Extremely Randomized Trees on GPUs

Jason Phang
Email: jasonphang@nyu.edu

Kenil Tanna
Email: kenil@nyu.edu

I. Introduction

Random Forests[1] are often the textbook example of highly parallel algorithms that are unsuitable for GPUs.

In this report, we propose and evaluate our implementation of Extremely Randomized Trees[2] (or *ExtraTrees*), a variant of Random Forests which are more amenable to GPU-driven parallel programming.

II. Literature Review

Most past work on parallelizing forests on GPUs has been done on Random Forests, the more popular algorithm, although there is some prior work on parallelizing ExtraTrees.

Liao et al.[3], Hakan et al.[4] and Hannes et al. [5] all implement the Random Forest algorithm on GPUs, with varying levels of success. Liao et al in particular stopped supporting updates to their implementation after they found that the scikit-learn implementation on CPU beat their GPU. Furthermore, their algorithm involves conditionally switching to CPU for small data sizes.

The primary dedicated implementation of ExtraTrees for the GPU was the gpuERT[6]. Jansson et al. follow a node wise approach, i.e each block operates on a single node rather than a tree and they process all the nodes on the same level together, and then move on to the next level. So this is a breadth-first approach rather than a depth-first approach which is more suitable for GPUs. Upon closer examination of the algorithm, we determined that the implementation does not follow the original algorithm strictly, allowing the implementation to obtain some performance gains. For example, rather than choosing a random point between the minimum and maximum of a sampled feature at that node, they sample 10 points and take the average value as the splitting point. Hence as the sample points are actually pre-defined and not conditional on the node, their algorithm is not inherently random as it should have been. This saves a considerable amount of time, but comes at a cost of not reproducing the same accurate results as if the ExtraTrees were constructed on CPUs. The second major drawback of this implementation is that it only supports a binary classifier, which saves computation time/space compared to computing counts for a data-dependent K classes. Finally, the distributed code only runs and compiles on Windows. Although we were able to obtain a GPU machine running on Windows, we were unable to run the distributed binary. We were similarly unable to compile the binary from source due to the Visual C++ 2010-specific environment and various library dependencies.

It should be noted that their implementation and source code support both Random Forests and ExtraTrees - we hope to be able to obtain new performance gains by focusing solely on ExtaTrees, possibly bypassing some unnecessary computation required to support both models.

Geary et al.[7] also implemented a dedicated ExtraTrees implementation, although we were unable to obtain a copy of their paper or code. The literature review by Jansson et al. cites their performance as being worse than CPU implementations.

III. Algorithm

The *ExtraTrees* algorithm is extremely similar to the Random Forest algorithm, and intuitively simple. We grow R independent trees. Starting with all data at the root note, we randomly drawing a $\tilde{F}$ number of non-constant features f_j, and drawing a random cut-value c_j for each of the chosen features. We slice the data $(X[:, f_j] < c_j)$ on each of the above cuts, and evaluate the cut that best separates the Ys based on some criteria (e.g. Gini impurity). This becomes chosen slice for slicing the data into left and right nodes (X_L, Y_L) and (X_R, Y_R). We then repeat the procedure on the left and right data to grow the left and right nodes of the tree. This is repeated recursively until every node reaches a termination criteria, often even the maximum depth, minimum number of elements remaining, or when Y is constant within a node. See Algorithm 1 for details.

The Random Forest algorithm is similar, except that each tree is grown in a bootstrap manner (sample the original data with replacement), and most importantly, instead of choosing random cuts for each feature, it chooses the optimal cut-value for each chosen feature. This is often the most compute-intensive portion of implementations of Random Forest, as it often involves sorting the data based

```
Data:
Data set: $\mathcal{X} \in \mathbb{R}^{N \times F}$, $\mathcal{Y} \in \mathbb{R}^1$
Parameters: number of trees $R$, number of features per node $\tilde{F}$ (default=$\sqrt{F}$), termination condition $\mathcal{C}$ , score function $S$
Result:
$R$ decision trees, each of which $T_r : \mathbb{R}^{N \times F} \rightarrow \mathbb{R}^1$
begin
    for $r = 1$ to $R$ do
        Initialize Tree $T$, with node $n_0$
        GrowTree($T, X, Y, n_0$)
    end
end

Function GrowTree($T, X, Y, n$):
    if termination condition $\mathcal{C}(T, X, Y)$ is met then
        Register mode of $Y$ to node $n$
        return
    end
    $f$, $c$ = SplitNode($X, Y$)
    Register $f$, $c$ to node $n$
    Split $(X, Y)$ into $(X_L, Y_L)$, $(X_R, Y_L)$ based on feature $X[:, f] < c$
    From node $n$, initialize left and right nodes $n_L, n_R$
    GrowTree($T, X_L, Y_L, n_L$)
    GrowTree($T, X_R, Y_R, n_L$) return $T$
Function SplitNode($X$, $Y$):
    for $j = 1$ to $\tilde{F}$ do
        Choose a random non-constant column $j$ of $X$
        Choose a random cut-value $c$ uniformly between the min and max above the chosen column $j$
        Compute the impurity score $s = S(j, c, X, Y)$
        store tuple $(s, j, c)$
    end
    Select the best tuple $(s', j', c')$ based on highest $s$ return Best split feature $j'$, best cut $c'$
```

Algorithm 1: ExtraTrees

on the feature. Given that ExtraTrees omits this step, this may pave the way for a GPU-efficient implementation of ExtraTrees.

However, even without the best-cut step, the ExtraTrees algorithm remains difficult to implement in a highly parallel manner. Importantly, as we grow the tree, the number of data points considered in each node grows smaller. In fact, because the data points are distributed across different nodes at each level (with no overlap), CPU implementations often simply sort the data points by nodes at each level, so growing the tree at each node need only consider a smaller and very local number of values. This form of sorting is tree-dependent, so CPU implementations will make copies of data for each tree it grows in parallel. For example, the *scikit-learn* implementation grows one tree per processor at a time by default, hence copying the data a number of times equal to the number of processors.

Our goal is to derive an algorithm and implementation that is able to grow a large number of trees in parallel without having to make multiple copies of the data. This is particularly of interest as data-sets grow larger and it becomes infeasible to flippantly make copies of the whole data set.

IV. GPU Algorithm

To begin discussing our implementation, we make several reasonable assumptions:

1) The number of trees R grown is less than the maximum number of threads per block/maximum thread-dimension per block. This is generally true - a modern GPU support up to 1024 threads per block, whereas the number of trees grown is often on the order of 50-200. Even if this assumption is violated, we can simply grow the tree we can fit in GPU sequentially.
2) The data fits on GPU. This assumption may change over time, and solving this issue will require efficient ingress and egress of data via streams. For now, we are choosing to omit consideration of this problem. Again, in comparison, most CPU and even GPU-based implementations simply store copies of data for each tree grown, so this assumption is already far more stringent.

It is instructive to consider the following dimensions for a sample experiment. MNIST, a data set of monochrome hand-written digits, has a training set of N=60,000 observations of $F = 768$ features (pixels) each, with 10 C classes. We will train 256 trees on MNIST.

We also restrict our implementation of ExtraTrees to support the following configuration, as a proof-of-concept:

1) This is a ExtraTree classifier, so we are evaluating classes and scoring splits by Gini impurity.
2) The features/cuts considered per node is $\tilde{F} = \sqrt{F}$.

We start by illustrating the major data structures that we will consider, which will support the discussion of our algorithm.

Our `trees` data structure is stored as a $6 \times NM \times R$ array. 6 is the number of fields per node that we will store (meta-data for each node, elaborated on shortly), M the number of nodes and R the number of trees. N is grown dynamically as required while growing the tree - in particular, it was fit the largest tree being grown in parallel. We note that it would be infeasible to preallocate the **tree** data structure naively ahead of time with a 2^d array where d is the max depth. For example, a typical tree grown on MNIST has a depth of 40-50, and 2^40 integers worth of space is infeasible to pre-allocate.

The six fields we will store are:

1) Left child node-id
2) Right child node-id
3) Node depth
4) Best feature
5) Best cut
6) Predicted class

Note that some fields (best feature/cut) are only relevant to branch nodes, and others (predicted value) only relevant to leaf nodes. Crucially, while growing the tree, as well as as terminal nodes, we enforce the following fact: a node's left and right node points to itself. This eases some issues of tree traversal in parallel and reduces branch-divergence somewhat.

As we grow the tree, we walk down the M dimension of the tree. By following a procedure whereby we register new child-nodes at the right-end of the tree at each step, we obtain a useful property: the nodes will be ordered in monotonically increasing depth. This will prove useful when we considering growing nodes in parallel. The currently analyzed/grown "node" in each tree is recorded in a variable `tree_pos`, which is common across all trees. We store the "end" of the tree by an array `tree_length`. Note that tree lengths will diverge only when trees terminate early (all leaf nodes have been completed), although trees can grow to different depths at a time.

The other important structure we maintain is `batch_pos`. This is a $R \times N$ array. During training, this maintains the current node-id that each data point is at the current (or previous) depth. Using this in conjunction with the above `tree` structure, it is possible to efficiently traverse many trees for many data points in parallel (at the cost of global memory accesses). To minimize the amount of address hopping, we use the fact that we will be growing the trees in increasing depth, so we only need to traverse down the trees one depth at a time, and only when we grow a new depth. During inference, we reuse the `batch_pos` data structure for whole-tree traversal, although in general inference is far less costly than training.

A. Kernels and Operations

We detail here the key kernels required in implementing the ExtraTrees algorithm

1) `batch_advance`: This kernel is responsible for advancing the node-positions of each data points for each tree in the `batch_pos` array. Because we grow the tree node by node, we only need to advance the positions by at most 1 level deep at each call. The `batch_pos` are used subsequently in filtering out irrelevant data points.

2) `check_node_termination`: This method checks if a node should be considered terminal for a given tree (i.e. turned into a leaf). Our terminal condition $\mathcal{C}$ is when a node has only data points belonging to a single class - the prediction is then that class. Unfortunately, this has to be performed by scanning across the ys for the data points at the currently `tree_pos` and confirming if they are all the same value. If so, the `trees` structure's prediction for this note is is updated, and this node is marked as not a branch node for this `treepos`. Being marked as not a branch node will cause the following kernels to short-circuit and return early for this `tree_pos`.

3) `collect_min_max`: In order to select candidates for our random features, we first need to obtain the minimums and maximums for each feature. This is useful both for identifying non-constant features, as well as having the minimum and maximum value for each feature to draw our cut from. Note that the minimums and maximums are for each feature in each tree.

This is the slowest portion of the algorithm currently based on profiling, as it requires many global reads from the X and `batch_pos` arrays. Unfortunately, there seem to be few ways to bypass this issue, especially if we are growing the tree node-wise (remember: we are unable to store information level-wise as the nodes per level could be prohibitively large). Furthermore, the kernel needs to scan each data point (at least for its position), and even such a simple "masking" operation takes up significant time due to the number of global reads. In a later section we identify potential areas for improvement.

It is important here to note that iterative algorithms here have the benefit here of storing the non-constant features as the grow their trees depth-wise, and are able to sort their data points. We are unable to sort our data points for trees grown in parallel, and storing the non-constant features would be infeasible for growing trees breadth-wise as we would have to store them for every node in the level, hence we have to recompute the minimums and maximums at each point. (Note that even iterative algorithms have to recompute the minimum and maximums - they only have the advantage of skipping already-known constant features.)

4) `count_num_valid_feat`*:* After collecting the relevant minimums and maximums for each feature per tree for the current node, we have to randomly sample $\tilde{F}$ feature out of the non-constant (or "valid") features. We do this by, for each tree, walking through our features and noting the ones where the minimum differs from the maximum. If so, we record this, in a separate array, as "valid". We also add to a count of the number of valid features for that tree.

5) `populate_random_feats_cuts`*:* Now we need to sample $\tilde{F}$ features randomly from our valid features. We draw randomly from $\text{Unif}(0, \text{num\_valid})$, and select from our array of valid features, which have indices pointing to the real feature IDs. We similarly pick out the minimums and maximums and sample $\text{Unif}(\text{minimum}, \text{maximum})$. (There is an alternative way to sample these features randomly that reduces address hopping, but we found that it was slower in practice due to larger divergence. Details can be found in the Appendix - although it was not used, it was a meaningful exercise in considering and working around some limitations in CUDA.)

6) `populate_class_counts`*:* Next we populate the number of data points in belonging to each class in the current node based on our random cuts. These results are written to two $R \times K \times \sqrt{F}$ arrays, representing the potential left and right branches for each randomly split per tree. This is done by again scanning through the relevant data points (filtering through `batch_pos`) and checking the relevant X and Ys to determine which left/right and class the value should be added to.

This is the second slowest kernel in our algorithm, again due to the large number of reads and writes. Because of the large scale of the kernel, even implementing the sum of counts via a reduce-sum method is infeasible, due to the large amount of shared memory that would be required to facilitate the sum of counts for each tree/random feature.

7) `place_best_feat_cuts`*:* After collecting the counts of classes in the prior step, we can now compute the Gini impurity for each potential slice via:

$$I_G = \sum_{i=1}^{K} p_i \sum_{k \neq i} p_k = 1 - \sum_{i=1}^{K} p_i^2 = 1 - \sum_{i=1}^{K} \frac{n_{b,i}}{N_b}$$

where $n_{b,i}$ is the number of elements in class i in branch b, and N_b is the total number of counts in branch b. This is computed for both branches, and we want to pick the split which minimizes impurity.

To shorten our computation, this can actually be computed by a metric that proxies for the improvement in impurity

$$\dot{I}_G(p) = -N_L \left(1 - \sum_{i=1}^{K} \frac{{n_{L,i}}^2}{N_L} \right) - N_R \left(1 - \sum_{i=1}^{K} \frac{{n_{R,i}}^2}{N_R} \right)$$

We then choose the best split based on the largest proxy improvement.

8) `update_trees`*:* Finally, with the best feature and cut in hand, we can update the tree. First, we register the best feature and cut to the current node in `trees`. We also extend the length (i.e. maximum node ID) of the tree by two - this is used for determining if we need to extend the `trees` array.

Next, we also update the left and right nodes, placing sensible placeholder values where necessary. We update the depth as well as setting both children of each node to themselves - preserving our idempotent property.

9) Repeat: After this, we repeat the process, starting by batch-advancing the trees. This will cause positions as the recently grown node to advance down to their next branch, and we continue to grow the tree until no nodes are being advanced at all. That concludes the growing of the trees.

10) `batch_traversal` *and Evaluation:* To use our model for evaluation / prediction, we create a fresh `batch_pos` array initialized to 0, and proceed to traverse our way down the tree, similar to `batch_advance`, except we continue until positions do not change (the idempotent property again). After this, we simply collecting the predictions from the corresponding nodes of the tree, and pick out the mode across trees for each data point as the predicted value for each data point. The picking of the mode is best done on CPU, so we transfer the predictions to host after the traversal.

V. Experiments

All experiments were performed with the following high-end hardware and software configuration:

1) CPU: Intel Core i7-6700 8M Skylake Quad-Core 3.4 GHz
2) Memory: 32GB DDR4 SDRAM
3) GPU: NVIDIA GTX 1080 (8GB)
4) OS: Ubuntu 16.04 LT

We used 2 standard machine learning data sets in our experiments (Table I). Iris is a small data set based on classifying flower types based on 4 features, while MNIST is a data-set of hand-written digits. We report the statistics for the number of observations for the training set alone, which is used to build the trees. Note that generating predictions after learning the trees is orders of magnitude faster than training.

Name	Rows	Features	Classes
Iris	100	4	3
MNIST	60000	784	10

TABLE I: Data Sets

Our main comparison is against the Python scikit-learn implementation of ExtraTrees, which is written in Cython (bridge language between C and Python, benefiting greatly from C-like speeds) and taking advantage of multi-core processing. The bulk of the algorithm should be run with C-like performance.

As mentioned above, the existing GPU implementations had either been surpassed by scikit-learn, or we were unable to reproduce their binaries to successfully run their algorithm. In addition, we adhered strictly to the algorithm specified by the ExtraTrees paper, whereas some GPU implementations took liberties, albeit sensible practically-driven ones, in changing the algorithm to adapt it better to GPU parallel computing.

All of our tests involve running only the training portion of the algorithm. For the Python implementation, we start timing from an already running session of Python with the data already loaded, to avoid any issues of Python's overhead. Our implementation includes the loading of the respective Iris and MNIST data, which incurs a small overhead, though as we see the overhead is negligible when taken into consideration. We run the algorithms for trees of powers of 2.

As a side note, we verified that our results are exactly within range of normal runs of standard implementations of ExtraTrees, achieving 98% test performance for Iris for most numbers of trees, and >95% performance for MNIST for 10 trees.

We observe that we perform admirably for the Iris data set (Fig. 1). While the scikit-learn algorithm scales linearly in the number of trees (demonstrating that the Python overhead is not dominating the running times), our implementation is largely flat in running time even up to 1024 trees. This likely demonstrates that our implementation is coming in significant under capacity for the GPU, partially due to our relatively economical use of data structures.

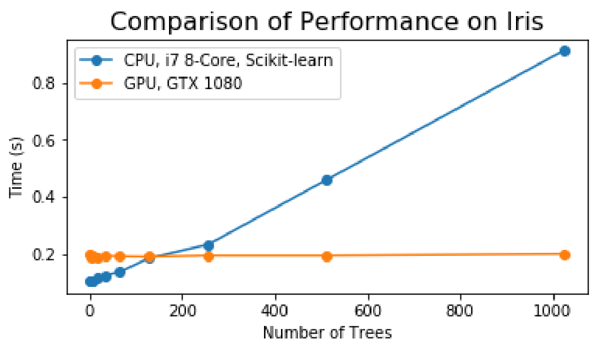

Fig. 1: Iris Running Times Comparison

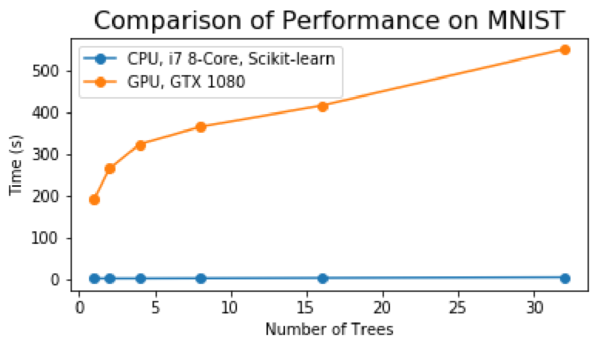

Fig. 2: MNIST Running Times Comparison

However, we perform much, much worse for MNIST - in fact seeing almost the opposite behavior (Fig. 2). The scikit-learn version continues to grow linearly in times, but is extremely fast, whereas our implementation struggles as we handle both the larger data set as well as larger number of trees, running against various constraints. Our investigation show that the primary hit to performance is the large number of global address reads involved in the algorithm, growing both in the size of the data as well as in the depth of the tree (which grows with the size of the data). The CPU implementation benefits from the heavy locality of sorting duplicates of the data, particularly as data sizes grow. We do, however, note that even on MNIST, our implementation experiences sub-linear growth in time - double the number of trees leads to less than double the running time.

While this is a setback, we aim to continue to build on the algorithm to scale better to larger problem sizes.

VI. Discussion

A. Performance

We detail in Table II the breakdown of the running-time of kernels in our implementation, from a 10-tree run on MNIST. This is the breakdown from `nvprof` of the kernel running times

As we can see the majority of the running time is in `kernel_check_node_termination`, `collect_min_max` and `populate_class_counts`. All three involve heavy reads from the largest data structures (`batch_pos` and X/Y), as well as fairly significant writes.

We attempted several approaches to optimizing all three kernels, but net few results. `populate_class_counts` in particular requires random reads of features across all data points, making it extremely difficult to find areas for improvement. `kernel_check_node_termination` and `collect_min_max` are somewhat more regular in terms of its data accesses as it deterministically goes through the entire X / Y but we found that even approaches like reduce-min/max or rearranging our order of reads to promote memory coalescing failed to net significant improvements.

Beyond the scope of this project, a strategy that we did not have the time to implement would be to process several nodes at once (currently we are growing 1 node for all trees). This would complicate the algorithm somewhat - for example, we would need to ensure that, if we were processing M nodes at once, we would only process nodes in the same level for that tree. This requires more housekeeping in terms of trees growth. Likewise, we would need to increase the memory allocated to the relevant data structures linearly with the number of nodes simultaneously processed.

The advantage of processing multiple nodes at once is clearest for `collect_min_max`, where in one-pass of the entire data we could collect the minimums and maximums for multiple nodes. For `populate_class_counts`, we are still constrained by the random feature reads, but processing multiple nodes at once would shift more of the time from global reads to managing the internal storage of counts for multiple nodes.

Our above analysis notwithstanding, we believe there are still many opportunities for further improving our implementation. A large portion of our efforts were spent on correctly replicating the algorithm in a sensibly parallel way, with thought towards the best representations

B. Memory

Because we generate our trees node-wise and do not make any copies of the data (barring a single transposed copy of X for more efficient memory accesses), our memory footprint is relatively small.

For example, MNIST consumes an array of $N \times F = 60000 \times 768$ Ints for storing just the X, approximately 300mb. The main structures that consume space in our implementation are `batch_pos` and `trees`. `batch_pos` consumes $N \times T$, and given our maximum T of 1024, it is at most 1.5x MNIST. `trees` consumes $T \times 6 \times M$ were M is the number of nodes, which empirically on MNIST is between 15,000 to 17,000. It is worth noting that `trees` is grown dynamically over time as the number of nodes increase. In the worst case, $M \leq 2N$, since each branch must separate at least 1 data point, so `trees` also consumes at most 12x MNIST. Hence, we can bound our total memory consumption from these data structures at at most 14.5x MNIST in the very worst case. Note that this assumes that we are building 1024 decision trees simultaneously while getting the very worst features and splits at every juncture, and we still only consume a little more than 10x the original memory. Empirically, we experience far less than that, and the algorithm runs comfortably on a GTX 1080.

VII. Conclusion

We completed a GPU-based parallel implementation of the ExtraTrees algorithm that is competitive with CPU-based libraries. While it has certain kinks and significant room for improvement in terms of large-data scalability, we believe that it is a foundational proof-of-concept of an approach toward parallelizing ExtraTrees, one of a family of algorithms normally deemed to be poor candidates for GPU-based parallelization. We look forward to working further in improving and optimizing this implementation.

References

[1] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[2] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10994-006-6226-1

[3] Y. Liao, A. Rubinsteyn, R. Power, and J. Li, "Learning random forests on the gpu," 2013. [Online]. Available: http://news.cs.nyu.edu/~jinyang/pub/biglearning13.pdf

[4] H. Grahn, N. Lavesson, M. H. Lapajne, and D. Slat, "Cudarf: A cuda-based implementation of random forests," 2013. [Online]. Available: https://www.computer.org/csdl/proceedings/aiccsa/2011/0475/00/06126612-abs.html

Time (%)	Time	Calls	Operation
44.92%	257.429 s	25775	kernel_check_node_termination
42.48%	243.420 s	25775	kernel_populate_class_counts
8.45%	48.4484 s	25775	kernel_collect_min_max
1.23%	7.04770 s	25775	kernel_populate_valid_feat_idx
1.10%	6.27610 s	25775	kernel_advance_trees
1.05%	6.03598 s	25775	kernel_populate_feat_cut
0.69%	3.95542 s	25775	kernel_place_best_feat_cuts
0.02%	132.87 ms	25775	kernel_update_trees
0.01%	69.734 ms	25775	kernel_collect_num_valid_feat
0.01%	60.097 ms	77327	[CUDA_memcpy_DtoH]
0.01%	48.073 ms	5	[CUDA_memcpy_HtoD]
0.01%	42.689 ms	25775	get_max_tree_length
0.01%	31.750 ms	25776	kernel_refresh_tree_is_done
0.01%	31.363 ms	25775	kernel_depopulate_valid_feat_idx
0.00%	662.56 us	1	kernel_traverse_trees
0.00%	163.01 us	120	[CUDA_memcpy_DtoD]
0.00%	94.176 us	1	init_random
0.00%	35.328 us	1	kernel_raw_predict
0.00%	10.657 us	2	kernel_initialize_batch_pos
0.00%	1.5360 us	1	kernel_initialize_trees

TABLE II: Breakdown of Operations in 10-Tree MNIST runs

[5] H. Schulz, B. Waldvogel, R. Sheikh, and S. Behnke, "Curfil: Random forests for image labeling on gpu." in *VISAPP (2)*, J. Braz, S. Battiato, and F. H. Imai, Eds. SciTePress, 2015, pp. 156–164. [Online]. Available: http://dblp.uni-trier.de/db/conf/visapp/visapp2015-2.html#SchulzWSB15

[6] K. Jansson, H. Sundell, and H. Boström, "gpurf and gpuert: Efficient and scalable GPU algorithms for decision tree ensembles," in *IPDPS Workshops*. IEEE Computer Society, 2014, pp. 1612–1621.

[7] M. Geary, H. Lee, D. Signorelli, and J. Vaughan, "Final project implementing extremely randomized trees in cuda."

Appendix

A. *Running the code*

In our submission zip file, we include the source code as well as data files. **If there are any issues getting the code to run or obtaining the data files, please let us know**.

The code can be compiled with:

```
nvcc extraTrees.cu -o extraTrees
```

The code can be run with

```
./extraTrees [dataset] [num_trees]
[seed]
```

where `dataset=0` for MNIST and `dataset=1` for MNIST, `num_trees` is an integer between 1 and 1024, and seed is the random seed.

Here is an example running MNIST with 10 trees:

```
./extraTrees 0 10 2
```

Here is an example running Iris with 128 trees:

```
./extraTrees 1 128 2
```

B. *Alternative Strategy for Sampling Random Features*

We considered an alternative strategy for sampling random features. This strategy was driven by our desire to minimize the number of address hops. We hoped to take advantage of the fact that features could potentially be sampled multiple times, as the sampling was done in replacement. This is particularly the case as we reach the end of the tree and the number of valid features becomes small, possibly less than $\tilde{F}$. At the same time, there could be memory fetching benefits from accessing the features in order rather than via random hops.

Here, we assume that we are sampling from f valid features, with a companion array of size f each containing the addresses of valid features. Hence, our goal was to derive an algorithm that samples uniformly (with replacement) n elements from f items, and returns the elements *in order*. We also needed to do this in $O(n)$ time, without any arrays to store intermediate counts (otherwise we could simple sample and add to counts, and then iterate in order), nor the ability to sort.

As it turns out, this can be achieved with the following algorithm. The key thing to note is that ultimately we simply wanted to sample how many elements of each item (feature) to take, so this can be solved in $O(f)$ time given the right statistical tools.

```
Data: $n$, $f$
Result: $\ell$: A list of $n$ elements sampled from $1 \cdots f$
        in sorted order
begin
    Set $ell = [\,]$
    Set $m = n$ The remaining number of elements
      to draw
    for $i = 1$ to $f - 1$ do
        Sample $d \sim$ Binomial $\left(m, \frac{1}{f-i+1}\right)$
        Add $i$ to $\ell$ $d$ times.
        Set $m \leftarrow m - d$
    end
    Return $\ell$
end
```

Algorithm 2: Binomial Sampling Algorithm (In-Order Sampling with Replacement)

Essentially, we draw from the binomial distribution with m and probability $\frac{1}{f-i+1}$ at each step, which translates to: out of the remaining m elements, how many will be of class i. This allows us to sample in order, and it can be shown that this process has the same distribution as sampling with replacement.

However, a further complication is that **cuRAND**, the CUDA library for random numbers, does not support the binomial distribution. Further more, if we were to replicate the binomial distribution ourselves, it turns out that it is unsuitable for CUDA (possibly a reason why cuRAND does not support it). Firstly, the formula for the binomial distribution involves the binomial coefficient, which is often computed iteratively due to its inclusion of ratios of factorials. Furthermore, even if we precomputed the table of binomial coefficients, drawing from the distribution would require us to compute the CDF for the discrete support and then iteratively walking through to draw from the distribution.

We attempted a compromise by using the normal approximation to the binomial distribution (as the normal distribution is supported by cuRAND) - although the assumptions on n and p might not be strongly held for the approximation to be accurate, importantly, the approximation would be largely unbiased across features - effectively, it would bias the algorithm toward taking a more uniform sampling of features compared to binomial, so features would cluster less.

In practice, we found that the above method incurred further costs down the road in terms of identifying the features iteratively as we walked through the sample array even though we would theoretically be faster for repeated features, and that uniformly sampling out of order and the address hopping would suffice.