Booster: An Accelerator for Gradient Boosting Decision Trees Training and Inference

Mingxuan He
Elmore Family School of ECE
Purdue University
West Lafayette, IN, USA
he238@purdue.edu

Mithuna Thottethodi
Elmore Family School of ECE
Purdue University
West Lafayette, IN, USA
mithuna@purdue.edu

T. N. Vijaykumar
Elmore Family School of ECE
Purdue University
West Lafayette, IN, USA
vijay@ecn.purdue.edu

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00106

***Abstract*—Recent breakthroughs in machine learning (ML) have sparked hardware innovation for efficient execution of the emerging ML workloads. For instance, due to recent refinements and high-performance implementations, well-established *gradient boosting decision tree (GBT)* models (e.g., XGBoost) have demonstrated their dominance in commercially-important contexts, such as table-based datasets (e.g., relational databases and spreadsheets). Unfortunately, GBT training and inference are time-consuming (e.g., several hours of training for large datasets). Despite their importance, GBTs have not been targeted for hardware acceleration as much as neural networks.**

We propose Booster, a novel accelerator for GBTs based on their unique characteristics. We observe that the dominant steps of GBT training and inference (accounting for 90-98% of time) involve simple, fine-grained, independent operations on small-footprint data structures (e.g., histograms and shallow trees) – i.e., GBT is on-chip memory bandwidth-bound. Unfortunately, existing multicores and GPUs do not support massively-parallel data structure accesses that are irregular and data-dependent. By employing a *scalable sea-of-small-SRAMs* approach and an SRAM bandwidth-preserving mapping of data record fields to the SRAMs called *group-by-field mapping*, Booster achieves significantly more parallelism (e.g., 3200-way parallelism) than multicores and GPUs. In addition, Booster employs a redundant data representation that significantly lowers the memory bandwidth demand. Our simulations reveal that Booster achieves 11.4x and 6.4x speedups for training, and 45x and 22x (21x and 11x) speedups for offline (online) inference, over an ideal 32-core multicore and an ideal GPU, respectively. Based on ASIC synthesis of FPGA-validated RTL using 45 nm technology, we estimate a Booster chip to occupy 60 mm^2 of area and dissipate 23 W when operating at 1-GHz clock speed.

***Index Terms*—Gradient boosting, accelerator**

I. Introduction

Machine learning has made enormous strides in recent years on multiple fronts. For visual recognition, there are deep neural networks (DNNs) [24]. Concurrently, *gradient boosting with decision trees (GBT)* [12] has continued to be a dominant method for table-based datasets (e.g., relational databases and spreadsheets which are indispensable in enterprises for capturing sales, revenue, payroll, inventory, and insurance data), especially those with categorical or discrete data as is common in many analytics (e.g., recommendation systems, insurance, regression). The advances and open-source software for offline training of GBT models (e.g., [9], [11], [16], [32]) have resulted in their extensive use. For instance, Facebook [14] and Bing [18] have used GBT to predict advertisement clicks, a key revenue source. Further, GBT has been used for botnet detection [19], exotic particle search in the Large Hadron Collider [7], insurance claim prediction [2], supervised ranking [23], and airline flight delay prediction [1], [22]. XGBoost [9], which has over 5800 citations, won the prestigious Infoworld's 2019 Technology of the Year award, the first place in the ACM 2017 RecSys Challenge (for recommendation systems), numerous Kaggle awards over the years as stated in the 'Awesome XGBoost' Web site [5], as well as the 2016 John Chambers award [33]. XGBoost alone is used by over 3200 companies, including eBay, Capital One, and Nvidia [15]. In Kaggle 2021 public solutions, GBT is used in 4 out of 15 winners as opposed to neural networks (NNs) in 9. Anecdotally, a Google search of GBT reveals its indisputable importance for tabular data. In addition to its empirical success, GBT's rich theoretical foundation gives confidence in the method [27]. While numerous accelerators (and proposed accelerators) exist for NNs [10], the most popular ML method, only a few exist for GBT, the second-most popular method (Section III-G).

The core idea of boosting is to combine many simple or *weak* models (i.e. low-accuracy models) to form a *strong* model (i.e., a highly-accurate model). Unlike DNNs, where the parameters of the model are trained via gradient descent, GBT [12] refines an aggregate function (the *strong* model) by taking approximate gradient steps in the function space (by incrementally adding *weak* models). Determining the next best decision point to grow a given tree by considering all possible points in all possible fields is computationally expensive. A common alternative is to replace the fields with discretized values via histograms that approximate the points—effectively reducing the number of possible decision points from n to the number of histogram bins k, where $k \ll n$ [9], [16], [17]. Empirically, these histogram-based methods are faster with only a slight loss of accuracy [16], [17].

Training GBT models is time-consuming (e.g., several hours on datasets with hundreds of millions of records). Further, increasing the accuracy typically requires more training data which in turn requires larger models (more trees) to avoid over-fitting. Both more training data and larger models lead to longer training times. On the other hand, inference latency is often critical in many online (e.g., advertisement click prediction) and offline (e.g., analytics) scenarios. As such, we propose an accelerator for GBT training and inference. Fortunately, our accelerator uses the *same* hardware for training and inference.

The GBT training algorithm grows the model one decision tree at a time; and grows each tree by starting with its root and

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00106

splitting the nodes to form a tree. The training time for GBT models is dominated by three key steps – histogram binning of records' gradient statistics, single-predicate evaluation, and evaluating each record on a single tree (Section II). Inference, on the other hand, involves traversing hundreds of shallow regression trees (e.g., 500, 6-deep trees).

The training algorithm has enormous parallelism, across both the *fields* of an input record (e.g., 100 *fields* expanded to some 4000 *features* due to one-hot encoding of *categorical data*, as explained in Section II) and different input records (e.g., 200 million). However, a single input record may fall into different histogram bins for different fields (e.g., bin 20 for field 1 and bin 38 for field 2) and the input records may fall into different histogram bins for a single field, imposing significant irregularity in histogram access. On the positive side, the data structures (e.g., histograms and decision trees) are small enough to fit on-chip (e.g., under 2 MB). While all the records are binned at the root, the other nodes receive only a subset of the records, filtered by the predicates in the path from the root to a given vertex. This subsetting makes the input record accesses in the binning and single-predicate evaluation irregular. Further, the single-predicate evaluation and the one-tree traversal use only one field and a subset of fields of a record, respectively, potentially wasting memory bandwidth on the unused fields. Similarly, inference has parallelism across the decision trees traversed by each record and across different records. Inference is also irregular in that the records may take different paths through a tree and, of course, different trees' predicates are different from each other. Thus, irrespective of training or inference, GBT has *enormous fine-grain, irregular access parallelism* where the compute work is mostly accessing histogram bins or decision tree nodes (i.e., on-chip memory bandwidth-bound). This data access parallelism is in sharp contrast to the compute parallelism of DNNs.

While CPUs can handle irregularity, their parallelism is limited (e.g., 64 threads in a 32-core multicore). GPUs, on the other hand, can exploit immense parallelism but face difficulty with irregularity. While the GPU Shared Memory can accommodate some irregularity, its limited capacity amounts to either almost entirely turning off multithreading or incurring considerable underutilization (e.g., over 90% idle GPU lanes), as we discuss in Section II. Clearly, DNN accelerators which are customized for tensor computations, such as the TPU, are a mismatch for GBT.

Based on the above workload characteristics, we propose an accelerator for GBT, called Booster, which makes the following contributions:

- To match the *fine-grained, irregular parallelism in accessing many small data structures*, we propose a *scalable sea-of-small-SRAMs* architecture. The SRAMs hold the histograms (for training) or the decision trees (for training or for inference), as appropriate. Each SRAM has an associated floating-point unit to perform some computation. The number of SRAMs determines the degree of parallelism required to match the memory bandwidth – across the fields of a record and across the input records (e.g., scale to 3200 SRAMs processing 64 fields per record and 50 records in parallel).
- Naively mapping one-hot encoded features to SRAMs (by packing the histogram bins into SRAMs, for example) would result in idle SRAMs as a given record would have only one active feature out of all the one-hot encoded features corresponding to a categorical field. Further, for accurate binning, the training algorithm bins the records with missing fields in a default histogram bin for each such field. Thus, while the one-hot encoded features appear to be sparse, the higher-level fields are dense where every record has every field exactly once. We exploit this observation to map a high-level field – i.e., all the corresponding one-hot encoded features including the bin for absence – to an SRAM, achieving full SRAM bandwidth utilization (exactly one access per SRAM). Though dense and parallel, the accesses remain irregular, rendering GPUs ineffective and CPUs insufficient.
- Finally, to save memory bandwidth in single-predicate evaluation and one-tree traversal in training, we employ *a per-field column-major format* for the input records, in addition to the natural *per-record row-major format*. Column-major format is well-known, our novelty is the redundancy. The input records already undergo offline pre-processing in software. The *redundant* format's pre-processing overhead is amortized over as many scans of the input data as the trees built by training.

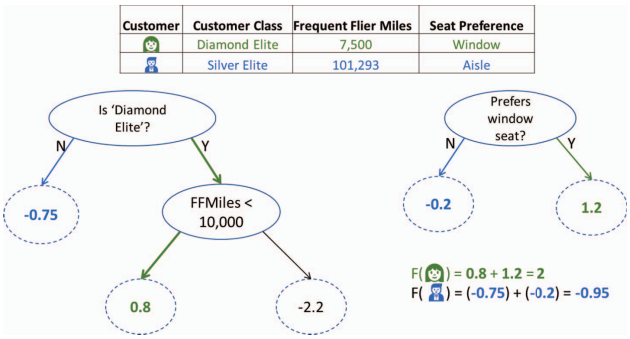

Fig. 1. Tree Ensemble Model: Inference example

We evaluate Booster via simulations and ASIC synthesis. Our simulations reveal that Booster achieves 11.4x and 6.4x speedups for training, and 45x and 22x (21x and 11x) speedups for offline (online) inference, over an ideal 32-core multicore and an ideal GPU, respectively. An ASIC synthesis of FPGA-validated RTL using 45 nm technology estimates a Booster chip to occupy 60 mm^2 of area and dissipate 23 W of power when operating at 1-GHz clock speed.

II. Gradient Boosting Trees

Our description is common to all state-of-the-art implementations of gradient boosting trees (GBT) (e.g., [9], [11], [16]). Section I refers to XGboost more than the others because XGBoost is the first. We summarize GBT with a focus on (1) offering high-level intuition on its operations, and (2) identifying key workload behavior relevant to Booster's design; we do not aim to describe *all* of GBT's details.

A. Model and Training

The tree ensemble models used by GBT have the following characteristics:

- The ensemble has K trees with decision rules in the interior nodes and weights (w) associated with the leaves.
- The model operates on table-based datasets. Each row of the table corresponds to a record and each column corresponds to a field.

TABLE I
GBT training steps (**key data structure**)

Step	Details
①	**histogram-binning** of the gradient statistics of each field of the input records that reach the leaf under consideration (by default, all records reach the root)
②	based on the gradient statistic summations in each histogram bin, **finding the best bin to split a leaf**; a split point adds children to the current leaf using a predicate based on a feature and a split criterion (e.g., age $\geq$ 22)
③	**applying the newly-determined splitting predicate** to partition the records reaching the new vertex into "predicate true" and "predicate false" subsets for the next iteration of the leaf-splitting algorithm
④	repeating the first three steps until the tree reaches the desired depth (or until the loss does not improve)
⑤	**evaluating the completed tree** to determine the residual loss of the model with the newly added tree, and to compute gradient statistics for the updated model
⑥	repeating the above steps for a new tree if the loss continues to decrease

- There are n training records, with known "golden" outputs; The i^{th} record has a known output value of y_i.
- Inference involves passing a record through all the individual trees to generate a weak prediction per tree (w associated with the leaf) which are combined for the final, strong prediction ($\hat{y}_i$). Figure 1 shows inference with $K = 2$ trees for a fictional airline's frequent-flier database. The lower-right computation shows the strong prediction for the green and blue customers. A positive score may mean (say) that the customer is more likely to respond to specific advertisement campaigns.
- GBT is agnostic about the loss function $l(\hat{y}_i, y_i)$ which should be differentiable and convex. In addition to the loss function, the objective function (which GBT minimizes during training) includes terms for penalizing tree complexity and for weight regularization.

The GBT training algorithm, common to all implementations [9], [9], [11], [16], arrives at the final K-tree model by incrementally growing the ensemble one tree at a time. Each tree is in turn grown by splitting leaf vertices, one leaf at a time as shown in Table I. At a high level, GBT training determines the split points of a leaf based on the distribution of first- and second-order gradient statistics (g_i and h_i) of the training records. We expand on this process to identify the key computational tasks.

To describe training, Figure 2 defines the fields of the database in Figure 1. (Only the field definitions, and not the data, are shown.) The fields include categorical data (fields 1 and 2 shown in blue and amber, respectively, in Figure 2) as well as numerical fields (field 3 in green). For simplicity, we first show a logical view of training. The real implementation eliminates some of the operations where possible. In preparation for training, the input records are pre-processed in software (1) to discretize floating-point fields into bins (e.g., 256 bins, including one bin for records with a missing field), (2) to one-hot encode categorical fields (i.e., the 3-category field 1 is expanded into three mutually-exclusive 'yes-no' binary *features*, where each binary feature has two bins), and (3) to include an 'absent' bin for each categorical field which may be missing in some records. For high inference accuracy, GBT accounts for missing features using the absent bins. Naive one-hot encoding results in much larger records and higher memory bandwidth demand (e.g., 40

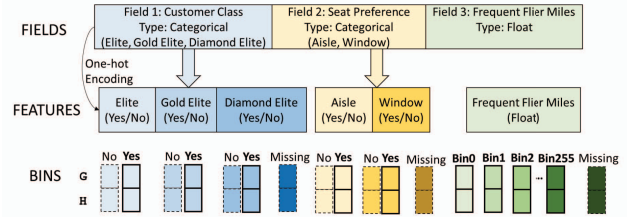

Fig. 2. Fields, features and histogram bins in training

original fields expand to 200 features). However, a key pre-processing optimization ensures that only one bin per field sees a gradient statistic update by counting only the 'yes' features and reconstructing the 'no' features by subtracting from the overall gradient statistic sum [16]. Similarly, the 'absent' bin update can also be elided. We incorporate this optimization into our GBT baseline.

The histogram-binning of the relevant records in Step ① (Table I) add a given record's gradient statistics (g_i and h_i) to the bin's summation (G and H) respectively. Step ② identifies the feature and bin within the feature that should be used as the split point for the children of the current leaf. To do so, this step evaluates every bin of every feature as a potential split point. For a given feature, the algorithm considers the bins in left to right order and starts with the split point to the left of all the bins so that the left cumulative count is zero and the right cumulative count is all the records reaching the leaf. The algorithm then successively moves the split point right by one bin and adds that bin's count, G and H to the left cumulative bucket and subtracts the bin's quantities from right cumulative bucket. Figure 3 illustrates the summation of the gradient statistics to the left and right of the split-point under consideration – the upper bin boundary of the i^{th} bin. (Though not shown, GBT considers placing records with missing fields in both the left and the right sub-trees to pick the best split option.) The quality of this split is evaluated based on the left and right cumulative buckets. The algorithm greedily picks the best split point across all fields and split points. We note that the computation to evaluate the quality of a split is often complex (i.e., hardware-unfriendly) and may vary across implementations.

Step ③ in Table I applies the newly-chosen predicate (corresponding to the split-point) to partition the relevant records (i.e., those that reach the current leaf after being filtered by the predicates in the path from the root to the leaf). The records are partitioned into predicate-true and predicate-false subsets which are used later for exploring the next vertices on the corresponding paths. An optimization in Step ① is that the histogram-binning can be performed only for the child vertex with the smaller subset and the other child vertex's (with the larger subset) bin counts, bin G and H can be calculated by simply subtracting the first child's bins from the parent vertex's bins without any explicit binning at the other child [9]. Step ④ repeats steps ① through ③ to the relevant records to complete the tree. With a fully-grown tree, Step ⑤ passes all the input records through the tree to update each record's first and second order gradient statistics (g_i and h_i) based on the new tree's prediction compared to the ground truth (i.e., the per-record loss), the loss function, and the record's previous g_i and h_i. This step also updates the new total loss across all the records for the new and old trees put together.

GBT implementations can be configured to proceed vertex

by vertex or level by level (i.e., explore together all the valid vertices at a level though some vertices may be missing). The above assumes the former, whereas the latter streams in all the input records and histogram-bins the relevant records at each vertex. Because multiple vertices are explored together, this configuration maintains a separate histogram per vertex.

In addition to the above qualitative description, we quantify the fraction of time spent in the algorithm's steps later in Section IV and Section V-A2.

B. Inference

In inference, each input record traverses all the decision trees. In a tree, each vertex's predicate is evaluated for the record to determine the next vertex. At the leaf, the tree outputs a value. All the trees' outputs are combined to compute the final prediction for the record. Unlike training, an input record for inference simply lists the category name for the corresponding categorical field to avoid the space overhead of holding one-hot-encoded features in the table or database. While the training obtains tree predicates testing one-hot encoded features, some post-processing of the trees after training (1) converts the features into the original category names for inference, and (2) encodes each field in the predicates with byte offsets of the field within the record format as field numbers do not suffice due to the variable lengths of the fields (categorical versus numerical). Further, missing fields in the input records for inference are padded with dummy values. Unlike training, inference involves only the tree traversal but is often user-facing and latency-critical (e.g., advertisement click prediction).

C. Parallelism and Access Patterns

There is enormous fine-grain parallelism in training both across the features of a record (e.g., 10s-1000s) and across different records (e.g., hundreds of millions). As mentioned in Section I, the computation is almost entirely data-access heavy (i.e., on-chip memory bandwidth-bound). Step ① in Table I has both intra- and inter-record parallelism. The histograms across the features can be updated in parallel for *intra-record* parallelism. Fortunately, the histograms can fit on-chip (e.g., 2-8 MB). However, because different features of a record often update different bins (e..g, feature 1 may update bin 8, feature 2 may update bin 0), the accesses are both fine-grained and irregular. An easy way to exploit *inter-record* parallelism is to replicate the histograms followed by a reduction across the replicas. Step ② is short as it iterates over all the features' bins which can be processed in parallel but are far fewer than the input records. Step ③'s parallelism is inter-record where the new predicate is evaluated for each record. Step ⑤'s parallelism is also inter-record where the new tree is traversed for each record. However, because only subsets of the records are likely relevant to vertices other than the root, the records' access is irregular (non-contiguous) in steps ①, ③, and ⑤. Further, Step ⑤'s tree traversal is also highly irregular because different records would follow different paths. While steps ① and ③ do not have any load imbalance, some records may see different path lengths in Step ⑤ (some paths may be terminated earlier than the pre-specified depth due to non-profitability). Fortunately, such modest load imbalance can be handled by averaging out over multiple records. Finally, steps ③ and ⑤ use only a feature and a subset of the features of the records, respectively, potentially wasting memory bandwidth.

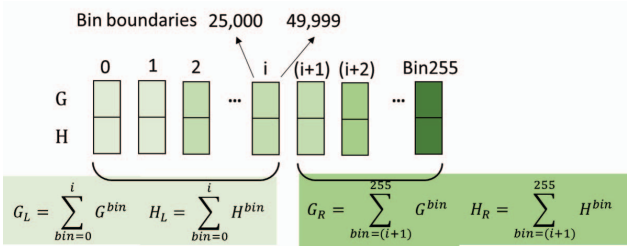

Fig. 3. Evaluating a split for `(ffmiles >=50,000)`

In inference, there are abundant intra-record parallelism (e.g., 500-1000) and inter-record parallelism (e.g., 128). However, each record takes a different path through the trees implying significant fine-grained but irregular access parallelism, like training. Further, path length variations for different records can be load-balanced by averaging across the records of the batch, like Step ⑤ in training.

D. Why are multicores, manycores, and GPUs insufficient?

Implementations of GBT on multicores, manycores, and GPUs leave significant opportunity unrealized. Multicores (and manycores, by extension) exploit GBT parallelism and cache locality in training while dealing with the irregularity. The input records are partitioned among the threads each of which has a private version of the histograms of Step ① (Table I) at the end of which the histograms are reduced. Step ③ is parallelized by partitioning the input records and replicating the current tree among the threads. The independent threads can handle the irregular accesses to the histograms (Step ①), the tree (Step ⑤), and the records (steps ①, ③, and ⑤). However, due to their modest parallelism (e.g., 32 and 80 threads) and limited on-chip cache to hold the replicated histograms, multicores and manycores exploit only a small fraction of the available parallelism (100-500 intra-record and abundant inter-record parallelism).

Similarly, in inference, multicores and manycores cannot exploit most of the inter-tree and inter-record parallelism.

In contrast, GPUs can exploit abundant fine-grained parallelism in training as long as the parallelism is regular. However, the data-dependent histogram bin updates are irregular. GPUs can even handle memory-access irregularity if the irregular-access footprint fits in special high-bandwidth structures like the GPUs' Shared Memory, which is typically limited to 48KB-96KB. Unfortunately, this option is not viable because of the read-modify-write nature of the updates. Specifically, multi-threading, fundamental for GPUs to hide memory latency, interacts poorly with read-modify-writes. The interleaving of GPU warps due to multi-threading will lead to incorrect results. Each of two possible solutions degrades performance in its own way. The first way, locking, limits multithreading by disallowing thread interleaving. The second way is to privatize the histogram state so that the multithreaded updates go to different copies. However, privatization needs as many copies of the state as the degree of multithreading, which will not fit in GPUs' Shared Memory. For example, even the smallest of our benchmarks (`Higgs`) has 28 numerical features yielding 7K bins (using 256 bins/field) of 8 bytes each – i.e., 56 KB per warp. The typical per-SM Shared Memory under 96 KB can accommodate at most one warp (i.e., no multi-threading) which would eliminate the GPU's latency tolerance. Alternatively, the GPUs would need an impractically-

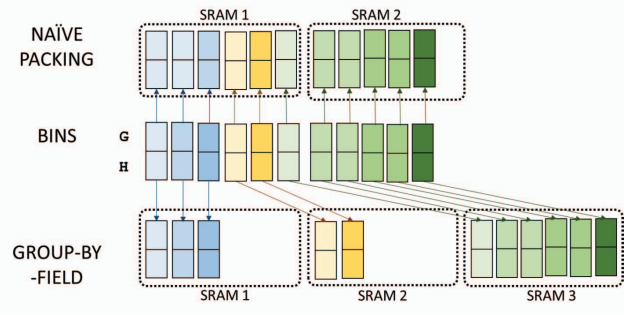

Fig. 4. Group-by-field mapping to SRAMs

large 1.75 MB of Shared Memory to accommodate 32 warps. Indeed, a recent GPU implementation achieves around 2.3x mean speedup over a 20-core multicore implementation [32], which we validate in Section V-A4.

For inference on GPUs, multithreading is possible because there are no read-modify-write problems. However, SIMT parallelism is significantly degraded as different trees (and different records) traverse different tree paths causing control flow and memory divergence. Using Shared Memory to accommodate irregular accesses requires holding as many trees (or copies) as the threads which will not fit.

III. Booster

Recall from Section I that Booster's key contributions are: (1) a *scalable sea of small SRAMs* to match GBT trees' abundant, fine-grained, irregular, small-data-structure access parallelism in both training and inference (e.g., scale to 3200 SRAMs), (2) mapping the gradient-accumulating histogram bins (Step ①) to SRAMs for categorical fields using a *group-by-field* strategy that outperforms a naive greedy-packing strategy, and (3) employing a redundant *per-field column-major format* in addition to the natural per-record row-major format to save memory bandwidth in single-predicate evaluation and one-tree traversal (steps ③ and ⑤). Booster uses the same hardware for both training and inference. We start with training.

A. Group-by-field mapping of bins in categorical fields to SRAMs for training

As discussed in Section II, the numerical fields are associated with a number of bins. Figure 4 illustrates the bins for the three fields of the frequent-flier database. The two categorical fields have three and two 'yes' bins respectively. Recall from Section II that the 'no' bin gradient summations can be reconstructed. For ease of illustration, Figure 4 assumes that FP features are discretized to six (6) bins; a real application would typically use 128-256 bins in practice.

The placement of the histogram bins in SRAMs has a significant impact on work serialization and balance in step ① . For example, if histogram bins belonging to multiple fields are placed in the same SRAM, multiple updates to those bins have to be serialized. Figure 4 illustrates the *naive packing* approach assuming an SRAM capacity of six bins. Bins are allocated to SRAMs (shown in dashed boundaries) till the SRAM is filled. Under this assignment, all three gradient updates corresponding to a record may be assigned to the first SRAM as shown in Figure 4 whereas the second SRAM may have no updates to process (load imbalance and bandwidth underutilization). However, histogram bins placed on different SRAMs can be updated concurrently. Booster's sea-of-SRAMs approach helps harness this parallelism.

We observe that there is *exactly* one update across all the bins of a field, whether numerical, categorical, or missing (recall from Section II that each field has a default 'absent' bin). Booster uses a *group-by-field mapping* wherein all the histogram bins of a categorical field are mapped to a single SRAM, utilizing near 100% SRAM and DRAM bandwidths. Figure 4 shows that all bins that are shades of blue (or amber or green) which are associated with a single categorical field are mapped to one SRAM (the mapping is needed for and applies only to categorical fields). If the number of bins associated with a single field exceed the capacity of an SRAM bank, those bins may be spread across multiple SRAMs (details in Section III-C). Our goal is to avoid bins of multiple fields sharing an SRAM bank which would lead to (avoidable) serialization.

B. Booster microarchitecture (training)

Booster's microarchitecture consists of a sea of small SRAMs each of which has a 32-bit floating-point (FP) adder for updating G and H (Figure 5). Each *Booster Unit (BU)* comprises one SRAM and the associated FP adder. Multiple BUs are organized in a cluster (e.g., 64) which may be replicated (e.g., 8). Like most GPUs and the TPU, Booster is connected to the host via PCIe and an I/O-like interface.

For the current tree leaf, the single-predicate step in the *previous* iteration of the inner step ④ loop produces a stream of pointers (record numbers) to the relevant records (all the records are relevant for the root). For the histogram-binning step (step ①), Booster fetches each relevant input record (one or more memory blocks, say 64 bytes) using the pointers, distributes each field to a BU, and broadcasts the record's g_i and h_i to each BU. The broadcast bus in Figure 5 performs a *logical* broadcast implemented as a simple, pipelined broadcast over point-to-point links (e.g., 16 BUs per link). Because the input has millions of records, this pipeline's fill and drain overheads are negligible (e.g., 3200/16 = 200 cycles). The distribution of the fields occurs in a simple, fixed left-to-right order of the fields and SRAMs. A cache block of the input record goes to a BU cluster without any broadcast. The fixed order implies that simple one-to-one buses from the fetch buffer to the BUs suffice. Each field specifies the feature number which the BU can use to index directly into the SRAM. Booster employs simple double-buffering to hide completely the fetch latency. Because the full set of pointers to the records relevant for every step is known a priori, the double-buffering removes memory latency as an issue. Further, though the relevant records would likely not be contiguous in memory, each record is one or more memory blocks of contiguous bytes, thus achieving good memory bandwidth.

For Step ①), each SRAM holds a field's histogram bin counters, G and H for each bin (Table II). Each BU simply increments its incoming bin's counter, and adds the record's G and H to the bin's G and H. The BUs are not synchronous as each BU has its own finite-state-machine control, but the BUs do not stray from each other beyond the pipelined-broadcast delay. For more parallelism, the records can be partitioned among the clusters so that each cluster generates a set of histograms which are written to Booster's memory at the end of the step after one pass over the entire dataset and reduced

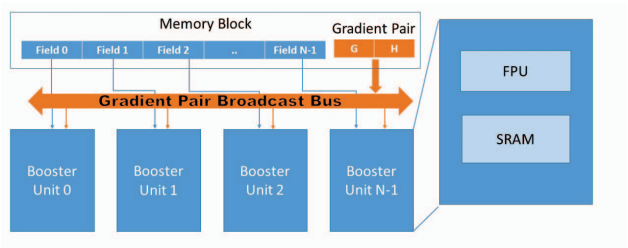

Fig. 5. Booster microarchitecture

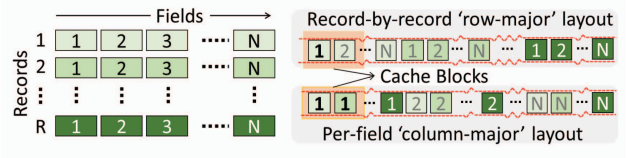

Fig. 6. Per-field column-major layout

by the host. While we use round-robin partitioning, any fine-grained partitioning that keeps the BUs busy is acceptable due to commutativity and associativity of summation. Because choosing the best split point (Step ②) (a) is a light-weight task whose execution time is proportional only to the number of bins (thousands) and not the number of records (millions), (b) is often complex (i.e., hardware-unfriendly), and (c) may vary across implementations (as noted in Section II-A), we offload the step along with the reduction at Step ① end to the host.

The best split point is expressed as a predicate used in the single-predicate step (Step ③). In this step, Booster first receives the predicate as a field number, a bin number, and a condition (e.g., `ffmiles` $\geq$ `50,000`) which may be encoded as (`Field-3` $\geq$ `upper-bin-boundary(`Bin_i`)`) as shown in Figure 2. This predicate is broadcast to all the BUs of one or more clusters (Table II). The field number enables Booster to fetch only the relevant field of each relevant record in the redundant *per-field column-major format*. Treating the input data as a table where, by default, each row is a record and each column is a field, this format redundantly stores the table in column-major layout so that each per-field column is contiguous in memory (Figure 6). As before, the relevant single-field columns are distributed to the BUs which evaluate the predicate for each record and place a pointer (record number) in either the "predicate true" or the "predicate false" buffers, which are streamed out to off-chip memory. Like the input records, the output streams are also double-buffered. Unlike the full relevant records, the relevant single-field columns would likely be more non-contiguous (e.g., in a memory block of a single-field column, only a subset may be relevant). Nevertheless, our redundant format saves significant off-chip memory bandwidth compared to the original per-record row-major format which would fetch the whole record but use only one field (Figure 6). Step ④ iterates over steps ① through ③ to grow the tree to the desired depth (or stop if the loss does not improve).

For Step ⑤ (one-tree traversal), we apply the well-known idea of mapping the newly-grown tree to a table where each entry captures a vertex by encoding its predicate (like the single-predicate step with a slight modification) and optional pointers to the vertex's children (the pointers can be elided by placing the children at a fixed offset from the vertex). The tree is broadcast to all the BUs and replicated in the SRAMs (Table II). Booster fetches the single-field columns relevant to the tree's predicates, and G and H of all the records. The modification is that instead of using the original field numbers, the predicates use a software renumbering among only the relevant fields (e.g., the original field 228 may be renumbered as the new field 7 among the fields relevant to the tree). Each BU operates on a record which traverses the tree sequentially by looking up the table in the SRAM. The new field number in the predicate indicates the field in the record. As the record exits the tree, the record's prediction is compared with the ground truth to update the record's G and H and the total loss. The records' new G and H are double-buffered and written back to off-chip memory in a stream whose efficiency motivates storing these fields separately.

Booster's implementation should choose the SRAM size to be the smallest that accommodates all the features of a field, for typical fields. SRAMs of different sizes are possible but may be hard to exploit because categorical and numerical field counts vary across workloads. The amount of parallelism – i.e., the number of BUs – should be enough so that the on-chip work is rate-matched with the off-chip memory bandwidth. For example, assuming a high-end 400 GB/s memory bandwidth (sustained average bandwidth assumed in Section IV), 1-GHz clock for Booster, and 64-byte blocks, leads to 6.25 blocks supplied every cycle. Each field consumes a byte to specify its feature (i.e., 64 fields in a block). The compute work for every field at its BU is a short integer subtract (to calculate the bin), an SRAM read, two pipelined floating-point adds, and an SRAM write, which can fit in, say, 8 cycles. The total SRAM occupancy associated with $6.25(blocks) * 64(fields/block) * 8(cycles/field) = 3200(cycles)$. Thus, 3200 SRAMs is adequate to saturate this high memory bandwidth (lower bandwidth would require linearly fewer SRAMs). Booster's high parallelism effectively ensures that it is memory bound with all computation hidden under memory latency (of record reads). Apart from the SRAMs, the double-buffering also incurs on-chip capacity. Input-side double-buffering hides Booster's latency which at worst (Step ⑤ takes 25 cycles for 5 tree levels where each level is an SRAM read and a predicate floating-point comparison) needs 25x400 B = 10 KB for all the BUs. Output-side double-buffering hides 100-cycle DRAM latency requiring 400x100 = 40KB for all the BUs, for a total of 50 KB.

C. Microarchitecture extensions (training)

We consider a few exceptional cases. (1) If there are more fields than the SRAMs, then the records are partitioned into subsets of fields with the G and H fetched for every partition. A partition of all the records is histogram-binned (Step ①) before the next. Single-predication evaluation (Step ③) and one-tree traversal (Step ⑤) are not affected by this case. (2) The partitioning strategy also applies to cases where the records are larger than the memory block. If the records are smaller than half the block then two records are packed into one block to reduce the off-chip memory bandwidth loss. (3) If a field has more features than SRAM entries, then multiple SRAMs are grouped logically to accommodate the field at a modest underutilization of SRAM bandwidth (each record would update only one of the SRAMs in a group). To maintain the one-to-one mapping between fields and SRAMs, pre-processing expands a field n times in the record if and only if the field is spread over n BUs. In the expansion, the field instance, corresponding to the BU into which the

TABLE II
DATA STRUCTURE MAPPING

Step	Mapping
①	Map a field's bins to a BU
②	Offload
③	Replicate predicate at BUs
④	Repeat steps ①-③
⑤	Replicate tree at BUs
⑥	Repeat steps ①-⑤

TABLE III
DATASET AND MODEL CHARACTERISTICS

Name	#Records (millions)	All fields	Categ.	#Features (one-hot)	Seq. Time (mins)	Comment
IoT [19]	7	115	0	115	15	Botnet attack detection
Higgs [7]	10	28	0	28	18.5	Exotic particle data
Allstate [2]	10	32	16	4232	1.6	Insurance claim predictn.
Mq2008 [23]	1	46	0	46	2.5	Supervised Ranking
Flight [1], [22]	10	8	7	666	5.5	Flight delay prediction

feature falls, is set to the *original feature modulo the maximum SRAM index*, so that the BU correctly updates its SRAM. The other field instances are set to a NOP feature so that the corresponding BUs perform a NOP (pre-processing discretizes numerical fields into 255 features instead of the usual 256 features, leaving a NOP feature).

(4) Conversely, packing multiple small fields to the same SRAM to save space reduces throughput. Each record would have to update the SRAM multiple times while other SRAMs idle. If the off-chip memory bandwidth limits the overall throughput with some spare SRAM throughput, such packing may not reduce throughput. As such, our group-by-field mapping achieves 89% capacity utilization and near 100% SRAM and DRAM bandwidth utilization for our benchmarks, which is a good trade-off for on-chip memory bandwidth-bound GBT. (5) A tree not fitting in the SRAM is unlikely given that GBT's core idea is to combine many weak models (i.e., shallow trees) into a strong model. While a larger tree can be partitioned into a group of SRAMs, we leave this case for future work.

D. Inference on Booster

Implementing inference on Booster is an extension of the one-tree traversal in training with the key difference being multiple decision trees in the former versus only one tree in the latter. In online (small batch) or offline (large batch) inference, each record traverses all the decision trees each of which is loaded as a table into a BU. Because each tree's predicates may involve tens of fields of a record (typically different sets of fields for different trees), Booster simply broadcasts (pipelined) each full record to all the BUs. At each BU, each record sequentially traverses the tree making as many SRAM accesses as the path length through the tree. Multiple records are double-buffered to hide memory latency. Each tree's output (typically a small value such as -1, 0, or 1) is buffered to be combined with the other trees' outputs for a final prediction. The outputs are also double-buffered so that the next record can start as soon as the previous record exits the tree. Thus, each tree is asynchronous with respect to the others, allowing load balancing across the trees and records, involving potentially different path lengths, by averaging over the records. Moreover, if there are too many trees to fit on one Booster chip, they can be distributed to multiple chips (round-robin). Input buffering for inference is identical to that for training Step ⑤. The output is a scalar per BU. Finally, there is *zero* inter-BU communication in training or inference. The *only* communications are the broadcasts of G and H in training, and of the records in inference. As such, (pipelined) buses suffice and a more elaborate on-chip network is likely unnecessary.

E. PCIe and broadcast overheads

As mentioned in Section III-B, Booster uses a PCIe 4.0 x16 interface whose bandwidth and *one-way* latency overhead (including software) are 32 GB/s and well under 2 μs (*round-trip* latency is 1 μs [21]). We analyze PCIe's performance impact on training, and offline and online inference, using Booster's configuration discussed so far.

Training involves copying the input records to and Step ② data from Booster. Because training handles millions of records, the PCIe latency overhead for the input and final results is negligible; as are the pipelined-broadcast overhead (Section III-B), assuming 64-B width and 200 links (3200 BUs with 16 BUs/link), for (a) the G/H pairs (2 32-bit values) in Step ① (200 cycles), (b) the predicate (6 bytes) in Step ③ (200 cycles), and (c) the tree (around 256 B) in Step ⑤ (204 cycles). Because the copied records are reused thousands of times (once per tree vertex), the PCIe bandwidth limit is immaterial. Step ② data copy latency of 2 (PCIe latency) + (3200 BUs * 2 KB SRAM)/32 GB/s (transfer) + 2 (PCIe latency for copying back the split decision) is under 210 μs per Step ④ iteration. Given one iteration per tree vertex, the total time is 210 μs * 500 (trees) * 32 (maximum number of vertices/tree) = 3.4 s versus the baseline run time of hours.

Offline inference involves copying the input records and final results. Because there are millions of records, the PCIe latency and input record broadcast overheads are negligible. Unlike training, inference uses the records only once. With Booster, however, each record (64-128 B) traverses 500 6-deep trees in parallel where each tree takes 6 *(3 SRAM read + 1 32-bit FP compare) cycles = 24 cycles. Assuming a 1-GHz clock, consuming around 100 B/record in 24 cycles needs 4.17 GB/s. Now, 3000 BUs running 6 parallel model copies need 25 GB/s which is under PCIe's 32 GB/s.

Finally, online inference also involves copying the input records and final results. For a batch of 256 records, an Intel 32-core multicore (5^{th} generation) takes around 176 μs. Thus, the PCIe latency overhead for the input and results (4 μs) and the input broadcast overhead of 256 (records) * 100 (B/record)/64 (B width) + 200 (links) = 600 cycles (0.6 μs) totaling to 4.6 μs impose an Amdahl's Law limit of 38x speedup, giving Booster room for significant speedups. The bandwidth calculations remain the same as those for offline inference.

All of these overheads are included in our experiments.

F. Realization

Booster can be realized as an FPGA or ASIC implementation while keeping in mind the usual FPGA-versus-ASIC trade-offs of power/performance penalties (ASICs are better with around 10x faster clock), time to market (FPGAs are better), and volume-dependent costs (FPGAs are less expensive

at low volumes). Our contribution is the Booster architecture, *independent of ASIC or FPGA implementation*. As such, our results show an ASIC implementation to compare Booster against high-performance CPUs and GPUs.

G. Previous GBT acceleration

One FPGA-based work [29] parallelizes GBT training only across records like a multicore by holding multiple histogram copies which degrade on-chip capacity. Booster's higher intra-record parallelism requires only one copy. A one-page paper [26] pipelines the processing but includes few other details. Another FPGA work [3] accelerates only GBT inference by exploiting three-way pipelining across records and coarse-grain inter-model parallelism (e.g., 10-way), in contrast to Booster's higher intra-record parallelism.

IV. Methodology

Datasets: We use the five datasets shown in Table III. For each dataset, we list the number of (a) training records, (b) fields per record, (c) categorical fields, and (d) features (after one-hot encoding of categorical features). For each dataset, we show sequential run times on an Intel 5^{th} generation CPU on Google Cloud to train a model of 500 trees, each with depth of upto 6.

We use datasets under 10 million records to reduce experiment time (training under 20 minutes). Full-scale datasets are larger. For instance, training 500 trees on an Intel Xeon using Terabyte Click Log's 1.7 billion records with 13 integer and 26 categorical features [30] exceeds 80 hours [31]; and a multi-modal dataset for wearables has 63 million records [28]. Unfortunately, large, commercial datasets are often not public (e.g., Facebook researchers state that the data is confidential and that "a small fraction of a day's worth of data can have many hundreds of millions of instances." [14]).

Dominant components of execution time: Figure 7 shows a breakdown of the normalized sequential training time for the steps in XGBoost (Table I). We see that Step ① (histogram binning), Step ③ (single-predicate evaluation), and Step ⑤ (one-tree traversal) constitute over 98% of run time except for `Mq2008` due to its small dataset. In contrast, Step ② (choosing the split point of a tree vertex) is short enough to be offloaded to the host.

Benchmarks other than `IoT` yield trees with leaves mostly at a depth of six. `IoT` has many shallow trees which increase Step ①'s share by processing larger fractions of records closer to the root (see Figure 7). This exception impacts `IoT`'s training and inference performance, as we show later. In the two benchmarks with categorical data (`Allstate` and `Flight`), the left versus right sub-tree split of the records at a vertex is extremely lopsided (99%-1%, for example). Because the histogram-binning step (Step ①) bins only the smaller sub-tree and sets the larger sub-tree bins to be the parent's bins minus the smaller sub-tree's bins (Section II-A), Step ①'s share is smaller for `Allstate` and `Flight` in Figure 7 despite their large datasets.

Software modifications: We verified that the software changes to XGBoost's input data format (for the redundant per-feature column representation) do not change the results.

Simulator: We build a cycle-level performance simulator for Booster. Although we model the delays of steps ①, ③, and ⑤ based on our RTL implementation, the delays are hidden entirely under memory latency, by construction

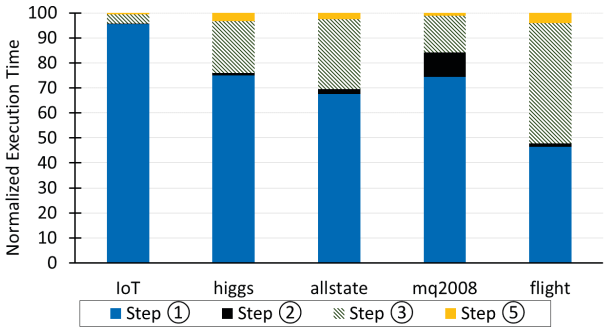

Fig. 7. XGBoost sequential training time breakdown

(Section III-B). For accurate memory latency and bandwidth simulation, we use DRAMSim2 [25] configured as a high-bandwidth, 24-channel memory, whose parameters are derived from the Hynix JESD235 standard and an Nvidia paper [8] (see Table IV). This memory achieves a sustained bandwidth varied as 200, 400 (default), 800 GB/s.

Simulated systems: Because directly comparing simulated systems and real (multicore or GPU) systems is not meaningful, we simulate *idealized* alternatives for a multicore, manycore, and a GPU called *Ideal Multicore*, *Ideal Manycore* and *Ideal GPU*, respectively (Table V). We conservatively assume that the *Ideal Multicore*, *Ideal Manycore* and *Ideal GPU* are constrained *only* by 32- and 80- 64-way parallelism *without any implementation artifacts* (based on the parallelism limits described in Section II-D). Specifically, recall that read-modify-write in the dominant Step ① in training precludes GPU warp parallelism *and* multithreading, leaving only SM parallelism as captured by *Ideal GPU*. We model *Ideal Manycore* after the company Ampere's 80-core Altra [4]. Because our *Ideal* configurations assume perfect pipelines and caches, and perfect, convergent SIMT behavior with zero kernel launch overheads, they are effectively upper-bounds on the performance of real multicores, manycores, and GPUs, respectively. All *Ideal Multicore*, *Ideal Manycore* and *Ideal GPU* use the same memory configuration as Booster (see Table IV). We do sanity-check these alternatives against a real multicore and a real GPU (we do not have access to a real manycore). Section V-A4 confirms that our *Ideal Multicore* and *Ideal GPU* configurations indeed outperform a real 32-core multicore (Intel 5^{th} generation) and a real GPU (Nvidia V100), respectively. We also compare to (a) an FPGA work [29], called *Inter-record*, which parallelizes training only across records like a multicore (Section III-G), and (b) another FPGA work, called *Inter-model* [3], which parallelizes inference only across coarse-grain models (Section III-G). For fair comparison, we simulate these architectures assuming ASIC implementations with the same area, clock speed, and memory bandwidth as Booster (i.e., the *only* difference is the architecture).

Because Booster offloads step ② to the host, we add the step's time on a real 32-core multicore host to the execution time of all the systems (Booster, *Ideal Multicore*, *Ideal Manycore* and *Ideal GPU*). We *further* add the PCIe and pipeline-broadcast overheads for the offload and input data copy in to Booster's training, and offline and online inference (Section III-E). As such, these overheads are negligible for training and offline inference where Booster's run time is in

TABLE IV
DRAM Configuration (default)

Channels, banks, row	24, 16, 1 KB
tCAS-tRP-tRCD-tRAS	12-12-12-28

TABLE V
Hardware parameters (Google Cloud hardware: [1]Intel 5^{th} generation, [2]Nvidia V100, [3]L1D Cache)

Configuration	# cores	Clock speed (GHz)	SRAM size (KB)	SRAM energy (norm.)
Real Multicore[1]	32 cores	2.2	32 KB[3]	NA
Ideal Multicore	32 cores	2.2	32 KB	1
Real GPU[2]	80 SMs	1.5	96 KB	NA
Ideal GPU	64 SMs	2.2	96 KB	2.64
Ideal Manycore	80 cores	3.3	32 KB	NA
Booster	3200 BUs	1	2KB	0.71

seconds. We include this overhead for online inference.

We model lower-bounds on energy for *Ideal Multicore* and *Ideal GPU* (in contrast to upper-bounds on performance). We conservatively model (1) the SRAM energy of each configuration's typical SRAM size (see Table V) using access activity from our simulator and per-access energy cost from CACTI 7.0 [6]. Similarly, we conservatively model the DRAM energy based on transfer activity, ignoring major energy overheads in real multicores and GPUs (e.g., multicores' superscalar cores, GPUs' large register files and heavily-banked Shared Memory) that Booster does not incur.

RTL implementation, FPGA validation and ASIC Synthesis: We implemented Booster in System Verilog and used Intel's Quartus Prime (v15.0) with Qsys system builder for synthesis. We verified our implementation's correctness via RTL simulation and tests on a scaled-down FPGA prototype (8 BUs in 1 cluster) using an Intel Cyclone IV, a modest prototyping FPGA. We synthesized an ASIC instance of Booster with 1 cluster of 64 BUs using Synopsys's Design Compiler to estimate Booster's area, power, and clock speed. We used the 45-nm technology FreePDK45 PDK/cell library [20] (newer technology nodes are unavailable). Because FreePDK45 does not include a memory compiler, we use Cacti 7.0 [6] to estimate area and power for the SRAMs.

V. Results

Our results for training show: (1) performance comparison of Booster, *Ideal GPU*, *Ideal Multicore*, and *Ideal Manycore*, (2) execution time breakdown for those schemes, and (3) the impact of Booster's components. Then, we show inference performance, followed by energy for training and inference. Finally, we show area and power results from ASIC synthesis.

A. Training performance

1) Performance: Figure 8 shows the speedup for training achieved by *Ideal GPU*, the Inter-record scheme (IR) [29] (Section III-G and Section IV), *Ideal Manycore*, and Booster over the *Ideal Multicore* baseline (Y-axis) for each of the benchmarks and the geometric mean (X-axis). We vary Booster's BU count as 1600 (Booster-1600), 3200 (Booster-3200, default), and 6400 (Booster-6400), and correspondingly the memory bandwidth as 200, 400 (default), and 800 GB/s. *Ideal Multicore*, *Ideal GPU*, *Ideal Manycore*, and IR performance do not change with memory bandwidth due to lack of parallelism (i.e., they do not saturate even 200 GB/s); hence there is only one bar each for these architectures.

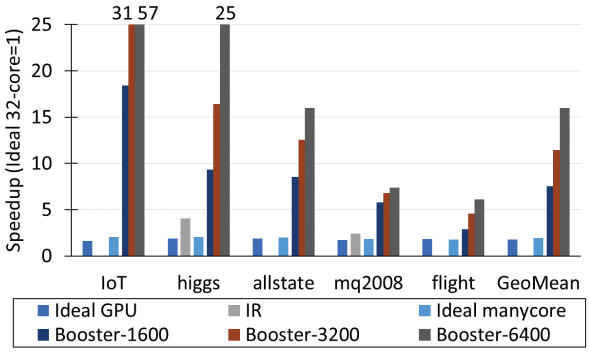

Fig. 8. Training performance comparison

Due to their similar levels of parallelism (64- and 80-way), *Ideal GPU* and *Ideal Manycore* achieve similar, modest speedups between 1.6x and 1.9x over *Ideal Multicore* across all benchmarks. We have also observed speedups on a real GPU similar to those of *Ideal GPU* (see Section V-A4). For the other benchmarks, even one copy does not fit, a case ignored by IR which shows results only for `Higgs` with FPGA-limited 64 copies. At the same clock speed as Booster, IR achieves some modest speedups over *Ideal Multicore* which essentially has 32 copies. IR's original speedup for `Higgs` is higher [29] because of (1) holding on-FPGA all 10,000 input records (zero DRAM traffic, infeasible for millions of records), and (2) weaker base cases of 12-thread CPU and Nvidia 1080TI GPU.

Booster-3200 (default) achieves speedups varying from 30.6x (for `IoT`) to 4.6x (for `Flight`), with a mean speedup of 11.4x, over the aggressive *Ideal Multicore* baseline. Because *Ideal Multicore*, *Ideal GPU*, and *Ideal Manycore* are limited only by their parallelism and not any implementation artifacts (32- and 64-way, respectively), these speedups isolate the impact of Booster's massive parallelism (3200-way). Similarly, IR's lower parallelism places IR well behind Booster. For the accelerated steps, Booster hits the memory bandwidth limit by capturing high SRAM parallelism. As such, the overall speedup is limited by bandwidth for the accelerated steps and by Amdahl's Law limits for the un-accelerated portion.

The performance trends across benchmarks are as expected; we see higher speedups on larger datasets (e.g., `IoT`, `Higgs`). Smaller datasets (e.g., `Mq2008`) or larger datasets that behave like smaller datasets (e.g., `Allstate` and `Flight`), due to the presence of categorical data and skewed data distributions (see Section IV), achieve lower speedups. Later, in Section V-B, we confirm the dependence of speedup on dataset size by scaling up the datasets. Finally, unlike the others, Booster's performance scales nearly linearly with memory bandwidth and BU count, dampened slightly by Step ②'s Amdahl's Law effect.

2) Training time breakdown: Figure 9 shows the training time breakdown (Y-axis) of the three architectures normalized to that of *Ideal Multicore* for our benchmarks (X-axis). Due to its similarity in speedups to *Ideal GPU*, *Ideal Manycore* is not shown. The execution time is broken down by the steps (shown as sub-bars) of the training algorithm (Section II-A). We observe that the speedups inversely correlate with the fraction of execution time of Step ②. Note that *Ideal Multicore*'s breakdown here is different from the *sequential* breakdown shown earlier in Figure 7. The 32-core baseline relatively increases Step ②'s fraction of time. *Ideal GPU* offers a modest

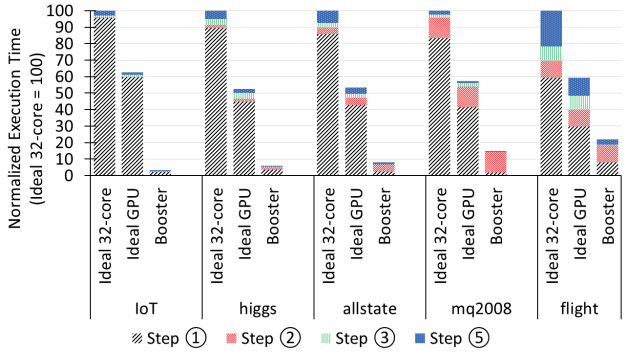

Fig. 9. Booster's training time breakdown (3200 BUs)

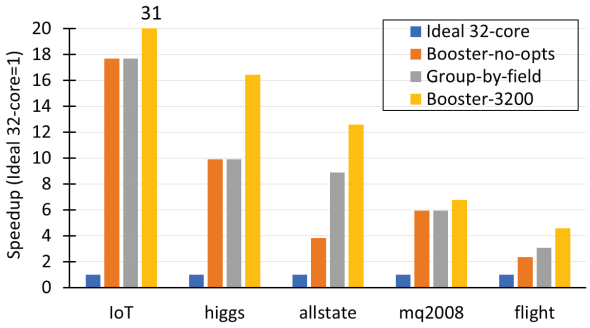

Fig. 10. Isolating Booster's training optimizations

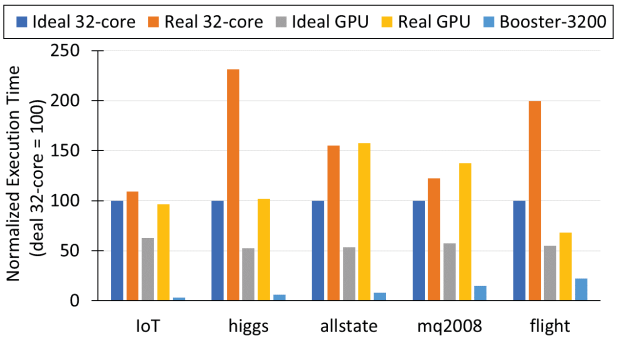

Fig. 11. Comparing *Ideal* and *Real* configurations for training

reduction in the three accelerated steps. Even on the GPU, Step ② does not see much improvement. In contrast, Booster-3200 makes all the accelerated steps vanishingly small so that the unaccelerated Step ② dominates the residual execution time.

For the high-opportunity benchmarks, *Ideal GPU* cuts the execution time in half. Step ② sees no speedup even on *Ideal GPU*. As such, the execution time for `Flight`, which has a significant Step ② fraction, is reduced only by 78%. The breakdown for Booster is barely observable at the given scale because of the high speedups that apply to steps ①, ③, and ⑤. Step ② (orange subbar), which is the un-accelerated portion, remains unchanged. Because the un-accelerated portion is approximately 3.5%, the maximum achievable speedup is limited to 28.8x. Booster achieves 11.4x speedup.

3) Isolating Booster's training optimizations: Figure 10 isolates the training speedup contribution of Booster-3200's components (Y-axis, *Ideal Multicore* = 1) for our benchmarks (X-axis). The speedups are over *Ideal Multicore* (first bar). The second bar *Booster-no-opts* uses naive packing to map bins to SRAMs (Section III-A) and no optimizations other than exploiting BU parallelism. The next bar shows the speedup due to group-by-field mapping (Section III-A) which shows improvements for the two benchmarks with categorical fields (`Allstate` and `Flight`). Recall that the mapping is designed to improve datasets with categorical fields. For benchmarks without any categorical fields, naive packing achieves the same effect as the mapping because our SRAMs are sized to accommodate numerical fields. Finally, the third bar shows the contribution of our redundant per-field, column-major representation which helps accelerate steps ③ and ⑤. This optimization works best for benchmarks with high speedups. Effectively, speeding up Step ① makes steps ③ and ⑤ take larger fractions of the residual execution time. As such, the impact of any optimization that targets steps ③ and ⑤ is magnified.

The redundant representation, a software-only optimization, may be applied to steps ③ and ⑤ in *Ideal GPU* and *Ideal Multicore*. However, *Ideal Multicore* and *Ideal GPU* see little benefits (under 4%) because (1) Step ① dominates the execution time for *Ideal Multicore* and *Ideal GPU* (see Figure 7), (2) Step ⑤ is compute-bound on *Ideal Multicore* and *Ideal GPU*, and (3) Step ③ does improve, but with minimal overall impact (Amdahl's Law effect).

4) Validating the Ideal *models:* Figure 11 compares the training times on real hardware (32-core multicore and GPU) to those of *Ideal Multicore* and *Ideal GPU* for all our benchmarks (X-axis). The last bar shows the execution time of Booster, included for viewing convenience. We use an Intel 5^{th} generation processor for the multicore and an Nvidia V100 for the GPU; both on Google Cloud. The execution times are normalized (Y-axis) to that of *Ideal Multicore* (first bar), the same baseline used in all our speedup measurements. Figure 11 confirms two important properties. First, the *Ideal Multicore* execution time is always lower than that of the real 32-core system. Similarly, *Ideal GPU* is always better-performing than the real GPU.

These results confirm that Booster's speedups are in comparison to aggressive, ideal baselines. Second, while *Ideal GPU* is always better-performing than *Ideal Multicore*, the GPU performs worse than the multicore for two of the five benchmarks (`Allstate` and `Mq2008`). This result confirms that the workload irregularity limits GPU performance, strengthening the case for an accelerator. While we compare to a V100, the A100's matrix-multiply tensor units would not help GBT. The A100 has 108 SMs (maximum clock under 1.5 GHz) and is performance-bound by 80-core *Ideal Manycore* (3.3 GHz) over which Booster is faster (Figure 8).

B. Sensitivity to training dataset size

Given that commercial datasets are larger (Section IV) and that data is growing faster than Moore's law [13], it is important to understand how Booster's performance is affected by growing dataset sizes. To that end, we performed experiments by scaling up (replicating) data sizes by a factor of 10. Figure 12 shows the performance comparison of *Ideal Multicore*, *Ideal GPU*, and Booster for the scaled-up benchmarks (X-axis). The speedups of *Ideal GPU* over *Ideal Multicore* (Y-axis) remain modest (< 2x) and similar to the speedups with the unscaled datasets. In contrast, Booster's speedups are dramatically higher for the larger datasets. Compared with the unscaled datasets (Figure 8), the speedup range improves from

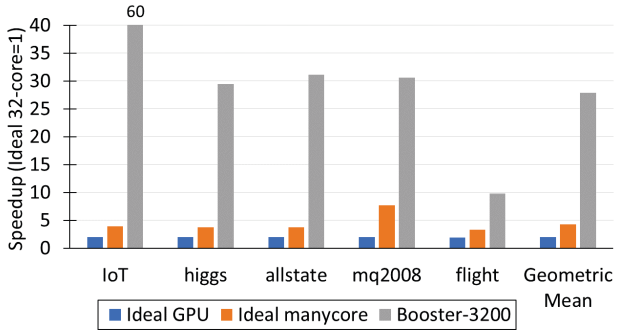

Fig. 12. Sensitivity to training dataset size

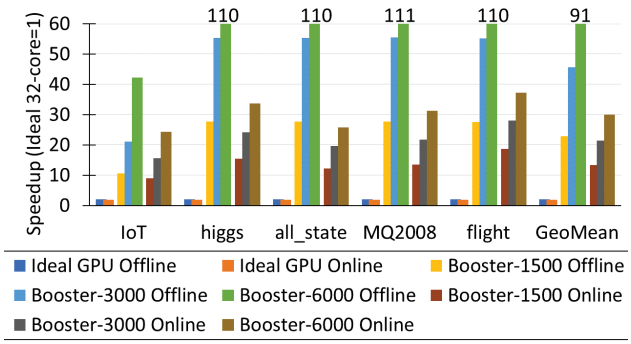

Fig. 13. Booster's performance for Inference

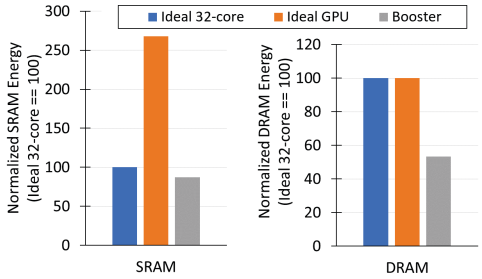

Fig. 14. Booster's energy savings (a) SRAM (training with 3200 BUs and inference with 3000 BUs) and (b) DRAM (training with 3200 BUs)

4.6-30.6 to 9.8-61.5 with the scaled datasets; the geometric mean speedup increases from 11.4 to 27.9. *Every* benchmark achieves higher speedup with the scaled datasets than with the unscaled ones (Figure 8); even `Flight`, the benchmark with the lowest speedup improves from 4.6 to 9.8 speedup. This result highlights Booster's strength that its performance will improve as data volume grows.

C. Inference performance

For inference, each record traverses the model's 500 trees. Although we did not implement inference in RTL, we did implement the key building block – one-tree traversal – in our design. We use the latency numbers from the one-tree traversal model to simulate batch inference. Our simulation uses 3000 of the 3200 BUs to create 6 replicas of the 500 trees to improve throughput. Figure 13 shows the speedups for *Ideal GPU* and Booster over *Ideal Multicore* (Y-axis) for inference with a batch of 1 million (offline) and 256 (online) records for each benchmark (X-axis). Like Figure 8, we vary Booster's BU count as 1500 (Booster-1500), 3000 (Booster-3000, default), and 6000 (Booster-6000), and correspondingly the memory bandwidth as 200, 400 (default), and 800 GB/s (PCIe 64 GB/s for Booster-6000). Like in Figure 8, *Ideal Multicore* and *Ideal GPU* performance do not change with memory bandwidth due to lack of parallelism. Unlike Figure 8, we do not compare to IR [29] which does not discuss inference. We found that *Intermodel* [3] (Section III-G and Section IV) performs on average 66% worse than *Ideal Multicore* due to insufficient parallelism (invisible, not shown).

Ideal GPU's low speedups are due to SIMT divergence in inference (Section II-D). Though our results show that Booster-3000 (default) achieves 45x mean speedup in offline inference, the results are better understood as two distinct clustered behaviors. Four of the five benchmarks with deep trees (as described in Section IV) behave similarly with 55.5x speedup over over *Ideal Multicore*. The outlier (`IoT`) achieves 21.1x speedup because of its shallow trees. Booster's absolute performance is unaffected as its performance depends on the *maximum* depth across all trees, which is usually 6. In contrast, *Ideal Multicore*'s absolute performance improves for shallow trees (Section IV) due to reduced overall work which leads to Booster's speedup being lower for `IoT`. Interestingly, `IoT`'s shallow trees cause higher speedups for Booster in training (Figure 8), as opposed to inference. Shallow trees have less left-right sub-tree work-pruning in the histogram-binning Step ①, so that Step ① dominates the other steps, as explained in Section IV. Booster parallelizes this dominant step to achieve high speedups in training.

Finally, online inference speedups, while substantial (21x on average for Booster-3000), are lower than offline inference speedups due to the PCIe overhead for input copy being relatively higher for the smaller batch (Section IV). Like in Figure 8, Booster's performance scales linearly with BU count and memory bandwidth for offline inference, while being throttled by the PCIe overhead for online inference.

D. Training and inference energy

We compare access energy in training and inference for SRAM and DRAM separately as we cannot measure the relative proportions of the two components. However, because Booster is strictly better in both SRAM energy *and* DRAM energy, Booster is guaranteed to achieve lower total energy regardless of the specific ratio of SRAM to DRAM energy.

Figure 14 compares the normalized SRAM (Figure 14(a)) and DRAM (Figure 14(b)) energy across *Ideal Multicore*, *Ideal GPU*, and Booster-3200. The per-access SRAM energy, and hence the normalized SRAM energy, in each architecture is the same for training and inference. The DRAM energy is the same for inference in all three architectures but different only for training where the per-field column-major format (Section III-B) is applicable (not to inference). The results are averaged across all the benchmarks and normalized to that of *Ideal Multicore*. *Ideal GPU*'s 32-banked, 96-KB Shared Memory incurs more SRAM energy than *Ideal Multicore*'s 32-KB L1 D-cache. In contrast, Booster accesses a smaller 2-KB SRAM, resulting in lower SRAM energy. For DRAM energy, *Ideal Multicore* and *Ideal GPU* are identical as they access the same set of blocks. In contrast, Booster has fewer DRAM transfers due to the optimized format.

E. ASIC synthesis results

Table VI shows the area and power estimates for a 50-cluster Booster-3200 chip with 64 BUs per cluster (using 40 nm technology). Our synthesis achieves 1-GHz clock speed.

TABLE VI
Area and power estimates for Booster

Component	Area (mm^2)	Power (W)
Control Logic	8.4	4.3
FPU	18.4	9.5
SRAM	33.1	9.4
Total	60.0	23.2

Almost half (55%) of Booster's area goes to the SRAMs and 31% to the 32-bit floating-point units (FPUs). Although the aggregate SRAM capacity is only 6.4 MB ($= 3200 * 2$KB), the area is 70% larger than that of a 1-bank 6.4-MB SRAM array as Booster uses the equivalent of 3200 banks for parallel access. The area would be much smaller for a recent technology node which is not available in FreePDK. Also, the SRAM capacity scales linearly with the off-chip memory bandwidth (Section III-B). This 3200-bank, 6.4-MB SRAM is needed to match the aggressive, sustained (not peak) bandwidth of 400 GB/s achieved by recent technology nodes. Because the static power dominates the dynamic power, the total power of Booster's SRAM is only around 59% higher than that of the 1-bank case despite the huge difference in the access parallelism. While the SRAM area is larger than the FPUs', the SRAM and FPUs dissipate nearly equal power, each accounting for around 41% of the total power.

VI. Conclusion

We proposed an accelerator, called Booster, for gradient boosting trees (GBT) training and inference based on the observation that (a) GBT training is dominated by three key computational steps – histogram binning, single-predicate evaluation, and one tree traversal – that are performed on a large number of records, and (b) GBT inference involves traversing hundreds of shallow trees. While there is abundant inter- and intra-record parallelism, traditional multicores and GPUs are unable to parallelize the memory accesses which are irregular, data-dependent, and to small data-structures (i.e., GBT is on-chip memory bandwidth-bound). Booster employs (1) a *scalable sea-of-small-SRAMs* approach to harness large-scale parallelism in training and inference (e.g. 3200-way), and (2) a bandwidth-preserving mapping of data fields to SRAMs called *group-by-field mapping*, resulting in significantly high parallelism. Our simulations show that Booster achieves 11.4x and 6.4x speedups for training, and 45x and 22x (21x and 11x) speedups for offline (online) inference, over an ideal 32-core multicore and an ideal GPU, respectively. Based on ASIC synthesis of FPGA-validated RTL using 45 nm technology, we estimate that a Booster chip would occupy 60 mm^2 of area and dissipate 23 W when operating at a 1-GHz clock speed. As such, Booster would be an attractive option for accelerating the highly-successful GBT models.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive feedback and suggestions for this paper. The authors also deeply thank Google Cloud Platform for generously providing free credits for the project.

References

[1] " Airline on-time performance:," http://stat-computing.org/dataexpo/2009/, 2009.
[2] " Allstate claim data:," https://www.kaggle.com/c/ClaimPredictionChallenge/, 2013.
[3] A. Alcolea and J. Resano, "Fpga accelerator for gradient boosting decision trees," *Electronics*, vol. 10, no. 3, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/3/314
[4] Ampere, "Ampere Altra," https://amperecomputing.com/ampere-altra-industrys-first-80-core-server-processor-unveiled/.
[5] "AWesome XGBoost," https://github.com/dmlc/xgboost/tree/master/demo.
[6] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, Jun. 2017.
[7] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.
[8] N. Chatterjee *et al.*, "Architecting an energy-efficient dram system for gpus," in *Proc. of HPCA*, 2017, pp. 73–84.
[9] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794.
[10] Y. Chen *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.
[11] A. V. Dorogush *et al.*, "CatBoost: gradient boosting with categorical features support," in *Workshop on ML Systems at NIPS*, 2017.
[12] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.
[13] J. Gantz and D. Reinsel, "Extracting value from chaos," *IDC IView*, pp. 1–12, 01 2011.
[14] X. He *et al.*, "Practical lessons from predicting clicks on ads at facebook," in *Proc. of the 8th Intl. Workshop on Data Mining for Online Advertising*, ser. ADKDD'14, 2014, p. 1–9.
[15] "HG Insights," (https://discovery.hgdata.com/product/xgboost).
[16] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *NeurIPS*, 2017, pp. 3146–3154.
[17] T. Keck, "Fastbdt: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification," *arXiv preprint arXiv:1609.06119*, 2016.
[18] X. Ling *et al.*, "Model ensemble for click prediction in bing search ads," in *Intl. Conf. on World Wide Web Companion*, 2017, p. 689–698.
[19] Y. Meidan *et al.*, "N-baIoT—network-based detection of IoT botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.
[20] NCSU, "Freepdk45," https://www.eda.ncsu.edu/wiki/FreePDK45:Contents#Current_Version.
[21] R. Neugebauer *et al.*, "Understanding pcie performance for end host networking," in *Proc. of the 2018 SIGCOMM*, 2018, p. 327–341.
[22] S. Pafka, " Flight delay data :," https://github.com/szilard/benchm-ml#data, 2016.
[23] T. Qin and T.-Y. Liu, "Introducing letor 4.0 datasets," *arXiv preprint arXiv:1306.2597*, 2013.
[24] W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, 09 2017.
[25] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comp. Arch. Letters*, vol. 10, no. 1, pp. 16–19, 2011.
[26] T. Sadasue and T. Isshiki, "Scalable full hardware logic architecture for gradient boosted tree training," in *2020 IEEE 28th FCCM*, 2020, pp. 234–234.
[27] R. E. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. Cambridge, MA: The MIT Press, 2012.
[28] P. Schmidt *et al.*, "Introducing wesad, a multimodal dataset for wearable stress and affect detection," in *Proc. Intl. Conf. on Multimodal Interaction*, ser. ICMI '18, 2018, p. 400–408.
[29] T. Tanaka, R. Kasahara, and D. Kobayashi, "Efficient logic architecture in training gradient boosting decision tree for high-performance and edge computing," *arXiv preprint arXiv:1812.08295*, 2018.
[30] "Terabyte Click Log," (https://labs.criteo.com/2013/12/download-terabyte-click-logs-2/).
[31] "LightGBM Experiments," (https://lightgbm.readthedocs.io/en/latest/Experiments.html).
[32] Z. Wen *et al.*, "Exploiting gpus for efficient gradient boosting decision tree training," *IEEE TPDS*, vol. 30, no. 12, pp. 2706–2717, Dec 2019.
[33] "Wikipedia," (https://en.wikipedia.org/wiki/XGBoost).