Accelerating Random Forest Classification on GPU and FPGA

Milan Shah
North Carolina State University
Raleigh, NC, USA
mkshah5@ncsu.edu

Reece Neff
North Carolina State University
Raleigh, NC, USA
rwneff@ncsu.edu

Hancheng Wu
North Carolina State University
Raleigh, NC, USA
hwu16@ncsu.edu

Marco Minutoli
Pacific Northwest National Laboratory
Richland, WA, USA
marco.minutoli@pnnl.gov

Antonino Tumeo
Pacific Northwest National Laboratory
Richland, WA, USA
Antonino.Tumeo@pnnl.gov

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

ABSTRACT

Random Forests (RFs) are a commonly used machine learning method for classification and regression tasks spanning a variety of application domains, including bioinformatics, business analytics, and software optimization. While prior work has focused primarily on improving performance of the training of RFs, many applications, such as malware identification, cancer prediction, and banking fraud detection, require fast RF classification.

In this work, we accelerate RF classification on GPU and FPGA. In order to provide efficient support for large datasets, we propose a hierarchical memory layout suitable to the GPU/FPGA memory hierarchy. We design three RF classification code variants based on that layout, and we investigate GPU- and FPGA-specific considerations for these kernels. Our experimental evaluation, performed on an Nvidia Xp GPU and on a Xilinx Alveo U250 FPGA accelerator card using publicly available datasets on the scale of millions of samples and tens of features, covers various aspects. First, we evaluate the performance benefits of our hierarchical data structure over the standard compressed sparse row (CSR) format. Second, we compare our GPU implementation with cuML, a machine learning library targeting Nvidia GPUs. Third, we explore the performance/accuracy tradeoff resulting from the use of different tree depths in the RF. Finally, we perform a comparative performance analysis of our GPU and FPGA implementations. Our evaluation shows that, while reporting the best performance on GPU, our code variants outperform the CSR baseline both on GPU and FPGA. For high accuracy targets, our GPU implementation yields a 5-9× speedup over CSR, and up to a 2× speedup over Nvidia's cuML library.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms.*

KEYWORDS

random forest classification, GPU, FPGA

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545067

ACM Reference Format:
Milan Shah, Reece Neff, Hancheng Wu, Marco Minutoli, Antonino Tumeo, and Michela Becchi. 2022. Accelerating Random Forest Classification on GPU and FPGA. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545067

1 INTRODUCTION

Random Forests (RFs) are a popular machine learning method for classification and regression and are widely used in various application domains such as business analysis, advertising systems, software optimization and bioinformatics [4, 11, 20]. RFs are an ensemble method that uses a set of trained trees to perform a prediction. Given an input query, the classification is done by first fitting the query information to each tree in the forest, and then performing a majority vote or averaging the prediction results over the trees.

The use of RFs involves training using existing sampled data and classification on future query data. Both steps exhibit parallelism and are suitable for hardware acceleration. During the training step, multiple decision trees can be trained in parallel. In the classification step, the final decision for a query is determined by traversing all decision trees and then averaging their individual classification results. Since queries and tree traversals are independent of each other, they can be processed in parallel. There have been several previous works focusing on accelerating the training of RFs [4, 11, 15, 20]. While fast training is essential for developing highly accurate RF models in a timely fashion, fast decision making is just as important since quick classification results are expected in many use cases. However, there exists only a few efforts targeting the acceleration of RF classification [10, 14, 19]. In this work, we focus on accelerating RF classification on GPU and FPGA.

An effective implementation of RF classification on GPU involves several challenges. First, the training step can generate large and sparse decision trees. Sparse trees require irregular data structures, often a performance limiting factor on GPU. In addition, large decision trees may not fit in GPU shared memory, causing expensive off-chip memory accesses during tree traversal. Second, each query can follow a unique traversal path for each decision tree, leading to thread divergence. Van Essen et al. [19] tried to mitigate these problems by placing decision trees into GPU texture and shared memory using a compact forest ensemble. To this end, they forced

```
Input:
Forest  { tree1 , tree2, ..., tree N}
Queries { query1 , query2, ..., query M}
Query   { feature1, feature2, ...}

1. for each query in queries:
2.   for each tree in forest:
3.      tmp += tree_traverse(query,tree.root);
4.   query.decision = tmp < N/2? A:B;
```

(a) Classification of many queries using random forest

```
6. tree_traverse(query,node):
7.   if (is_leaf(node)):
8.      return node.value;
9.   go_left = check_path(query, node);
11.  if (go_left):
12.     tree_traverse (query,node.left());
13.  else:
14.     tree_traverse (query,node.right());
```

(b) Decision tree traversal function

Figure 1: Pseudocode of RF classification

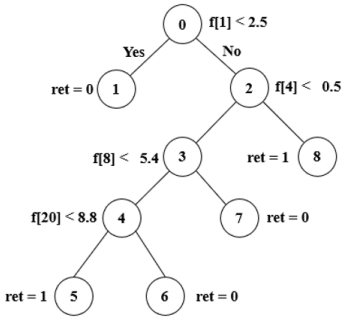
the decision trees to be small by limiting the maximum tree depth to 6 during the training step. However, the tree depth affects classification accuracy.

In this work, we target general RF classification without imposing limitations on the size and depth of the decision trees during the training phase. We propose several optimizations to speed up RF classification. First, we devise a hierarchical tree data structure that trades off the space efficiency of the compressed sparse row (CSR) format with the time efficiency of more regular array-based representations. Specifically, we divide each decision tree into subtrees. The maximum depth of the subtrees is a configurable and tunable parameter. Each subtree is populated to a complete tree, allowing subtrees to be represented through arrays of fixed size [2]. Given the topological characteristics of complete subtrees, node accesses within each subtree can be performed efficiently by indexing into the corresponding array using an arithmetic formula, thus avoiding indirect memory accesses. We then use the CSR format to represent connections between subtrees. Second, we propose three tree traversal code variants that differ in the parallelization approach and memory layout. Lastly, we evaluate the performance of our methods on an Nvidia Xp GPU and a Xilinx Alveo U250 FPGA board. We use publicly available datasets leading to large trees, and we factor accuracy into our analysis. On GPU, our experimental results show a 5-9× speedup over a CSR-based implementation and up to a 2× speedup over Nvidia's cuML library [17]. On FPGA, our implementation achieves a 5.5× speedup over CSR with a single compute unit, and up to a 109.5× speedup with compute unit replication.

2 BACKGROUND

2.1 Classification with Random Forests

Random forests (RFs) generally lead to more accurate classification than single decision trees, since a single decision tree can easily overfit the training dataset. In Fig. 1a we show the operation of RF classification. The code assumes that M queries must be classified into two classes – A and B – using a trained forest with N decision trees. To achieve this, for every query (line 1), the code invokes a tree traversal function on each decision tree in the forest (lines 2-3). The decisions made by the different trees are accumulated in a temporary variable (`tmp`). The final classification of the query is acquired by performing a majority vote over the results of individual decision trees. If we assume that the `tree_traverse` function returns value 0 for class A and value 1 for class B, this can be accomplished by comparing the value of variable `tmp` with N/2 (line 4). Fig. 1b shows the tree traversal pseudocode. We illustrate its operation on the simple decision tree shown in Fig. 2a. During traversal, inner nodes are used to perform the comparison that guides the traversal (visit left or right child), while leaf nodes return a classification value. As can be seen, root node 0 includes the comparison "f[1] < 2.5", which checks whether the value of feature 1 is less than 2.5 (line 9). If the comparison is true, the left child is visited next (lines 11-12), otherwise the right child is visited next (line 13-14). If we assume that feature 1 of the query has value 1.25, the traversal goes left and reaches leaf node 1 (line 7-8). Since node 1 returns 0, the sample query is classified as class A (line 4).



(a) Example decision tree

node_id	0	1	2	3	4	5	6	7	8
children_arr_idx	0	2	2	4	6	8	8	8	8
children_arr	1	2	3	8	4	7	5	6	

(b) Tree structure with CSR

node_id	0	1	2	3	4	5	6	7	8
feature_id	1	-1	4	8	20	-1	-1	-1	-1
value	2.5	0	0.5	5.4	8.8	1	0	0	1

(c) Node attributes with CSR

Figure 2: Decision tree with representation in CSR format

2.2 High Level Synthesis for FPGA

Traditionally, programming FPGAs has required logic design expertise and the mastering of hardware description languages (HDLs), such as VHDL and Verilog. This has added a barrier to the adoption of FPGAs. To increase ease of programming and reduce development time, both industry and academia have been involved in efforts focused on providing high-level synthesis (HLS) capabilities.

HLS tools convert code written using popular programming languages and standards (such as C, C++ and OpenCL) into an HDL specification that can be synthesized to an FPGA bitstream.

In this work, we use the Xilinx Vitis HLS tool to convert C++ kernels into FPGA bitstreams and the OpenCL host API to handle data transfer and task enqueueing. Vitis HLS breaks the C++ kernel up into various pipeline stages, taking the target frequency and initiation interval (II) into consideration. The II is the smallest number of cycles that can pass before another work-item can be entered into the pipeline. Ideally, in working with Vitis HLS, the resulting architecture will have a deep pipeline with a low initiation interval to take advantage of the pipeline parallelism unique to FPGAs compared to CPUs and GPUs. To achieve greater parallelism, each execution pipeline (also called "compute unit") can be replicated. The Xilinx Alveo U250 FPGA card used in this work has four super logic regions (SLRs), an SLR being a single chip containing lookup tables (LUTs), registers, digital signal processors (DSPs), block RAMs (BRAMs), and UltraRAMs (URAMs). There is interconnect logic between all SLRs to allow resource sharing, and each SLR has its own external memory to access large data that may not fit on the BRAM and URAM. Compute unit replication can be done both within and across SLRs. Typically, data transferred from the host CPU to the FPGA are stored in the FPGA's external memory and accessed from it.

2.3 CSR Tree Traversal and Its Bottlenecks

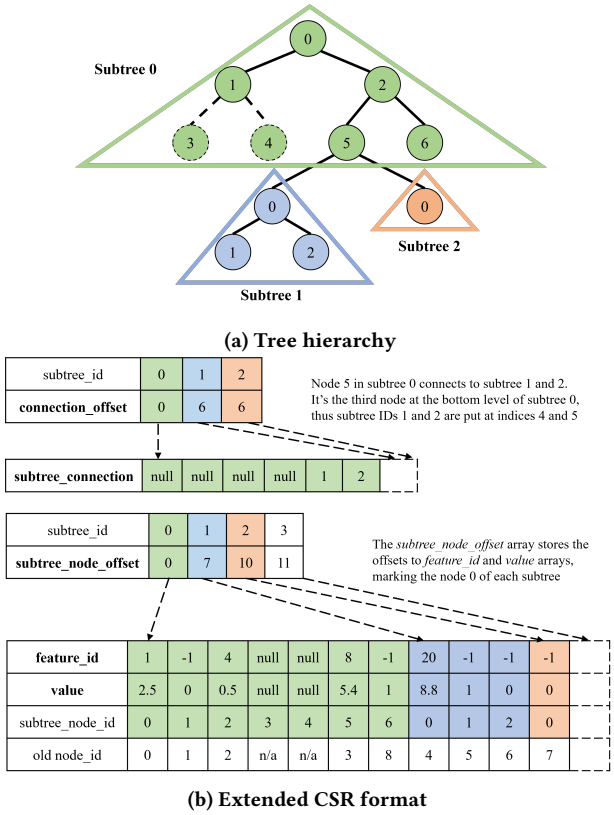
CSR Tree Representation A trained RF contains many decision trees that represent different decision strategies and are independent of each other. Without loss of generality, we discuss how a single decision tree can be laid out in memory. A decision tree representation needs to store two types of information: the attributes associated with the nodes of the tree and the tree topology. For inner nodes, the attributes are the feature identifier and the value used in the comparison; for leaf nodes, they correspond to the associated return value. Node attributes can be stored in arrays and directly indexed using the corresponding node identifier. One common approach to store the tree structure on CPU is through pointer-based representations. Pointer-based data structures, however, are not GPU-friendly, as pointer-chasing leads to poor performance on GPU [12]. A better approach to represent the tree topology on GPU is using the Compressed Sparse Row (CSR) format [3]. In this paper, we use the CSR format as the baseline representation.

Fig. 2b illustrates the CSR representation of the topology of the decision tree in Fig. 2a. For each node, array `children_arr` stores the identifiers of its children, while array `children_arr_idx` stores the starting index of the children within array `children_arr`. For example, node 4 has two children (nodes 5 and 6), and their identifiers 5 and 6 are stored in `children_arr[6:7]`. Thus, we set `children_arr_idx[4]` to 6 to indicate the children of node 4 begin at `children_arr[6]`. These indirect accesses to child nodes are indicated by dotted arrows in the figure. Arrays `feature_id` and `value` in Fig. 2c store the node attributes, and are directly indexed by node identifier. For inner nodes, `feature_id` stores the feature number `n` and `value` stores the value `val` used in the comparison "f[n] < val". For leaf nodes, `feature_id` is set to default value −1 and `value` stores the return value.

Performance Bottlenecks on GPU In this work we use Nvidia GPUs and the CUDA programming model. Streaming multiprocessors (SMs) of a GPU share an L2 cache and an off-chip memory (called global or device memory), while cores on the same SM share fast, on-chip memory (used as L1 cache or shared memory). Two important factors affect GPU performance. First, since threads within a warp (32 threads executing in lock-step) are forced to execute the same instruction, control flow divergence within a warp leads to thread serialization and core underutilization. Second, since global memory is accessed in 128-byte transactions, the memory bandwidth is best utilized when threads within a warp access consecutive 128 bytes memory, allowing memory accesses issued by the warp to be coalesced into a single transaction. On the contrary, if threads within a warp access memory irregularly and the accesses span across several 128-byte chunks, one transaction must be issued for each chunk, leading to underutilization of memory bandwidth.

Fig. 1b shows RF classification requires executing multiple queries against the set of trees in the forest (lines 1-2). Assuming M queries and N trees, the tree traversal function (line 3) will be executed M*N times, once for each query-tree pair. Note these tree traversals are independent of each other and can be performed in parallel. When we assign GPU threads to handle different traversals, threads in a warp can follow different paths, leading to irregular memory accesses to all four arrays that represent a tree, harming performance.

Performance Bottlenecks on FPGA Previous works on accelerating binary tree traversal on FPGA either loaded the entire tree into the FPGA's on-chip memory (i.e., BRAM or URAM) and then performed the tree traversal there [13, 14], or implemented the tree itself into the synthesizable logic (e.g., LUTs) [10]. However, the first approach limits the maximum tree depth supported, while the second limits the number of trees that can be traversed in parallel before having to reprogram the FPGA with other trees. A tree of depth 30, for example, would require 4.2GB of on-chip memory. The Xilinx Alveo U250 FPGA card used in this work only supports 13.5MB per SLR. In practice, this FPGA would only be able to support storing a tree up to a tree depth of around 18 or 19 depending on additional resource utilization within the kernel, which is insufficient for this work as tested trees can scale up to a depth of 35. In addition, the number of queries processed (up to 1.5 million or more) in this work also prevents storing all query data in on-chip memory and instead requires the data to be stored in external memory along with the tree data structures. To be able to process such a large amount of trees and queries, we need to design an FPGA implementation that can process any size tree and any number of queries as long as they can fit in the FPGA's 16GB of external memory. Splitting each tree into subtrees helps in producing a faster design than the baseline CSR version as it results in fewer external memory accesses to the trees, but memory accesses for all other data structures must be considered when analyzing trade-offs between subtree buffering, pipeline depth/initiation interval, and memory access latency as discussed in Section 3.2.



(a) Tree hierarchy

(b) Extended CSR format

Figure 3: Hierarchical tree representation for the tree in Fig. 2a (maximum subtree depth set to 3)

3 HIERARCHICAL RANDOM FOREST CLASSIFICATION

We propose a hierarchical decision tree traversal approach aimed to improve classification speed by limiting irregular memory accesses and improving data locality within each traversal. To this end, we first propose a hierarchical tree format aimed to overcome the shortcomings of the CSR layout. Then, we introduce three RF classification code variants that use the proposed tree layout and differ in their traversal approach and their way to map decision trees to on-chip and off-chip memory. Finally, we implement the proposed code variants on GPU and FPGA, and we discuss platform-specific considerations.

3.1 Hierarchical Decision Tree Layout

We recall that, when the tree topology is encoded using the CSR format, calculating the identifiers of the children of a node involves indirect indexing into array `children_arr` through array `children_arr_idx`, leading to two memory accesses, both potentially irregular, per child node. In our design, each decision tree is divided into triangle-shaped subtrees. The maximum depth of the subtrees is defined as a tuning parameter. We split the tree into subtrees recursively starting from the root node; for each subtree, we stop when either the maximum subtree depth is reached or there is no node at the next depth to be included. We enforce each subtree to be a *complete binary tree*, where all levels, except the last, are completely filled, and the last level has all its nodes to the left side. As complete binary trees, subtrees can be represented through arrays of fixed size [2]. In some cases, this constraint requires adding unused nodes with null values. In Fig. 3a we show the hierarchical structure corresponding to the decision tree in Fig. 2a, assuming a maximum subtree depth of 3. As can be seen, subtree 0 has been expanded with two nodes – nodes 3 and 4 (dotted) – to become a complete binary tree. Since subtree 0 has reached the maximum subtree depth, two subtrees are spawned from the children of its leaf nodes. The use of complete binary tree is key to the design, as it enables the indirect addressing-free array-based layout for subtrees described below. For indexing purposes, the nodes in each subtree are relabeled in breadth first order (starting from node identifier 0).

Fig. 3b shows the memory layout of the transformed hierarchical tree. In the figure, we highlight array elements belonging to different subtrees in different colors (subtree 0: green, subtree 1: blue, subtree 2: orange). We illustrate our proposed memory layout from bottom to top. The `feature_id` and `value`, which store the node attributes, are indexed through the node identifiers within the subtrees (`subtree_node_id`). As a reference, in the figure we also show the original node identifiers from Fig. 2b (`old node_id`). We note that the added nodes 3 and 4 of subtree 0 have been associated null attributes. Array `subtree_node_offset` allows indexing into the above arrays. Specifically, it has an entry per subtree, each storing the offset to root node 0 of that subtree within the `feature_id` and `value` arrays (in the figure, those offsets are also highlighted using dashed arrows). Finally, arrays `connection_offset` and `subtree_connection` store the interconnections among subtrees. Specifically, `subtree_connection` has two entries for each leaf node of a subtree, storing the identifiers of the left and right subtrees that node is connected to (if any). For example, subtree 0 has 4 leaf nodes (note that entries for leaf node 6 can be omitted). The `connection_offset` array has an entry per subtree, each pointing to the first leaf node of the subtree within the `subtree_connection` array (again, in the figure those offsets are also highlighted through dashed arrows).

We note that this approach to store subtree interconnections is similar to the CSR format, and it leads to irregular memory accesses as well. However, accesses to arrays `connection_offset` and `subtree_connection` are only necessary when the traversal moves from a subtree to another, significantly reducing the irregular memory accesses compared with the CSR encoding. If a node n is an inner node of a subtree, its left and right child identifiers are $2n + 1$ and $2n + 2$, respectively. On the other hand, leaf nodes of subtrees connect to the root nodes of different subtrees down the path. For example, node 5 in subtree 0 connects to the root nodes of subtrees 1 and 2. In this case, determining the children of node 5 of subtree 0 requires accessing the `connection_offset` and `subtree_connection` arrays to acquire topological information. The selection of the maximum subtree depth involves a space-time tradeoff: deeper subtrees can potentially require more extra nodes to be made complete, but they lead to fewer indirect indexing operations. We study this tradeoff in the evaluation section.

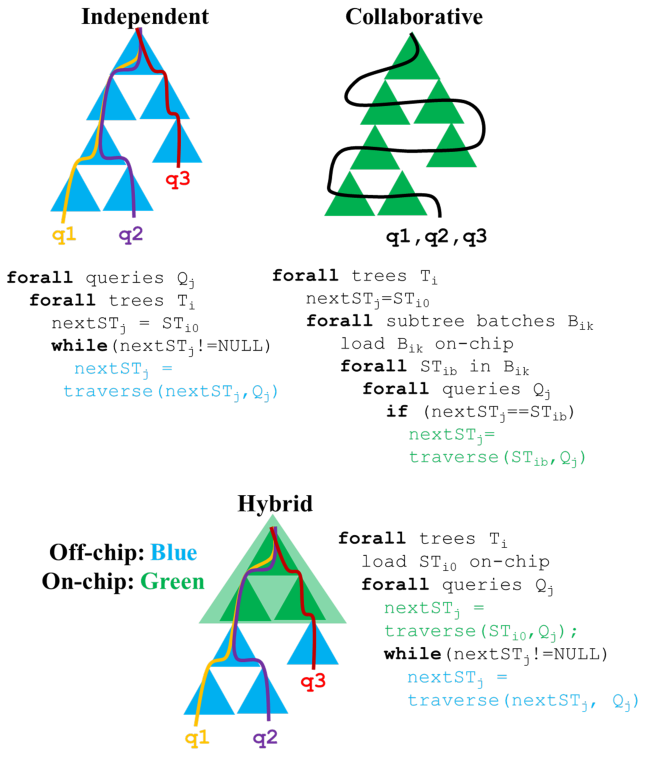


Figure 4: Hierarchical RF classification: independent, collaborative, and hybrid code variants

3.2 RF Classification Code Variants

We recall that, given M queries and N trees, the `tree_traverse` function shown in Fig. 1b needs to be executed $M * N$ times. Since these tree traversals are independent of each other, they can be performed in parallel. In this section, we propose three code variants differing in the mapping of decision trees to memory and in the traversal approach. The proposed kernels are based on the hierarchical tree layout described above and vary in their ability to leverage the GPU and FPGA platforms. Fig. 4 illustrates the traversal method, memory mapping, and pseudocode of the three kernels: *independent*, *collaborative*, and *hybrid* code variants. In the figure, each tree is composed of its subtrees, with blue subtrees accessed from off-chip memory and green subtrees accessed from on-chip memory. Across all variants, traversal within a single subtree remains the same: **1.** Begin at node 0 of the subtree **2.** Check query's feature value against node feature value to determine if next node is left or right child **3.** If next node is a subtree leaf, fetch next subtree. If next node is a tree leaf, write result **4.** If next node is not a leaf, repeat steps 2-3 for next node. For memory access estimates, q = # queries, s = subtree depth, d = tree depth, and t = # trees.

Independent code variant The independent code variant (first in Fig. 4) processes subtrees belonging to the same tree in an iterative fashion. Each query of the test set is mapped to a single thread (or work-item) and threads independently traverse each tree for their assigned query. The trees, in the form of their subtree nodes and subtree connections, are located on off-chip memory. Fig. 4 highlights how three threads, each assigned one query (q1, q2, or q3), traverse a tree. Threads that reach the leaf node of a subtree (or tree) can progress to traverse their next subtree. This variant can lead to a high degree of divergence among threads, with control flow varying greatly depending on a query's feature values. Since query traversals between subtrees are undetermined, the opportunity for data-reuse is limited. Query data can be stored on either off-chip or on-chip memory. The independent code variant performs roughly qtd (worst case scenario) irregular global memory accesses.

Collaborative code variant In the independent code variant, subtrees are accessed from off-chip memory and re-accessed for each query. To alleviate this need, we introduce a code variant that batch loads the subtrees into on-chip memory, and then lets the queries traverse the batched subtrees. The subtree access pattern and pseudocode are shown in Fig. 4. Note that all queries are considered in each subtree, even if the query is not "present" in the subtree (i.e., the query does not need to traverse the subtree). The **if** statement at the end of the pseudocode ensures the query is not processed if it is not present in the current subtree. The maximum subtree depth s that can be processed is $s = \log_2 \frac{M}{48}$, where M is the amount of on-chip memory available in bits and 48 is the required number of bits to store a node's attributes. While this ensures each subtree is only accessed from off-chip memory once instead of several times, going deeper into the tree results in fewer queries being processed. This leads to wasted cycles checking on non-present queries in the subtree. In the worst case, this variant will result in approximately $qt2^{s(\lfloor \frac{d}{s} \rfloor + 2)}$ off-chip memory accesses; however, these accesses are coalesced as the arrays containing subtree data are contiguous.

Hybrid code variant To overcome the control divergence and slow memory accesses of the independent code variant while reducing wasted work present in the collaborative variant, we propose the hybrid code variant (third in Fig. 4). For each tree, this code variant begins the traversal process by loading the first subtree, or "root subtree", into on-chip memory. We note that, for every tree, every query must traverse the root subtree. As such, cooperatively loading the root subtree into on-chip memory can lead to data-reuse across queries. Threads traverse the root subtree in a synchronized fashion and traverse subsequent subtrees in accordance with their query's path. Once all queries have traversed a tree, the root subtree of the next tree is loaded into on-chip memory and traversal begins again. When creating the hierarchical memory layout, we explored the performance of the hybrid variant with larger root subtrees to increase the portion of a tree that requires only on-chip memory accesses. We note that decision trees tend to be denser closer to the root node, thus larger root subtrees do not have to allocate as many unused nodes to make the root subtree complete. Increasing the size of the root subtree, as opposed to loading the first several subtree levels, eliminates accessing the subtree connection arrays while simultaneously enabling high bandwidth node accesses. The greatest constraint on the size of the root subtree is the size of on-chip memory. A root subtree with depth RSD has $2^{RSD} - 1$ nodes, each with an associated value, feature ID, and label indicating if the node is a leaf, while on-chip memory is typically small relative to off-chip memory. The effects of increasing the depth of the root subtree are explored in the experimental evaluation section. The hybrid code variant, in the worst case, has approximately $qt2^s + qt(d - s)$ global

memory accesses across the two sections, with the first section's $qt2^s$ accesses coalesced and the second section's $qt(d-s)$ irregular.

3.2.1 GPU-specific considerations. In all GPU code variants, queries are assigned to threads and processed independently.

Independent GPU kernel Threads run in lock-step with the threads of their warp on GPU and for random forest classification, independent traversal can lead to high degrees of branch divergence within a warp. A thread may finish traversing a tree or subtree with their assigned query before another thread in the same warp, lowering SM utilization and data-reuse. A thread may request a subtree previously accessed by another thread, but a cache eviction could have occurred between the two accesses. Additionally, since all nodes reside on global memory, threads must either access global memory or rely on limited caching with a high number of cache evictions due to irregular traversal across threads. However, the independent GPU kernel does ensure that threads do not have to load all subtrees into on-chip memory, a process that can be costly if many subtrees are never visited. Upon evaluation of loading queries in shared memory versus leaving queries in global memory, we found no significant difference in performance since node accesses remain the primary bottleneck.

Collaborative GPU kernel The collaborative kernel can better leverage memory coalescing and shared memory resources available on GPU compared to the independent kernel. Threads cooperatively load subtrees into shared memory through accessing adjacent data elements, leading to a coalesced access in a warp. Threads also traverse the tree in lock-step and visit every subtree in a tree across all queries, unless no threads in the warp need to visit the subtree. To accommodate subtree batching, threads must store the next subtree to visit for a query if the subtree is in the next batch. Figure 4 indicates that all queries mapped to threads visit each subtree and cannot advance until all threads in the block have completed the tree. The greatest disadvantage of this variant is starvation that can occur if many threads in a warp must wait for a few threads to finish subtree traversal. Queries may not visit a subtree but still must load the subtree into shared memory. Evaluating the collaborative GPU kernel on the three datasets yields a slowdown of $10-20\times$ compared to the independent variant, thus we focus only on independent and hybrid kernels in the experimental evaluation.

Hybrid GPU kernel The hybrid GPU kernel presents two significant advantages over the independent and collaborative kernels: 1. leveraging memory coalescing and shared memory and 2. reducing branch divergence. Loading the root subtree of a tree can be done on a block-level, where adjacent threads access adjacent elements of the root subtree to coalesce global memory requests, allowing for better utilization of the available global memory bandwidth. In addition, on GPU shared memory has significantly lower latency compared to global memory. Furthermore, threads in a warp traverse each tree together since they all rely on accessing the root subtree nodes at the same time. For subsequent subtrees, threads need not visit every subtree and only must wait until all other threads have completed traversing the tree within a thread-block. Larger root subtrees further increase the degree of memory coalescing and shared memory accesses with the main limit being the size of shared memory, 48 KB for the GPU used in the experimental evaluation. In the experimental evaluation, the effects of using the hybrid kernel compared to the independent kernel are reported with respect to global memory loads and branch efficiency (ratio of uniform control flow decisions to all branches).

Other optimizations tested Several additional optimizations have been tested, all yielding either no effect or slowdown of 2-10× relative to the independent variant across three ML datasets. These optimizations include: **1)** using K-Means clustering to place trees accessing similar features adjacent to each other in the memory layout, **2)** assigning each thread-block one tree to traverse for all queries, and **3)** modifying the collaborative variant so that each thread is assigned a query first, then batches of subtrees are loaded into shared memory for traversal. Optimization 1, aimed at promoting data locality, did not yield any significant performance benefit while Optimizations 2 and 3, aimed at promoting data re-use, resulted in significant slowdown relative to the independent variant.

3.2.2 FPGA-specific considerations. In the FPGA kernels, queries are processed sequentially, and parallelism is enabled through pipelining. We recall that the pipeline efficiency is defined by the initiation interval (II), which indicates the number of clock cycles between two consecutive work-items entering the pipeline. Therefore, while implementing similar iterative structures as the pseudocode of Fig. 4, we focused on reorganizing the memory access patterns to achieve a smaller initiation interval (II) on the innermost loop. On the more efficient code variants (the independent and hybrid ones), we implemented compute unit (CU) replication with multiple SLRs in order to add a level of parallelism on top of pipelining while fully utilizing the available external memory bandwidth. We remind that CU replication creates multiple copies of the execution pipeline on a single fabric (aka an SLR), and multi-SLR replicates those copies across several SLRs. Each SLR has its own external memory, so replicating across SLRs won't impact external memory contention the same way replicating within an SLR will where each CU will have to compete for external memory transfers. In practice, performing CU and SLR replication increases throughput as each CU can manage their own set of work-items dispatched to it by the host. Memory stalling from external memory contention and lower frequency must also be considered when performing replication, and the impacts are discussed more in depth in Section 4.4.

Independent kernel The high latency of the off-chip memory where the tree data structures are located leads to memory stalls, which, in turn, translate into a high II (the minimum number of cycles a new work-item can enter the pipeline) and low pipeline efficiency. To alleviate this problem, we stored the query features into BRAM. This reduced the II of the subtree traversal loop from 147 cycles to 76 cycles. However, the II cannot be reduced any further due to a RAW dependency on the variable tracking the current node.

Collaborative kernel In this implementation, each subtree is burst loaded into low latency on-chip memory (BRAM and URAM) and processed on all queries before the next subtree is accessed. Because the burst version loads most of the data structures into low-latency BRAM/URAM (except for query data such as its current subtree, node, and features), we were able to achieve an II of 3 cycles for the subtree traversal loop. We recall that each query has to be sent through the subtree pipeline even if it does not traverse the current subtree, leaving empty pipeline stages. This leads to reduced

Table 1: Machine Learning Datasets

Dataset	*Num Samples*	*Num Features*	*Source*
Covertype	581,012	54	UCI
Susy	3,000,000	18	UCI
Higgs	2,750,000	28	UCI

efficiency, and the efficiency decreases when moving deeper in the tree. In addition, subtree batching adds the startup overhead of loading each subtree before processing all queries. Thus, despite the deep tree traversal pipeline with low II, the collaborative code variant resulted to be the least efficient.

Hybrid kernel The hybrid design consists of two stages: (1) processing the root subtree stored in on-chip memory (BRAM and URAM), and (2) processing the remaining subtrees stored in off-chip memory. The design allows the pipeline to retain full utilization in stage 1, as all queries are guaranteed to traverse the root subtree stored in on-chip memory, while the utilization drops significantly in stage 2. On average, utilization drops to 2^{-s}, where s is the subtree depth. As discussed previously, the hybrid version results in around $qt2^s + qt(d-s)$ global memory accesses across the two stages, with $qt2^s$ at an II between 1 and 3 cycles and $qt(d-s)$ at an II of 76 cycles.

4 EXPERIMENTAL EVALUATION

We train the forests for each of the datasets outlined in Table 1 using the scikit-learn python library [16]. Susy and Higgs [1] come from the domain of particle physics and Covertype is a binarized form of a dataset containing cartographic information to predict the covertype of a wilderness forest. All three datasets are from the UCI Machine Learning Repository [5]. The input data sets are sliced into training set and test set with a 1:1 ratio. We generated forests of varying maximum tree depths and number of trees to determine forest parameters meeting accuracy targets for each dataset.

We analyze the effects of varying the maximum subtree depth in the hierarchical layout and compare the performance of independent and hybrid kernel variants against a CSR-based code and the implementation in Nvidia's open-source cuML library. Additionally, we select the best performing subtree depth and systematically increase the size of the root subtree, comparing against CSR and cuML again. We evaluate how varying parameters of the hierarchical format affects the memory usage. We also evaluate compare the GPU and FPGA performance results.

We use CUDA 11 to compile our code and test the GPU traversal kernels on a Pascal TITAN Xp GPU, a device with 128 cores per streaming multiprocessor (SM), 30 SMs, and a shared memory size of 48 KB per SM. For FPGA, we created C++ kernels and synthesized them on a Xilinx Alveo U250 FPGA with 4x16GB 2400MHz DDR4 RAM. The Alveo U250 is equipped with four Super Logic Regions (SLRs) with combined SLR resources totaling at 2,000 36Kb Block RAMs (BRAMs), 1,280 288Kb UltraRAMs (URAMs), 1.7 million look up tables (LUTs), 3.5 million registers/flip-flops (FFs), and 12,228 DSP slices. The algorithms were compiled and synthesized with Xilinx Vitis Unified Software Platform 2020.2 [9].

Covertype	Number of Trees						
Maximum Tree Depth	10	25	50	75	100	125	150
5	71.4%	71.2%	70.7%	70.6%	71.4%	72.3%	72.4%
10	78.5%	79.6%	80.0%	80.1%	80.1%	80.4%	80.7%
15	81.7%	82.8%	83.0%	83.1%	83.2%	83.3%	83.3%
20	84.4%	85.5%	85.8%	85.9%	86.0%	86.0%	86.0%
25	86.1%	87.3%	87.6%	87.8%	87.8%	87.8%	87.8%
30	87.0%	88.2%	88.4%	88.7%	88.7%	88.6%	88.6%
35	87.2%	88.4%	88.6%	88.9%	88.8%	88.8%	88.8%
40	87.2%	88.5%	88.7%	88.9%	88.9%	88.8%	88.8%
45	87.2%	88.5%	88.7%	88.9%	88.9%	88.8%	88.8%
50	87.2%	88.5%	88.7%	88.9%	88.9%	88.8%	88.8%

Susy	Number of Trees						
Maximum Tree Depth	10	25	50	75	100	125	150
5	77.3%	77.7%	77.8%	77.8%	77.8%	77.7%	77.7%
10	79.3%	79.4%	79.4%	79.5%	79.4%	79.4%	79.4%
15	79.7%	79.9%	80.0%	80.0%	80.0%	80.0%	80.0%
20	79.6%	80.0%	80.1%	80.2%	80.2%	80.2%	80.2%
25	79.2%	79.8%	80.0%	80.1%	80.2%	80.2%	80.2%
30	78.7%	79.6%	79.9%	80.0%	80.1%	80.1%	80.1%
35	78.5%	79.5%	79.9%	80.0%	80.0%	80.1%	80.1%
40	78.5%	79.5%	79.8%	79.9%	80.0%	80.1%	80.1%
45	78.4%	79.5%	79.8%	79.9%	80.0%	80.1%	80.1%
50	78.4%	79.5%	79.8%	79.9%	80.0%	80.1%	80.1%

Higgs	Number of Trees						
Maximum Tree Depth	10	25	50	75	100	125	150
5	67.0%	67.7%	67.8%	68.1%	67.9%	68.0%	68.3%
10	70.5%	70.9%	71.0%	71.0%	71.1%	71.1%	71.1%
15	72.0%	72.6%	72.7%	72.8%	72.8%	72.7%	72.8%
20	71.8%	72.9%	73.3%	73.5%	73.5%	73.6%	73.6%
25	71.1%	72.7%	73.4%	73.6%	73.7%	73.8%	73.9%
30	70.3%	72.6%	73.3%	73.6%	73.8%	73.9%	73.9%
35	70.1%	72.5%	73.2%	73.6%	73.8%	73.9%	74.0%
40	70.2%	72.5%	73.3%	73.7%	73.8%	73.9%	74.0%
45	70.2%	72.4%	73.3%	73.6%	73.7%	73.9%	73.9%
50	70.1%	72.5%	73.3%	73.6%	73.8%	73.9%	73.9%

Figure 5: Accuracy Scores of Forests with Varying Tree Depth and Number of Trees on ML Datasets

4.1 Accuracy-Guided Parameter Selection

The search space for parameters of the random forest algorithm is immense, with many factors determining the shape of each tree and computational complexity of the inference problem. The scikit-learn API RandomForestClassifier trains forests based on tuning parameters provided by the user including maximum tree depth, number of trees, minimum samples to create a leaf node, and maximum number of leaf nodes. The values of parameters are typically selected using accuracy scores of models with varying parameter combinations on a validation dataset [7]. For this paper, we focus on the maximum tree depth and the number of trees in the forest. To target practical applications, we train forests of tree depths varying from 5 to 50 and number of trees ranging from 5 to 150 using all three machine learning datasets. The accuracy scores are reported in the heat-maps of Fig. 5, where lower accuracy scores are more red and high accuracy scores are more green. Accuracy in RF classification is the percent of queries correctly classified by the RF model.

Given the reported scores, we select tree depth values in the green band of values for each dataset and use 100 trees to generate forests. Since the number of trees does not yield a significant difference in accuracy as we visit values near 100 trees, this focal point is sufficient in determining the effects of the hierarchical tree format and kernels. CSR, cuML, and the kernel variants all scale linearly in execution time with the number of trees, thus we do not report speedup for varying number of trees. We chose the lowest maximum tree depths that can support a sufficiently high accuracy ($\approx$ 0.3% of the max reported accuracy) along with neighboring values: 25-35 for Higgs, 15-25 for Susy, and 30-40 for Covertype. Balancing computational complexity with accuracy reflects the practical motivation of the experimental evaluation.

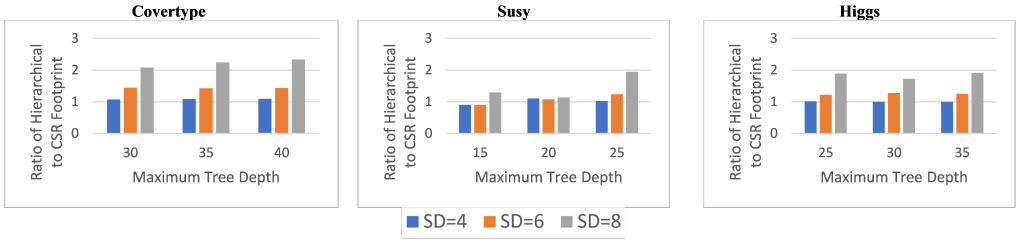


Figure 6: Comparison of Hierarchical Representation Memory Footprint to CSR (Maximum Subtree Depth = *SD*)

4.2 CSR Memory Footprint Analysis

In this section, we report the memory footprint of the proposed hierarchical tree representation. We vary the maximum subtree depth with the values 4, 6, and 8. Note the maximum number of nodes in a subtree is $2^{SD} - 1$ where *SD* is the maximum subtree depth. In Fig. 6, CSR and hierarchical representations are compared with varying subtree depths using all datasets. The y-axis is the ratio of hierarchical memory used to CSR memory used. The x-axis is the maximum tree depth of each forest, trained using 100 trees. *SD* indicates the max subtree depth value of the generated forest.

Results in Fig. 6 indicate that in terms of memory footprint, lower maximum subtree depth values, such as subtree depth of 4 and 6, can occupy nearly the same memory space as the CSR format. Larger subtrees tend to require more space than smaller subtree forests since there are more likely to be empty nodes in a larger subtree. The maximum number of nodes in a subtree being equal to $2^{SD} - 1$ implies an exponential relationship between depth and nodes in a subtree, thus a large increase in memory footprint is seen between $SD = 6$ and $SD = 8$. Fig. 6 further suggests relative memory usage as trees grow vertically (i.e. deeper trees) since the deeper trees of the Covertype forests have a larger footprint than the shallower trees of the Susy forests. Memory footprint can relate to the composition of the tree, specifically the ratio of leaves to total nodes. A greater proportion of leaves in levels of the tree higher than the lowest level can lead to more memory usage in the hierarchical format since empty nodes are allocated to fill a subtree to be complete. The following sections will explore how the hierarchical layout performs on both GPU and FPGA, with kernels targeting platform strengths. The speedup from using the hierarchical layout will be shown to greatly offset the memory benefits of the CSR format.

4.3 GPU Timing Analysis

In this section, we report GPU performance results of random forest classification under different parameters settings. In all cases, we report the speedup over CSR-based traversal of the independent variant, the hybrid variant, and NVIDIA's cuML forest inference. On random forests of 100 trees and ranges of tree depths chosen from the above parameter selection, the execution time of CSR-based RF classification varies as follows: 0.4 s to 0.6 s for Covertype, 1.4 s to 3.2 s for Susy, and 4.3 s to 5.2 s for Higgs. The number of queries is the size of the test set for each dataset, or half of the number of samples listed in Table 1. We omit showing the results achieved by the collaborative code variant, which is consistently 10-20x slower than the independent code variant.

Maximum Subtree Depth: Fig. 7 reports results from tree depths that met the accuracy scores outlined in Fig. 5 for each of the datasets labeled. The maximum subtree depth, *SD*, is selected with values 4, 6, and 8. As can be seen, the hybrid kernel consistently outperforms the independent kernel across all subtree depth values tested. Speedup against CSR ranges from approximately 2.5× to 4× for the independent and from 4.5× to 9× for the hybrid code variants. Fig. 8 reports differences in global memory loads and warp divergence for the two kernels. The hybrid kernel allows for offloading of global memory loads to shared memory and less branch divergence as the root subtree of each tree is traversed by all threads. As the maximum subtree depth increases, the proportion of global loads between the hybrid and the independent variants decreases substantially, since a larger fraction of the loads is serviced by shared memory. Nodes from subsequent subtrees will also be less likely to be evicted from the L1 cache since all queries undergoing traversal visit the same tree. In the independent variant, threads mapped to a query can finish traversing a tree before others in the same thread-block, leading to greater warp divergence and requests for node data not already present in the L1 cache.

The hybrid kernel generally performs better than cuML, especially for larger maximum subtree depth values. cuML speedup over the CSR code ranges from approximately 4× to 5×. Large maximum subtree depths enable less indirect memory accesses to traverse from subtree to subtree and more arithmetic indexing to access node values. $SD = 4$ for the hybrid variant approximately matches or outperforms cuML for Susy and Higgs datasets and is slightly outperformed for the deeper tree depths present in the Covertype forests. Across all datasets, deeper subtrees generally lead to better performance compared to shallower subtrees. Performance changes as a result of increasing tree depth reflect how well a particular traversal method can scale with the exponentially growing computational complexity of deeper trees. Speedup of the various traversal methods remain consistent when tree depth is fixed and number of trees changes. This occurs due to a linear scaling of execution time with the number of trees. Since the number of nodes in a forest scales linearly with the number of trees, speedup remains relatively constant.

Root Subtree Depth: Table 2 reports the hybrid variant speedup against CSR for forests generated with maximum root subtree depth, *RSD*, varying from 8 to 12 and subsequent subtree depth fixed at 8 using the same datasets. Maximum tree depths are varied with speedup reported in each of the *GX* columns.

Increasing the root subtree depth typically increases the speedup of the hybrid variant as the overall maximum tree depth increases,

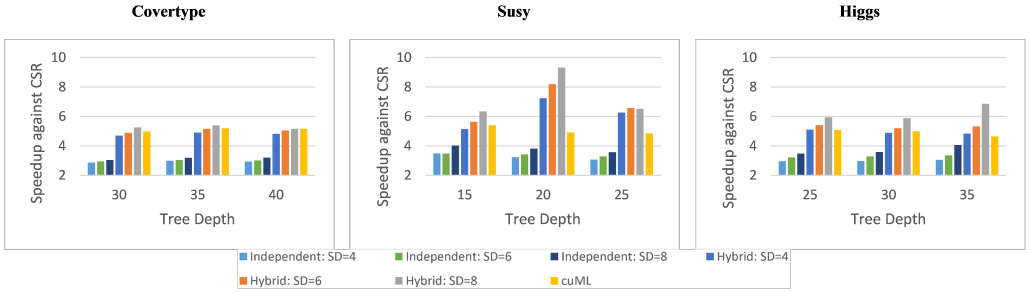


Figure 7: Effects of Tree Depth and Subtree Depth (SD) on GPU Variant Speedup over CSR (Num Trees = 100)

Table 2: Effects of Root Subtree Depth (RSD) on GPU Hybrid and FPGA Independent Variants. GX and FX denote GPU (in speedup) and FPGA (in seconds) runs with a RSD of X, respectively. d is tree depth.

Dataset	d	G8	G10	G12	F8	F10	F12
Covtype	30	5.3	5.4	5.5	6.2	6.2	6.0
	35	5.4	5.5	5.8	6.5	6.3	6.1
	40	5.2	5.4	5.6	6.5	6.3	6.2
Susy	15	6.4	7.2	8.1	22.5	22.7	22.7
	20	9.3	9.4	9.1	30.0	29.9	29.6
	25	6.5	7.9	8.3	35.3	33.4	33.1
Higgs	25	6.0	6.3	6.5	32.3	31.0	30.7
	30	5.9	6.5	7.1	33.8	32.5	31.6
	35	6.9	6.9	7.0	32.8	32.3	32.3

with the exception of the forest generated with tree depth of 20 and 100 trees in the Susy results. Larger root subtrees result in more nodes stored in shared memory, increasing high bandwidth memory accesses and a greater proportion of coalesced memory requests to total memory requests, in line with the data presented in Fig. 8. Fig. 8 indicates larger subtrees reduce global memory accesses thus larger root subtrees can further reduce global memory accesses. Speedup can increase from 6× to nearly 8× as seen in the Susy forest of tree depth 15 and number of trees 100 when root subtree depth increases from 8 to 12. In all, larger root subtrees make the hierarchical layout more cache-friendly and can provide additional performance gain to the hybrid kernel variant.

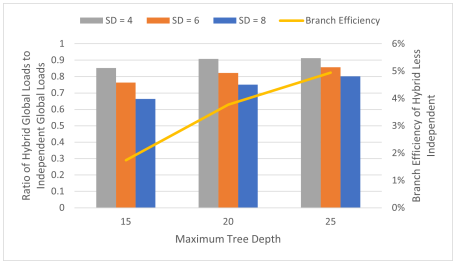


Figure 8: Comparison of Global Load Requests and Branch Efficiency for Hybrid and Independent Variants for Susy Dataset (Maximum Subtree Depth = SD)

Table 3: Comparison of FPGA versions with a synthetic dataset ($d = 15, s = 10, t = 40, q = 250k$). Unless otherwise noted, the version contains a single CU. Frequency (f) is in MHz and II is in cycles. xSyC = x SLRs with y CUs in each.

Version	Time (s)	Stall %	vs CSR	f	II
Baseline (CSR)	162.47	10.97%	1.00	300	292
Independent	54.59	10.76%	2.98	300	76
Collaborative	1957.80	90.68%	0.08	300	3
Hybrid	29.76	25.09%	5.46	300	3/76
Independent 4S12C	1.48	30.39%	109.48	300	76
Hybrid 4S12C	2.44	79.80%	66.58	300	3/76
Hybrid Split 4S10C	2.23	-	72.92	245	3/76

4.4 FPGA Timing Analysis

Code Variants Comparison: Table 3 reports the execution time of all the FPGA code variants on a synthetic dataset with 250k queries (q), 40 trees (t) with depth 15 (d), and a maximum subtree depth of 10 (s). We show the results of implementations with a single and multiple compute units (CU). In the latter case, we use 4 super logic regions. As can be seen, all versions based on our hierarchical decision tree layout achieve a speedup over CSR except for the collaborative version due to the massive query workload starvation on child subtrees not making up for the subtree burst overhead. The speedup over CSR can be attributed to fewer external memory accesses per pass and the buffering of data structures resulting in a much lower II. Looking only at single CU versions, the hybrid variant has the best speedup followed by the independent variant. When looking at parallel scalability (i.e., multiple CUs), however, the independent kernel is the most scalable, providing the best performance over replicating the hybrid compute units. We believe that replicating stage one of the hybrid version was causing too much external memory stalling at 70% (as replicating the collaborative version resulted in slow down due to memory contention), so we created a split version where there was a single stage one compute unit for each SLR, and stage two was then replicated as usual. While we did see speedup over the non-split version, kernel complexity limited the amount of compute units we could replicate (10 per SLR instead of 12), and also resulted in lower frequency (245MHz vs 300MHz). Comparing this to the replicated independent version, we see stage one took around 1.3 seconds, and stage two took 0.8 seconds compared to independent's

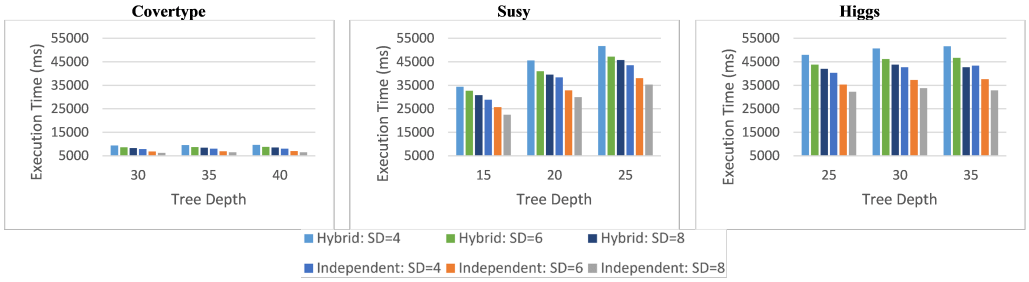


Figure 9: Effects of Tree Depth and Subtree Depth (*SD*) on FPGA Variant Runtime (Num Trees = 100)

1.48 seconds total. This shows the non-scalable stage (stage one) takes up the majority of the execution time and, while faster on a single compute unit, the independent version ended up being more scalable and able to take advantage of more parallelism.

Tree Configuration Analysis: Figure 9 shows the results reported when varying the maximum subtree depth (SD) with both the independent and collaborative code variants. As for the synthetic dataset, the independent code variant outperforms the hybrid code variant in almost all configurations with the same SD, showing the stage one size in the hybrid version is not enough to offset the scalability trade-off. Due to the large execution time of the CSR version, we were unable to collect its data for datasets as large as these, and instead included comparisons with a smaller synthetic dataset in Table 3. Similarly to GPU, deeper subtrees result in lower execution times for both versions as shown in Table 2, and execution time rises with the number of trees in the dataset.

4.5 FPGA vs GPU

Figure 10 indicates GPU massively outperforms FPGA. This large gap in performance is due to its much faster clock frequency, higher peak memory bandwidth, and parallelism (thousands of cores vs FPGA's 40-48). FPGAs typically make up for this by utilizing hardware level pipelining, but the high II in the independent version due to a RAW dependency inhibits one of the FPGA's main feature as an accelerator. The lower theoretical peak memory bandwidth ($\approx$ 77 GB/s on the Alveo U250 vs $\approx$ 547.5 GB/s on the Titan Xp) in the collaborative version also inhibits it from performing more similarly.

5 RELATED WORK

GPU work: There exist several attempts to accelerate general recursive tree traversal algorithms on GPUs. First, Karras [8] proposed using an explicit stack to keep the order in which the children nodes are visited, eliminating the cost of recursion from general tree traversal algorithms and leading to more effective GPU deployment.

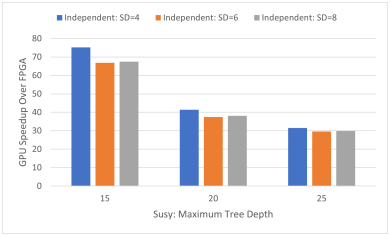


Figure 10: GPU vs FPGA on SUSY (Max Subtree Depth = *SD*)

This method does not apply to our work on decision tree traversal which inherently follows a top-down path [18]. As only one child node is selected to be visited at each node, RF classification does not require the use of stacks to track the order of other child nodes. Second, Goldfarb et al. [6] accelerated general recursive tree traversal by introducing lockstep execution to the explicit stack method above, together with sorting of queries as a preprocessing step [18]. Although they have not reported the cost of presorting, their results show that lockstep execution, together with the presorting, significantly boosts performance of general recursive tree traversal on GPUs. As similar queries are grouped together, lockstep traversal reduces warp divergence and provokes coalesced memory accesses. Although their method for general tree traversals can be used on decision tree traversals, our work differs in two ways. With our collaborative GPU kernel similarly employing lockstep execution, we do not consider presorting as a preprocessing step. As ML datasets often are high-dimensional and can use non-numeric features, presorting the queries would lead to an extra cost that cannot be amortized. In addition, with the proposed hierarchical tree representation, our hybrid GPU kernel can fit sets of subtrees into shared memory, leading to faster tree accesses. Lastly, Wu and Becchi [21] target different general recursive tree traversal patterns and propose a greedy tree traversal variant for binary search tree, which allows a GPU thread to immediately fetch and start another query when the current assigned query finishes. According to their profiling data, this greedy variant helps reduce thread divergence (preventing idle threads), but increases the chance of uncoalesced memory accesses, leading to performance degradation. Thus, we do not consider applying this variant to our work on decision trees.

FPGA work: Lin et al. [10] implemented three random forest architectures: *memory-centric*, with the tree stored in the on-chip memory, *comparator-centric*, using LUTS to store tree parameters and comparisons, and *synthesis-centric*, with the tree synthesized as boolean functions in LUTs. Synthesis-centric uses the fewest resources, but has the highest context switch time whereas memory-centric has the fastest context switch time but uses the most resources. Comparator-centric finds a middle ground between the two in both area and context switching. For all three versions, the entire tree (represented as the tree or boolean functions) must fit in the FPGA. Nakahara et al. [14] accelerated random forest classification with Altera on OpenCL, applying burst memory transfer optimization, utilizing fixed point bits instead of floating point bits, and full pipelining to achieve a speedup 14× higher than CPUs and

10× higher than GPUs. For this work, the entire tree must fit within the FPGA on-chip memory to provide such speedups.

These prior works require that entire trees fit in the relatively small on-chip memory, a limitation that our work has aimed to remove. The hierarchical representation still enables the use of fast on-chip memory, however, the tree is partitioned into portions that fit in this level of the memory hierarchy. As such, forests are able to grow trees to be far deeper than the imposed limits of prior works to meet high accuracy targets while improving performance.

6 CONCLUSION

In this work, we have proposed and evaluated a novel memory layout for random forest inference across two platforms: GPU and FPGA. We have designed and tested three kernels that leverage the memory layout and accelerate random forest inference for hundreds of thousands to millions of queries. For GPU, the hybrid kernel variant outperformed CSR and cuML the greatest, especially for larger subtree depths. Utilizing shared memory in the hybrid version suits both the cache-friendly nature of the hierarchical layout as well as the need for fast node value accesses. On FPGA, the independent version was 109.5× faster than the CSR version and outperformed the hybrid version when full scalability was realized. Despite this, FPGA still trails GPU due to the nature of the algorithm inhibiting deep pipelining at a low initiation interval and GPU having a much higher bandwidth and clock frequency. For both, increasing both subtree depth and root subtree depth improved inference time.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation awards CNS-1812727 and CCF-1741683 at North Carolina State University, and by the U.S. DOE ExaGraph project and the ADOX-V project at the Pacific Northwest National Laboratory (PNNL).

REFERENCES

[1] P. Baldi, P. Sadowski, and D. Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications* 5, 1 (jul 2014). https://doi.org/10.1038/ncomms5308

[2] Paul E Black et al. 2020. Dads: The on-line dictionary of algorithms and data structures. *NIST: Gaithersburg, MD, USA* (2020).

[3] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[4] Chuan Cheng and Christos-Savvas Bouganis. 2013. Accelerating Random Forest training process using FPGA. In *2013 23rd International Conference on Field programmable Logic and Applications*.

[5] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[6] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[7] Trevor Hastie and Robert Tibshirani. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2 ed.). Springer.

[8] Tero Karras. 2012. Thinking Parallel, Part II: Tree Traversal on the GPU. https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/

[9] Vinod Kathail. 2020. Xilinx Vitis Unified Software Platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Stephen Neuendorffer and Lesley Shannon (Eds.).

[10] Xiang Lin, R.D. Shawn Blanton, and Donald E. Thomas. 2017. Random Forest Architectures on FPGA for Multiple Applications. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*.

[11] Diego Marron, Albert Bifet, and Gianmarco De Francisci Morales. 2014. Random Forests of Very Fast Decision Trees on GPU for Mining Evolving Big Data Streams. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*.

[12] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[13] Oyku Melikoglu, Oguz Ergin, Behzad Salami, Julian Pavon, Osman Unsal, and Adrian Cristal. 2019. A Novel FPGA-Based High Throughput Accelerator For Binary Search Trees. https://doi.org/10.48550/ARXIV.1912.01556

[14] Hiroki Nakahara, Akira Jinguji, Tomonori Fujii, and Simpei Sato. 2016. An acceleration of a random forest classification using Altera SDK for OpenCL. In *2016 International Conference on Field-Programmable Technology (FPT)*. 289–292. https://doi.org/10.1109/FPT.2016.7929555

[15] Oleksandr Pavlyk and Olivier Grisel. 2020. Accelerate Your scikit-learn Applications. (2020). https://medium.com/intel-analytics-software/accelerate-your-scikit-learn-applications-a06cacf44912

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[17] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803* (2020).

[18] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. 1995. Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *J. Parallel and Distrib. Comput.* 27, 2 (1995), 118–141. https://doi.org/10.1006/jpdc.1995.1077

[19] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. 2012. Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*.

[20] Zeyi Wen, Hanfeng Liu, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2020. ThunderGBM: Fast GBDTs and Random Forests on GPUs. *J. Mach. Learn. Res.* 21, 108 (2020), 1–5.

[21] Hancheng Wu and Michela Becchi. 2017. An Analytical Study of Recursive Tree Traversal Patterns on Multi- and Many-Core Platforms. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. 586–595. https://doi.org/10.1109/ICPADS.2017.00082