Redwood: Flexible and Portable Heterogeneous Tree Traversal Workloads

Yanwen Xu
University of California, Santa Cruz
yxu83@ucsc.edu

Ang Li
Princeton University
angl@princeton.edu

Tyler Sorensen
University of California, Santa Cruz
tysorens@ucsc.edu

***Abstract*—Shared memory heterogeneous systems are now mainstream, with nearly every mobile phone and tablet containing integrated processing units. However, developing applications for such devices is difficult as workloads must be decomposed across different processing units, and the decomposition must be flexible to account for the growing diversity of devices, each with different relative processing unit throughput. Furthermore, many devices require distinct programming front ends, requiring significant effort to write cross-platform applications.**

In this work, we identify a pragmatic class of applications, which we call traverse-compute applications, that are ideal for shared memory heterogeneous systems. These applications have a flexible heterogeneous decomposition where CPUs excel at traversing a tree structure, while accelerators excel at node computations. Leveraging this insight, we present Redwood: a framework for writing heterogeneous traverse-compute workloads. Redwood provides a simple processing unit abstraction and a tree traversal library that enables heterogeneous optimizations. Using Redwood, we implement Grove, a benchmark suite containing nine pragmatic tree traversal applications, e.g., k-nearest neighbors. We instantiate Redwood for three different heterogeneous programming platforms: CUDA, SYCL, and High-Level Synthesis; we use Grove to evaluate five shared memory heterogeneous systems. Our evaluation highlights the importance of flexible heterogeneous decomposition as the optimal parameters differ widely across platforms and applications. However, once optimally configured, heterogeneous implementations can provide up to $13.53\times$ speedups (geomean of $3.01\times$) over homogeneous implementations, showcasing the potential of heterogeneous computing for these workloads.

***Index Terms*—Heterogeneous Computing, Benchmarks, FPGAs, GPUs, Tree Traversals**

I. Introduction

As Moore's Law and Dennard's scaling come to an end, the demand for ever-increasing performance and energy efficiency has driven the development of *Shared-Memory Heterogeneous Systems* (SMHSs), particularly in mobile System-on-Chips (SoCs), e.g., an Apple A12 SoC has over 80% of the die area consisting of accelerators [45]. SMHSs incorporate diverse specialized processing units (PUs), including traditional CPUs and *Programmable Accelerating PUs* (PAPUs), such as integrated GPUs and embedded FPGAs, all interconnected through a shared-memory hierarchy on the same chip. In contrast to conventional *accelerator-oriented* heterogeneous systems (e.g., [23], [41]), SMHSs architecture enables efficient communication and data sharing between different PUs, compared to discrete heterogeneous systems where data is typically transferred via PCIe, as studied in [12], [19], [33].

In recent years, there has been a growing trend in academia to explore SMHSs, as shown by the increasing number of studies focusing on the integration of accelerators and their interactions with a shared memory hierarchy in SoC designs [22], [27]. Moreover, in [13], [30], [60], these studies have shown that shared memory is a key factor in improving Heterogeneous SoCs' performance and energy efficiency.

Apart from academic explorations, there are many deployed SMHSs, largely on mobile devices, such as phones and tablets. Many different vendors produce these devices, e.g. Apple, Qualcomm, and ARM, each of which contains vendor-specific PAPU architectures. This diversity necessitates the need for benchmarks to evaluate and compare the efficiency of these SMHSs. However, SMHSs remain difficult to target with workloads that can efficiently utilize their different PUs.

In this work, we identify a pragmatic class of workloads, which we call *traverse-compute*, that are ideal for SMHSs. These workloads aim to harness the capabilities of various PUs in SMHSs effectively, enabling comprehensive performance evaluation and comparison across different SoC designs.

A. Heterogeneous Traverse-Compute Workloads

Trees are a fundamental data structure and form the foundation for many useful workloads [18]. A tree traversal is characterized by irregular memory accesses, as nodes can be scattered across the memory. Many tree-based workloads combine tree traversal with some computation, largely occurring when leaf nodes are visited. We call such workloads *traverse-compute* workloads, which can often be found in statistical learning [14], and particle simulations [5].

Traverse-compute workloads utilize spatial partitioning trees like k-d trees [3], allowing a point cloud to be efficiently searched in $O(\log(n))$ instead of $O(n)$. An example of traverse-compute workload is *nearest-neighbor* (NN); its applications include facial recognition on surveillance cameras [39], object detection in robotics [35], and anomaly detection in portable monitoring systems [50]. These tree-based approaches offer an attractive energy-compute trade-off, making them a suitable alternative to DNN-based approaches for resource-constrained devices [40], [57], where SMHSs are often employed.

These workloads have been well-studied and can be found in other benchmarks: e.g., Splash3 [43] and [6] includes *Barnes-Hut* algorithms; Hetero-Mark [51] includes *k nearst-neighbor*.

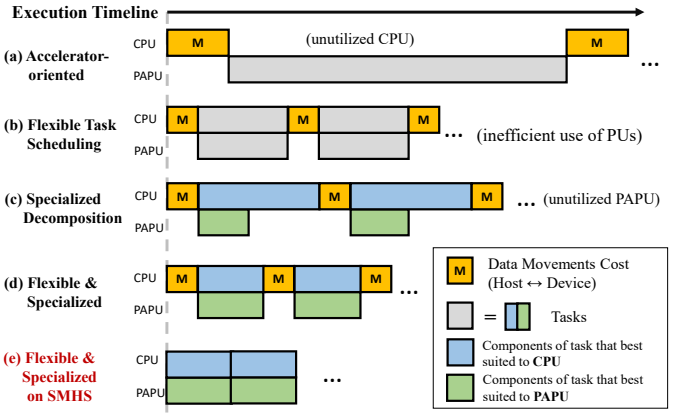

Fig. 1. (a-d) Execution timelines for different heterogeneous workload paradigms. Blue/Green shading represents tasks, decomposed from the original algorithm (Grey), that are best suited to CPU/PAPU, respectively. Redwood combines flexible task scheduling, specialized decomposition, and efficient communication between PUs, as illustrated in (e).

However, the traverse-compute abstraction boundary has not been exploited in the way that we are proposing.

B. Heterogeneous Computing Paradigms

While there are many works on heterogeneous computing, e.g., see [33], we believe that several factors make this workload, along with our corresponding programming framework and benchmark suite, unique. We start by discussing four common heterogeneous computing paradigms, illustrated in Figure 1, and show how traverse-compute workloads have a flexible and specialized mapping to SMHSs.

a) Accelerator-oriented: In many discrete, accelerator-oriented systems, the trend is to offload most computation to the accelerator, as shown in Fig. 1-a. This approach can result in an underutilized CPU, especially in a SMHS, where different PUs share a more or equal portion of the system resources, e.g., as is the case on many smartphone SoCs [45]. Thus, to utilize the entire system, tasks should be decomposed so that the workload is distributed across PUs.

b) Flexible Task Scheduling: Other frameworks have task schedulers that assign similar tasks across both the CPU and GPU, such as work stealing in [10]; this is shown in Fig. 1-b, where both PUs are occupied. Furthermore, due to task granularity and dynamic task management, these heterogeneous frameworks can adapt to different PAPU throughputs. Using Nvidia's SMHS Jetson chips as an example, the Xavier reports 1.7 GPU TFLOPs, while the Orin reports 5.3 GPU TFLOPs [38]. Given these differences, a portable heterogeneous workload requires a *flexible* decomposition, i.e., so that the workload executing on the Orin can be configured to have more work mapped to the GPU and the workload configured to the Xavier will have less work mapped to the GPU.

c) Specialized Heterogeneous Decomposition: In the above approach, identical tasks are mapped to different PUs, which may result in inefficient use of the available resources; because different PUs may have different strengths and weaknesses, e.g., CPUs excel at sequential computation with unpredictable control flow operations, while GPUs are optimized for data-parallel operations. In an ideal decomposition, each PU will perform tasks that are best suited for its unique architectural strength. As illustrated in Fig. 1-c, one maybe extract components from the original task (grey) into parts that are most suited for PAPU (green) and CPU (blue). However, the downside is that not all tasks can be decomposed in a balanced manner: For example, as shown in the figure, it may be the case that only a small fraction of the task contains components that are well suited to GPU. This can lead to an underutilized PAPU.

d) Flexible & Specialized: To achieve an efficient heterogeneous implementation in SMHSs, a balance between flexible and specialized decomposition is needed (Fig. 1-d,e). In both of these cases, the CPU and PAPU are fully utilized with tasks that are suited to their architectural strength. Furthermore, in a SMHS (Fig. 1-e), the shared memory hierarchy can be used to reduce data movement between PUs, improving performance and efficiency. This allows even finer-grained tasks to be offloaded from the CPU to the PAPU. However, programming a workload with flexible and specialized decomposition on SMHSs can be challenging, as communication and synchronization between PUs must be carefully managed. In this work, we provide an API and runtime that can be used to efficiently implement flexible and specialized heterogeneous traverse-compute workloads.

C. Challenges in Programmability

As a more practical concern, due to the diversity of SMHSs, many devices support a limited set of programming frontends. For example, CUDA can only be executed on Nvidia devices, SYCL has the most support on Intel devices, and FPGAs are most easily programmed through High-level Synthesis (HLS). Although portability is often advertised, it can be difficult in practice, as shown in OpenCL [49]. Given this, it is difficult for workloads to target cross-vendor SMHSs.

D. Mapping Traverse-Compute Workloads to a SMHS

Traverse-compute workloads have a natural heterogeneous decomposition (as in Fig. 1-e), and a simple programming interface, as follows:

- **Flexible Scheduling**: Trees can be configured such that the partitioning stops at different levels; all points that have not been partitioned can be assigned to a leaf node. A shallow tree will have many un-partitioned points, i.e., heavier compute tasks for the PAPU and fewer traversal steps for the CPU. A deep tree will be the opposite. The tree can easily be reconfigured to adapt to SMHSs that have different relative throughputs between PUs.
- **Specialized Heterogeneous Decomposition**: Tree workloads can be decomposed such that the CPU traverses the tree, utilizing architecture components such as load-store queues to tolerate long-latency memory accesses; while the PAPU performs the node computation, utilizing the high architectural parallelism.

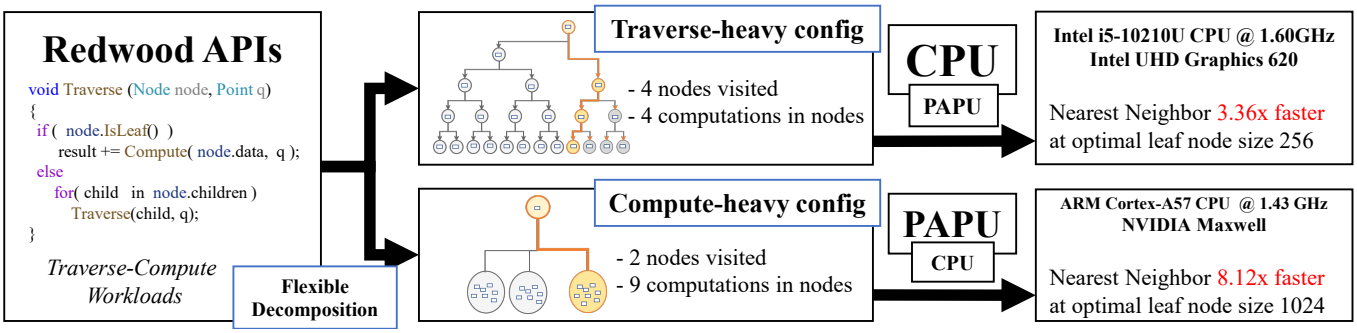

Fig. 2. The flow of Redwood: (left) User uses Redwood APIs to implement traverse-compute workloads. (middle) The workload can be easily configured to target SMHSs with different relative throughputs between CPU and PAPU. (right) Workload now can be used to evaluate different SMHSs.

- **Simple Abstraction Boundaries:** and can utilize common accelerator routines. The core accelerator function is a reduction, which has been widely studied for programmable accelerators [17]. Thus, traverse-compute applications require only a simple interface to target a wide range of SMHSs.

E. Redwood and Grove

While the above insights are conceptually sound, they require significant effort to implement in a modular and efficient way. To this end, we present **Redwood**: *a framework for writing heterogeneous traverse-compute workloads for SMHSs*. Redwood has three components:

1) **Backend**: provides a simple abstraction over PAPUs and can be instantiated to target different platforms.
2) **Data structures**: spatial tree structures that have a configurable height.
3) **Traverse-compute API**: an interface for leaf node computations that hides heterogeneous optimizations.

We show how Redwood can be used to develop heterogeneous traverse-compute workloads by implementing **Grove**: *a benchmark suite of nine tree traversal applications*. Each benchmark contains distinct characteristics in terms of tree traversal patterns and node computations. Grove contains important workloads such as *k-nearest neighbors* (KNN), a widely used supervised learning workload. Grove also contains Barnes-Hut (BH), a common HPC application for particle simulation with additional applications in learning [54].

We highlight the programmability of Redwood by implementing the backend for CUDA, SYCL, and high-level synthesis (HLS). We then use Grove to evaluate five SMHSs: two Nvidia Jetson devices, two Intel CPU/GPU processors, and one tightly integrated CPU/FPGA device. These systems all have different performance profiles across their respective PUs, and Grove can be used to highlight the importance of the flexible heterogeneous decomposition and a simple backend. We identify key heterogeneous performance characteristics of each platform, including their optimal CPU/PAPU load balance and task granularity. We show that tree traversals have the potential to benefit significantly from heterogeneous computation, with tuned heterogeneous implementations outperforming their homogeneous counterparts by up to $13.53\times$, with a geomean of $3.01\times$.

Figure 2 shows an overview of our contribution: Redwood is used to implement a traverse-compute workload. The workload can then be configured to be either a shallow or deep tree depending on the relative throughput of the PUs. The application can then be executed across different SMHSs (depending on the availability of a backend). In the figure, we highlight the NN application on an Nvidia Jetson device (executing on the CUDA backend), which has a higher PAPU throughput, and an Intel i7-9700K (executing on the SYCL backend), which has a lower PAPU throughput.

Contributions: In summary, our contributions are:

- The classification of traverse-compute workloads, which have a flexible heterogeneous decomposition (Sec. II-A).
- Redwood: a framework for developing heterogeneous traverse-compute workloads (Sec. III).
- Grove: a suite of 9 traverse-compute workloads (Sec. IV).
- An evaluation of Grove across five SMHSs, identifying key performance characteristics (Sec. VI).

We aim to make it easier for researchers and developers to analyze and design SMHS, by open-sourcing Redwood and Grove at https://github.com/xuyanwen2012/redwood-rt.

II. Background

A. Spatial Partitioning Trees

As mentioned in Sec. I-A, trees can be used for a structured representation of a *point cloud* [3], [31]. In such cases, these trees are called *Spatial Partitioning Trees* (SPT). A point cloud is a set of points P in N-dimensional space. Each point consists of a coordinate vector $\vec{v}$ and potentially some data, e.g., its mass.

A SPT will structure P in the following way: each node represents either: (1) a disjoint set of particles $P' \subset P$, or (2) a bounding box b. The root node is a bounding box around all particles. The space is then partitioned into sub-bounding boxes. These sub-bounding boxes are assigned to the children of the root node. This process continues recursively until the sub-bounding box contains a small (but configurable) number of particles. These particles then get set as the data for a leaf node, and the recursion terminates.

A k-d tree subdivides the bounding box using by creating a splitting plane through one of the axes. A quadtree, or octree, is limited to $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively. These trees subdivide their bounding box into four or eight equally sized subspaces. A k-d tree is guaranteed to have at most logarithmic depth, while an octree is easier to be modified dynamically, e.g., particles can be easily reorganized when they move out of their bounding box in particle simulations.

a) Formalizing Traverse-Compute Workloads: Traverse-compute workloads can be formalized as such: Given two point clouds P and Q, create a spatial tree for Q, and call it $T(Q)$. For each $p \in P$, a traverse-compute routine, call it tr, is performed on $T(Q)$. The tr operation starts at the root of $T(Q)$ and traverses the tree. At each node, $n \in T(Q)$, some computation is performed on the node data to determine the traversal path. When the traversal reaches a leaf node l, all particles $w \in l$ are computed. This computation consists of: (1) a distance computation between $p.\vec{v}$ and $w.\vec{v}$, giving a distance d; (2) an interaction computation based on $p.data$, $w.data$ and d, giving a value i; and (3) a computation across the i values, e.g., a reduction. After a leaf node visit, the traversal returns to the parent to make the next traversal decision. Traversals that depend on leaf node computations are said to be *guided*, while traversals that do not depend on leaf node computations are said to be *unguided* [18]. Section IV shows examples of nine traverse-compute workloads, which utilize a variety of trees, distance metrics, and traversal patterns.

B. Shared Memory Heterogeneous Systems

The focus of this work is integrated, shared memory heterogeneous systems (SMHSs), also known as *Unified Memory Architectures* (UMA); these systems are composed of several PUs, where all PUs (e.g., CPUs and GPUs) have access to the same main memory region. In some cases, the different PUs even have coherent caches [27]. Thus, communication between PUs is more efficient than communication in traditional discrete accelerator systems, i.e., communication does not require expensive data transfers across PCIe.

One popular CPU-GPU implementation of SMHSs is NVIDIA's Tegra SoC. In Tegra, both the CPU (host) and an integrated GPU (device) share the same DRAM memory on the SoC. This is in contrast to traditional discrete GPU systems, which have a separate main memory for the GPU. In this paper, we refer to *Unified Shared Memory* (USM) as a memory region that is allocated in the same physical SoC main memory. In the case of Tegra, the device memory, host memory, and unified memory are all allocated on the same physical SoC DRAM.

The PAPUs we consider are integrated GPUs and FPGAs; we now overview each PU, including its architectural strengths and its programming platforms.

a) CPUs: CPUs are often the foundation of a computer system. Their design is optimized for sequential, unpredictable workloads. For example, they contain complex hardware components, such as reorder buffers and load-store queues, so that other instructions can be executed while memory requests are buffered. This architectural complexity allows CPUs to excel at workloads with irregular memory accesses. For example, some sparse computations have shown CPUs to outperform their GPU counterparts, e.g., by up to 3.5× [8]. However, CPUs have limited computation throughput.

b) GPUs: While GPUs began as graphics accelerators, due to general-purpose GPU (GPGPU) programming frameworks, they have now been applied to more general-purpose workloads (see [44, ch. 1.5] for an overview). GPU architectures are designed for parallel workloads and have high computational throughput. They contain several streaming multiprocessors (SMs), each of which has many small cores that execute instructions in a SIMT (single-instruction, multiple threads) manner. GPU cores work best when memory accesses are contiguous, as they share load/store units that coalesce memory requests. For example, [55] shows that fully connected DNNs, containing large matrix multiplications are 10-100× faster on a GPU system than on a CPU system. However, the shared load/store units on GPUs are unable to efficiently service irregular memory accesses; and divergent control flow across threads will cause sequentialized execution under the SIMT execution model.

c) FPGAs: Field-Programmable Gate Arrays (FPGA) are increasingly being integrated into SoCs due to their gate-level emulation capability which enables post-silicon hardware specialization. Unlike CPUs and GPUs which are time-multiplex generic ALUs with sequentially-executed instructions, FPGA-accelerated datapaths are spatially mapped onto the reconfigurable fabric. As a result, although FPGA-emulated accelerators rarely achieve the same amount of data-level parallelism as GPUs, they are capable of exploiting pipeline parallelism. For example, it has been shown that for some computer vision tasks, FPGAs can outperform CPUs by up to 22.3× and GPUs by up to 3.0× in terms of energy per frame [42].

III. Redwood: A Framework for Developing Heterogeneous Traverse-compute Workloads

We now present Redwood: a framework for developing heterogeneous traverse-compute workloads. The core insight is that the traversal can be performed on the CPU while the leaf node computations can be performed on the PAPU.

Redwood is made up of three components: (1) a backend, which can be instantiated for different heterogeneous programming platforms; (2) tree data structures, which have a configurable height; and (3) a leaf node computation API and runtime that allows heterogeneous computation, even on workloads where traversals are dependent on leaf node computations.

A. Redwood Backend

The Redwood backend abstracts several common features in heterogeneous programming platforms, and thus, we argue can easily be instantiated. We highlight this in Sec. III-A by discussing implementations for three programming platforms: CUDA, SYCL, and HLS. The different abstractions are:

a) Unified Shared Memory (USM): This memory allows data to be accessed both on the CPU and PAPU in a SMHS requiring no copying. Redwood requires only a simple `Redwood_malloc` and `Redwood_free` which returns a pointer to a USM memory region.

b) Accelerator Work Queue (AWQ): Redwood requires an AWQ to launch a PAPU kernel. To enable heterogeneous computing, the kernel launch must be asynchronous, so that the CPU can perform tree traversal tasks while the accelerator is performing compute tasks. To provide this, Redwood provides an `Redwood_AWQ` object which can `launch` a task, and `sync` a task that was previously launched.

c) Compute Routines: Redwood is built on a small number of common PAPU compute routines:

- **distance metrics**: because traverse-compute workloads can utilize different distance metrics, the Redwood backend requires a PAPU distance computation routine to compute the distance between two points. These routines are typically very simple, consisting only of a few lines of arithmetic.
- **interaction computation**: some workloads require an interaction in addition to the distance, e.g., for particle simulations. Again, these routines are typically only a few arithmetic instructions.
- **reduction**: this routine performs a reduction across an array of values, using a configurable binary operator. Reductions have been widely studied for PAPUs [17] and optimized implementations are often available in libraries.
- **sort**: some traverse-compute workloads (e.g., KNN) can be implemented using sorting. This can be mapped to the PAPU if an efficient implementation is available. Much like reduction, sorting has been widely studied for PAPUs, e.g., GPUs [48] and FPGAs [61].
- **batched reduction**: this routine is simply a reduction applied over a batch of arrays. It is useful when a single reduction is too lightweight to justify the overhead of a PAPU kernel launch.

B. Redwood Data Structures

Redwood provides three tree structures that can be used as the basis of the traverse-compute workloads. Each tree takes in a point cloud to partition, along with a configurable set of data to store for each node. The trees provide a common API, such as the ability to retrieve the root node and query the children from a node. Redwood provides a *k-d tree*, which is the most general spatial decomposition tree. It can structure point clouds in arbitrary dimensions. Redwood also contains quadtrees and octrees, which are restricted to point clouds in $\mathbb{R}^3$ and $\mathbb{R}^2$, respectively. Leaf node data is allocated in USM (allocated using the Redwood backend) so that the PAPU can easily access the memory without requiring any explicit copies. If a bounding box has fewer particles than the leaf node is configured with, then dummy particles are stored, with a special data value as an identifier.

a) Flexible Heterogeneous Decomposition: Redwood trees are parameterized by a leaf node size. This is important so that Redwood applications can be efficiently executed across a wide range of SMHSs that may have different relative PU throughputs. For example, systems that have less relative PAPU throughput can configure the tree to have smaller leaf node sizes, as this will require less leaf node computation (performed on the PAPU) and more tree traversals (performed by the CPU). We show that different SMHSs and different applications have different optimal leaf node sizes in Sec. VI-A.

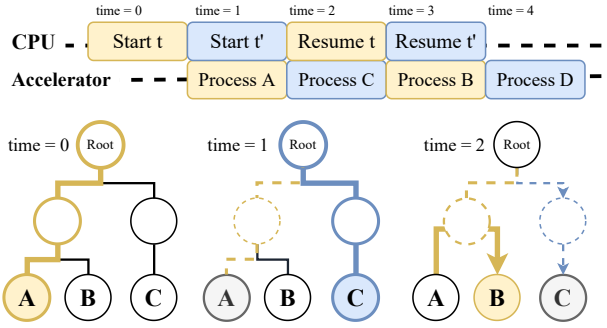

Fig. 3. Traverser execution timeline: At each time stamp, the CPU traversal trace is highlighted with a solid colored line. The leaf node that is being processed by the PAPU is shaded in grey. The dotted color lines are traversals that are suspended and will be resumed later.

We note that there is a computational trade-off space when increasing leaf node size. Spatial partitioning trees essentially prune the computation space. Shallow trees have less pruning, and thus will require extra computations. For example, Fig. 2 shows the shallow tree as having 9 point computations, with the deeper tree having only 4. However, as our results will show (Sec. VI), this extra computation can result in significant end-to-end performance increases as PAPUs are extremely efficient at the required computation, and irregular memory accesses in tree structures have significant overhead.

C. Node Computation API and Traversers

Recall that a traverse-compute workload has query points P and a tree $T(Q)$. A query point $p \in P$ starts at the root of $T(Q)$ and traverses the tree. When the traversal reaches a leaf node, the node computation could be executed immediately. However, to enable heterogeneous optimizations, Redwood provides an asynchronous API pair: `Compute_leaf_node_start` and `Leaf_nodes_sync`. The first call takes in three arguments: an `Redwood_AWQ`, the query point, and a leaf node; it registers a leaf node computation with Redwood. The second API call takes a `Redwood_AWQ` as an argument and waits until all leaf node computations on the `AWQ` are finished. This allows the CPU to continue traversing the tree while the PAPU executes the leaf node computation. In an unguided traversal, the CPU can simply call `compute_leaf_node_start` on each leaf node as they are visited, only calling `leaf_nodes_sync` at the end of all the query points.

a) Traversers: For a guided traversal, i.e., the traversal is dependent on the leaf node computations, the asynchronous leaf node computation calls are not sufficient to enable the CPU and PAPU to execute in parallel. This is because the

traversal must wait for the leaf node computation before it can continue. To address this, we developed *traversers*: lightweight co-routines that encode a query point and its traversal state. A traverser can suspend and resume its traversal at leaf nodes.

Figure 3 shows how traversers can enable heterogeneous computation: a traverse-compute workload can start the traversal for query point p using traverser t (shown in yellow); this traversal can then start a leaf node computation (node A) and then suspend t. Another traverser t' on query point p' can then start (shown in blue). This independent traversal can traverse to a leaf node (labeled C) and register the leaf node computation. At that point, t' can suspend its traversal, and t can resume. At this point, t's leaf node computation may be finished, so t can continue its traversal. The timeline shows how this enables the CPU and PAPU to execute in parallel. Traversers are implemented using a lightweight data structure to record the query point and a pointer to its recursive traversal stack. Suspending a traverser returns the traverser object from the traverse-compute routine. The traverse-compute routine can be restarted with a traverser object. In our experiments, we found our traverser implementation to have roughly $2\times$ less overhead than C++ co-routines.

b) Batching Computations: Leaf node computations may be small, and as such, the computation might not contain sufficient parallelism to fully utilize the PAPU; additionally, on some platforms, the overhead of launching a kernel may be prohibitive for small workloads (discussed in Sec. VI-C). To address these issues, we implement batched computations in Redwood. When a call to `compute_leaf_node_start` is executed, Redwood does not immediately start executing the node computation. Instead, it batches a configurable number of leaf nodes in USM. Redwood also implements ping-pong buffering; once one buffer is filled, the PAPU kernel is launched and a second buffer starts to be populated. When a batch is full, one PAPU kernel is launched to compute all the leaf nodes. This optimization takes advantage of the batched reduction computation required in the backend.

As an additional optimization, Redwood will detect if more than one leaf node is registered with the same query point (which is the usual case for unguided traversals). In these cases, the reductions can be merged into a larger reduction, which is more efficient than many small reductions.

IV. Grove: Benchmarks

Using Redwood, we implement a set of traverse-compute workloads called Grove, summarized in Tab. I, which have applications in scientific computing (particle simulation) and supervised learning (KNN). We categorize our benchmarks into three high-level algorithms, which are then instantiated with different node computations.

A. Grove Algorithms

a) Nearest Neighbor (NN): Given two point clouds P and Q, for each $p \in P$, NN finds its closest neighbor $q \in Q$. This algorithm iteratively performs a traverse-compute routine on $T(Q)$ for each $p \in P$. Given a leaf node L, the leaf node computation consists of computing the distance from p to every point $l \in L$ and performs a `min` reduction over the distances. The traversal then pops up the tree and determines if more leaf nodes need to be searched. Because of this dependency, this algorithm has a guided traversal. Thus, a configurable number of traversers are used to execute different points from P. In Grove, we implement NN using a k-d tree and experiment with particles in $\mathbb{R}^4$.

TABLE I
Summary of Grove: our traverse-reduce benchmark suite implemented with Redwood

Alg.	Input	Distance/Interaction	Array Analysis	Tree
BH	$\mathbb{R}^3$	Euclidean + Gravity Euclidean + Gaussian Euclidean + TopHat	reduction `sum`	octree
NN	$\mathbb{R}^4$	Euclidean Manhattan Chebyshev	reduction `min`	kd-tree
KNN	$\mathbb{R}^4$	Euclidean Manhattan Chebyshev	sort `min`	kd-tree

b) k-Nearest Neighbor (KNN): A natural extension to NN is KNN, which finds the k nearest neighbors. Instead of performing a reduction over the distances, the best k neighbors are appended to the leaf node points and sorted. The first k elements of the sorted array are then kept. While this might not be the optimal way to implement KNN, we utilize this approach due to well-supported sorting routines for PAPUs. This algorithm could be improved in future work by implementing a heterogeneous heap (e.g., following [9]). KNN is also guided and requires traversers. We implement KNN using a k-d tree and experiment with particles in $\mathbb{R}^4$. To ease our implementation, we fix k to be 32.

c) Barnes-Hut (BH): BH is an algorithm for approximating particle simulations. Given a set of particles P, every $p \in P$ needs to compute a force interaction with every other $p' \in P$. Directly performing this computation requires $|P|^2$ interaction computations. However, the BH algorithm allows the force to be approximated by particles that are sufficiently far apart from each other. For each $p \in P$, a traverse-compute routine is performed on a spatial partitioning tree of P, $T(P)$. At each interior node, p checks its distance to the centroid of the bounding box B. If the distance is far enough (set by a parameter θ), then the particles in B are approximated.

Unlike in the NN algorithms, BH doesn't require traversals ever actually reach leaf nodes. Thus, increasing the leaf node size does not have as much of a dramatic increase in performance. However, our heterogeneous implementation can still provide a considerable speedup (Sec. VI). Additionally, BH has an unguided traversal pattern as the traversal does not change from the results of a leaf node computation. Thus BH is implemented without traversers. In Grove, we instantiate BH in $\mathbb{R}^3$ using an octree. We set θ to 0.01, which is in similar to other works [5].

B. Node Computations

a) Distance Metrics: Each algorithm can be instantiated with different distance metrics. The different metrics can change the intensity of node computations, which has a significant impact on the speedups (discussed in Sec. VI-B). For NN and KNN, Grove implements three distance metrics; given two points p and w (and their coordinate vector $\vec{v}$) the three metrics are:

- Euclidean (Euc): $K(p.\vec{v}, w.\vec{v}) = \sqrt{\sum (p.\vec{v} - w.\vec{v})^2}$
- Manhattan (Man): $K(p.\vec{v}, w.\vec{v}) = \sum |p.\vec{v} - w.\vec{v}|$
- Chebyshev (Che): $K(p.\vec{v}, w.\vec{v}) = \max |p.\vec{v} - w.\vec{v}|$

b) Interaction: BH requires an interaction computation on top of the distance computation. In scientific computing, the interaction is often a particle force, like gravity. However, BH can also be used in learning applications, e.g., for kernel density estimation [54], which takes a point cloud and estimates its probability density. The interaction can be a variety of probability functions. Grove implements three interactions for BH: one scientific interaction and two learning interactions:

- Gravity (Gra): $K(p, w) = -G\frac{m_p m_w}{|r_{pw}|^2}\hat{r}_{pw}$, where G is the gravitational constant, m_p, m_w is mass of particle p, w, and $|r_{pw}|$ is the distance between p, w, and $\hat{r}_{pw}$ is the unit vector from p to w.
- Gaussian (Gau): $K(x) = \exp(-\frac{x^2}{2h^2})$, where h is a constant, and x is the euclidean distance.
- Top hat (Top): $K(x) = 1$ if $x < h$, where x is the same as in Gaussian.

V. Experimental Methodology and Platforms

A. Backend Implementations

We implement the Redwood backend for three SMHS programming platforms: CUDA, SYCL, and HLS. In all cases, the basic node computation kernels (excluding reductions are sorting) were implemented using simple functions. We compile Redwood front end with `g++` with optimization level `-O3`. We link the front end to the corresponding back end once it is compiled with its required framework.

a) CUDA: We implement USM with CUDA's unified memory [37, App. N]; on Nvidia's SMHSs, e.g., the Jetson series [38], this memory is directly shared between the CPU and GPU, without requiring explicit copies. The `Redwood_AWQ` is implemented using CUDA streams [37, Sec. 3.2.7.5]; which allows for asynchronous kernel launches and synchronization. For reductions and sorting, we utilize the high-performance CUB library [32]. The CUDA backend is compiled with `nvcc`.

b) SYCL: We implement USM with SYCL's unified memory [15, Sec. 4.8.3.4]. The `Redwood_AWQ` is implemented using SYCL queues [15, Sec. 4.6.5]. We were unable to find optimized reduction kernels for SYCL, so we implemented batched reductions to be per-thread parallel over each batch. For larger reductions, we implemented a tree reduction. We perform sorting on the CPU, as we were unable to find optimized SYCL sorting kernels. The SYCL backend could be improved if optimized libraries were provided. The SYCL backend is compiled with Intel's OneAPI framework.

c) High-Level Synthesis: The tightly-integrated CPU-FPGA system we target can straightforwardly share CPU-allocated memory, and peak throughput can be achieved by reading/writing an entire cache line per memory access. Thus USM can be implemented with a simple `aligned_alloc` call. We implemented our kernels using Algorithmic C [46], synthesized them with Catapult HLS [47], and recorded the post-HLS timing information. To utilize an optimized sorting network for KNN, we utilize the Spiral Project: Sorting Network IP Generator [61], which generates customized sorting networks in synthesizable RTL Verilog.

B. Evaluation Platforms

We evaluate Grove on five platforms, summarized in Tab. II: two Nvidia Jetsons, two Intel chips, and one experimental, tightly-integrated CPU/FPGA system called Duet [27]. The Nvidia and Intel devices can simply run the Grove benchmarks that have been compiled with their backend.

Because the Duet system is new, it is most easily evaluated using simulation. To simulate tightly-integrated FPGA accelerators, we used the gem5 [4] simulator with a Duet extension [26]. We chose Duet over commodity CPU/FPGA systems (e.g., Xilinx ZynQ [56]) as we are particularly interested in the combination of many-core processors and multiple small, embedded FPGAs. Most commodity CPU/FPGA systems are FPGA-centric SoCs with a monolithic FPGA and a few cores, and their acceleration paradigm is similar to the large GPU-oriented systems. In contrast, the Duet system represents a very different family of SMHSs that have a low accelerator invocation overhead, facilitate fine-grained memory sharing, and excel at pipelined parallelism. To represent realistic offload overhead, the simulation models multistage asynchronous FIFOs, and CPU/FPGA clock penalties are also accurately modeled with 1.5GHz/333MHz.

C. Evaluation Methodology

Inputs are synthetically generated data of one million points in the $\mathbb{R}^3/\mathbb{R}^4$, in uniform and multivariate Gaussian distributions (Sec. VI-D). Each Grove benchmark was run three times on each system and an average was taken, although we did not observe high variance, which has been reported for some SMHSs [1]. The runs for Duet were only executed once as it executes on a cycle-level simulator. Additionally, given that simulation time can be prohibitively long, we reduced the dataset size by a factor of 10.

We do a run of pilot experiments and find the best reduction batch size for each platform to be 1024, 512, and 1, for Nvidia, Intel, and Duet, respectively. Using ping-pong buffering, this leads to 2048, 1024, and 2 traversers for each platform, respectively. Duet does not benefit from batching as it is optimized for pipelining and does not have kernel launch overhead like GPU systems have. We then run all benchmarks with leaf node sizes varying between 32 and 1024 (considering powers of two). As a homogeneous baseline, we implement a sequential CPU backend. We sweep through the leaf node

TABLE II
SHARED MEMORY HETEROGENEOUS SYSTEMS EVALUATED WITH GROVE

Device	Backend	CPU	Accelerator
Nvidia Jetson Nano	CUDA	ARM Cortex-A57	128-core Maxwell
Nvidia Jetson Xavier	CUDA	Carmel ARMv8.2	384-core Volta
Intel i7-9700K	SYCL	i7-9700K	Intel UHD Graphics 630
Intel i5-10210U	SYCL	i5-10210U	Intel UHD Graphics 620
Duet	HLS	RISC-V TimingSimpleCPU	Duet FPGA

sizes to find the best homogeneous CPU configuration. Our baseline does not incorporate traversers or batches.

In this work, we consider only the traverse-compute portion of the application and discount the cost of building the tree. Indeed, our pilot experiments show that the traverse-reduce routine is over $150\times$ more costly than building the tree. However, building the tree will start to become the bottleneck once traversing is optimized enough. In future work, we hope to explore optimizing tree construction, e.g., following [7].

Finally, we note that Redwood is not meant to provide the most optimal implementation of traverse-reduce workloads. Instead, Redwood and Grove are meant to provide a benchmark suite to show the potential for heterogeneous computing for this workload domain. Our results capture heterogeneous performance characteristics, and our design captures important abstraction insights. Future work can further optimize Redwood, e.g., using optimizations discussed in [52].

VI. BENCHMARKING SMHS

We begin by highlighting the potential of heterogeneous computing for traverse-compute workloads. Figure 4 shows the speedup of heterogeneous implementations of Grove over homogeneous implementations, in both cases, using their optimal leaf node sizes. We discuss the results per system:

- **Duet** has the highest speedups among all platforms, with a $13.53\times$ highest speedup and a $6.43\times$ geomean speedup. This is because Duet is a simple in-order CPU, which doesn't have the ability to tolerate latency from irregular tree-traversal computations. Thus, it is highly beneficial to offload a high amount of computation to the FPGA.
- **Nvidia** has the next highest speedups, e.g., with the Xavier achieving an $8.12\times$ highest speedup and a $5.13\times$ geomean speedup, and the Nano achieving a $6.9\times$ highest speedup and a $4.71\times$ geomean speedup. This is because the powerful Nvidia GPUs can offset the cost of the node computations from the smaller ARM CPUs.
- **Intel** has the smallest speedups, and sometimes slowdowns, with KNN on the i5 slowing down by almost $2\times$. However, NN on the same device almost achieves a $3.36\times$ speedup. The Intel CPUs are very powerful, and the Intel GPUs are relatively small, thus there is less to be gained by offloading node computations.

A. Impact of Leaf Node Flexibility

The flexible data structures provided with Redwood allow us to highlight how the leaf node size, i.e., which allows a flexible heterogeneous decomposition, can influence the performance of Grove. Table III shows the optimal leaf node sizes for each application on heterogeneous and homogeneous implementations. On average, PAPUs have $4.33\times$ bigger optimal leaf node sizes than CPU baselines. In computing intense workloads like BH, the optimal leaf size between PAPUs can have up to a $256\times$ difference (Duet BH Gaussian).

The same workload can have different optimal leaf node sizes on different platforms. Figure 5 shows the breakdown of BH Euclidean ran on each platform. The Intel i5 achieves at most $1.45\times$ speedup at leaf size 64, it drops immediately and turns into a slowdown after leaf size 256. In contrast, Duet has the optimal leaf size at 512, achieving a $7.58\times$ speedup, $5.22\times$ faster than i5-10210 at its optimal configuration.

B. Impact of Distance Metrics

The abstractions provided in Redwood allow us to evaluate the performance impact of different intensities of node computations. Using one platform from each backend, Fig. 6 shows how different distance metrics affect the performance of the NN applications. We show results across a selection of leaf nodes to show that the different distance metrics retain the optimal leaf node size (per platform), but have different speedup potentials. We see that the Chebyshev distance, whose computation only consists of low-cycle instructions, has less performance gain than Euclidean distance, which contains square root floating point operations.

The shape of these graphs over different leaf node sizes shows that performance increases as the PAPU is able to handle more and more computation. At high leaf node sizes, the shallow tree will prune too little of the traversal space, and the extra computation overwhelms the PAPU, as seen with the performance drop.

C. Impact of Batching

We next examine the impact of batching leaf nodes. We found that batching is vitally important to the CUDA and SYCL backend, for different reasons. In CUDA, we found the kernel launch overhead was significant (as has been reported, e.g., in [33]). Figure 7 shows the breakdown CPU execution time spent in NN Euclidean when run on the Nano with different batching sizes (measured with `nvprof`). In the case of a batch size of 32, 83% of the time was used on overhead (kernel launch overhead and synchronization). When we increase the batch size to 1024, the overhead was reduced to 16% of the time. On the other hand, SYCL has little overheads from kernel launching, i.e., consistently $\leq 3\%$ according to *VTune*. However, batching still allows our batched reduction kernel to exploit more parallelism, and thus, we found an optimal batch size for SYCL to be 512.

The Duet system has even lower overhead because eFPGA-emulated accelerators are invoked by one or very few accesses to certain memory-mapped control registers. In addition, since the Duet system implements bi-directional cache coherence between the CPUs and the eFPGAs, there is no need to explicitly batch and transfer data. Thus, leaf node data are implicitly shared with the eFPGA-emulated accelerators and

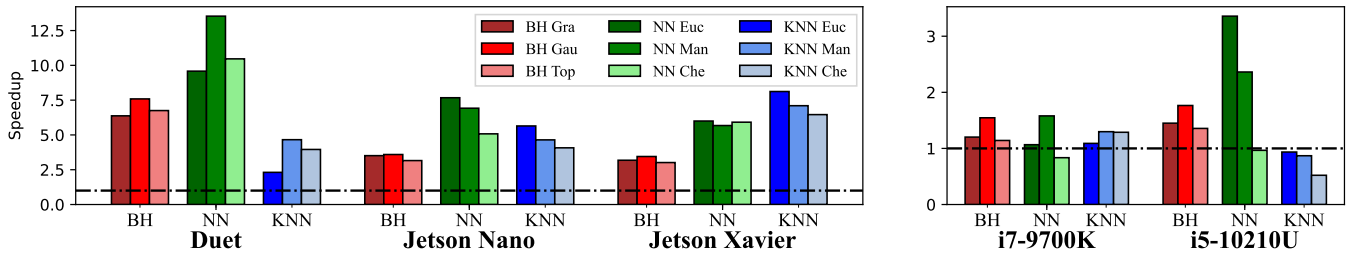

Fig. 4. Speedups of the best heterogeneous configuration vs. the best homogeneous configuration of Grove.

TABLE III
OPTIMAL LEAF NODE SIZES FOR HETEROGENEOUS IMPLEMENTATION (LEFT-HAND SIDE) AND HOMOGENEOUS CPU IMPLEMENTATION (RIGHT-HAND SIDE) FOR EACH WORKLOAD AND PLATFORM. THE OPTIMAL LEAF NODE SIZES ARE DETERMINED BY RUNNING THROUGH EACH CONFIGURATION.

	BH Gra	BH Gau	BH Top	NN Euc	NN Man	NN Che	KNN Euc	KNN Man	KNN Che	Average	Ratio
Jetson Nano	256/8	256/8	128/8	512/256	512/256	512/256	256/128	256/128	256/256	327/115	2.26
Jetson Xavier	128/16	128/8	128/8	1024/128	1024/256	512/64	512/64	512/64	512/64	498/75	6.67
i7-9700k	64/8	128/8	128/8	256/64	256/64	128/64	64/64	64/64	64/64	128/45	2.82
i5-10210U	64/8	64/8	64/8	256/64	256/64	256/64	32/64	32/64	32/64	117/45	2.59
Duet	512/4	512/2	512/4	512/32	512/32	256/16	128/16	128/32	128/32	355/19	18.82
Average	205/9	218/7	192/7	512/109	512/134	333/93	198/67	198/70	198/96	285/66	4.33

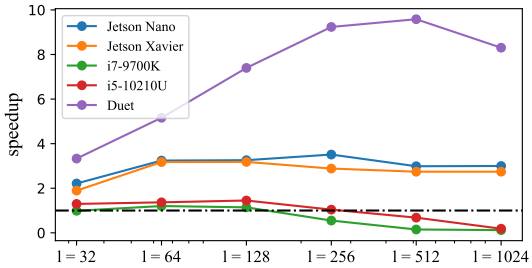

Fig. 5. Speedups of BH Gau application across different leaf node sizes on our five evaluation platforms.

passed simply by writing the base memory addresses into the aforementioned control registers.

D. Impact of Input Distribution

We consider input particle distribution in terms of *spatial sparsity*, where the input data can impact the depth and irregularity of a constructed spatial partitioning tree. We ran our tests on both uniform distributions, in which the particles have evenly distributed across the computation domain (balanced tree), and multivariate Gaussian distribution, which forms clustered particles (imbalanced tree). We observed the sparse dataset generally takes longer to complete, as the nature of BH and NN/KNN algorithms do (e.g., uniform NN is $1.24\times$ faster than multivariate Gaussian NN). However, the speedup ratio remains the same across systems and applications.

E. Impact of Threading

We now experiment with scaling Grove benchmarks to multiple CPU threads. In this experiment, each CPU thread has its own accelerator work queue, and thus each thread can submit leaf node computations to the PAPU. We use one representative platform from each backend and experiment with 1, 2, 4, and 8 threads on the NN Euclidean benchmark, and show two leaf node sizes: the optimal size for 1 thread (as shown in Tab. III) and the minimal size of 32. Our results are shown in Fig. 8, with the baseline for each platform/leaf-node size pair being the single-threaded heterogeneous runtime.

We observe different trends for different platforms. For Duet and Nvidia, we see that when the optimal leaf node size is picked, there is no speedup gain (horizontal lines). Thus, this leaf node size is saturating the PAPU with one thread; adding more threads does not provide any improvement. However, they still have some scalability when a smaller leaf size is picked, as the PAPU is no longer saturated and can take work from additional threads. We see that the Intel device can scale to multiple threads, as the PAPU is not fully saturated (potentially due to our less optimal SYCL kernel implementations). In future work, we hope to have Redwood configurable so that some CPU threads might perform the homogeneous traverse-reduce computation if the PAPU is saturated.

F. Heterogeneous System Design Insights

Our results from running Grove across our evaluation platforms have provided several insights, both for new heterogeneous system designers, as well as optimization targets for existing systems.

a) PU Relative Throughputs: Our Duet backend achieved the most speedups, where only simple in-order RISC-V cores were used, which lack the hardware components to tolerate long memory latency as seen in OoO cores like the Intel ones. OoO cores are generally more complex and larger

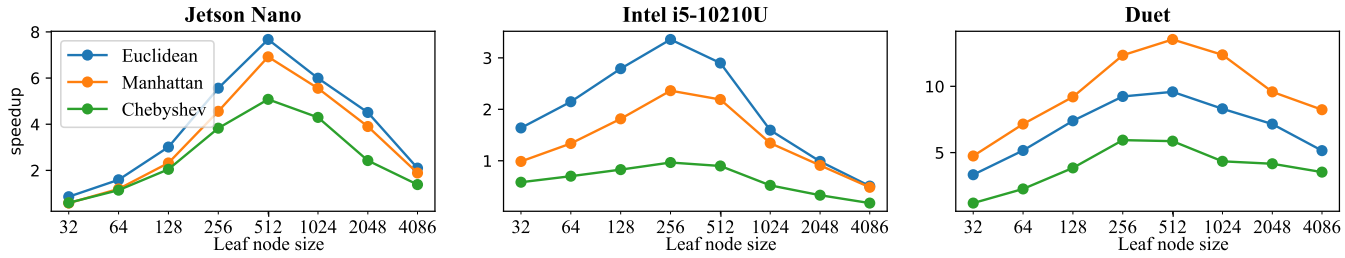

Fig. 6. The choice of the backend can also have a significant impact on the performance of distance metric calculations. We found Manhattan distance performs better on FPGA, showcasing the different characteristics of the underlying architecture.

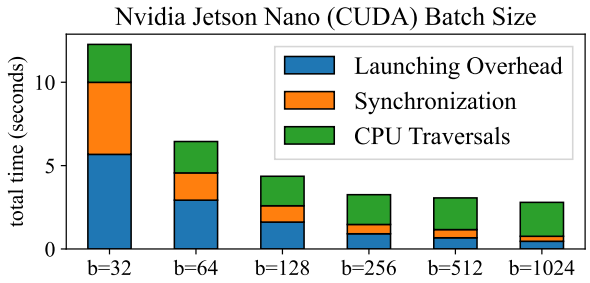

Fig. 7. Batch size affecting CUDA backends running the NN Euclidean with leaf size 1024. The less number of batches, the more kernel launching and synchronization overhead.

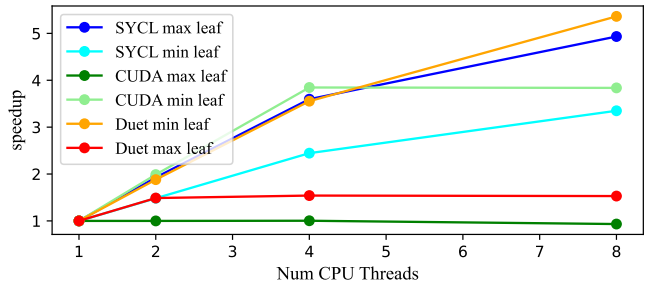

Fig. 8. Scaling heterogeneous implementations to multiple threads

in die area, and consume more power. But a combination of simple in-order cores and accelerators, it may be possible to achieve similar or even better performance compared to OoO cores on the same die area. This could result in more energy-efficient and cost-effective systems that are better suited to the needs of resource-constrained environments. Future research could explore the design space exploration of simple in-order cores and accelerators to identify the optimal configurations for specific workloads using Redwood.

b) Low-cost Kernel Submission: Having a low-cost kernel submission can benefit systems to efficiently execute tree traversal workloads on SMHSs, as had in Duet FPGA. Kernel launch overhead can be expensive on GPU backends. As seen in Section VI-C, there is potential for further optimization of kernel submission on GPUs to gain additional performance. Applying the persistent thread approach [16] to GPU kernel submission could potentially lead to further gains in performance, especially for workloads that involve frequent small kernel launches.

G. Integrated vs. Discrete

We conclude our results by discussing if Redwood is useful for discrete PAPU systems, i.e., which would require memory transfers, as shown in Figure 1-d. Because Nvidia allows the unified memory abstraction on discrete GPUs (with memory copies happening behind the scenes), we can run Grove on discrete systems. We execute the NN Euclidean application on a Quadro RTX 4000 with an Intel i7 CPU. We still observe a $3\times$ speedup on the heterogeneous implementation over the homogeneous implementation; thus, Redwood can also be useful in this domain.

However, we believe Redwood is still most suited for SMHSs for several reasons: (1) discrete GPU systems often have very large, powerful GPUs. In which case it may be more beneficial to use a homogeneous PAPU-based implementation, such as [6]. Additionally, the memory transfer overhead requires additional consideration; at low batch sizes, we found that memory transfers were taking over 30% of the total execution time. Because of this, we aim to continue prioritizing SMHSs as the main target of Redwood.

VII. Related Work

As noted in Sec. I, there have been many works on heterogeneous computing, as seen in this survey [33]. Here, we focus on the most closely-related works and themes.

a) Homogeneous Tree Traversals: Prior works on tree applications were optimized for only one type of PU, e.g., BH implemented solely on the CPU in [59], in discrete GPUs [2], [6], and in discrete FPGAs [11]. Sparse tree/graph workloads such as Unbalance Tree Search [20] and Lonestar [25] have been studied. However, these works are also GPU-centric implementations, which is illustrated in Fig. 1-a.

None of these works fully utilize the PU resources that are available on a SMHS. Additionally, these works are highly specialized to their specific architecture and do not provide abstraction layers that support portability, as Redwood does. Other works have proposed further optimizations to tree traversal computations, e.g., [52]. While they are evaluated on CPU-

only platforms, we hope to incorporate these optimizations into future versions of Redwood.

b) Adaptive Heterogeneous Task Scheduling: Another line of work utilizes the *workload partition* collaboration pattern (as defined in [51]), where identical tasks (e.g., iterations from a `parallel_for` construct) are executed across both a CPU and GPU system, as illustrated in Fig. 1-b. This has been implemented for the BH application in [24], [36], [53]; these approaches use adaptive partitioning strategies so that runtime is equalized across CPU and GPU. Other prior works [10] have used a more dynamic approach, e.g. work-stealing so that each PU is more fully occupied.

However, in the above approaches, the CPU and GPU are performing identical tasks that are not specialized to their respective architecture. Redwood maps tasks to PUs that can efficiently compute them: traversals on CPUs and node computations on PAPUs. Furthermore, these works have yet to examine tree traversal workloads in-depth, as we have done in this work. Tree workload naturally has specialized heterogeneous decomposition, knowing which parts of the application map best for each PU.

c) Specialized Heterogeneous Decomposition: Other works map subtasks to different PUs based on where they are most efficiently computed, as illustrated in Fig. 1-c. For example, [21] proposes a decomposition for the Fast Multipole Method for the N-body problem (similar to BH). Much like Redwood, they map node computations to the GPU and data structure traversal to the CPU. Another related approach in [28] decomposes medical image analysis into regular and irregular tasks, which are mapped to the CPU and GPU, respectively. However, their work does not have a *flexible* decomposition, which enables the workload to be evenly split between the different PUs of a SMHS, as is provided by Redwood. This is essential for performance portability given the variety of different heterogeneous systems.

d) Accelerating Workloads on SMHSs: To address the expensive data movement in a heterogeneous workload, [29], [34], [58] propose data-movement engines to overcome data transfer overhead on discrete heterogeneous systems. However, these are hardware solutions, whereas Redwood is a software solution, that works on existing SMHSs.

Other works have investigated accelerating workloads on SMHSs, such as radix sort [12] and key-value stores [19]. Both workloads show that low CPU-GPU data transfer overhead is a key advantage in shared memory CPU-GPUs, and is more energy efficient than discrete GPUs [33]. However, these works have not yet examined tree traversal workloads in-depth, as we have done in this work.

VIII. Conclusions

In this work, we identified traverse-compute workloads as an ideal heterogeneous workload, where CPUs and PAPUs can efficiently collaborate using distinct workloads that fit their unique architectural strengths. We developed Redwood: a framework that has a simple backend and allows for flexible heterogeneous decompositions so that traverse-compute workloads can be tuned across the ever-growing diversity of shared-memory heterogeneous systems. We implemented nine benchmarks using Redwood and evaluated five SHMS backends, highlighting the importance of configurable heterogeneous parameters and showing significant heterogeneous speedups for this workload domain.

IX. Acknowledgment

We thank the anonymous reviewers for their feedback, which greatly improved the clarity and rigor of this work. We also thank the research team behind the DECADES project, with PIs David Wentzlaff, Margaret Martonosi, and Luca Carloni, for their support and feedback. We especially thank the software subgroup, with members: Esin Tureci, Marcelo Orenes Vera, Aninda Manocha, Naorin Hossain, Stojche Nakov, and Juan Luis Aragón for their frequent discussions around this work. This research was supported in part by the DARPA SDH Program under agreement No. FA8650-18-2-7862, NSF Award No.1763838, and the U.S. Government. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Sergio Afonso and Francisco Almeida. Rancid: Reliable benchmarking on android platforms. *IEEE Access*, PP:1–1, 08 2020.

[2] Robert G Belleman, Jeroen Bédorf, and Simon F Portegies Zwart. High performance direct gravitational N-body simulations on graphics processing units ii: An implementation in CUDA. *New Astronomy*, 13(2):103–112, 2008.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[5] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. *Parallel Algorithms*, 30, 1997.

[6] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU computing Gems Emerald edition*, pages 75–92. Elsevier, 2011.

[7] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *SIGGRAPH*, pages 57–64. Eurographics Association, 2008.

[8] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. SLIDE : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems, 2019. arXiv, 10.48550/ARXIV.1903.03129.

[9] Yanhao Chen, Fei Hua, Chaozhang Huang, Jeremy Bierema, Chi Zhang, and Eddy Z. Zhang. Accelerating concurrent heap on GPUs, 2019. arXiv: 10.48550/ARXIV.1906.06504.

[10] Lin Cheng. Intelligent scheduling for simultaneous CPU-GPU applications. 2017.

[11] James Coole, John Wernsing, and Greg Stitt. A traversal cache framework for FPGA acceleration of pointer data structures: A case study on barnes-hut n-body simulation. In *International Conference on Reconfigurable Computing and FPGAs*, pages 143–148. IEEE, 2009.

[12] Michael Christopher Delorme. *Parallel sorting on the heterogeneous amd fusion accelerated processing unit*. PhD thesis, University of Toronto, 2013.

[13] Davide Giri, Paolo Mantovani, and Luca P Carloni. Accelerators and coherence: An SoC perspective. *IEEE Micro*, 38(6):36–45, 2018.

[14] Alexander Gray and Andrew Moore. N-body'problems in statistical learning. *Advances in neural information processing systems*, 13, 2000.

[15] The Khronos SYCL Working Group. SYCL 2020 Specification (revision 6), November 2022.

[16] Kshitij Gupta, Jeff A Stuart, and John D Owens. *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[17] Mark Harris et al. Optimizing parallel reduction in CUDA. *Nvidia developer technology*, 2(4):70, 2007.

[18] Nikhil Hegde, Jianqiao Liu, Kirshanthan Sundararajah, and Milind Kulkarni. Treelogy: A benchmark suite for tree traversals. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 227–238. IEEE, 2017.

[19] Tayler H Hetherington, Timothy G Rogers, Lisa Hsu, Mike O'Connor, and Tor M Aamodt. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 88–98.

[20] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous-race-free memory models. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 427–440, 2014.

[21] Qi Hu, Nail A Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[22] Tianyu Jia, Paolo Mantovani, Maico Cassel Dos Santos, Davide Giri, Joseph Zuckerman, Erik Jens Loscalzo, Martin Cochet, Karthik Swaminathan, Gabriele Tombesi, Jeff Jun Zhang, et al. A 12nm agile-designed soc for swarm-based perception with heterogeneous ip blocks, a reconfigurable memory hierarchy, and an 800mhz multi-plane noc. In *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pages 269–272. IEEE, 2022.

[23] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17. Association for Computing Machinery, 2017.

[24] Ji Yun Kim and Christopher Batten. Accelerating irregular algorithms on GPGPU using fine-grain hardware worklists. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 75–87. IEEE, 2014.

[25] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76. IEEE, 2009.

[26] Ang Li. gem5-duet: A Gem5-based Simulator for Tightly-Integrated CPU-FPGA Systems. https://github.com/angl-dev/gem5-duet.

[27] Ang Li, August Ning, and David Wentzlaff. Duet: Creating Harmony between Processors and Embedded FPGAs (to appear). In *The 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)*, 2023.

[28] Yixun Liu, Andriy Fedorov, Ron Kikinis, and Nikos Chrisochoides. Real-time non-rigid registration of medical images on a cooperative parallel architecture. In *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 401–404.

[29] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2020.

[30] Paolo Mantovani, Emilio G Cota, Christian Pilato, Giuseppe Di Guglielmo, and Luca P Carloni. Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 1–10, 2016.

[31] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic, 1980.

[32] Duane Merrill. Cub. *NVIDIA Research*, 2015.

[33] Sparsh Mittal and Jeffrey S Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[34] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1009–1022, 2019.

[35] R Muralidharan. Object recognition using k-nearest neighbor supported by eigen value generated from the features of an image. *International Journal of Innovative Research in Computer and Communication Engineering*, 2(8), 2014.

[36] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *The Journal of Supercomputing*, 70(2):756–771, 2014.

[37] Nvidia. CUDA C++ Programming Guide, Version 11.8, November 2022.

[38] Leela S. Karumbunathan (Nvidia). Nvidia Jetson AGX Orin Series, 2022. Technical Brief.

[39] Pallabi Parveen and Bhavani Thuraisingham. Face recognition using multiple classifiers. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 179–186. IEEE, 2006.

[40] Yuliang Pu, Jun Peng, Letian Huang, and John Chen. An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 167–170. IEEE, 2015.

[41] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 13–24. IEEE Press, 2014.

[42] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8.

[43] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[44] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[45] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. The aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3), 2015.

[46] Siemens. Algorithmic C. https://github.com/hlslibs.

[47] Siemens. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[48] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, 2018.

[49] Tyler Sorensen and Alastair F. Donaldson. The hitchhiker's guide to cross-platform opencl application development. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16. Association for Computing Machinery, 2016.

[50] Ming-Yang Su. Real-time anomaly detection systems for denial-of-service attacks by weighted k-nearest-neighbor classifiers. *Expert Systems with Applications*, 38(4):3492–3498, 2011.

[51] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10.

[52] Kirshanthan Sundararajah and Milind Kulkarni. Composable, sound transformations of nested recursion and loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 902–917, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rubén Gran, and María Garzarán. Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. *Procedia Computer Science*, 51:140–149, 2015.

[54] Shitong Wang, Jun Wang, and Fu-lai Chung. Kernel density estimation, kernel methods, and fast learning in large data sets. *IEEE Transactions on Cybernetics*, 44(1):1–20, 2014.

[55] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU platforms for deep learning, 2019. arXiv, 10.48550/ARXIV.1907.10701.

[56] Xilinx, Inc. Zynq UltraScale+ MPSoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[57] Hamoud Younes, Ali Ibrahim, Mostafa Rizk, and Maurizio Valle. An efficient selection-based KNN architecture for smart embedded hardware accelerators. *IEEE Open Journal of Circuits and Systems*, 2:534–545, 2021.

[58] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices*, 53(2):593–607, 2018.

[59] Junchao Zhang, Babak Behzad, and Marc Snir. Optimizing the barnes-hut algorithm in upc. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2011.

[60] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P Carloni. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 350–365, 2021.

[61] Marcela Zuluaga, Peter Milder, and Markus Püschel. Streaming Sorting Networks. *ACM Trans. Des. Autom. Electron. Syst.*, 21(4), may 2016.