Bonsai: High-Performance Adaptive Merge Tree Sorting

Nikola Samardzic*, Weikang Qiao*, Vaibhav Aggarwal, Mau-Chung Frank Chang, Jason Cong
University of California, Los Angeles Los Angeles CA, USA
{nikola.s, wkqiao2015, vaibhav.a}@ucla.edu, mfchang@ee.ucla.edu, cong@cs.ucla.edu

***Abstract*—Sorting is a key computational kernel in many big data applications. Most sorting implementations focus on a specific input size, record width, and hardware configuration. This has created a wide array of sorters that are optimized only to a narrow application domain.**

In this work we show that merge trees can be implemented on FPGAs to offer state-of-the-art performance over many problem sizes. We introduce a novel merge tree architecture and develop Bonsai, an adaptive sorting solution that takes into consideration the off-chip memory bandwidth and the amount of on-chip resources to optimize sorting time. FPGA programmability allows us to leverage Bonsai to quickly implement the optimal merge tree configuration for any problem size and memory hierarchy.

Using Bonsai, we develop a state-of-the-art sorter which specifically targets DRAM-scale sorting on AWS EC2 F1 instances. For 4-32 GB array size, our implementation has a minimum of 2.3x, 1.3x, 1.2x and up to 2.5x, 3.7x, 1.3x speedup over the best designs on CPUs, FPGAs, and GPUs, respectively. Our design exhibits 3.3x better bandwidth-efficiency compared to the best previous sorting implementations. Finally, we demonstrate that Bonsai can tune our design over *a wide range of problem sizes* (megabyte to terabyte) *and memory hierarchies* including DDR DRAMs, high-bandwidth memories (HBMs) and solid-state disks (SSDs).

***Index Terms*—merge sort, performance modeling, memory hierarchy, FPGA**

I. Introduction

There is a growing interest in FPGA-based accelerators for big data applications, such as general data analytics [1]–[3], genomic analysis [4], compression [5] and machine learning [6]–[11]. In this paper we focus on sorting given its importance in many data center applications. For example, MapReduce keys coming out of the mapping stage must be sorted prior to being fed into the reduce stage [12]. Thus, the throughput of the sorting procedure limits the throughput of the whole MapReduce process. Large-scale sorting is also needed to run relational databases; the sort-merge join algorithm has been the focus of many research groups, with sorting as its main computational kernel [13], [14].

Data processing systems like Hive [15], Spark SQL [16], and Map-Reduce-Merge [17] implement relational functions on top of Spark and MapReduce; sorting is a known bottleneck for many relational operations on these systems.

CPUs offer a convenient, general-purpose sorting platform. However, implementations on a single CPU are shown to be many times inferior to those on GPUs or FPGAs (e.g., [18], [19]). For example, PARADIS [20], the state-of-the-art CPU sorter, works at <4 GB/s for inputs over 512 MB in size. Additionally, the CPU architecture is specialized to work with 32/64-bit data and sorting wide records usually leads to much lower performance, as CPU has no efficient support for gathering 32- or 64-bit portions of large keys together [21], [22].

Regarding GPU-based sorting, hybrid radix sort (HRS) [18] offers state-of-the-art performance for arrays up to 64 GB in size, with the best reported throughput of over 20 GB/s for 2 GB arrays. In general, GPU sorters perform well (>5 GB/s) only when the entire array fits into the GPU's memory (4-8 GB) [23]–[26]. HRS is able to sort up to 64 GB of data by using the CPU to merge smaller subsets of the input data that are initially sorted on the GPU. Nonetheless, this CPU-side merging dominates the computation time for large enough arrays: for 32 GB arrays, GPU-based sorters spend the majority of their compute time on the CPU. Most GPU sorters also focus exclusively on 32- or 64-bit wide records, which are more amenable to GPU's parallel instructions. As many workloads require wider records [27] [28], this restriction is often unacceptable.

The two best performing FPGA sorters are SampleSort [19] and Terabyte sort [29]. SampleSort sorts up to 14 GB of data at 4.44 GB/s, offering the best FPGA-accelerated sorting performance after our work. However, SampleSort relies on the CPU for sampling and bucketing, which limits scalability: indeed, for arrays over 16 GB, the performance drops 3x. Terabyte Sort [29] implements a merge tree-based FPGA sorting system that can sort up to 1 TB of data. However, our analysis shows that their design misses many optimization opportunities and does not perform well on smaller-scale sorting tasks (§IV-C). Other FPGA sorters focus exclusively on arrays that can fit in on-chip memory (<1 MB) [30]–[35] and will be discussed in §VII-A.

Table I shows the best known solutions for different problem sizes use both different algorithms and different hardware platforms, with none of them performing well across all problem sizes. In fact, most solutions are not able to sort at all outside a preset range of input sizes (dashes in Table I indicate no reported result). This implies that data center engineers need to procure multiple different hardware platforms and familiarize themselves with many different sorting approaches in order to achieve good performance across all problem sizes. Further, almost all sorters consider only a small range of record widths.

*indicates co-first authors and equal contribution for this work

TABLE I: Sorting time in ms per GB (lower is better). Best sorters for CPU, GPU, and FPGA across different problem sizes compared to Bonsai. Performance of distributed sorters [36], [37] multiplied by number of server nodes used. Dashes ('-') indicate no reported result.

	4 GB	8 GB	16 GB	32 GB	64 GB	128 GB	512 GB	2 TB	100 TB
CPU PARADIS [20]	436	436	395	388	363	-	-	-	-
CPU distributed [36]	-	-	-	-	-	508	508	508	466
GPU HRS [18]	208	208	208	224	260	267	-	-	-
GPU distributed [37]	-	-	-	-	-	-	2,909	3,368	-
FPGA SampleSort [19]	215	217	220	643	-	-	-	-	-
FPGA TerabyteSort [29]	-	-	-	-	3,401	4,366	4,347	4,347	6,210
Bonsai	**172**	**172**	**172**	**172**	**172**	**250**	**250**	**250**	**375**

To overcome these problems, we present Bonsai, a merge tree-based *adaptive* sorting solution which can tune our merge tree architecture to sort a wide range of array sizes (megabytes to terabytes) on a single CPU-FPGA server node. It considers the scalability of sorting kernels with respect to not only the problem size, but also available computational resources, memory sizes, memory bandwidths, and record width. Bonsai uses a set of analytical performance and resource models in order to configure our adaptive merge tree architecture to any combination of available hardware and problem sizes. Furthermore, our general approach helps computer architects better understand what performance benefits future compute and memory technology may bring, as well as how these improvements can best be integrated with our merge tree sorter.

In particular, we explore the unique reconfigurability of the FPGA, which allows Bonsai to adapt the sorting kernel to meet changing sorting demands within hundreds of milliseconds [38]. The problem size, record width, latency vs. throughput trade-offs, and available memory and compute hardware are all considered when optimizing our merge tree configurations.

We choose to base our design on the merge sort algorithm [39] as it has both the asymptotically optimal number of operations and predictable, sequential memory access patterns (ideal for memory burst and coalescing). Furthermore, due to its asymptotically optimal I/O complexity [40], merge sort is generally regarded as the preferred technique for sorting large amounts of data within a single computational node [25].

Alternatively, bucketing algorithms like sample or radix sort have been effectively used in many data center-scale distributed sorters, but are only useful when the ratio of problem size to the number of processors is relatively low [41]. In comparison, our implementation exhibits much better per-node performance on terabyte-scale problems than any such distributed sorting system (Table I).

Our contributions include:

- A novel *adaptive merge tree* (AMT) architecture, which allows for the independent optimization of merge tree throughput and the number of concurrently merged arrays (§II). Our architecture can be *optimally adapted* to the available on-chip resources and off-chip memory bandwidth. We also introduce new features necessary for high performance of merge trees (§V).
- We develop Bonsai, a complete and adaptive sorting solution which uses a set of comprehensive models to tune the AMT configuration to reduce sorting time (§III). We explain how the models can be used to optimize our design for different memory systems, such as DRAM, high-bandwidth memory (HBM), and flash memory (§IV).
- We use Bonsai to develop a state-of-the-art complete sorting system for DRAM-scale sorting on AWS F1. Our design has a minimum of 2.3x, 1.3x, 1.2x and up to 2.5x, 3.7x, 1.3x speedup over the best sorters on CPUs, FPGAs, and GPUs, respectively and exhibits 3.3x better bandwidth-efficiency compared to all previous sorting implementations.
- We further demonstrate the versatility of our approach by using Bonsai to create an SSD sorter whose projected performance is 17.4x better than any other single-node sorter [29] and 2x better per-node latency than any distributed terabyte-scale sorting implementation [37].

A. Hardware Mergers

Hardware mergers are the basic building blocks of merge trees. We call a k-merger a hardware merger that can merge two sorted input streams at a rate of k records per cycle. The k-merger is designed to expect k-record tuples at its two input ports and outputs one k-record tuple each cycle. We use some combination of 1-, 2-, 4-, 8-, 16-, and 32-mergers in our AMT designs, although using even bigger mergers is also possible.

In order to output k records per cycle, mergers use a pipeline of two $2k$-record *bitonic half-mergers* [42]. A $2k$-record bitonic half-merger is a fully-pipelined network that merges two k-record sorted arrays per cycle [43]. The network is made up of $\log k$ steps. In each step, k compare-and-exchange operations are executed in parallel on different pairs of records. Thus, the bitonic half-merger merges with latency $\log k$ and requires $k \log k$ logic units.

The logic required by the $2k$-merger is asymptotically dominated by the two bitonic half-mergers. Thus, the logic utilization of a $2k$-merger is $\Theta(k \log k)$. The mergers consume no on-chip block memory other than registers.

Note that Bonsai and the AMT architecture can use any underlying merger design as a building block, and are not limited to our specific merger implementation.

B. Problem Formulation

In this paper, we focus our attention on optimizing sorting on a single FPGA for input size ranging from megabytes to terabytes of data. Nonetheless, our design can also be used as a building block for a larger distributed sorting system.

TABLE II: Bonsai input parameters.

Symbol	Definition
N	Number of records in array
r	Record width in bytes

(a) Array parameters.

Symbol	Definition
β_{DRAM}	Bandwidth of off-chip memory
$\beta_{\text{I/O}}$	Bandwidth of I/O bus
C_{DRAM}	Off-chip memory capacity in bytes
C_{BRAM}	On-chip memory capacity in bytes
C_{LUT}	Number of on-chip logic units
b	Size of read batches in bytes

(b) Hardware parameters.

Symbol	Definition
f	Merger frequency
m_k	Logic utilization of a k-merger
c_k	Logic utilization of a k-coupler

(c) Merger architecture parameters.

Bonsai's goal is to minimize sorting time by optimizing the choice of adaptive merge trees (AMTs) to the available hardware, merger architecture, and input size (Table II).

AMTs are constructed by connecting mergers and couplers together (details in §II). We view mergers/couplers as black box components and do not optimize their design with Bonsai. Thus, the resource utilization and frequency of mergers/couplers are treated as input parameters to our model.

We present how mergers and couplers can be connected to create various AMTs in §II. We explain how AMTs can be configured to minimize sorting time (§III-A). We introduce a resource utilization model that we use to understand if an AMT configuration is implementable on available hardware (§III-B). Finally, we present the Bonsai-optimal AMT configurations for three different types of off-chip memories and synthesize the respective configurations on an FPGA (§IV). We report performance of the DRAM sorter and experimentally validate Bonsai's predictions in §VI.

II. AMT Architecture Framework

Adaptive merge trees (AMTs) are constructed by connecting mergers into a complete binary tree. AMTs are uniquely defined by their throughput and number of leaves. The throughput of an AMT is the number of merged elements it outputs per cycle out of the tree root (we call this value p). The number of leaves of an AMT, denoted ℓ, represents the number of sorted arrays the AMT concurrently merges; ℓ is important because it determines the number of recursive passes the data needs to make through the AMT.

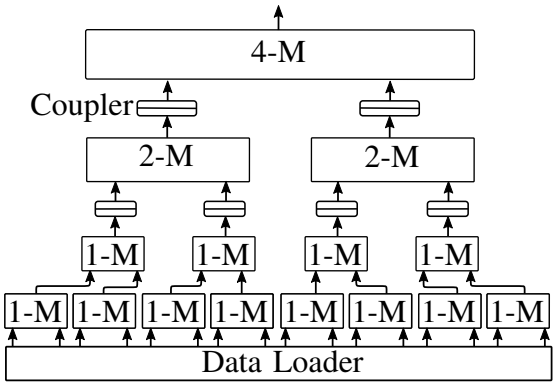

Fig. 1: Example architecture of an AMT with throughput $p = 4$ and number of leaves $\ell = 16$. 1-M, 2-M, and 4-M represent 1-, 2-, and 4-mergers, respectively.

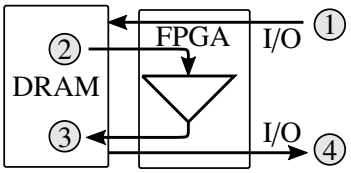

Fig. 2: Illustration of single AMT tree configuration and its associated data movement flow. The triangle represents the AMT tree.

Our AMT architecture allows for any combination of p and ℓ to be implemented as long as there are sufficient on-chip resources. To implement a p and ℓ AMT, we put a p-merger at the root of the AMT, two $p/2$-mergers as its children, then four $p/4$-mergers as their children, etc., until the binary tree has $\log_2 \ell$ levels and can thus merge ℓ arrays. In general, the tree nodes at the k-th level are $p/2^k$-mergers. If for a given level k, we have $2^k > p$, we use 1-mergers. For example, for the AMT with throughput $p = 4$ and 16 leaves ($\ell = 16$), we would use a 4-merger at the root, two 2-mergers as the root's children, four 1-mergers as the root's grandchildren, and eight 1-mergers as the root's great-grandchildren (Figure 1).

In order to feed the output of a $p/2$-merger to the parent p-merger, a *p-coupler* is used between tree levels to concatenate adjacent $p/2$-element tuples into p-element tuples suitable for input into the parent p-merger (Figure 1).

Merge sort is run by recursively merging arrays using AMTs. The array is first loaded onto DRAM (step 1, Figure 2) via an I/O bus (either PCI-e from the host or SSD, or an Ethernet port from another FPGA or host). Then we stream the data through the AMT, which merges the input elements into sorted subsequences (steps 2-3). Steps 2-3 are then recursively repeated until the entire input is merged into a single sorted array. We call each such recursive merge a *stage*. Once the data is sorted, it is output back via the I/O bus (step 4).

During the first stage, the AMT merges unsorted input data from DRAM and outputs ℓ-element sorted subsequences back onto DRAM. In the second stage, the ℓ-element sorted subsequences are loaded back into the AMT, which in turn merges ℓ different ℓ-element sorted subsequences; thus, the output of the second stage are ℓ^2-element sorted subsequences. In general, the k-th stage will produce ℓ^k-element sorted subsequences. Therefore, the total number of merge stages required to sort an N-element array is $\lceil \log_\ell N \rceil$.

As recognized in [29] and [44], merging more arrays (i.e., increasing ℓ) reduces the total number of merge stages required to sort an array, thereby reducing the sorting time. On the other hand, using an AMT with higher throughput (i.e., increasing p) reduces the execution time of each stage. Thus, there is

TABLE III: AMT configuration parameters.

Symbol	Definition
p	Number of records output per cycle by a merge tree
ℓ	Number of input arrays of a merge tree
λ_{unrl}	Number of unrolled merge trees
λ_{pipe}	Number of pipelined merge trees

a natural trade-off between p and ℓ, as increasing either of them requires using additional limited on-chip resources. The Bonsai model shows that different choices of p and ℓ are optimal for different problem sizes, as described in §III, and §IV.

Our AMT architecture can be configured to work with any key and value width up to 512 bits without any resource utilization overhead or performance degradation; if necessary, even wider records can be implemented by using bit-serial comparators in the mergers [45].

In order to read/write from/to off-chip memory at peak bandwidth, reads and writes must be batched into 1-4 KB chunks. The *data loader* implements batched reads and writes, thereby abstracting off-chip memory access to the AMT. The data loader may consume considerable amounts of on-chip memory, as it needs to store ℓ pre-fetched batches on-chip. Nonetheless, it allows us to utilize the full bandwidth of off-chip memory. Further microarchitecture details are presented in §V.

III. AMT Architecture Extensions and Performance Modeling

In this section, we introduce AMT configurations and explain how different configurations impact performance (§III-A) and resource utilization (§III-B), which in turn help to create Bonsai, an optimizer that finds the optimal AMT configuration parameters (Table III) given the input parameters (Table II). Bonsai is introduced in §III-C.

A. AMT Configurations

An *AMT configuration* (summarized in Table III) is defined by specifying: the AMT throughput p, the AMT leaf count ℓ, the amount of AMT unrolling λ_{unrl} (§III-A2), and the amount of AMT pipelining λ_{pipe} (§III-A3).

Each AMT is uniquely defined by its throughput (p) and leaf count (ℓ), which we denote as $AMT(p, \ell)$. In order to ease implementation, we use the same p and ℓ values for all AMTs within a configuration. A λ_{unrl}-*unrolled* configuration means λ_{unrl} AMTs are implemented to work independently in parallel. Conversely, a λ_{pipe}-*pipelined* configuration implies ordering λ_{pipe} AMTs in a sequence so that the output of one AMT is used as input to the next AMT. We allow for both unrolling and pipelining to be used by replicating a λ_{pipe}-pipelined configuration λ_{unrl} times (§III-A4).

1) Optimizing single-AMT configurations: In this section, we model the performance of $AMT(p,\ell)$.

As discussed in §II, the total number of merge stages required to sort an N-element array is $\lceil \log_\ell N \rceil$. The amount of time required to complete each stage depends on the throughput of the AMT ($= pfr$) and off-chip memory bandwidth, denoted β_{DRAM}. Thus, the time needed to complete each stage is $Nr/\min\{pfr, \beta_{\text{DRAM}}\}$. The sorting time is equal to the amount of time needed to complete all $\lceil \log_\ell N \rceil$ stages:

$$\text{Latency} = \frac{Nr \cdot \lceil \log_\ell N \rceil}{\min\{pfr, \beta_{\text{DRAM}}\}}. \quad (1)$$

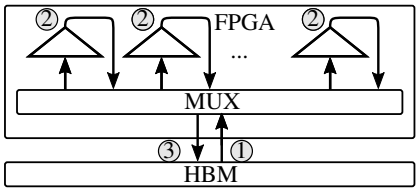

Fig. 3: Design and data movement of an unrolled tree configuration. Each triangle represents a merge tree.

In general, our model's predictions and experimental results suggest that increasing p is more beneficial than increasing ℓ up until the AMT throughput reaches the DRAM bandwidth.

2) AMT unrolling: The total sorting time can be further improved by employing multiple AMTs to work independently. Of course, this is only useful if the off-chip memory bandwidth can meet the increased throughput demands of using multiple AMTs. When λ_{unrl} AMTs are used to sort a sequence, we first partition the data into λ_{unrl} equal-sized disjoint subsets of non-overlapping ranges and then have each AMT work on one subset independently (Figure 3). This partitioning can be pipelined with the first merge stage and thus has no impact on sorting time. To ensure the merge time of each AMT will be approximately the same, all AMTs within a configuration are chosen to have the same p and ℓ value. As each AMT sorts its subset independently, the sorting time of unrolled configurations is the same as the time it takes for a single AMT to sort N/λ_{unrl} elements, assuming no other bottlenecks. Importantly, the off-chip memory bandwidth available to each AMT is no longer β_{DRAM}, but $\beta_{\text{DRAM}}/\lambda_{\text{unrl}}$ as the unrolled AMTs are required to share the available memory bandwidth. Thus, for a λ_{unrl}-unrolled configuration, we have

$$\text{Latency} = \frac{Nr \cdot \lceil \log_\ell (N/\lambda_{\text{unrl}}) \rceil}{\min\{pfr,\ \beta_{\text{DRAM}}/\lambda_{\text{unrl}}\}}. \quad (2)$$

As partitioning data into λ_{unrl} non-overlapping subsets may cause interconnect issues for large values of λ_{unrl}, another approach is to forgo partitioning and let each AMT sort a pre-defined address range. After each AMT finishes sorting its address range, we rely on merging these sorted ranges by using a subset of the AMTs from the original configuration. This approach is preferred when λ_{unrl} is larger than a certain range, but incurs a performance penalty because the final few merge stages cannot use all available AMTs. [1]

3) AMT pipelining: We will assume that DRAM bandwidth (β_{DRAM}) will be multitudes greater than I/O bandwidth, denoted $\beta_{\text{I/O}}$ [46]. For large data stored on the SSD, the array is sent over the I/O bus to the sorting kernel at throughput

[1] The comparison of non-overlapping and address-based partitioning is left for future work.

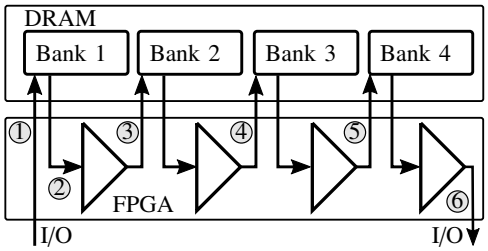

Fig. 4: Design and data movement of a pipelined tree configuration.

$\beta_{\text{I/O}}$; the kernel then sorts the array and returns it back to the SSD over the I/O bus. If we use λ_{unrl} unrolled AMTs to sort the input array in parallel as in §III-A2, the I/O bus will idle until the sorting procedure is completed. Since I/O bandwidth is a scarce resource, it would be better if the I/O bus would never idle. Therefore, we introduce AMT pipelining, which configures AMTs so that data can be read from and written to the I/O bus at a constant rate over time.

We can pipeline multiple AMTs in such a way that each merge stage of the sorting procedure is executed on a different AMT (Figure 4). Thus, at any point in time, each AMT executes its stage on a different input array. Concretely, when the first array comes over the I/O bus, it is sent to the first AMT in the pipeline and merged into ℓ-element sorted subsequences (steps 1-3, Figure 4). Once this initial stage is completed, the array is forwarded via a DRAM bank to a second AMT which performs the second merge stage (step 4). Concurrently, a second array can be fed into the first AMT in the pipeline (steps 1-2). Once this stage completes, a third array is fed into the first AMT, while the second and first arrays are independently merged by the second and third AMTs in the pipeline, respectively. Thus, the pipelined approach ensures a constant throughput of sorted data to the I/O bus (step 6). AMT pipelining is useful when multiple arrays need to be sorted. Thus, we use AMT pipelining in the first phase of the SSD sorter, where the input data is first sorted into DRAM-size subsequences (§IV-C). Specifically, using pipelining with $\lambda_{\text{pipe}} = 4$ lowers the execution time of the first phase of the SSD sorter by 2x.

Similarly to unrolling, pipelining divides the available DRAM bandwidth between the AMTs in the pipeline: when λ_{pipe} AMTs are used, the bandwidth of the pipeline will be limited to $\beta_{\text{DRAM}}/\lambda_{\text{pipe}}$. Further, the throughput of the pipeline is limited by the I/O bandwidth ($\beta_{\text{I/O}}$), as well as by the throughput of the AMTs used in the pipeline ($= pfr$). Thus, the throughput of a p and ℓ λ_{pipe}-pipeline is

$$\text{Throughput} = \min\{pfr,\ \beta_{\text{DRAM}}/\lambda_{\text{pipe}},\ \beta_{\text{I/O}}\}, \tag{3}$$

with the sorting time being

$$\text{Latency} = \frac{Nr \cdot \lambda_{\text{pipe}}}{\min\{pfr,\ \beta_{\text{DRAM}}/\lambda_{\text{pipe}},\ \beta_{\text{I/O}}\}}. \tag{4}$$

In contrast to unrolling, the total amount of data an AMT pipeline can sort is limited by two factors. First, each AMT in a pipeline must store its intermediate output onto DRAM. Specifically, in a λ_{pipe}-pipelined configuration, the biggest array that can be sorted without spilling data out of DRAM is $C_{\text{DRAM}}/\lambda_{\text{pipe}}$. Second, in a λ_{pipe}-pipelined $AMT(p,\ell)$ configuration, each array passes through at most λ_{pipe} merge stages (the data cannot be sent backwards in the pipeline). Thus, the maximum amount of data this pipeline can sort is $\ell^{\lambda_{\text{pipe}}}$. This constraint can be mitigated by pre-sorting small subsequences of the input data before the initial merge stage. In summary, the greatest number of records N that a p and ℓ λ_{pipe}-pipelined configuration can sort is

$$N \leq \min\{C_{\text{DRAM}}/\lambda_{\text{pipe}}, \ell^{\lambda_{\text{pipe}}}\}. \tag{5}$$

4) Combining pipelining and unrolling: We allow for both unrolling and pipelining to be used in configurations; this is done by replicating a λ_{pipe}-pipelined configuration λ_{unrl} times. Combining Equations 2, 3, and 4, we get the sorting time of a λ_{pipe}-pipelined, λ_{unrl}-unrolled configuration:

$$\text{Latency} = \frac{Nr \cdot \lambda_{\text{pipe}}}{\min\{pfr,\ \beta_{\text{DRAM}}/(\lambda_{\text{pipe}}\lambda_{\text{unrl}}),\ \beta_{\text{I/O}}\}}, \tag{6}$$

$$\text{Throughput} = \lambda_{\text{unrl}} \cdot \min\{pfr,\ \beta_{\text{DRAM}}/(\lambda_{\text{pipe}}\lambda_{\text{unrl}}),\ \beta_{\text{I/O}}\}. \tag{7}$$

B. Resource Utilization

In order for Bonsai to decide which AMT configurations can be implemented on a given chip, we need to develop good models for logic and on-chip memory utilization. We discuss resource utilization of a single $AMT(p,\ell)$; if k AMTs are used in a configuration, the resource utilization of the configuration will be exactly k times higher than that of a single AMT.

1) Logic utilization: AMTs are made up of mergers and couplers. Thus, we approximate the look-up table (LUT) utilization of an AMT by adding up the LUT utilization of the mergers and couplers used to build the AMT; the LUT utilization of an $AMT(p,\ell)$ can be written as:

$$\text{LUT}(p,\ell) = \sum_{n=0}^{\log \ell} 2^n (m_{\lceil p/2^n \rceil} + 2c_{\lceil p/2^n \rceil}), \tag{8}$$

with c_{2^n} and m_{2^n} being the number of LUTs used by a 2^n-coupler and 2^n-merger, respectively;the n-th summand corresponds to the LUT utilization at depth n of the tree.

Our experiments show that this simple model predicts LUT utilization of AMTs within 5% of that reported by the Vivado synthesis tool for all AMTs we were able to synthesize (i.e., AMTs for which $p \leq 32$ and $\ell \leq 256$) (Figure 10).

To ensure that an $AMT(p,\ell)$ can be synthesized on a chip, we require that

$$\text{LUT}(p,\ell) < C_{\text{LUT}}, \tag{9}$$

where C_{LUT} is the number of LUTs available on the FPGA.

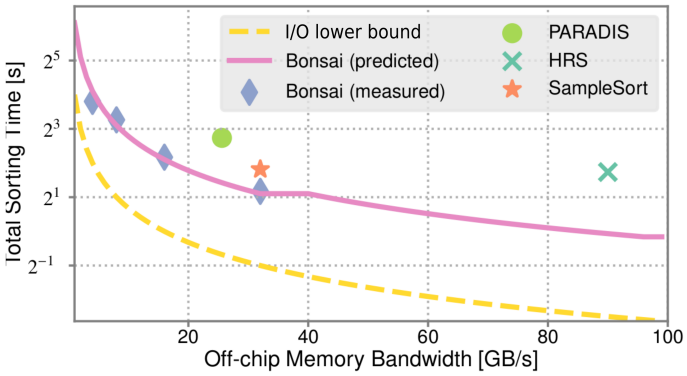

Fig. 5: The sorting time of optimal AMT configurations for different values of off-chip memory bandwidth compared to best sorters on CPU (PARADIS) [20], GPU (HRS) [18], and FPGA (SampleSort) [19]. The time required to stream the entire data from and to memory is also included (I/O lower bound). We use a 16 GB input size with 32-bit records.

2) On-chip memory utilization: The data loader is tasked to read the input data from DRAM in 1-4 KB sequential batches. Read batching is necessary for the DRAM to operate at peak bandwidth. As each of the ℓ input leaves to the AMT are stored in separate segments on DRAM, each leaf requires a separate input buffer for storing batched reads. In order for an $AMT(p,\ell)$ to be synthesizable on chip, we must ensure all ℓ input buffers can fit in on-chip memory. Thus, we have

$$b \cdot \ell \leq C_{\text{BRAM}}, \tag{10}$$

where b is the size of the read batches and C_{BRAM} is the amount of on-chip memory. When an FPGA is used, C_{BRAM} equals the amount of on-chip BRAM.

C. Bonsai AMT Optimizer

We now put the performance and resource models together to define Bonsai. Bonsai is an optimization strategy that exhaustively prunes all AMT configurations that fit into on-chip resources and picks the one with either minimal sorting time (latency-optimal) or maximal throughput (throughput-optimal). Specifically, Bonsai outputs the optimal AMT configuration (Table III) given array, hardware, and merger architecture parameters (Table II).

Formally, Bonsai's latency optimization model finds

$$\underset{p,\ell,\lambda_{\text{unrl}}}{\text{argmin}} \left\{ \frac{N \lceil \log_\ell(N/\lambda_{\text{unrl}}) \rceil}{\min\{\beta_{\text{DRAM}}/\lambda_{\text{unrl}},\ pfr\}} \right\},$$

subject to

$$\begin{cases} \lambda_{\text{unrl}} \cdot \text{LUT}(p,\ell) & \leq C_{\text{LUT}} \\ \lambda_{\text{unrl}} \cdot b\ell & \leq C_{\text{BRAM}}. \end{cases}$$

Pipelining is not used in the latency optimization model, because it does not improve sorting time. However, pipelining is used for optimizing sorting throughput.

In case many N-element arrays need to be sorted, optimizing for throughput gives better total time than optimizing for the latency of sorting a single N-element array; notably, we optimize for throughput in the first phase of the SSD sorter, where the data is first sorted into many DRAM-scale subsequences (details in IV-C). When optimizing for throughput, Bonsai finds

$$\underset{p,\ell,\lambda_{\text{unrl}},\lambda_{\text{pipe}}}{\text{argmax}} \left\{ \lambda_{\text{unrl}} \cdot \min\{\beta_{\text{I/O}},\ \beta_{\text{DRAM}}/(\lambda_{\text{pipe}}\lambda_{\text{unrl}}),\ pfr\} \right\},$$

subject to

$$\begin{cases} \lambda_{\text{pipe}}\lambda_{\text{unrl}} \cdot \text{LUT}(p,\ell) & \leq C_{\text{LUT}} \\ \lambda_{\text{pipe}}\lambda_{\text{unrl}} \cdot b\ell & \leq C_{\text{BRAM}} \\ \min\{C_{\text{DRAM}}/(\lambda_{\text{pipe}}\lambda_{\text{unrl}}),\ \ell^{\lambda_{\text{pipe}}}\} & \geq N. \end{cases}$$

Bonsai can pick AMT configurations that optimally utilize any off-chip memory bandwidth. The predicted and measured sorting time of Bonsai on a modern FPGA is presented as a function of available DRAM bandwidth in Figure 5 along with previously best performing CPU, FPGA, and GPU sorters.

Importantly, Bonsai can list all implementable AMT configurations in decreasing order of performance. Therefore, if the most optimal design is impossible to synthesize due to constraints not anticipated by the model, other close-to-optimal configurations can be tried.

IV. Optimal Configurations in Modern Hardware

In this section we present optimal AMT configurations on an FPGA for three vastly different off-chip memories: DDR DRAM (~32 GB/s bandwidth, 16-64 GB capacity), HBM (256-512 GB/s bandwidth, ~16 GB capacity), and SSD (<10 GB/s I/O bandwidth, 512 GB-2 TB capacity).

A. DRAM Sorting

As a concrete case study, we use Bonsai to construct a sorter on the AWS EC2 F1.2xlarge instance, which offers a modern FPGA connected to a 64 GB DDR4 DRAM that runs at 32 GB/s concurrent read and write.[2]

The latency-optimized configuration for this setup uses a single $AMT(32, 256)$ (Figure 2). The throughput of the $p = 32$ AMT for 32-bit records is exactly 32 GB/s when it runs at 250MHz. Thus this configuration matches the peak bandwidth of DRAM and then builds as many leaves (ℓ) as can be implemented on the FPGA. The reason why ℓ cannot be made larger than 256 is that the data loader uses up the on-chip memory (Equation 10).

The data is loaded from the host onto DRAM through the PCI-e (step 1, Figure 2). Then the data is streamed onto the FPGA (step 2), where the AMT merges the data and writes the merged subsequences back onto DRAM (step 3). The throughput of this streamed merge is 32 GB/s. Steps 2-3 are then repeated as many times as are necessary for all the subsequences to be merged into a single array. Finally, the sorted data is streamed back over the PCI-e (step 4).

[2]The measured DRAM read and write speed is roughly 29GB/s.

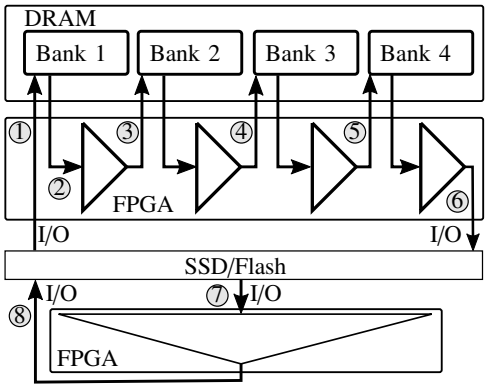

Fig. 6: Optimal terabyte-scale tree configuration and data flow. The top FPGA implements phase one, and the bottom FPGA implements phase two. The two phases can be implemented on a single FPGA via reprogramming.

Our experimental results show that our DRAM sorter has state-of-the-art sorting time for 4-32 GB array size, with a minimum of 2.3x, 1.3x, 1.2x and up to 2.5x, 3.7x, 1.3x lower sorting time than best CPU, FPGA, and GPU sorters, respectively. Details are presented in §VI-C.

B. High-Bandwidth Memory Sorting

Recently, Intel and Xilinx announced a release of a high-bandwidth memory (HBM) for FPGAs that is expected to achieve up to 512 GB/s bandwidth and has a capacity of up to 16 GB [47]. We now show that our design scales well with memory bandwidth by describing the optimal AMT configuration for such HBMs.

When sorting 32-bit integers with the HBM acting as DRAM, our model decides the optimal configuration to be $\lambda_{\text{unrl}} = 16$ $AMT(32, 2)$ (Figure 3). Data is streamed onto the sorting kernel (step 1) and each AMT independently merges a subsequence of the input data (step 2). Since λ_{unrl} is large, we do not partition the data into non-overlapping intervals before sending them to AMTs. Instead, each AMT sorts a predefined address range. Once each AMT sorts its range, there are only 16 sorted subsequences left, meaning that there will not be sufficient input for all AMTs in the next stage. Therefore, half of the AMTs are idled, and the remaining $16/2 = 8$ AMTs do one more merge stage. Then half of these AMTs are idled, and the remaining $8/2 = 4$ AMTs perform one more merge stage. This continues until the entire array is eventually merged. Therefore, all merge stages except the last four use all 16 AMTs and thus utilize the full memory bandwidth of 512 GB/s. We experimentally support our HBM predictions in §VI-D.

C. SSD Sorting

Our analysis can be expanded to allow for multi-tier off-chip memory hierarchies in order for it to be applicable to problem sizes beyond the capacity of DRAM. We now discuss how Bonsai can be used to optimize sorting on an FPGA that uses a 64 GB DRAM with 32 GB/s concurrent read and write bandwidth together with a 2 TB SSD with 8 GB/s I/O bandwidth.

The key insight for such two-level hierarchies is that the sorting procedure should be divided into two distinct phases, with each phase using a different AMT configuration. In the first phase, data is streamed from SSD to DRAM. Once data is on DRAM, we aim to sort as much data as would fit onto DRAM before sending the data back to SSD. We process the entire input in this way. Thus, at the end of the first phase, the SSD will contain many DRAM-size sorted subsequences.

In the second phase, these sorted subsequences are merged in as few stages as is possible. It is crucial that the number of stages in the second phase is minimized, as every merge stage requires a full round trip to SSD and is thus limited in throughput by the relatively low SSD bandwidth.

In the first phase, the entire input data completes exactly one round trip from SSD to DRAM and back. Thus, we can minimize the total execution time of this phase by using throughput-optimal Bonsai optimization.

The throughput-optimal configuration for the AWS F1 instance with 32-bit records sorting 8 GB arrays is shown in Figure 4. The pipeline contains 4 $AMT(8, 64)$. The DRAM has four memory banks, each connecting to the FPGA at four different ports. Each DRAM bank can read and write to the FPGA simultaneously at a peak rate of 8 GB/s. Thus, each AMT saturates the bandwidth capacity of one bank. Further, the I/O bandwidth is 8 GB/s. This implies that the data stream out of the last AMT in the pipeline (step 6) will saturate the I/O bandwidth. Thus, the throughput of this pipeline will be $\min\{\beta_{\text{I/O}}, \beta/\lambda_{\text{pipe}}, pfr\} = 8$ GB/s. The greatest amount of data we can sort with this pipeline is 8 GB, assuming we pre-sort the input data into 256-element subsequences (Equation 5).

In the second phase of SSD sorting, the SSD effectively acts as the only off-chip memory, as each stage in this phase requires a round trip to SSD. Thus, we can minimize latency directly by reconfiguring the FPGA using the latency-optimized AMT configuration. The latency-optimized design when the SSD is off-chip memory consists of one $AMT(8, 256)$. Note that p of our AMT is not high because peak SSD bandwidth is relatively low (<9 GB/s). Conversely, the AMT will have many leaves (ℓ) in order to minimize the total number of stages required to do the necessary merges. Therefore, this AMT configuration effectively mitigates the impact of low SSD bandwidth by reducing the total amount of memory accesses.

The optimal AMT configuration for SSD sorting of 32-bit records with SSD on AWS F1 is given in Figure 6. Steps 1-6 refer to the first phase and use the throughput-optimal configuration for 8 GB arrays. Once the first phase is completed, we reprogram the FPGA to a latency-optimized configuration for the second phase (steps 7-8).

The first phase will operate at 8 GB/s and output 8 GB sorted subsequences. As the second phase merges 256 subsequences concurrently, it can merge a total of $256 \cdot 8$ GB = 2 TB of data in only one SSD round trip. This round trip is also executed at 8 GB/s. Thus, this system is expected to sort 2 TB

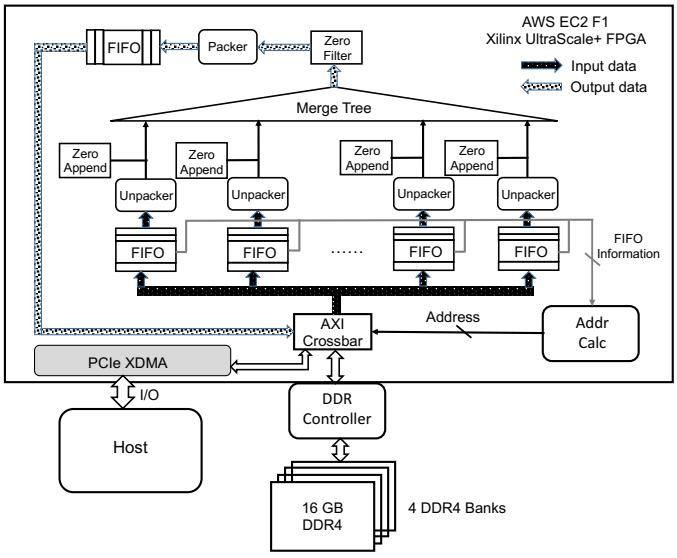

Fig. 7: The organization of a single merger tree system on the AWS EC2 F1 instance. The host can configure the merge tree kernel and prepare the data to be sorted through the PCIe DMA channels. To make full use of the external DRAM bandwidth, the communication between the sorting kernel and the DDR controller is always through a 512-bit wide AXI-4 interface, regardless of the record width: the Unpacker will extract one record from the 512-bit FIFOs per cycle automatically once the record width is set by the user and the packer will concatenate the output of the merge tree into 512-bit wide data. The role of FIFOs, zero append and zero filter are discussed in detail in V-A and V-B.

of data in 512 s (4 GB/s). This is over 17x less time than the state-of-the-art single-server-node sorter in [29].

In order to sort up to 256·2 TB=512 TB of data, we only need to run one more merge stage of the $AMT(8, 256)$. Thus, we can sort up to 512 TB of data at 8/3 =2.66 GB/s on a single FPGA. This is significantly better than all previous scalability results (Table I). In general, our design can sort even bigger arrays just by increasing the number of merge stages. Our SSD sorter offers 17.3x lower latency than the best previous single server node terabyte-scale sorter (§VI-E).

V. Other Microarchitecture Considerations

In this section we describe the microarchitecture of the proposed merge tree. Figure 7 shows a block diagram of the major components of our sorting system on an AWS EC2 F1 instance. Below we specifically present two important design considerations that enable the merge tree to work efficiently.

A. Data Loader

All AMT configurations read and write data in a stream. All reads from a specific leaf of the AMT are from continuous memory addresses. Thus, the AMT architecture has amenable off-chip memory access patterns.

The data loader performs the data reads and ensures off-chip memory is operating at peak bandwidth. Reads to any specific leaf occur in batches and are initiated by the data loader. Each leaf has an input buffer that is implemented as a FIFO, which is as wide as the DRAM bus (512 bits) and can hold two full read batches. The data loader checks in a round-robin fashion if any input buffer has enough free space to hold a new read batch. Whenever the data loader encounters an input buffer with sufficient free space, it performs a batched load into the buffer. In order to be able to do this, the data loader maintains a pointer to which address was last loaded into each input buffer.

In case one input buffer becomes empty, the AMT will automatically stall until the data loader feeds the buffer with more data. While running our experiments, we did not have any input buffer become empty (unless we were pausing the data loader in order to ensure the AMT behaves correctly with empty input buffers).

Due to batched and sequential reads/writes, the data loader allows the off-chip memory to operate at peak bandwidth, as verified by our experiments.

B. Intermediate State Flushing

During a merge stage, the AMT's intermediate state needs to be flushed and prepared for new input from another distinct set of inputs. Assume we sort an N-record array using an AMT with ℓ leaves. If at the current stage we have a-record sorted subsequences at the input, the AMT's intermediate state will have to be flushed $N/(\ell \cdot a)$ times for this stage alone. Even when sorting MB-scale data, the AMT's state will be flushed hundreds of thousands of times over the entire sorting procedure, especially at the early stages of the sorting process where we have many short runs to merge. Therefore, it is crucial that the state flushing scheme is efficient.

To address this issue, we use a reserved record value (called a *terminal record*) that is propagated through the datapath to signify to mergers that one of their input arrays has been fully processed. This approach improves on the work in [48]. Our design feeds exactly one terminal record between adjacent input arrays. The terminal record propagates through the AMT causing only a single-cycle delay when flushing each merger's state in preparation for new input. We use the value zero as the terminal record (*zero append* and *zero filter* in Figure 7). The *zero append* will append a zero as a terminal record whenever an entire sorted subsequence is fed into an input buffer. At the output of the merge tree, these terminal records are filtered out using a *zero filter*. Although we reserve zero for the terminal record, any other value may be used.

VI. Experimental Results

In this section, we present experiments to validate the model's resource and performance predictions (§VI-B), demonstrate the performance of our DRAM Sorter (§VI-C), and validate our performance projections for the HBM sorter (VI-D) and SSD Sorter (§VI-E). Finally, we discuss the scalability in input size and record width (VI-F) .

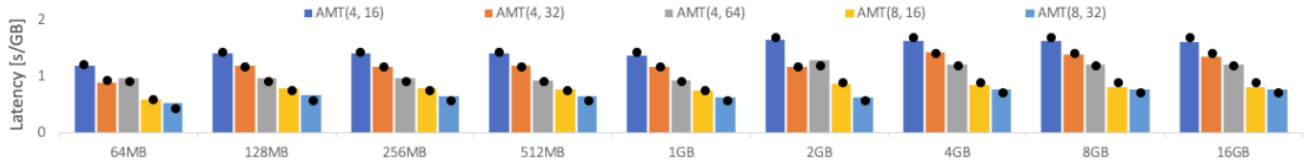

Fig. 8: Sorting time per GB of various AMTs on an AWS F1 instance (bars) and predicted by our performance model (·).

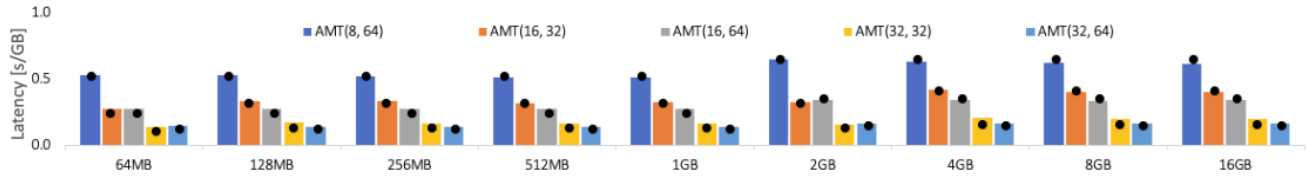

Fig. 9: Sorting time per GB of various AMTs on an AWS F1 instance (bars) and predicted by our performance model (·).

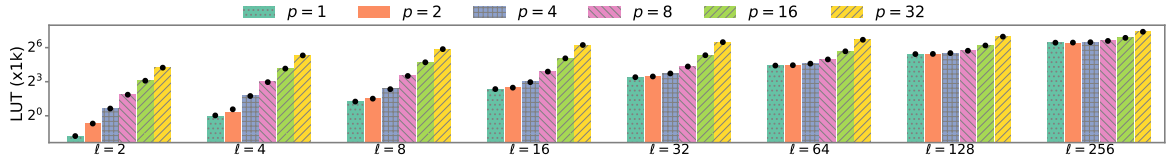

Fig. 10: Measured LUT utilization of various AMTs (bars) vs. our resource model's predictions (·).

A. Experimental Setup

We implement all experiments on an Amazon AWS F1.2xlarge instance that uses a single Virtex UltraScale+ 16nm VU9P FPGA with a 64 GB DDR DRAM that has 4 banks, each with 8 GB/s concurrent read and write bandwidth and a capacity of 16 GB. We use Xilinx Vivado version 2018.3 for synthesis and implementation of our designs, which we developed in Verilog. Our designs are running at 250 MHz or higher frequency.

We benchmark our sorter on 32-bit integers generated uniformly at random. We also use `gensort` to generate a benchmark of 100 byte records (10-byte key, 90-byte value) in accordance with the guidelines in Jim Gray's sort benchmark [49]. We hash the 90-byte value to a 6-byte index, which allows us to feed the 10-byte key and 6-byte value into a 16-byte AMT sorter. We generate datasets that are 64 MB-32 GB in size.

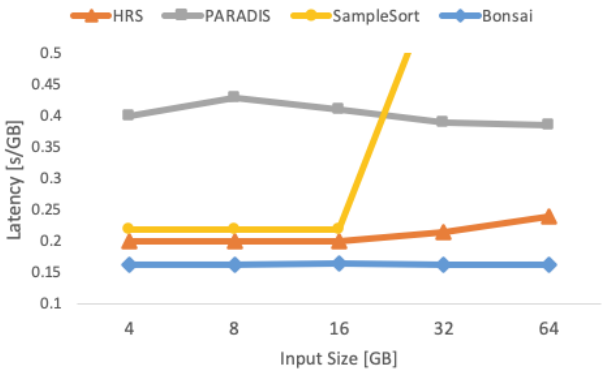

Fig. 11: Comparison of our DRAM sorter (Bonsai) to state-of-the-art results on CPU (PARADIS) [20], GPU (HRS) [18], and FPGA (SampleSort) [19]. Results presented in sorting time per GB (lower is better).

B. Model Validation

1) Resource Model Results: The resource utilization results reported by the synthesis tool are within 5% of our resource utilization predictions for all AMTs we were able to implement on the AWS EC2 F1 instance; that is, all AMTs such that $p \leq 32$ and $\ell \leq 256$. We report LUT utilization for all trees we were able to implement on the AWS FPGA in Figure 10.

2) Performance Model Results: In order to validate our performance model, we measured the sorting time of various AMTs for input sizes ranging 512 MB-16 GB and report results in Figures 8 and 9. All sorting time results are within 10% of those predicted by our performance model.

We notice that when AMTs have the same throughput p, the AMT with the greater number of leaves ℓ gives better or equal performance. Similarly, when two AMTs have the same number of leaves ℓ, the AMT with the higher throughput p has better performance, as long as DRAM bandwidth is not saturated. Once DRAM bandwidth is saturated, increasing throughput p does not decrease sorting time; however, increasing the number of leaves ℓ reduces the total number of merge stages, thus reducing sorting time even when the AMT throughput is high enough to saturate DRAM bandwidth. Thus, optimal single-AMT configurations always have throughput p exactly high enough to saturate DRAM bandwidth, with as many leaves ℓ as on-chip resources permit.

C. DRAM Sorter Results

1) DRAM sorter latency: We implement the latency-optimized DRAM sorter from §IV-A, which consists of a single $AMT(32, 64)$. We limit ℓ to 64 because designs with more leaves have lower frequency due to FPGA routing congestion. Our experiments show that for 4-32 GB input size

TABLE IV: Resource utilization breakdown of the optimal DRAM sorter on AWS F1.

Component	LUT	Flip Flop	BRAM
Data loader	110,102	604,550	960
Merge tree	102,158	100,264	0
Presorter	75,412	64,092	0
Total	287,672	768,906	960
Available	862,128	1,761,817	1,600
Utilization	33.3%	43.6%	60%

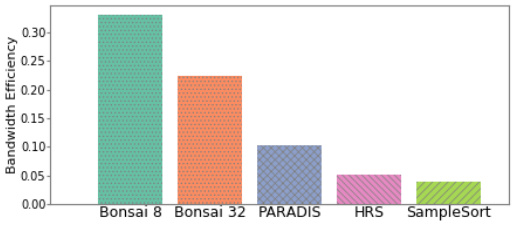

Fig. 12: Bandwidth-efficiency at 16 GB input size. Comparison of Bonsai's DRAM sorter to state-of-the-art results on CPU (PARADIS) [20], GPU (HRS) [18], FPGA (SampleSort) [19].

TABLE V: Execution time breakdown of sorting 2 TB of data.

Phase	Time	Percentage
Phase One	256s	49.6%
Reprogramming	4.3s	0.8%
Phase Two	256s	49.6%
Total	516.3s	

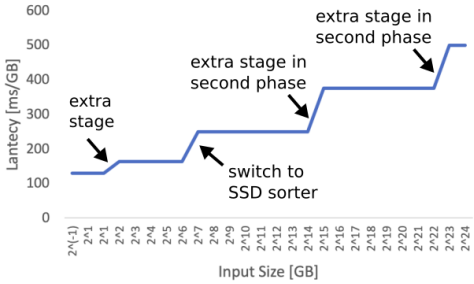

Fig. 13: Latency per GB of latency-optimized Bonsai sorters across 0.5 GB-100 TB input size, along with reasons for increases in latency.

the latency-optimized DRAM sorter has better performance than all previous sorters on any hardware.

As shown in Figure 11, when sorting 32GB data our implementation has 2.3x, 3.7x, and 1.3x lower sorting time than the best designs on CPUs, FPGAs, and GPUs, respectively. The DRAM sorter does not perform well for input size beyond that of DRAM capacity (64 GB). For input size over 64 GB, the SSD Sorter offers better performance.

The resource utilization breakdown of the latency-optimized DRAM sorter is given in Table IV. We use a 16-record bitonic network to presort the data into 16-record subsequences before the first merge stage. This reduces the total number of stages by one, and the total execution time by 10-20%, depending on input size. Table IV demonstrates that the FPGA has additional resources available to leverage future improvements in DRAM bandwidth, which is the bottleneck in our DRAM sorter.

2) Bandwidth-efficiency: Formally, bandwidth-efficiency is defined as the ratio of the throughput of the sorter to the available bandwidth of off-chip memory; for example, the DRAM-scale sorter used in the first phase of terabyte-scale sorting sorts at a throughput of 7.19 GB/s; since the DRAM bandwidth is 32 GB/s, the bandwidth-efficiency of our DRAM sorter is 7.19/32 = 0.225. As implementations of many algorithms are bottlenecked by memory bandwidth, bandwidth-efficiency is one of the most important scalability concerns in many problem domains. Additionally, memory accesses account for most of the energy consumed by many computer systems [50]. Thus, bandwidth-efficiency is directly related to energy consumption. We show the bandwidth-efficiency of our sorter in comparison to other bandwidth-efficient implementations in Figure 12[3]. To the best of our knowledge, our implementation has the highest bandwidth efficiency: specifically 3.3x better bandwidth-efficiency than any other sorter, when working with a 8 GB/s DRAM (labeled 'Bonsai 8'). When working with 32 GB/s DRAM bandwidth using 4 DRAM banks, Bonsai still gives 2.25x improvement in bandwidth-efficiency compared to all other sorters (labeled 'Bonsai 32').

[3]For SampleSort we use 1/latency as the throughput since the original paper doesn't report throughput at this data scale

D. HBM Sorter Results

The new Xilinx U50 board has a high-bandwidth memory tile that incorporates 32 DDR4 memory banks, with each bank providing up to 8 GB/s read/write bandwidth [51]. As current DDR4 DRAM has 4 DRAM banks, each providing up to 8 GB/s read/write bandwidth, we verify our HBM performance projections by implementing designs that saturate a single DRAM bank; then we make resource utilization projections for what would be required to scale the design to 32 banks.

In order to experimentally verify our results, we first synthesized 16 $AMT(16, 2)$ on an AWS F1 instance. The synthesis tool shows these AMTs can perform at 250 MHz, as predicted. Since we did not have access to HBM, we verified that unrolling scales well by using DRAM banks. As the DRAM bandwidth is 32 GB/s, we showed that two $p = 16$ AMTs saturate DRAM bandwidth, with each AMT using two DRAM banks. We also showed that four $p = 8$ AMTs saturate DRAM bandwidth, with each AMT working independently on a single DRAM bank. This demonstrates that unrolling scales both performance and resource utilization linearly with the unrolling amount, λ_{unrl}.

E. SSD Sorter Results

We validate our conclusions from §IV-C by independently verifying the throughput of each of the two phases. We throttled the DRAM throughput to that of modern SSD Flash (8 GB/s), and run the pipeline in phase one (Figure 4) on AWS F1. The pipeline effectively saturates I/O bandwidth of 8 GB/s, as predicted by our model.

In order to verify the second phase, we again throttle the DRAM to operate at 8 GB/s and implement an $AMT(8, 256)$. Again, the design operates at 8 GB/s. This validates our performance predictions for SSD sorting. We also experimentally verify that reprogramming the FPGA between the two phases will take an average time of 4.3s. The summary of our results is given in Table V. Our SSD sorter offers 17.3x lower latency on sorting 1 TB of data compared to the best previous single server node terabyte-scale sorter [29].

F. Scalability Analysis

We measure scalability as sorting time (latency) per GB of input data. If a sorter is able to achieve the same latency per GB for a wide range of input sizes, then it has good scalability. As demonstrated by Table I, our sorter exhibits scalability that is multiple times better than all previous work.

1) Scalability in Input Size: In order to understand the scalability of our system, we now describe the reasons for latency per GB increases across a wide range of input size. Figure 13 shows the latency per GB of the Bonsai sorter for input size 0.5 GB to 1024 TB.

The latency per GB increases at four distinct points, marked by arrows on Figure 13. The first latency increase (labeled 'extra stage') happens at 2 GB because the DRAM sorter needs to feed the data to the AMT one additional time; that is, the data requires an extra merge stage to be sorted (1.33x performance penalty). The second latency increase (labeled 'switch to SSD sorter') happens at 128 GB when the input data is too large to fit onto DRAM; at this point, data is initially stored in an SSD (1.33x performance penalty). The third latency increase (labeled 'extra stage in second phase') happens at 32 TB because an extra merge stage in the second phase of the SSD sorter is needed to merge all the data (1.5x performance penalty). The fourth performance penalty happens at 4096 TB and increases latency per GB by 1.33x; this is again caused by a need for an extra merge stage in the second phase of the SSD sorter.

2) Scalability in Record Width: Table VI shows that the AMT building blocks for 32- and 128-bit records exhibit comparable resource utilization for equal-throughput elements, with 128-bit records offering somewhat better throughput per LUT. For example, a 128-bit record 4-merger has the same throughput as a 32-bit record 16-merger, but almost 50% less logic utilization. This is because the bigger the record width, the less data shuffling is required within each merger. Specifically, the 128-bit record 4-merger has the same throughput as the 32-bit 16-merger, but the 128-bit 4-merger needs a much smaller number of compare-and-swap operations to output 4 records per cycle versus the 16 records per cycle that the 32-bit 16-merger must output. More formally, the logic complexity of the compare-and-swap unit grows linearly with record width, while the number of compare-and-swap units within a merger grows superlinearly ($\Theta(k \log k)$) with the number of records. Thus, 1 GB of wider records requires less resources to be sorted in the same amount of time as one GB of narrower records.

TABLE VI: LUT utilization and throughput of building-block elements.

Element	Th-put	LUT	Element	Th-put	LUT
1-merger	1 GB/s	300	FIFO	1 GB/s	50
2-merger	2 GB/s	622	2-coupler	1 GB/s	142
4-merger	4 GB/s	1,555	4-coupler	2 GB/s	273
8-merger	8 GB/s	3,620	8-coupler	4 GB/s	530
16-merger	16 GB/s	8,500	16-coupler	8 GB/s	1,047
32-merger	32 GB/s	18,853	32-coupler	16 GB/s	2,079

(a) 32-bit records

Element	Th-put	LUT	Element	Th-put	LUT
1-merger	4 GB/s	1,016	FIFO	4 GB/s	134
2-merger	8 GB/s	2,210	2-coupler	4 GB/s	576
4-merger	16 GB/s	5,604	4-coupler	8 GB/s	1,938
8-merger	32 GB/s	13,051	8-coupler	16 GB/s	2,081
16-merger	64 GB/s	29,970	16-coupler	32 GB/s	4,142
32-merger	128 GB/s	77,732	32-coupler	64 GB/s	8,266

(b) 128-bit records

VII. Related Work

A. FPGA Sorting

In addition to [29] and [19] (§I), [30] and [32] give a fairly comprehensive analysis of sorting networks on FPGAs, but limit the discussion to sorting on the order of MB elements, with [32] arguing for a heterogeneous implementation where small chunks are first sorted on the FPGA and then later merged on the CPU. Still, their reported performance has little advantage over a CPU for larger input sizes. The authors in [33] present a domain-specific language to automatically generate hardware implementations of sorting networks; they consider area, latency, and throughput. The unbalanced FIFO-based merger in [34] presents an interesting approach to merging arrays, but is not applicable to large sorting problems. In [35] the authors use a combination of FIFO-based and tree-based sorting to sort gigabytes of data. This work also removed any global intra-node control signals and allowed for larger trees to be constructed. However, it lacks an end-to-end implementation and focuses only on building the sorting kernel and reporting its frequency and resource utilization. Further, due to the recent innovation in hardware merger designs, memory and I/O, and increases in FPGA LUT capacity, their analysis has become more limited. Our work extends their analysis and improves performance by using higher throughput merge trees.

B. GPU Sorting

The work in [52] models sorting performance of GPUs. The model allows researchers to predict how different advances in hardware would impact the relative performance of various state-of-the-art GPU sorters. Their results indicate that performance is limited by shared and global memory bandwidth. Specifically, the main issue with GPU sorters compared to CPU implementations seems to be that GPU's shared memory is multiple times smaller than CPU RAM. This implies global memory accesses are more frequent with GPUs than disk or flash accesses in CPU implementations.

The work in [18] focuses on building a bandwidth-efficient radix sort GPU sorter with an in-place replacement strategy that mitigates issues relating to low PCIe bandwidth. To the best of our knowledge, their strategy provides state-of-the-art results on GPU, reporting they sort up to 2 GB of data at over 20 GB/s. When integrated as a CPU-GPU heterogeneous sorter with CPU does the merging, they are able to sort 16 GB in roughly 3.3s. Nonetheless, this approach is not scalable, as it relies on executing merge stages on CPU. Specifically, at 32 GB, the CPU computation dominates the execution time of the heterogeneous sorter.

VIII. Conclusions

In this paper we present Bonsai, a comprehensive model and sorter optimization strategy that is able to adapt sorter designs to available hardware. When Bonsai's optimized design is implemented on an AWS F1 FPGA, it yields a minimum of 2.3x, 1.3x, 1.2x and up to 2.5x, 3.7x, 1.3x speedup over the best sorters on CPUs, FPGAs and GPUs as well as exhibits 3.3x better bandwidth-efficiency compared to the best previous sorting implementation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported in part by the NSF CAPA REU Supplement award # CCF-1723773, the CRISP center under the JUMP program, Mentor Graphics and Samsung under the CDSC Industrial Partnership Program. The authors also thank Xilinx for equipment donation and Amazon for AWS credits.

Nikola Samardzic owes special thanks to the Rodman family for their continued support through the Norton Rodman Endowed Engineering Scholarship at UCLA. He also thanks the donors that contributed to the UCLA Achievement Scholarship and the UCLA Womens' Faculty Club Scholarship.

References

[1] B. Sukhwani, T. Roewer, C. L. Haymes, K.-H. Kim, A. J. McPadden, D. M. Dreps, D. Sanner, J. Van Lunteren, and S. Asaad, "Contutto: A novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[2] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An appliance for big data analytics," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[3] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[4] L. Wu, D. Bruns-Smith, F. A. Nothaft, Q. Huang, S. Karandikar, J. Le, A. Lin, H. Mao, B. Sweeney, K. Asanovic, D. A. Patterson, and A. D. Joseph, "FPGA accelerated INDEL realignment in the Cloud," in *Proceedings of the 25th Annual International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[5] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled cpu-fpga platforms," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[7] J. Fowers, K. Ovtcarov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. W. L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burder, "A configurable cloud-scale DNN processor for real-time AI," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[8] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *International Symposium on Microarchitecture (MICRO)*, 2017.

[9] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: Qos for accelerated machine learning systems," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[10] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[11] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2019.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Operating System Design and Implementation (OSDI)*, 2004.

[13] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," in *Very Large Data Bases (VLDB)*, 2017.

[14] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," in *Operating System Design and Implementation (OSDI)*, 2004.

[15] A. Thusoo, J. S. Sarma, N. Jain, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *International Conference on Data Engineering (ICDE)*, 2010.

[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *International Conference on Management of Data*, 2015.

[17] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: Simplified relational data processing on large clusters," in *International Conference on Management of Data*, 2007.

[18] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on GPUs," in *International Conference on Management of Data (SIGMOD)*, 2017.

[19] H. Chen, S. Madaminov, M. Ferdman, and P. Mildred, "Sorting large data sets with FPGA-accelerated samplesort," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[20] M. Cho, D. Brand, and R. Bordawekar, "PARADIS: An efficient parallel algorithm for in-place radix sort," in *Very Large Data Bases (VLDB)*, 2015.

[21] H. Inoue and K. Taura, "SIMD- and cache-friendly algorithm for sorting an array of structures," in *Very Large Data Bases (VLDB)*, 2015.

[22] J. Chhugani, W. Macy, and A. Baransi, "Efficient implementation of sorting on multi-core SIMD CPU architecture," in *Very Large Data Bases (VLDB)*, 2008.

[23] "Nvidia thrust." https://developer.nvidia.com/thrust. Accessed: 2019-10-30.

[24] "Nvidia CUB." https://github.com/NVlabs/cub. Accessed: 2019-10-30.

[25] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[26] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, 2011.

[27] C. Binnig, S. Hildenbrand, and F. Farber, "Dictionary-based order-preserving string compression for column stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

[28] P. Bohannon, P. Mcllroy, and R. Rastogi, "Main-memory index structures with fixed-size partial keys," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[29] S.-W. Jun, S. Xu, and Arvind, "Terabyte sort on FPGA-accelerated flash storage," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[30] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on FPGA," in *International Symposium on Field-programmable Gate Arrays (FPGA)*, 2015.

[31] K. Fleming, M. King, and M. C. Ng, "High-throughput pipelined mergesort," in *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2008.

[32] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, FPGA '16, 2016.

[33] M. Zuluaga, P. Milder, and M. Püschel, "Computer generation of streaming networks," in *Design Automation Conference (DAC)*, 2012.

[34] R. Marcelino, H. C. Neto, and J. M. P. Cardoso, "Unbalanced FIFO sorting for FPGA-based systems," in *International Conference on Electronics, Circuits, and Systems (ICECS)*, 2009.

[35] D. Koch and J. Tørresen, "FPGAsort: A high performance sorting architecture expoiting run-time reconfiguration on FPGAs for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011.

[36] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub, "Tencent sort." Technical Report.

[37] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, and S. Matsuoka, "GPU-accelerated large-scale distributed sorting coping with device memory capacity," in *Transactions on Big Data*, 2016.

[38] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," in *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2011.

[39] D. E. Knuth, *Art of Computer Programming: Sorting and Searching*. Addison-Wesley Professional, 2nd ed., 1998.

[40] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," in *Research Report, RR-0725 INRIA*, 1988.

[41] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Intl. Symp. on Parallel Algorithms and Architectures*, 1991.

[42] A. Farmahini-Farahani, H. J. Duwe, M. J. Schulte, and K. Compton, "Modular design of high-throughput, low-latency sorting units," in *Transactions on Computers*, 2008.

[43] K. E. Batcher, "Sorting networks and their applications," in *American Federation of Information Processing Societies*, 1968.

[44] K. Manev and D. Koch, "Large utility sorting on FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, 2018.

[45] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2nd ed., 2004.

[46] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance analysis of NVMe SSDs and their implication on real world databases," in *International Systems and Storage Conference (SYSTOR)*, 2015.

[47] M. Deo, J. Schulz, and L. Brown, "Intel Stratix 10 MX devices solve the memory bandwidth challenge," 2019.

[48] S. Mashimo, T. V. Chu, and K. Kise, "A high-performance and cost-effective hardware merge sorter without feedback datapath," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[49] "Sort benchmark home page." http://sortbenchmark.org/. Accessed: 2019-10-30.

[50] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Saving power in the control path of embedded processors," in *Design & Test of Computers*, 1994.

[51] "Alveo u50 data center accelerator card." https://www.xilinx.com/products/boards-and-kits/alveo/u50.html. Accessed: 2019-10-30.

[52] B. Karsin, V. Weichert, H. Casanova, J. Iacono, and N. Sitchinava, "Analysis-driven engineering of comparison-based sorting algorithms on GPUs," in *International Conference on Supercomputing (ICS)*, 2018.