GPUTreeShap: Fast Parallel Tree Interpretability

Rory Mitchell [1 2] **Eibe Frank** [2] **Geoffrey Holmes** [2]

Abstract

SHAP (SHapley Additive exPlanation) values (Lundberg & Lee, 2017) provide a game theoretic interpretation of the predictions of machine learning models based on Shapley values (Shapley, 1953). While SHAP values are intractable in general, a recursive polynomial time algorithm specialised for decision tree models is available, named TreeShap (Lundberg et al., 2020). Despite its polynomial time complexity, TreeShap can become a significant bottleneck in practical machine learning pipelines when applied to large decision tree ensembles. We present GPUTreeShap, a software package implementing a modified TreeShap algorithm in CUDA for Nvidia GPUs. Our approach first preprocesses the input model to isolate variable sized sub-problems from the original recursive algorithm, then solves a bin packing problem, and finally maps sub-problems to streaming multiprocessors for parallel execution with specialised hardware instructions. With a single GPU, we achieve speedups of up to 19x for SHAP values, and 340x for SHAP interaction values, over a state-of-the-art multi-core CPU implementation. We also experiment with an 8 GPU DGX-1 system, demonstrating throughput of 1.2M rows per second—equivalent CPU-based performance is estimated to require 6850 CPU cores.

1 Introduction

Explainability and accountability are important for practical applications of machine learning, but the interpretation of complex models such as neural networks, gradient boosting ensembles, or random forests is challenging. Recent literature (Ribeiro et al., 2016; Selvaraju et al., 2017; Guidotti et al., 2018) describes methods for "local interpretability" of these models, enabling the attribution of predictions for individual examples to component features. One such method calculates so-called SHAP (SHapley Additive exPlanation) values quantifying the contribution of each feature to a prediction. In contrast to other methods, SHAP values exhibit several unique properties that appear to agree with human intuition (Lundberg et al., 2020). Although SHAP values generally take exponential time to compute, the special structure of decision trees admits a polynomial time algorithm. This algorithm, implemented alongside state-of-the-art gradient boosting libraries such as XGBoost (Chen & Guestrin, 2016) and LightGBM (Ke et al., 2017), enables complex decision tree ensembles with state-of-the-art performance to also output interpretable predictions.

However, despite improvements to algorithmic complexity and software implementation, computing SHAP values from tree ensembles remains a computational concern for practitioners, particularly as model size or size of test data increases: generating SHAP values can be more time-consuming than training the model itself. We address this problem by reformulating the recursive TreeShap algorithm, taking advantage of parallelism and increased computational throughput available on modern GPUs. We provide an open source module named GPUTreeShap implementing a high throughput variant of this algorithm. GPUTreeShap is integrated as a backend to the XGBoost library, providing significant improvements to runtime over its existing multi-core CPU-based implementation.

[1]Nvidia Corporation [2]Department of Computer Science, University of Waikato, Hamilton, New Zealand. Correspondence to: Rory Mitchell <ramitchellnz@gmail.com>.

2 Background

In this section, we briefly review the definition of SHAP values for individual features and the TreeShap algorithm for computing these values from decision tree models. We also review an extension of SHAP values to second-order interaction effects between features.

2.1 SHAP Values

SHAP values are defined as the coefficients of the following additive surrogate explanation model g, a linear function of binary variables

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i \tag{1}$$

where M is the number of features, $z' \in \{0,1\}^M$, and $\phi_i \in \mathbb{R}$. z'_i indicates the presence of a given feature and ϕ_i its relative contribution to the model output. The surrogate model $g(z')$ is a *local* explanation of a prediction $f(x)$ generated by the model for a feature vector x, meaning that a unique explanatory model may be generated for any given

x. The SHAP values are computed as

$$\phi_i = \sum_{S \subseteq M \setminus \{i\}} \frac{|S|!(|M| - |S| - 1)!}{|M|!} [f_{S \cup \{i\}}(x) - f_S(x)] \tag{2}$$

where M is the set of all features and $f_S(x)$ describes the model output restricted to feature subset S. Equation 2 considers all possible subsets, and so has runtime exponential in the number of features.

Given a trained decision tree model f and data instance x, it is not necessarily clear how to restrict model output $f(x)$ to feature subset S—when feature j is not present in subset S along a given branch of the tree, and a split condition testing j is encountered, then how do we choose which path to follow to obtain a prediction for x? (Lundberg et al., 2020) defines a conditional expectation for the decision tree model $E[f(x)|x_S]$, where the split condition on feature j is evaluated using a random variable with distribution drawn from the training set used to build the model. In effect, when a decision tree branch is encountered and the feature to be tested is not in the active subset S, we take the output of *both* the left and right branch in proportion to some distribution for that feature. More specifically, we use the proportion of weighted instances that flow down the left or right branch during model training. This process is also how the C4.5 decision tree learner deals with missing values (Quinlan, 1993). We refer to it as "cover weighting" in what follows.

Given this interpretation of missing features, (Lundberg et al., 2020) gives a polynomial time algorithm for efficiently solving Equation 2, named TreeShap. The algorithm exploits the specific structure of decision trees, where the model is additive in the contribution of each leaf. Equation 2 can thus be independently evaluated for each unique path from root to leaf node. These unique paths are then processed using a quadratic-time dynamic programming algorithm. The intuition of the algorithm is to keep track of the proportion of all feature subsets that flow down each branch of the tree, weighted according to the length of each subset $|S|$, as well as the proportion that flow down the left and right branches when the feature is missing.

For ease of reference, Algorithm 1 shows this recursive polynomial time algorithm, where m is a list representing the path of unique features split on so far. Each list element has four attributes: d is the feature index, z is the fraction of paths that flow through the current branch when the feature is not present, o is the fraction of paths that flow through the branch when the feature is present, and w is the proportion of sets of a given cardinality that are present. The decision tree is represented by the set of lists $\{v, a, b, t, r, d\}$, where each list element corresponds to a given tree node, with v containing leaf values, a pointers to the left children, b pointers to the right children, t the split condition, r the weights of training instances, and d the feature indices. At a high level, the algorithm proceeds by stepping through a path in the decision tree of depth D from root to leaf. According to Equation 2, we have a different weighting for the size of each feature subset, although we can accumulate feature subsets of the same size together. As the algorithm advances down the tree, it calls the method EXTEND, taking a new feature split and accumulating its effect on all possible feature subsets of length $1, 2, \ldots$ up to the current depth. The UNWIND method is used undo a feature which has been added to the path via EXTEND. UNWIND and EXTEND are commutative and can be called in any order. UNWIND may be used to remove duplicate feature occurrences from the path and to compute the final SHAP values. When the recursion reaches a leaf, the SHAP values ϕ_i for each feature present in the path are computed by calling UNWIND on feature i (line 7), temporarily removing it from the path; then, the overall effect of switching that feature on or off is

Algorithm 1 TreeShap

```
1:  function TS(x, tree)
2:      φ = array of len(x) zeroes
3:      function RECURSE(j, m, p_z, p_o, p_i)
4:          m = EXTEND(m, p_z, p_o, p_i)
5:          if j == leaf then
6:              for i ← 2 to len(m) do
7:                  w = sum(UNWIND(m, i).w)
8:                  φ_{m_i} = φ_{m_i} + w(m_i.o − m_i.z)v_j
9:          else
10:             h, c = x_{d_j} ≤ t_j ? (a_j, b_j) : (b_j, a_j)
11:             i_z = i_o = 1
12:             k = FINDFIRST(m.d, d_j)
13:             if k ≠ nothing then
14:                 i_z, i_o = (m_k.z, m_k.o)
15:                 m = UNWIND(m, k)
16:             RECURSE(h, m, i_z r_h / r_j, i_o, d_j)
17:             RECURSE(c, m, i_z r_c / r_j, 0, d_j)
18:     function EXTEND(m, p_z, p_o, p_i)
19:         l = len(m)
20:         m = copy(m)
21:         m_{l+1}.(d, z, o, w) = (p_i, p_z, p_o, l = 0 ? 1 : 0)
22:         for i ← l − 1 to 1 do
23:             m_{i+1}.w = m_{i+1}.w + p_o m_i.w(i/l)
24:             mi.w = p_z m_i.w[(l − i)/l]
25:         return m
26:     function UNWIND(m, i)
27:         l = len(m)
28:         n = m_l.w
29:         m = copy(m_{1...l−1})
30:         for j ← l − 1 to 1 do
31:             if m_i.o ≠ 0 then
32:                 t = m_j.w
33:                 m_j.w = n · l/(j · m_i.o)
34:                 n = t − m_j.w · m_i.z((l − j)/l)
35:             else
36:                 m_j.w = (m_j.w · l)/(m_i.z(l − j))
37:         for j ← i to l − 1 do
38:             m_j.(d, z, o) = m_{j+1}.(d, z, o)
39:         return m
40:     RECURSE(1, [], 1, 1, 0)
41:     return φ
```

adjusted by adding w multiplied by $(p_0 - p_z)$ to ϕ_i.

Given an ensemble of T decision trees, Algorithm 1 runs in time proportional to $O(TLD^2)$, using memory $O(D^2+M)$, where L is the number of leaves for each tree, D is the tree depth, and M the number of features. In this paper, we adapt Algorithm 1 for massively parallel GPUs.

2.2 SHAP Interaction Values

In addition to the first-order feature relevance metric defined above, (Lundberg et al., 2020) also provides an extension of SHAP values to second-order relationships between features, termed SHAP Interaction Values. This method applies the game theoretic SHAP interaction index (Fujimoto et al., 2006), defining a matrix of interactions as

$$\phi_{i,j} = \sum_{S \subseteq M \setminus \{i,j\}} \frac{|S|!(M - |S| - 2)!}{2(M-1)!} \nabla_{ij}(S) \tag{3}$$

for $i \neq j$, where

$$\nabla_{ij}(S) = f_{S\cup\{i,j\}}(x) - f_{S\cup\{i\}}(x) - f_{S\cup\{j\}}(x) + f_S(x) \tag{4}$$

$$= f_{S\cup\{i,j\}}(x) - f_{S\cup\{j\}}(x) - [f_{S\cup\{i\}}(x) - f_S(x)] \tag{5}$$

with diagonals

$$\phi_{i,i} = \phi_i - \sum_{j \neq i} \phi_{i,j} \tag{6}$$

Interaction values can be efficiently computed by connecting Eq. 5 to Eq. 2, for which we have the polynomial time TreeShap algorithm. To compute $\phi_{i,j}$, TreeShap should be evaluated twice for ϕ_i, where feature j is alternately considered fixed to present and not present in the model. To evaluate TreeShap for a unique path conditioning on j, the path is extended as normal, if feature j is encountered, it is *not included* in the path (the dynamic programming solution is not extended with this feature, instead skipping to the next feature). If j is considered not present, the resulting ϕ_i is weighted according to the probability of taking the left or right branch (cover weighting) at a split on feature j. If j is considered present, we evaluate the decision tree split condition $x_j < t_j$ and discard ϕ_i from the path not taken.

To compute interaction values for all pairs of features, TreeShap can be evaluated M times, leading to time complexity of $O(TLD^2M)$. Interaction values are challenging to compute in practice, with runtimes and memory requirements significantly larger than decision tree induction itself. In Section 3.5, we show how to adapt this algorithm to the GPU and how to improve runtime to $O(TLD^3)$ (tree depth D is normally much smaller than the number of features M present in the data).

3 GPUTreeShap

Algorithm 1 has properties that make it unsuitable for direct implementation on GPUs in a performant way. GPUs are massively parallel processors optimised for throughput, in contrast to conventional CPUs which optimise for latency. GPUs in use today consist of many streaming multiprocessor (SM) units with 32 single instruction, multiple thread (SIMT) lanes: threads operate in lockstep in groups of 32 called a "warp". Warps are generally executed on SMs without order guarantees, enabling latency in warp execution (e.g., from global memory loads) to be hidden by switching to other warps that are ready for execution (NVIDIA Corporation, 2020). Conventional multithreaded CPU implementations of Algorithm 1 achieve parallel work distribution by instances (Chen & Guestrin, 2016; Ke et al., 2017). For example, interpretability results for input matrix X are computed by launching one parallel CPU thread per row (i.e., data instance being evaluated). While this approach is embarrassingly parallel, CPU threads do correspond directly to the notion of GPU threads. If threads in a warp take divergent branches, performance is reduced, as all threads must execute identical instructions when they are active (Harris & Buck, 2005). Moreover, GPUs can suffer from per-thread load balancing problems—if work is unevenly distributed between threads in a warp, finished threads stall until all threads in the warp are finished. Additionally, GPU threads are more resource constrained than their CPU counterparts, having a smaller number of available registers due to limited per-SM resources. Excessive register usage results in reduced SM occupancy by limiting the number of concurrent warps. It also results in register spills to global memory, causing memory loads at significantly higher latency.

To mitigate these issues, we segment the TreeShap algorithm to obtain fine-grained parallelism, observing that each unique path from root to leaf in Algorithm 1 can be constructed independently (see from Algorithm 1 that the resulting ϕ_i at each leaf are additive and depend only on features encountered on that unique path from root to leaf). Instead of allocating one thread per tree, we allocate a *group* of threads to cooperatively compute SHAP values for a given unique path through the tree. We launch one such group of threads for each (unique path, evaluation instance) pair, computing all SHAP values for this pair in a single GPU kernel. This method requires preprocessing to arrange the tree ensemble into a suitable form, avoid some less GPU-friendly operations of the original algorithm, and partition work efficiently across GPU threads. Our GPUTreeShap implementation can be summarised by the following high-level steps:

1. Preprocess the ensemble to recover all unique decision tree paths.
2. Combine duplicate features along each path.

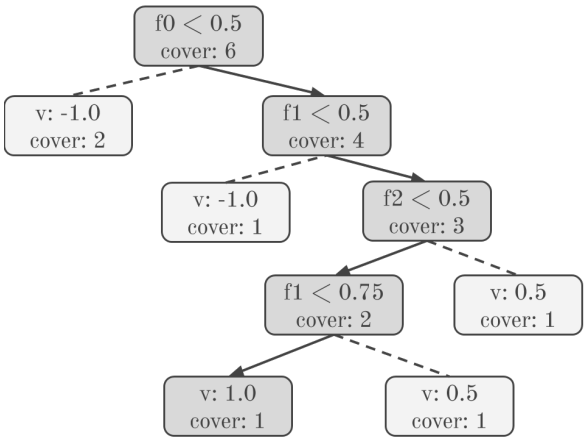

Figure 1. Unique decision tree path

```
struct PathElement {
  // Unique path index
  size_t path_idx;
  // Feature of this split, -1 is root
  int64_t feature_idx;
  // Range of feature value
  float feature_lower_bound;
  float feature_upper_bound;
  // Probability of following this path
  // when feature_idx is missing
  float zero_fraction;
  // Leaf weight at the end of path
  float v;
};
```

Listing 1: Path element structure

3. Partition path subproblems to GPU warps by solving a bin packing problem.
4. Launch a GPU kernel solving the dynamic programming subproblems in batch.

These steps are described in more detail below.

3.1 Recover Paths

Figure 1 shows a decision tree model, highlighting a unique path from root to leaf. The SHAP values computed by Algorithm 1 are an additive combination of the SHAP values from every unique path in the tree. Note that the decision tree encodes information about the weight of training instances that flow down paths in the *cover* variable. We preprocess the input ensemble into sub-lists for all possible unique paths. Path elements are represented as per Listing 1.

We use a lower and upper bound to represent the range of feature values that can flow through a particular branch of the tree when the feature is present. For example, the root node in Figure 1 has split condition $f_0 < 0.5$. Therefore, if the feature is present, the left branch from this node contains instances where $-\infty \leq f_0 < 0.5$, and the right branch contains instances where $0.5 \leq f_0 < \infty$. This representation is useful for the next step of combining duplicate features. Figure 2 shows the unique paths extracted from Figure 1. An entire tree ensemble can be represented in this form, containing sufficient information to compute its SHAP values.

3.2 Remove Duplicate Features

Part of the complexity of Algorithm 1 comes from a need to detect and handle multiple occurrences of a feature in a single unique path. In line 12, the candidate feature of the current recursion step is checked against existing features in the path. If a previous occurrence is detected, it is removed from the path using the UNWIND function. The p_z and p_o values for the old and new occurrences of the feature are multiplied, and the path extended with these new values. Unwinding previous features is problematic for GPU implementation. It requires threads to cooperatively evaluate FINDFIRST and then UNWIND, introducing branching as well as extra computation.

Instead, we take the approach of preprocessing the ensemble in path element form, combining duplicate features into a single occurrence. To do this, recognise that a path through a decision tree from root to leaf represents a single hypercube in the M dimensional feature space, with boundaries defined according to split conditions along the path. The boundaries of the hypercube may alternatively be represented with a lower and upper bound on each feature. Therefore, any number of decision tree splits over a single feature can be reduced to a single range, represented by these bounds.

The next observation we make is that the ordering of features within a path is irrelevant to the final SHAP values. As noted in (Lundberg et al., 2020), the EXTEND and UNWIND functions defined in Algorithm 1 are commutative; therefore, features may be added or removed to the path in any order, and we can sort unique path representations by feature index, combining consecutive features into a single path element.

3.3 Bin Packing

Each unique path sub-problem identified above is mapped to GPU warps for hardware execution. A decision tree ensemble contains L unique paths, where L is the number of leaves, and each path has length between 1 and maximum tree depth D. Given a 32 thread warp, multiple paths may be resident and executed concurrently on a single warp. As we wish to use fast warp intrinsics and avoid synchronisation, sub-problems are constrained to not overlap across warps[1]. To achieve the highest device utilisation, given this

[1]Note that our approach does constrain the maximum depth of a decision tree to less than or equal to the GPU warp size of 32. Given that the number of nodes in a balanced decision tree

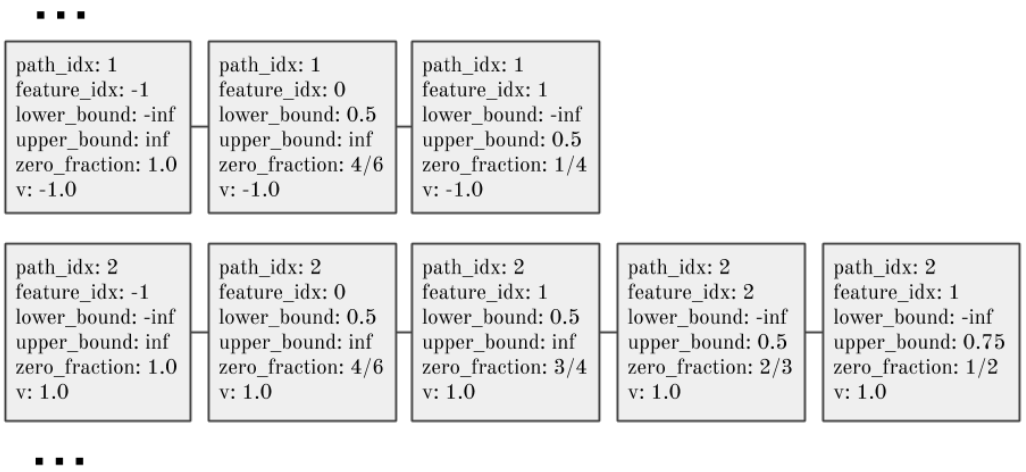

Figure 2. Two unique paths from Figure 1

constraint, path sub-problems should be mapped to warps such that the total number of warps is minimised. This is a standard bin packing problem. Given a finite set of items I, with sizes $s(i) \in Z^+$, for each $i \in I$, and maximum bin capacity B, I can be partitioned into the disjoint sets $I_0, I_1, \ldots, I_K$ such that the sum of sizes in each set is less than B. The partitioning minimising K is the optimal bin packing solution. In our case, the bin capacity, $B = 32$, is the number of threads per warp and our item sizes, $s(i)$, are given by the unique path lengths from the input ensemble. In general, finding the optimal packing is strongly NP-complete (Garey & Johnson, 1979), although there are heuristics that can achieve close to optimal performance in many cases. In Section 4.1, we evaluate three standard heuristics for the off-line bin packing problem, *Next-Fit* (NF), *First-Fit-Decreasing* (FF), and *Best-Fit-Decreasing* (BFD), as well as a baseline where each item is placed in its own bin. We briefly describe these algorithms and refer the reader to (Martello & Toth, 1990) or (Coffman et al., 1997) for a more in-depth survey.

Next-Fit is a simple algorithm, where only one bin is open at a time. Items are added to the bin as they arrive. If bin capacity is exceeded, the bin is closed and a new bin is opened. *First-Fit-Decreasing* sorts the list of items by non-increasing size. Then, beginning with the largest item, it searches for the first bin with sufficient capacity and adds it to the bin. *Best-Fit-Decreasing* also sorts items by non-increasing size, but assigns items to the feasible bin with the smallest residual capacity. FFD and BFD may be implemented in $O(n \log n)$ time using a tree data structure allowing bin updates and insertions in $O(\log n)$ operations (see (Johnson, 1974) for specifics).

increases exponentially with depth, and real-world experience showing $D \leq 16$ almost always, we believe this to be a reasonable constraint.

Table 1. Bin packing time complexities and worst-case approximation ratios

ALGORITHM	TIME	R_A
NF	$O(n)$	2.0
FFD	$O(n \log n)$	1.222
BFD	$O(n \log n)$	1.222

Existing literature gives worst-case approximation ratios for the above heuristics. For a given set of items I, $A(I)$ describes the number of bins used by algorithm A, and $OPT(I)$ the number of bins for the optimal solution. The approximation ratio $R_A \leq \frac{A(I)}{OPT(I)}$ describes the worst case performance ratio for any possible I. Time complexities and approximation ratios for each bin packing heuristic are listed in Table 1, as per (Coffman et al., 1997).

As this paper concerns the implementation of GPU algorithms, we would ideally formulate the above heuristics in parallel. Unfortunately, the bin packing problem is known to be hard to parallelise. In particular, FFD and BFD are P-complete, indicating that it is unlikely that these algorithms may be sped up significantly via parallelism (Anderson & Mayr, 1984). An efficient parallel algorithm with the same approximation ratio as FFD/BFD is given in (Anderson et al., 1989), but the adaptation of this algorithm to GPU architectures is nontrivial and beyond the scope of this paper. As shown by our evaluation in Section 3.3, CPU based bin packing heuristics give acceptable performance for this task, such that the main burden of computation still falls on the GPU kernels computing SHAP values in the subsequent step. We perform experiments comparing the three bin pack-

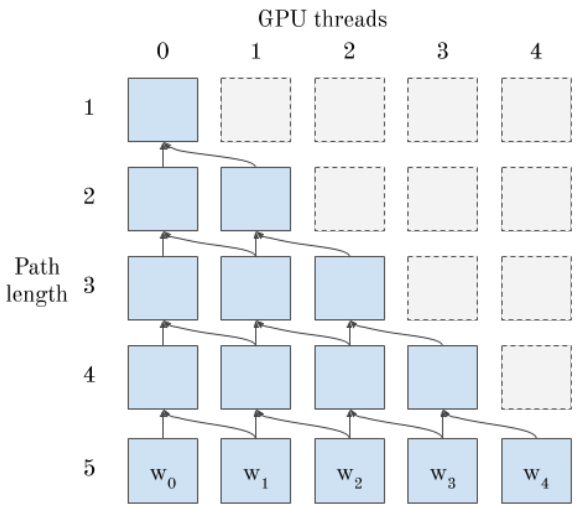

Figure 3. Data dependencies of EXTEND

ing heuristics in Section 4.1 in terms of runtime and impact on efficiency for GPU kernels.

3.4 GPU Kernel

Given a unique decision tree path, extracted from the recursive Algorithm 1 and with duplicates removed, we allocate one path element per GPU thread and cooperatively evaluate SHAP values for each row in the dataset X. Listing 2 provides a simplified overview of this process. The dataset X is assumed to be queryable from the device. To evaluate Algorithm 1, it remains to compute permutation weights for each possible feature subset with the EXTEND function (line 4), as well as to UNWIND each feature in the path and calculate the sum of permutation weights (line 7). The EXTEND function represents a single step in a dynamic programming problem. Figure 3 shows the data dependency of each call to EXTEND on previous iterations. Each thread depends on its previous result and the result of the thread to its left. This dependency pattern leads to a natural implementation using warp shuffle instructions, where threads directly access registers of other threads in the warp at considerably lower cost than shared or global memory. Listing 3 shows CUDA code for a single step of the EXTEND function on the device. Given the permutation weights for the entire path, it is then necessary to UNWIND the effect of each individual feature from the path (Algorithm 1, line 7) to evaluate its relative contribution. We distribute this task among threads, with each thread "unwinding" a unique feature. Device code for UNWOUNDSUM is given in Listing 4.

```
__device__ float GetOneFraction(
    const PathElement& e, DatasetT X, size_t row_idx) {
  // First element in path (bias term) is always zero
  if (e.feature_idx == -1) return 0.0;
  // Test the split
  // Does the training instance continue down this
  // path if the feature is present?
  float val = X.GetElement(row_idx, e.feature_idx);
  return val >= e.feature_lower_bound &&
         val < e.feature_upper_bound;
}

template <typename DatasetT>
__device__ float ComputePhi(
    const PathElement& e, size_t row_idx,
    const DatasetT& X,
    const ContiguousGroup& group,
    float zero_fraction) {
  float one_fraction = GetOneFraction(e, X, row_idx);
  GroupPath path(group, zero_fraction, one_fraction);
  size_t unique_path_length = group.size();
  // Extend the path
  for (auto unique_depth = 1ull;
       unique_depth < unique_path_length;
       unique_depth++) {
    path.Extend();
  }
  float sum = path.UnwoundPathSum();
  return sum * (one_fraction - zero_fraction) * e.v;
}

template <typename DatasetT, size_t kBlockSize,
          size_t kRowsPerWarp>
__global__ void ShapKernel(
    DatasetT X, size_t bins_per_row,
    const PathElement* path_elements,
    const size_t* bin_segments, size_t num_groups,
    float* phis) {
  __shared__ PathElement s_elements[kBlockSize];
  PathElement& e = s_elements[threadIdx.x];
  // Allocate some portion of rows to this warp
  // Fetch the path element assigned to this
  // thread
  size_t start_row, end_row;
  bool thread_active;
  ConfigureThread<DatasetT, kBlockSize, kRowsPerWarp>(
      X, bins_per_row, path_elements,
      bin_segments, &start_row, &end_row, &e,
      &thread_active);
  if (!thread_active) return;
  float zero_fraction = e.zero_fraction;
  auto labelled_group =
      active_labeled_partition(e.path_idx);
  for (int64_t row_idx = start_row;
       row_idx < end_row; row_idx++) {
    float phi =
        ComputePhi(e, row_idx, X, labelled_group,
                   zero_fraction);
    // Write results
    if (!e.IsRoot()) {
      atomicAdd(&phis[IndexPhi(
                    row_idx, num_groups, e.group,
                    X.NumCols(), e.feature_idx)],
                phi);
    }
  }
}
```

Listing 2: GPU kernel overview

```
// Cooperatively extend the path with a group of
// threads Each thread maintains pweight for its
// path element in register
__device__ float Extend(int unique_depth,
                        float zero_fraction,
                        float one_fraction,
                        float pweight, GroupT g) {
  size_t rank = g.thread_rank();
  float inv_unique_depth =
      1.0f / static_cast<float>(unique_depth + 1);
  float new_zero_fraction =
      g.shfl(zero_fraction, unique_depth);
  float new_one_fraction =
      g.shfl(one_fraction, unique_depth);
  float left_pweight = g.shfl_up(pweight, 1);
  // pweight of threads with rank >= unique_depth_
  // is 0 We use max(x,0) to avoid using a branch
  pweight *= new_zero_fraction *
             max(unique_depth - rank, 0llu) *
             inv_unique_depth;
  // left_pweight is undefined for leftmost thread
  // but rank = 0, so expression evaluates to 0
  pweight += new_one_fraction * left_pweight *
             rank * inv_unique_depth;
  return pweight;
}
```

Listing 3: EXTEND device code

```
// Each thread unwinds the path for its feature
// and returns the sum rank 0 returns a bias term
// - the product of all pz
__device__ float UnwoundSum(int unique_depth,
                            float pweight,
                            float zero_fraction,
                            float one_fraction,
                            GroupT g) {
  float next_one_portion =
      g.shfl(pweight, unique_depth);
  float total = 0.0f;
  const float zero_frac_div_unique_depth =
      zero_fraction /
      static_cast<float>(unique_depth + 1);
  for (int i = unique_depth - 1; i >= 0; i--) {
    float ith_pweight = g.shfl(pweight, i);
    float precomputed =
        (unique_depth - i) *
        zero_frac_div_unique_depth;
    float tmp =
        (next_one_portion * unique_depth + 1) /
        (i + 1);
    total += tmp * one_fraction;
    next_one_portion =
        ith_pweight - tmp * precomputed;
    total +=
        ((1.0f - one_fraction) * ith_pweight) /
        precomputed;
  }
  if (g.thread_rank() == 0) {
    return pweight * (unique_depth + 1);
  }
  return total;
}
```

Listing 4: UNWOUNDSUM device code

3.5 Interaction values

Computation of SHAP interaction values makes use of the same preprocessing steps as above, and the same basic kernel building blocks, except that each row/path pair evaluates SHAP values multiple times, iterating over each unique feature and conditioning on that feature as fixed or not present. There are some difficulties in conditioning on features with our algorithm formulation so far—conditioning on a feature j requires ignoring it and not adding it to the active path. This introduces complexity when neighbouring threads are communicating via shuffle instructions (see Figure 3). Each thread must adjust its indexing to skip over a path element being conditioned on. We found a more elegant solution is to swap a conditioned path element to the end of the path, then simply stop before adding it to the path (recall that the ordering of path elements is irrelevant). Thus, to evaluate SHAP interaction values, we use a kernel similar to Listing 2, except that we loop over each unique feature, conditioning on that feature as on or off. Device code for this is shown in Listing 5.

One major difference that arises between our GPU implementation and the CPU implementation of (Lundberg et al., 2020), is that we can easily avoid conditioning on features that are not present in a given path. See Listing 5, line 63, which describes a loop over path elements, thus avoiding iteration over all features. It is clear from Equation 5 that if we condition on feature j, and it is not present in the path, then $f_{S\cup\{i,j\}}(x) = f_{S\cup\{i\}}(x)$, $f_{S\cup\{j\}}(x) = f_S(x)$ and $\nabla_{ij}(S) = 0$. Therefore, our approach has runtime proportional to $O(TLD^3)$ instead of $O(TLD^2M)$ by exploiting the limited subset of possible feature interactions in a tree branch. This modification has a significant impact on runtime in practice (normally $M >> D$), as shown by our evaluation in Section 4.3.

4 Evaluation

We train a range of decision tree ensembles for evaluation using XGBoost on the datasets listed in Table 2. Our goal is to evaluate a wide range of models representative of different real-world settings, from simple exploratory models to large ensembles of thousands of trees. For each dataset, we train a small, medium, and large variant by adjusting the number of boosting rounds (10, 100, 1000) and maximum tree depth (3, 8, 16). The learning rate is set to 0.01 to prevent XGBoost learning the model in the first few trees and producing only stumps in subsequent iterations. Using a low learning rate is also common in practice to minimise generalisation error. Other hyperparameters are left as default. Summary statistics for each model variant are listed in Table 3, and our testing hardware is listed in Table 4.

```
template <typename DatasetT>
__device__ float ComputePhiCondition(
    const PathElement& e, size_t row_idx,
    const DatasetT& X,
    const ContiguousGroup& group,
    int64_t condition_feature) {
  float one_fraction =
      GetOneFraction(e, X, row_idx);
  GroupPath path(group, e.zero_fraction,
                 one_fraction);
  size_t unique_path_length = group.size();
  float condition_on_fraction = 1.0f;
  float condition_off_fraction = 1.0f;
  // Extend the path
  for (auto i = 1ull; i < unique_path_length;
       i++) {
    // Skip conditioned feature
    if (group.shfl(e.feature_idx, i) ==
        condition_feature) {
      condition_on_fraction =
          group.shfl(one_fraction, i);
      condition_off_fraction =
          group.shfl(e.zero_fraction, i);
    } else {
      path.Extend();
    }
  }
  float sum = path.UnwoundPathSum();
  if (e.feature_idx == condition_feature) {
    return 0.0f;
  }
  float phi = sum *
              (one_fraction - e.zero_fraction) *
              e.v;
  return phi *
         (condition_on_fraction -
          condition_off_fraction) *
         0.5f;
}

template <typename DatasetT, size_t kBlockSize,
          size_t kRowsPerWarp>
__global__ void ShapInteractionsKernel(
    DatasetT X, size_t bins_per_row,
    const PathElement* path_elements,
    const size_t* bin_segments, size_t num_groups,
    float* phis_interactions) {
  // Configure start/end row
  // Load path element for each thread
  // ...

  for (int64_t row_idx = start_row;
       row_idx < end_row; row_idx++) {
    // Compute phi_i for the diagonal
    float phi =
        ComputePhi(*e, row_idx, X, labelled_group,
                   e->zero_fraction);
    // Write phi to diagonal
    // ...

    // Iterate over all present features
    for (auto condition_rank = 1ull;
         condition_rank < labelled_group.size();
         condition_rank++) {
      e = &s_elements[threadIdx.x];
      int64_t condition_feature =
          labelled_group.shfl(e->feature_idx,
                              condition_rank);
      // Swap the path element we are conditioning
      // on to the end of the path
      SwapConditionedElement(&e, s_elements,
                             condition_rank,
                             labelled_group);

      float x = ComputePhiCondition(
          *e, row_idx, X, labelled_group,
          condition_feature);
      // Write x to off diagonal
      // Subtract x from diagonal
      // ...
    }
  }
}
```

Listing 5: Feature interactions kernel

Table 2. Datasets

NAME	ROWS	COLS	TASK	CLASSES	REF
COVTYPE	581012	54	CLASS	8	(BLACKARD, 1998)
CAL_HOUSING	20640	8	REGR	-	(PACE & BARRY, 1997)
FASHION_MNIST	70000	785	CLASS	10	(XIAO ET AL., 2017)
ADULT	48842	14	CLASS	2	(KOHAVI, 1996)

Table 3. Models

MODEL	TREES	LEAVES	MAX_DEPTH
COVTYPE-SMALL	80	560	3
COVTYPE-MED	800	113888	8
COVTYPE-LARGE	8000	6636440	16
CAL_HOUSING-SMALL	10	80	3
CAL_HOUSING-MED	100	21643	8
CAL_HOUSING-LARGE	1000	3317209	16
FASHION_MNIST-SMALL	100	800	3
FASHION_MNIST-MED	1000	144154	8
FASHION_MNIST-LARGE	10000	2929521	16
ADULT-SMALL	10	80	3
ADULT-MED	100	13074	8
ADULT-LARGE	1000	642035	16

4.1 Bin packing

We first evaluate the performance of the NF, FFD and BFD bin packing algorithms from Section 3.3. We also include "none" as a baseline, where no packing occurs and each unique path is allocated to a single warp. All bin packing heuristics are single threaded and run on the CPU. We report the execution time (in seconds), utilisation, and number of bins used (K). Utilisation is defined as $\frac{\sum_{i \in I} s(i)}{32K}$, the total weight of all items divided by the bin space allocated, or for our purposes, the fraction of GPU threads that are active for a given bin packing. Poor bin packings waste space on each warp and underutilise the GPU.

Results are summarised in Table 5. "None" is clearly a poor choice, with utilisation between 0.1 and 0.3, with worse utilisation for smaller tree depths (small models with max depth 3 allocate items of size 3 to warps of size 32). The simple NF algorithm provides decent results with fast runtimes, but it can lag behind FFD and BFD when item sizes are larger, having utilisation as low as 0.79 for *fashion_mnist-large*. FFD and BFD achieve better utilisation than NF in all cases, as reflected in their superior approximation guarantees. Interestingly, FFD and BFD achieve the same efficiency on every example tested. We have verified they can produce different packings on contrived examples, but there is no discernible difference for our application. FFD and BFD have longer runtimes than NF, reflecting their $O(n \log n)$ time complexity. FFD is slightly faster than BFD because it uses

Table 4. Hardware - Nvidia DGX-1

PROCESSOR	DETAILS
CPU	2X 20-CORE XEON E5-2698 V4 2.2 GHZ
GPU	8X TESLA V100-32

Table 5. Bin packing performance

Model	Alg	Time(s)	Utilisation	Bins
covtype-small	none	0.0018	0.105246	560
covtype-small	nf	0.0041	0.982292	60
covtype-small	ffd	0.0064	0.998941	59
covtype-small	bfd	0.0086	0.998941	59
covtype-med	none	0.0450	0.211187	113533
covtype-med	nf	0.0007	0.913539	26246
covtype-med	ffd	0.0104	0.940338	25498
covtype-med	bfd	0.0212	0.940338	25498
covtype-large	none	0.0346	0.299913	6702132
covtype-large	nf	0.0413	0.851639	2360223
covtype-large	ffd	0.8105	0.952711	2109830
covtype-large	bfd	1.6702	0.952711	2109830
cal_housing-small	none	0.0015	0.085938	80
cal_housing-small	nf	0.0025	0.982143	7
cal_housing-small	ffd	0.0103	0.982143	7
cal_housing-small	bfd	0.0001	0.982143	7
cal_housing-med	none	0.0246	0.181457	21641
cal_housing-med	nf	0.0126	0.931429	4216
cal_housing-med	ffd	0.0016	0.941704	4170
cal_housing-med	bfd	0.0031	0.941704	4170
cal_housing-large	none	0.0089	0.237979	3370373
cal_housing-large	nf	0.0225	0.901060	890148
cal_housing-large	ffd	0.3534	0.933114	859570
cal_housing-large	bfd	0.8760	0.933114	859570
fashion_mnist-small	none	0.0022	0.123906	800
fashion_mnist-small	nf	0.0082	0.991250	100
fashion_mnist-small	ffd	0.0116	0.991250	100
fashion_mnist-small	bfd	0.0139	0.991250	100
fashion_mnist-med	none	0.0439	0.264387	144211
fashion_mnist-med	nf	0.0008	0.867580	43947
fashion_mnist-med	ffd	0.0130	0.880279	43313
fashion_mnist-med	bfd	0.0219	0.880279	43313
fashion_mnist-large	none	0.0140	0.385001	2929303
fashion_mnist-large	nf	0.0132	0.791948	1424063
fashion_mnist-large	ffd	0.3633	0.958855	1176178
fashion_mnist-large	bfd	0.8518	0.958855	1176178
adult-small	none	0.0016	0.125000	80
adult-small	nf	0.0023	1.000000	10
adult-small	ffd	0.0061	1.000000	10
adult-small	bfd	0.0060	1.000000	10
adult-med	none	0.0050	0.229014	13067
adult-med	nf	0.0066	0.913192	3277
adult-med	ffd	0.0575	0.950010	3150
adult-med	bfd	0.1169	0.950010	3150
adult-large	none	0.0033	0.297131	642883
adult-large	nf	0.0035	0.858728	222446
adult-large	ffd	0.0684	0.954377	200152
adult-large	bfd	0.0954	0.954377	200152

a binary tree packed into an array, yielding greater cache efficiency, but its implementation is more complicated. In contrast, BFD is implemented easily using *std::set*.

Based on these results, we recommend BFD for its strong approximation guarantee, simple implementation, and acceptable runtime when packing jobs into batches for GPU execution. Its runtime is at most 1.6s for our largest model (*covtype-large*) with 6.7M items, and is constant with respect to the number of test rows (the bin packing occurs once per ensemble and is reused for each additional data point), allowing us to amortise its cost over improvements in end-to-end runtime from improved kernel efficiency. The gains in GPU thread utilisation from using BFD over NF directly translate into performance improvements, as fewer bins used means fewer GPU warps are launched. On our large size models, we see improvements in utilisation of 10.1%, 3.2%, 16.7% and 9.6% from BFD over NF. We use BFD in all subsequent experiments.

4.2 SHAP Values

We evaluate the performance of GPUTreeShap as a backend to the XGBoost library (Chen & Guestrin, 2016), comparing its execution time against the existing CPU implementation of Algorithm 1[2]. The baseline CPU algorithm is efficiently multithreaded using OpenMP, with a parallel for loop over all test instances. See https://github.com/dmlc/xgboost for exact implementation details for the baseline and https://github.com/rapidsai/gputreeshap for GPUTreeShap implementation details.

[2]We do not benchmark against TreeShap implementations in the python SHAP package or LightGBM because they are written by the same author, also in C++, and are functionally equivalent to XGBoost's implementation.

Table 6. V100 vs. 40 CPU cores - 10000 test rows

Model	CPU(s)	std	GPU(s)	std	Speedup
covtype-small	0.04	0.02	0.02	0.01	2.27
covtype-med	8.25	0.07	0.45	0.03	18.23
covtype-large	930.22	0.56	50.88	0.21	18.28
cal_housing-small	0.01	0.01	0.01	0.01	0.96
cal_housing-med	1.27	0.02	0.09	0.02	14.59
cal_housing-large	315.21	0.30	16.91	0.34	18.64
fashion_mnist-small	0.35	0.14	0.17	0.04	2.09
fashion_mnist-med	15.10	0.07	1.13	0.08	13.36
fashion_mnist-large	621.14	0.14	47.53	0.17	13.07
adult-small	0.01	0.00	0.01	0.01	1.08
adult-med	1.14	0.00	0.08	0.01	14.59
adult-large	88.12	0.20	4.67	0.00	18.87

Table 6 reports the runtime of GPUTreeShap on a single V100 GPU as compared to TreeShap on 40 CPU cores. Results are averaged over 5 runs and standard deviations are also shown. We observe speedups between 13-18x for medium and large models evaluated on 10,000 test rows. We observe little to no speedup for the small models as insufficient computation is performed to offset the latency of launching GPU kernels.

Figure 4 plots the time to evaluate varying numbers of test rows for the *cal_housing-med* model. We plot the average of 5 runs; the shaded area indicates the 95% confidence interval. This illustrates the throughput vs. latency trade-off for this particular model size. The CPU is more effective for < 180 test rows due to lower latency, but the throughput of the GPU is significantly higher at larger batch sizes.

SHAP value computation is embarrassingly parallel over dataset rows, so we expect to see linear scaling of per-

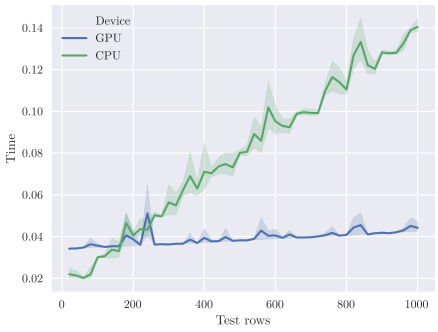

Figure 4. V100 vs. 40 CPU cores - *cal_housing-med*

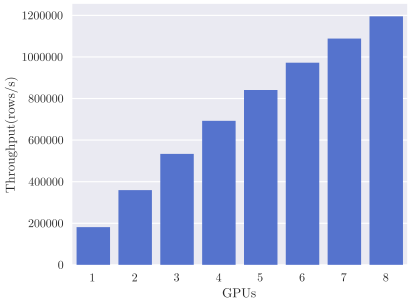

Figure 5. 8x V100 - *cal_housing-med*

Figure 6. 40 CPU cores - *cal_housing-med*

formance with respect to the number of GPUs or CPUs, given sufficient data. We set the number of rows to 1 million and evaluate the effect of additional processors for the *cal_housing-med* model, measuring throughput in rows per second. Figure 5 reports throughput up to the 8 GPUs available on the DGX-1 system, showing the expected linear scaling and reaching a maximum throughput of 1.2M rows per second. Reported throughputs are from the average of five runs—error bars are too small to see due to relatively low variance. Figure 6 shows scaling with respect to CPU cores, also scaling linearly, up to a maximum throughput of 7000 rows per second. The shaded area indicates the 95% confidence interval from 5 runs. We speculate that the dip at 40 cores is due to contention with the operating system requiring threads for other system functions, and so ignore it for this scaling analysis. We can reasonably approximate from Figure 6, using a throughput of 7000 rows/s per 40 cores, that it would require 6850 Xeon E5-2698 v4 CPU cores, or 343 sockets, to achieve the same throughput as 8 V100 GPUs for this particular model.

4.3 SHAP Interaction Values

Table 7 compares single GPU vs. 40 core CPU runtime for SHAP interaction values. For this experiment, we lower the number of test rows to 200 due to the significantly increased computation time over standard SHAP values. Computing interaction values is challenging for datasets with larger numbers of features, in particular for *fashion_mnist* (785 features). Our GPU implementation achieves moderate speedups on *cal_housing* and *adult* due to the lower number of features; these speedups are roughly comparable to those obtained for standard SHAP values (Table 6). In contrast, for *covtype-large* and *fashion_mnist-large*, we see speedups of 114x and 340x, in the most extreme case reducing runtime from six hours to one minute. This speedup comes from both the increased throughput of the GPU over the CPU and the improvements to algorithmic complexity due to omission of irrelevant features described in Section 3.5. Note that it may be possible to reformulate the CPU algorithm to take advantage of the improved complexity with similar preprocessing steps, but investigating this is beyond the scope of this paper.

Table 7. Feature interactions - 200 test rows

MODEL	CPU(S)	STD	GPU(S)	STD	SPEEDUP
COVTYPE-SMALL	0.14	0.01	0.02	0.01	8.32
COVTYPE-MED	21.50	0.32	0.19	0.02	114.41
COVTYPE-LARGE	2055.78	4.19	28.85	0.06	71.26
CAL_HOUSING-SMALL	0.01	0.00	0.01	0.00	1.44
CAL_HOUSING-MED	0.53	0.04	0.04	0.01	12.05
CAL_HOUSING-LARGE	93.67	0.28	8.55	0.04	10.96
FASHION_MNIST-SMALL	11.35	0.87	4.04	0.67	2.81
FASHION_MNIST-MED	578.90	1.23	4.91	0.71	117.97
FASHION_MNIST-LARGE	21603.53	622.60	63.53	0.78	340.07
ADULT-SMALL	0.06	0.09	0.01	0.00	11.25
ADULT-MED	1.74	0.30	0.04	0.01	39.38
ADULT-LARGE	67.29	6.22	2.76	0.00	24.38

5 Conclusion

SHAP values have proven to be a useful tool for interpreting the predictions of decision tree ensembles. We have presented GPUTreeShap, an algorithm obtained by modifying the TreeShap algorithm to enable efficient computation on GPUs. Our modifications exploit warp-level parallelism by cooperatively evaluating quadratic programming problems for each path in a decision tree ensemble, thus providing massive parallelism for large ensemble predictors. We have shown how standard bin packing heuristics can be used to effectively schedule problems at the warp level, maximising GPU utilisation. Additionally, our rearrangement leads to improvement in the algorithmic complexity when computing SHAP interaction values, from $O(TLD^2M)$ to $O(TLD^3)$. Our library GPUTreeShap provides significant improvement to SHAP value computation over currently available software, allowing scaling onto one or more GPUs and reducing runtime by 1 to 2 orders of magnitude.

REFERENCES

Anderson, R. and Mayr, E. Parallelism and greedy algorithms. Technical Report 1003, Computer Science Department, Stanford University, 1984.

Anderson, R. J., Mayr, E. W., and Warmuth, M. K. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, 1989.

Blackard, J. A. *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD thesis, Colorado State University, 1998.

Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *KDD*, pp. 785–794. ACM, 2016.

Coffman, E. G., Garey, M. R., and Johnson, D. S. Approximation algorithms for bin packing: A survey. In Hochbaum, D. S. (ed.), *Approximation Algorithms for NP-Hard Problems*, pp. 46–93. PWS Publishing, 1997.

Fujimoto, K., Kojadinovic, I., and Marichal, J.-L. Axiomatic characterizations of probabilistic and cardinal-probabilistic interaction indices. *Games and Economic Behavior*, 55:72–99, 2006.

Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman &; Co., 1979.

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5), 2018.

Harris, M. and Buck, I. GPU flow-control idioms. In Pharr, M. (ed.), *GPU Gems 2*, pp. 547–555. Addison-Wesley, 2005.

Johnson, D. S. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, 1974.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. LightGBM: A highly efficient gradient boosting decision tree. In *NIPS*, pp. 3149–3157. Curran Associates, 2017.

Kohavi, R. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, pp. 202–207. AAAI Press, 1996.

Lundberg, S. M. and Lee, S.-I. A unified approach to interpreting model predictions. In *NIPS*, pp. 4765–4774. Curran Associates, 2017.

Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., and Lee, S.-I. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

Martello, S. and Toth, P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

NVIDIA Corporation. CUDA C++ programming guide, 2020. Version 11.1.

Pace, R. K. and Barry, R. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

Ribeiro, M. T., Singh, S., and Guestrin, C. "Why Should I Trust You?": Explaining the predictions of any classifier. In *KDD*, pp. 1135–1144. ACM, 2016.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *ICCV*, pp. 618–626. IEEE, 2017.

Shapley, L. S. A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317, 1953.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv*, abs/1708.07747, 2017.