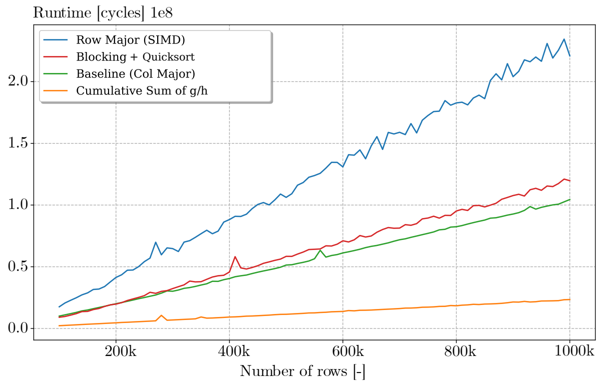

Fig. 8: Runtime of Approximate Splitfinding

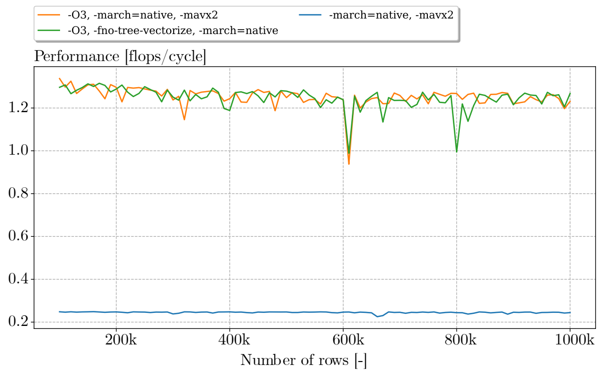

Fig. 9: Performance of Splitfinding Using Different Flags

half the maximum possible performance. We also tried loop unrolling for the vectorized implementations to drive them even closer to the roof but that didn't result in a meaningful increase in performance.

Approximate Splitfinding. Based on the results of the experiments on the exact algorithm, we implemented the best performing optimisations for the approximate algorithm in each of the three approaches, namely- ”row major with SIMD”, ”cumulative sum of g/h” and ”blocking with quicksort”. Fig. 7 shows that blocking with quicksort worked best for single tree creation and achieves a speedup of 2x over the baseline for larger inputs. This reinstates the observation wrt. optimal performance of blocking as discussed in the previous sections. Fig. 8 shows the variation of runtime of the approximate splitfinding algorithm with different input sizes. We observe that the ”Cumulative Sum of g/h” gives a speedup of roughly 4x on large inputs and has the best improvement in the runtime amidst others. This is mainly due to the fact that the cumulative sum calculation, a major numerical step in determining split points in the rest of the methods, is circumvented in this particular case because of the precomputed sums. It is important to note that the flop counts varied widely across these optimisations for approximate case. Thus we restrict ourselves to the discussion on the runtime improvements only.

Different Compiler Flags. Lastly, we tried different levels of compiler optimizations with the dual-purpose of evaluating our implementations (i.e. how much the compiler would be able to further optimize) and gaining an additional performance boost. These experiments were only done on our best performing version i.e. blocking with SIMD, for the exact greedy algorithm. However, we expect them to generalize to the approximate splitfinding algorithm as well. Figures 9 and 10 show the performance and runtime of the splitfinding and overall tree respectively with different levels of optimizations enabled. As expected, enabling compiler optimizations, without vectorizing, gave on average 2.4x and maximum of 3.3x improvement in total runtime. For the splitfinding, enabling optimizations without vectorizing, gave an average 5.1x and maximum 5.64x performance improvement.

When we enable vectorization with maximum optimization level, there is no significant improvement in the splitfinding function. This could be because our implementation is already close to both performance and memory bounds. As a result, the scope for improvement is limited. On the other hand, vectorization leads to a 4.3x improvement in overall runtime. This could be possible if the compiler vectorizes sections of the code that are dominated by memory I/O e.g. by reading/writing to memory using vector load/store operations which were not the focus of our work.

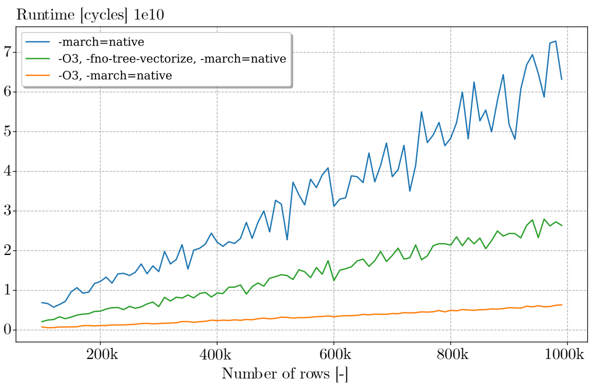

Fig. 10: Runtime of Complete Tree Using Different Flags

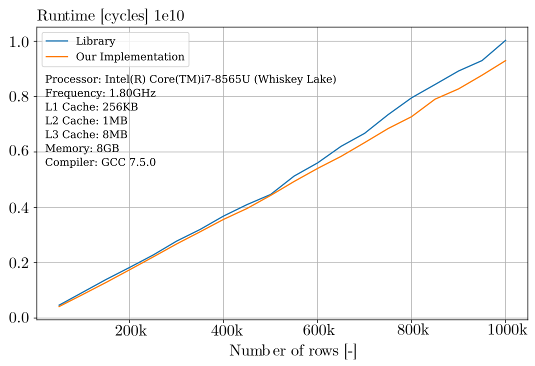

Fig. 11: Runtime Comparison with Library Implementation

4.2. Analysis

Building Multiple Trees. In practical situations, XGBoost algorithm is used to create multiple (not just one) trees. As shown in Figure 2(a), when building a single tree, using an efficient sorting algorithm helps more than presorting the data. However, this becomes less significant as more trees are built. Fig. 2(b) shows the fraction of time taken by each function versus the number of trees built. The contribution of sorting decreases as more trees are built. This is because, with presorting, the same data can be used to build all the trees; we do not have to sort again and again. Therefore, in a realistic scenario, optimizing sorting did not make sense. We can observe that the contribution of node splitting increases with the number of trees. Optimizing split required more algorithmic than implementation changes. Moreover, because it is dominated by I/O, with a lot of branching, vectorizing that function is non-trivial.

Memory Usage. We also performed analysis of the memory consumption of the entire framework with varying number of trees. Experiments show that our implementation scales well with number of trees of the order of 100. When creating trees with depth 7 using 100k instances, the memory usage increases from 50 MB to 53 MB when scaling from 1 to 10 tree and to 57 MB when creating 100 trees.

5. COMPARISON TO LIBRARY IMPLEMENTATION

We use the official implementation[3] of XGBoost mentioned in [3] for comparison. Since it was difficult to benchmark splitfinding routine independently due to the structure of library implementation, we decided to compare the runtime for entire tree creation framework. Figure 11 shows that our implementation is slightly faster than the library implementation for large input sizes, while giving almost similar runtime for smaller inputs.

We would like to bring it to attention of the reader that the focus of this work was optimizing splitfinding algorithm. Even though, we started off with a baseline implementation which was much slower as compared to the XGBoost library for entire tree creation, we are still able to beat the library by only optimizing the splitfinding algorithm. As we have pointed earlier that splitfinding only constitutes a small percentage of the total runtime and therefore, the improvements we achieve are more profound than it appears in Figure 11

These experiments were performed on a different machine as it was non-trivial to run the library implementation on macOS. Therefore, we decided to perform this particular experiment on a Ubuntu 18.04.4 machine. This experiment also confirms the applicability of our optimizations on a different machine with similar micro-architecture.

6. CONCLUSIONS

In this work, we optimized the performance of XGBoost using different techniques. We started by implementing the basic versions of exact and approximate splitfinding algorithms and identified the performance bottleneck of the framework. We then proceeded in two orthogonal directions simultaneously, when data is in row and column major order, until we could achieve the most optimal performing version in each direction. We leveraged the insights gained from both these experiments and finally implemented blocking, which gave the best performing results. We also tried SIMD vectorization in each of these experiments to obtain the most optimised code. Our results show that we are within 2x of the optimal performance which prove the efficacy of the optimisations. We finally compare our best performing code with the official library implementation of XGBoost and show that our work has an improvement in runtime. We also highlight some of the challenges in optimizations such as - 1. having relatively low algorithmic computations compared with data load and stores, 2. variability of flop counts in different optimizations 3. ineffectiveness of using instruction level parallelism near roof. We were able to identify these challenges and address them accordingly in our work.

[3]https://github.com/dmlc/xgboost/

7. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Given the nature of the problem statement for each individual's effort is highly overlapping, it required recurrent knowledge-sharing and constant help from the rest of the team members to realise any individual task.

Amory. Lead the effort for the Blocking implementation. This encompasses changing the access patterns and data storage to allow usage of the blocking format as well as optimizing the implementation by removing optimization blockers using techniques such as applying loop unrolling, strength reduction and other relevant techniques. Also worked on vectorizing the blocking implementation using SIMD intrinsics. To achieve this, the realloc function had to be implemented to allow for aligned memory reallocations and implemented the bit masking trick illustrated in table 1 to achieve full performance potential for the SIMD blocking implementation. In the end, also experimented with converting the data format to floats and using `qsort` instead of `mergesort`. Also helped analyzing the cost of the different functions, analyzed the memory usage, examined the runtime contributions of the different functions of our XGBoost implementation and identified possible bottlenecks using profiling tools.

Fizza. Lead the effort for Column Major optimizations in the first algorithm. Re-implemented the baseline in column major order and built further on it. This included implementing presorting, eliminating pointer chasing and precomputing the derivative statistics. To this end, worked on restructuring the entire code base to work with the new data format without compromising the correctness of the program. Tried standard C optimizations such as loop unrolling, code motion, data prefetching etc. Also tried to optimize the Split and Splitfinding functions using SIMD. Reasoned with all the others about why an optimization does or doesn't work. Benchmarking of all the optimizations/implementations (for both algorithms) documented in this report. Creating runtime/performance plots. Worked on roofline plot, including benchmarking to determine maximum memory bandwidth. Played around with compiler flags. Cost and memory usage analysis of both algorithms (including for independent functions) and their verification in every implemented optimization.

Noman. Lead the effort for Row Major optimizations which was the initial platform for all the further optimizations. Performed bottleneck measurements to identify sorting as the bottleneck. Focused on optimizations specific to row major such as presorting, using bucketsort and using vector intrinsics. Tried out further optimizations from the course such as loop unrolling, code motion, strength reduction and so on. The effort was made with the objective to achieve the best performing code within the constraints of row major data layout. Also worked on optimizing memory footprints of the entire framework whilst keeping all the memory leaks in check. Helped analyzing the cost of various routines and giving reasoning for all the observations. Collaborated with Rishabh on installation and benchmarking of XGBoost Library implementation with our best performing implementation. Worked with Amory on further reduction of intrinsics calls in blocking SIMD implementation.

Rishabh. Lead the effort on optimizing variant 2 of the splitfinding algorithm, precisely the approximate split finding. It first required understanding and implementing baseline code. Identified the potential bottlenecks with the help of rest of the team members and commenced optimising them with standard optimisation tricks mentioned in the class, namely loop unrolling, strength reduction, code motion, data-prefetching etc. Optimised the row major implementation of the algorithm using SIMD vectorisation. Optimised the column major implementation of the algorithm. Improved this particular case with cumulative summation of first and second order derivatives and storing the precomputed values beforehand for faster runtime. Collaborated with Amory in order to come up with blocking technique for approximate variant of splitfinding algorithm . Implemented the blocking optimisation and analysed the implications of the results consulting other team members. Collaborated with Noman on installing and benchmarking the official library implementation of the XGBoost. Collaborated with Fizza for the cost analysis, memory usage and verification of each of the optimizations in the approximate algorithm.

8. REFERENCES

[1] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2017, NIPS'17, p. 3149–3157, Curran Associates Inc.

[2] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo, "Planet: Massively parallel learning of tree ensembles with mapreduce," in *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB-2009)*, 2009.

[3] Tianqi Chen and Carlos Guestrin, "Xgboost: A scalable tree boosting system," 08 2016, pp. 785–794.

[4] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: Simplified data processing on large clusters," in

OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004, pp. 137–150.

[5] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tie-Yan Liu, "A communication-efficient parallel algorithm for decision tree," 2016.

[6] Stephen Tyree, Kilian Weinberger, Kunal Agrawal, and Jennifer Paykin, "Parallel boosted regression trees for web search ranking," 01 2011.

[7] Huan Zhang, Si Si, and Cho-Jui Hsieh, "Gpu-acceleration for large-scale tree boosting," 2017.

[8] Rory Mitchell, Andrey Adinets, Thejaswi Rao, and Eibe Frank, "Xgboost: Scalable gpu accelerated learning," 2018.

[9] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, "Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

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