Chai: Collaborative Heterogeneous Applications for Integrated-architectures

Juan Gómez-Luna[*], Izzat El Hajj[†], Li-Wen Chang[†], Víctor García-Flores[‡§], Simon Garcia de Gonzalo[†], Thomas B. Jablin[†¶], Antonio J. Peña[§] and Wen-mei Hwu[†]

[*]Universidad de Córdoba, [†]University of Illinois at Urbana-Champaign, [‡]Universitat Politècnica de Catalunya, [§]Barcelona Supercomputing Center, [¶]MulticoreWare, Inc.
el1goluj@uco.es, {elhajj2, lchang20}@illinois.edu, vgarcia@ac.upc.edu, {grcdgnz2, jablin}@illinois.edu, antonio.pena@bsc.es, w-hwu@illinois.edu

***Abstract*—Heterogeneous system architectures are evolving towards tighter integration among devices, with emerging features such as shared virtual memory, memory coherence, and system-wide atomics. Languages, device architectures, system specifications, and applications are rapidly adapting to the challenges and opportunities of tightly integrated heterogeneous platforms. Programming languages such as OpenCL 2.0, CUDA 8.0, and C++ AMP allow programmers to exploit these architectures for productive collaboration between CPU and GPU threads. To evaluate these new architectures and programming languages, and to empower researchers to experiment with new ideas, a suite of benchmarks targeting these architectures with close CPU-GPU collaboration is needed.**

In this paper, we classify applications that target heterogeneous architectures into generic collaboration patterns including data partitioning, fine-grain task partitioning, and coarse-grain task partitioning. We present Chai, a new suite of 14 benchmarks that cover these patterns and exercise different features of heterogeneous architectures with varying intensity. Each benchmark in Chai has seven different implementations in different programming models such as OpenCL, C++ AMP, and CUDA, and with and without the use of the latest heterogeneous architecture features. We characterize the behavior of each benchmark with respect to varying input sizes and collaboration combinations, and evaluate the impact of using the emerging features of heterogeneous architectures on application performance.

I. Introduction

Throughput-oriented, massively parallel processor architectures such as GPUs have become a central part of modern computing systems because of their ability to provide high performance at low energy costs. As a result, applications are increasingly requiring tighter integration of CPUs and GPUs to reduce the overhead incurred when leveraging the computing power of the entire system [1]. For this reason, heterogeneous system architectures and programming models are moving towards such tighter integration [2], [3] by introducing features such as shared virtual memory, memory coherence, and system-wide atomics to enable fine-grained CPU and GPU collaboration.

While the concepts of these features may appear simple, their design and implementation involve complex tradeoffs that must be guided with compelling use cases and benchmarks. Furthermore, to explore new ideas and innovations for future computing systems, the community needs a suite of collaborative heterogeneous benchmarks. Such a suite must cover a wide variety of CPU-GPU collaboration patterns, it must exercise a wide range of architectural features, and it must include implementations in both high- and low-level programming models. Chai is designed to serve these needs.

Applications that execute on heterogeneous architectures with close collaboration between different processors have different collaboration patterns. Some applications perform data partitioning where different processors perform the same operation on different data elements. Other applications perform task partitioning where different processors carry out different tasks based on their strengths. Moreover, task partitioning could be fine-grain or coarse-grain depending on whether data-parallel tasks are partitioned on an individual basis or on a group basis at global synchronization points.

Many benchmarks suites [4], [5], [6], [7], [8] have been widely used for evaluating heterogeneous platforms. However, these suites are originally designed for evaluating GPU computing use cases and do not capture true collaboration between CPUs and GPUs that new heterogeneous architectures enable. A few suites [9], [10] are designed for benchmarking collaboration between CPUs and GPUs. However, these suites either fall short in considering close collaboration using latest features of heterogeneous architectures, or miss collaboration patterns and sub-patterns that are needed for well-rounded studies of the new generations of heterogeneous systems. For example, Hetero-Mark [10], [11], [12] is a recent benchmark suite that covers collaborative CPU and GPU execution. However, it only provides one data partitioning benchmark without variability in partitioning granularity, input vs. output partitioning, use of system-wide atomics, inter-worker synchronization, and load balance of data parallel tasks. All these features are essential for a complete and well-rounded performance analysis and characterization of heterogeneous architectures and software stacks.

To fill this gap, Chai provides 14 benchmarks that cover various collaboration patterns, and within each pattern, cover different computation behaviors to exercise different features of the architecture. Chai encompasses a well-rounded combination of aspects such as partitioning granularity, use of system-wide atomics, inter-worker synchronization, and load balance. Moreover, each benchmark in Chai has seven different implementations in different programming models such as OpenCL, C++ AMP, and CUDA, and with and without the use of the latest heterogeneous architecture features.

We make the following contributions:

- We propose classifications for heterogeneous applications based on the nature of collaboration between processors.
- We implement 14 benchmarks in 5 different programming models that cover the proposed classifications and that exercise different features of heterogeneous architectures. These benchmarks have been open-sourced.
- We use our benchmarks to evaluate the impact of using the latest features of heterogeneous architectures over traditional methods.
- We characterize the behavior of each benchmark with respect to varying heterogeneity and input size.

The rest of this paper is organized as follows. Section II gives a brief overview of recent features of heterogeneous architectures and programming models that are targeted by Chai. Section III describes the collaboration patterns. Section IV describes the benchmarks. Section V shows the diversity of the benchmarks within each pattern. Section VI summarizes how the benchmarks are implemented in alternative programming models. Section VII presents different experiments to evaluate heterogeneous architecture features, characterize the benchmarks, and compare programming models. Section VIII discusses related work. Section IX concludes.

II. Heterogeneous Architecture Features

This section gives a very brief overview of the latest features of heterogeneous architectures and programming models that are important for collaboration.

Shared Virtual Memory (SVM) allows host and device processors to share the same virtual address range. This feature improves programmability by eliminating the need for double allocation of data on host and device, tracking contents of memory buffers and explicit copying of data. With this support, CPU and GPU can access data structures through the same pointers. It also has the potential to improve performance via dedicated memory transfer mechanisms and copy avoidance, otherwise data in collaborative programs might be copied back and forth multiple times between processors. SVM is a great feature of modern heterogeneous architectures for writing simpler and more efficient code.

With the introduction of SVM, **memory coherence** becomes an important issue. In programs with close collaboration between processors, multiple devices might access or even update the same addresses frequently in SVM. Thus heterogeneous architectures are moving towards adding support for memory coherence.

System-wide atomics are introduced to allow visible atomic updates across all processors in the system. This feature enables fine-grain communication and synchronization across devices [1] and provides essential support for a variety of CPU-GPU collaboration patterns.

A variety of programming models have introduced support for the heterogeneous architecture features described in this section, including low-level ones such as OpenCL 2.0 and CUDA 8.0 and high-level ones such as C++ AMP. Shared Virtual Memory is OpenCL 2.0 terminology. In CUDA, it is called Unified Memory which first appeared with CUDA 6.0, with memory coherence implemented later with CUDA 8.0 and the Pascal architecture. In HSA, it is called Unified Memory Space. System-wide atomics is CUDA terminology. In OpenCL 2.0, they are just C++11 atomics with the memory scope set appropriately. In HSA, they are called platform atomics. In the literature, they have also been called system-scope atomics [10] and cross-device atomics [13].

III. Collaboration Patterns

The benchmarks in this paper are categorized into collaboration patterns based on how work is partitioned between devices. The main patterns are data partitioning, fine-grain task partitioning, and coarse-grain task partitioning. These patterns are summarized in Figure 1 and described in this section.

Figure 1(a) shows an example application that consists of a coarse-gain task composed of two coarse-grain sub-tasks with a dependence between them. The execution of the bottom task depends on the result of the top task. Each coarse-grain sub-task consists of a collection of data-parallel fine-grain tasks. Each fine-grain task is a chain of dependent sub-tasks with the dependence between them going from top to bottom.

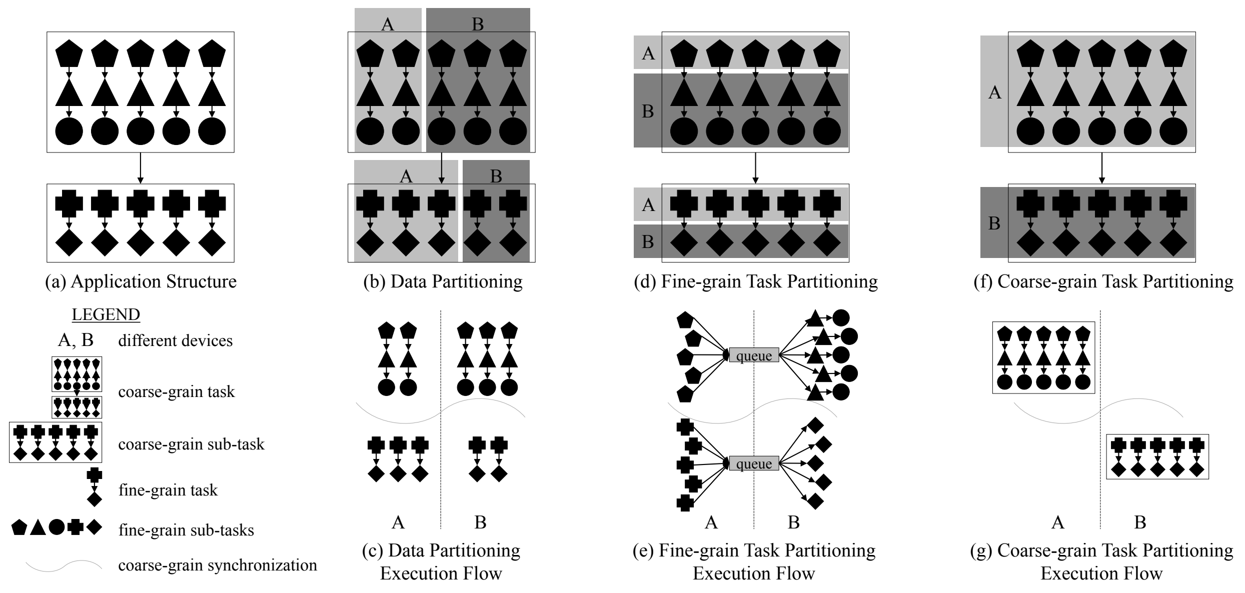
A. Data Partitioning

In *data partitioning*, different devices perform the same task on different parts of the data concurrently. Figure 1(b) illustrates how data partitioning can be applied to the example in Figure 1(a), and the corresponding execution flow for data partitioning is shown in Figure 1(c). Data partitioning benefits from heterogeneous architectures because input and output data can be stored in SVM which helps avoid explicit copying of data between devices and merging of final result. Moreover, system-wide atomics can be used in applications that require atomic updates to an output value or synchronization flag.

One of the challenges with this computation pattern is identifying the best strategy for partitioning data. Our heterogeneous benchmark suite provides researchers with a flexible interface to experiment with different partitioning strategies. Such strategies could be static, dynamic, data-dependent, profiling-based, learning-based, etc. By default, we use a naive dynamic strategy where workers from all participating devices use system-wide atomics to grab tiles of work from a shared worklist in SVM.

B. Fine-grain Task Partitioning

In *fine-grain task partitioning*, different devices perform different sub-tasks of the fine-grain parallel tasks in a computation. Figure 1(d) illustrates how fine-grain task partitioning can be applied to the example in Figure 1(a), and the corresponding execution flow is shown in Figure 1(e). While sub-tasks of the same fine-grain task cannot be performed in parallel because



A
B
queue
LEGEND
A, B different devices
coarse-grain task
coarse-grain sub-task
fine-grain task
fine-grain sub-tasks
coarse-grain synchronization
(a) Application Structure
(b) Data Partitioning
(c) Data Partitioning Execution Flow
(d) Fine-grain Task Partitioning
(e) Fine-grain Task Partitioning Execution Flow
(f) Coarse-grain Task Partitioning
(g) Coarse-grain Task Partitioning Execution Flow

Fig. 1. Computation Patterns

of the dependence between them, sub-tasks of different parallel tasks can be parallelized which enables different devices in this pattern to execute concurrently. For example, in Figure 1(e), all pentagon sub-tasks execute on device A in parallel and deliver their results to device B where the triangle sub-tasks wait for them. A triangle sub-task executes as soon as it receives its value in parallel with other triangle sub-tasks on device B, and also in parallel with other pentagon sub-tasks on device A from different fine-grain tasks. Fine-grain task partitioning benefits from heterogeneous architectures because the use of SVM and system-wide atomics makes fine-grain communication of intermediate results between devices practical.

C. Coarse-grain Task Partitioning

In *coarse-grain task partitioning*, different devices perform different coarse-grain sub-tasks of the entire computation. Figure 1(f) illustrates how coarse-grain task partitioning can be applied to the example in Figure 1(a), and the corresponding execution flow is shown in Figure 1(g). Coarse-grain task partitioning benefits from heterogeneous architectures because the use of SVM eliminates the need to copy data across devices between task partitions. Furthermore, the efficiency of global barrier synchronization depends on the latency and throughput of the implementation of system-wide atomics.

Different devices cannot execute concurrently in this pattern because a dependence exists between coarse-grain tasks. However, it is possible to achieve concurrency if multiple data sets are processed together in a pipeline fashion. In Chai, both kinds of benchmarks are provided.

IV. Heterogeneous Benchmarks

Table I shows a summary of the benchmarks included in the Chai benchmark suite, categorized according to the patterns described in Section III. The suite contains 14 programs from 10 unique benchmarks with some benchmarks having multiple implementations. This section describes each program in the suite, providing details about the implementation of each.

A. Bézier Surface (BS)

Bézier tensor-product surfaces are geometric constructions widely used in engineering and computer-graphics. In particular, we use a parametric non-rational formulation of Bézier surfaces on a regular 2D surface.

This implementation of Bézier Surface performs data partitioning on the output surface, dividing it into square tiles and assigning them to different CPU threads or GPU work-groups. The GPU work-group size is chosen to be the same as the tile size such that each surface point in the tile is computed by one GPU work-item. SVM is used to store the input matrix of control points and the output surface points. As an optimization, the input control points are loaded into the local GPU on-chip memory.

B. Canny Edge Detection - Data Partitioning (CEDD)

Canny Edge Detection [14] is a widely used edge detection algorithm in image processing. Multiple frames of a video stream are each processed through four stages: (1) a Gaussian filter to remove noise, (2) a Sobel filter to obtain the intensity and direction of edge gradients, (3) non-maximum suppression to make edges thinner, and (4) hysteresis to suppress weak edges not connected to strong ones.

This implementation of Canny Edge Detection performs data partitioning on the video, assigning different frames to the CPU and the GPU. Each stage is implemented as a different kernel, but all kernels for a frame are executed all on the CPU or all on the GPU. The data partitioning is considered coarse-grain because a frame is processed by the entire CPU thread pool or the entire GPU kernel. SVM is used to store all input, output, and partial results of the imaging pipelines.

Collaboration Pattern		Short Name	Benchmark
Data Partitioning		BS	Bézier Surface
		CEDD	Canny Edge Detection
		HSTI	Image Histogram (Input Partitioning)
		HSTO	Image Histogram (Output Partitioning)
		PAD	Padding
		RSCD	Random Sample Consensus
		SC	Stream Compaction
		TRNS	In-place Transposition
Task Partitioning	Fine-grain	RSCT	Random Sample Consensus
		TQ	Task Queue System (Synthetic)
		TQH	Task Queue System (Histogram)
	Coarse-grain	BFS	Breadth-First Search
		CEDT	Canny Edge Detection
		SSSP	Single-Source Shortest Path

TABLE I
SUMMARY OF BENCHMARKS INCLUDED IN CHAI.

C. *Image Histogram - Input Partitioning (HSTI)*

Histograms count the number of observations in an input that fall into disjoint bins. They are widely used in many applications, notably in image processing and pattern recognition. This program calculates a histogram of the pixel values in a monochrome image.

We provide two implementations of image histogram based on input and output data partitioning. This implementation, based on *input partitioning*, performs data partitioning on the input image, dividing it into chunks of pixels and assigning the chunks to different CPU threads and GPU work-groups. The GPU work-group size is chosen to be the same as the chunk size such that each pixel is processed by one GPU work-item. SVM is used to store the input image and output histogram bins, and system-wide atomics are used by the CPU threads and GPU work-items to update the bins atomically.

As an optimization, private histograms of partial results are allocated for each CPU thread or GPU work-group and merged into the global one at the end to reduce contention of atomic operations on the bins. The CPU per-thread private histograms do not need to be updated with atomics since they are only accessed by a single thread. The GPU per-work-group private histograms are stored in local on-chip memory.

D. *Image Histogram - Output Partitioning (HSTO)*

This implementation of image histogram, based on *output partitioning*, performs static data partitioning on the output histogram bins, dividing them into two mutually exclusive sets, one for the CPU, and one for the GPU. Both the CPU and the GPU must go through the entire input, but they only vote on a bin if it falls in their own partition. This arrangement eliminates conflicts between CPU threads and GPU work-items on the output bins.

E. *Padding (PAD)*

Padding is a data manipulation primitive that inserts space between elements of an array. It is commonly used for memory alignment adjustment and matrix transposition. We use in-place padding which unidirectionally shifts data in memory to expand an array, while ensuring that shifted elements do not overwrite old values before they are consumed.

This implementation of padding performs data partitioning on the input matrix, assigning each row to a different CPU thread or GPU work-group. It uses synchronization flags to notify adjacent workers that it is safe to write to the locations it loaded from [15]. SVM is used to store the array being padded and the synchronization flags, and system-wide atomics are used to set the flags.

F. *Random Sample Consensus - Data Partitioning (RSCD)*

Random Sample Consensus (RANSAC) is an iterative method to estimate parameters of a mathematical model from a set of input data using random sampling [16]. Each iteration includes two main stages: (1) an inherently sequential model fitting stage using random samples of input, and (2) a massively parallel evaluation stage measuring outlier counts and model errors. The iteration space is parallelized such that different random sample sets are attempted simultaneously.

We provide two implementations of RANSAC based on data and task partitioning. The data partitioning implementation partitions the iteration space, assigning each iteration to a different CPU thread or GPU work-group. On the GPU, the fitting stage of each iteration is computed by only one work-item in the work-group, since it is inherently sequential, while the evaluation stage is computed by all work-items. SVM is used to store the input data and resulting models. System-wide atomics are used for enqueueing successful models.

G. *Stream Compaction (SC)*

Stream compaction, also known as filtering, is a data manipulation primitive that removes elements from an array, keeping those satisfying a certain predicate. It is commonly used in tree traversal, image processing, and databases. We use in-place stream compaction which, like padding, unidirectionally shifts data in memory. Unlike padding, it shrinks the array and is more irregular, removing a variable number of elements.

This implementation of stream compaction performs data partitioning on the input array, assigning tiles to a different CPU threads or GPU work-groups. It uses synchronization flags like padding to synchronize with adjacent workers. SVM is used to store the array being compacted and the synchronization flags, and system-wide atomics are used to set the flags

H. *In-place Transposition (TRNS)*

Transposition is an important data manipulation primitive that converts between data layouts such as Array-of-Structures (AoS), Structure-of-Arrays (SoA), and Array-of-Structure-of-Tiled-Arrays (ASTA). It is important in heterogeneous systems because it reshapes data according to the memory access preferences of different devices (e.g., stride-one access for CPU vs. coalesced access for GPU). We use in-place transposition which reduces memory consumption, but requires sophisticated cycle dependency checks [17].

This implementation of transpose performs data partitioning on the set of dependency cycles, assigning each cycle to one or more CPU threads or GPU work-groups. Only dynamic partitioning is possible with this program because the full set of cycles is not known at the beginning to be statically partitioned. SVM is used to store the matrix being transposed and the flags used to synchronize between CPU threads or GPU work-groups collaborating on the same cycle. System-wide atomics are used for updating the collaboration flags.

I. Random Sample Consensus - Task Partitioning (RSCT)

This implementation of RANSAC (see Section IV-F) performs task partitioning of each iteration, executing the sequential fitting stage on the CPU and the parallel evaluation stage on the GPU. Once a CPU thread computes the model of an iteration, it sets a flag that enables a GPU work-group to evaluate the model. The work-group then atomically updates the best model and checks if convergence has been reached. SVM is used to store the input data and the variables used to track the convergence point. System-wide atomics are used for updating the convergence point.

J. Task Queue System (TQ)

A task queue system is a generic computation where the host generates and enqueues tasks on several queues residing in device memory. Chen *et al.* [18] present a task queue system that uses asynchronous data transfers, zero-copies, and events to create a task queue system for GPUs. Such a system can be implemented with fewer lines of code using SVM and system-wide atomics because there is no need for double declaration/allocation and data transfers are not necessary.

We provide two implementations of task queue system based on a synthetic and a histogram application. In this implementation, CPU threads generate and enqueue two types of synthetic tasks (heavy and light) that GPU work-groups dequeue and execute these tasks. Each work-item in the work-group performs some arithmetic additions on an input data element and updates it. SVM is used to store the queue and queue counters, and system-wide atomics are used to update the counters in order to enqueue/dequeue tasks. These counters include the number of enqueued tasks, the number of consumed tasks, and the current number of tasks in queue.

K. Task Queue System - Histogram Application (TQH)

This is a histogram application that is built on top of TQ, where the CPU reads and enqueues frames of a video sequence and the GPU calculates a histogram of pixel brightness in each frame. Since atomic operations contend on the histogram bins, the execution time for each frame can be vary significantly based on the frame's pixel distribution.

L. Breadth-First Search (BFS)

Breadth-First Search is a well-known graph traversal algorithm. We use a queue-based version which starts at a single source node, and on each iteration, visits the neighbors of every node in the current frontier and enqueues previously unvisited neighbors in the next frontier.

Our implementation of BFS applies coarse-grain task partitioning on the set of frontiers. Because graphs are irregular, different frontiers often have a different number of nodes and node neighbors to process. Small frontiers are more efficiently processed on the CPU while large ones are better suited for the GPU. The CPU loops over nodes in the current frontier and enqueues unvisited neighbors sequentially, while the GPU assigns different work-items to different frontier nodes which use atomics to enqueue unvisited neighbors concurrently.

As an optimization, a hierarchical queueing system [19] is used on the GPU where each work-group updates a local queue in on-chip memory then merges the result to the global queue at the end. SVM is used to allocate data which makes it easy to switch between the CPU and the GPU between frontiers. System-wide atomics are used to implement global synchronization across the CPU and the GPU between frontiers before a switch takes place (if any).

M. Canny Edge Detection - Task Partitioning (CEDT)

This implementation of Canny Edge Detection (see Section IV-B) performs task partitioning on the four imaging stages. Gaussian and Sobel filters are executed on the GPU because they are more regular. Non-maximum suppression and hysteresis are executed on the CPU because they contain control flow statements that can make GPU threads diverge, thereby underutilizing the vector lanes. SVM is used to store all data which makes it easy to switch between CPU and GPU between stages. Different devices can execute different stages from different frames in parallel forming a pipeline pattern.

N. Single-Source Shortest Path (SSSP)

Single-Source Shortest Path (SSSP) is a well-known graph traversal algorithm which finds the path with the minimal sum of edge weights from a given source node to each node in the graph. All nodes except the source are initialized to having infinite cost. Frontiers are then constructed in a manner similar to BFS, but at the enqueueing of each neighbor, the minimum cost of reaching that neighbor is also calculated and updated.

This implementation of SSSP is similar to BFS, with additional work needed to calculate and atomically update the minimum cost. It turns out that this additional work significantly affects the characteristics of the benchmark with respect to the impact of using SVM and system-wide atomics, which is why we choose to include both in the suite.

V. Heterogeneous Benchmark Suite Diversity

In addition to covering the three general collaboration patterns, Chai is designed to exhibit diversity within each pattern as well. Different benchmarks in each pattern are chosen to have different computation and memory access patterns such that they exercise different features of the architecture with varying intensity. Table II summarizes the differences between benchmarks in each pattern, and the following section elaborates more on each aspect.

Different data partitioning benchmarks have different levels of data partitioning granularity: fine-grain, where work-groups grab tiles of data parallel tasks but each work-item processes an independent task, medium-grain, where data parallel tasks

Data Partitioning

Benchmark	Partitioning Granularity	Partitioned Data	System-wide Atomics	Load Balance
BS	Fine	Output	None	Yes
CEDD	Coarse	Input, Output	None	Yes
HSTI	Fine	Input	Compute	No
HSTO	Fine	Output	None	No
PAD	Fine	Input, Output	Sync	Yes
RSCD	Medium	Output	Compute	Yes
SC	Fine	Input, Output	Sync	No
TRNS	Medium	Input, Output	Sync	No

Fine-grain Task Partitioning

Benchmark	System-wide Atomics	Load Balance
RSCT	Sync, Compute	Yes
TQ	Sync	No
TQH	Sync	No

Coarse-grain Task Partitioning

Benchmark	System-wide Atomics	Partitioning	Concurrency
BFS	Sync, Compute	Iterative	No
CEDT	Sync	Non-iterative	Yes
SSSP	Sync, Compute	Iterative	No

TABLE II
HETEROGENEOUS BENCHMARK SUITE DIVERSITY.

are assigned to work-groups as a whole, and coarse-grain, where data parallel tasks are assigned to kernels as a whole. The reason we do not consider partitioning granularity to be a major pattern classification criterion like in task partitioning is that granularity has a more fundamental impact on benchmark structure in task partitioning than it does in data partitioning.

There are other aspects that also differentiate data partitioning benchmarks. One aspect is whether the input and output are each partitioned or shared. This aspect is useful for studying read-only and read-write accesses to shared caches. Another aspect is the use of system-wide atomics. Some benchmarks use system-wide atomics for computation to update output values and counters. Other benchmarks exhibit inter-worker synchronization and use system-wide atomics to update and spin-lock on synchronization flags. This aspect is useful for evaluating performance of atomics as well as software optimizations related to the use of atomics. Another aspect is whether or not data parallel tasks are load balanced. This aspect is useful for evaluating scheduling policies and other optimizations related to load balancing.

Task partitioning benchmarks are already classified according to partitioning granularity into fine-grain or coarse-grain partitioning. Fine-grain task partitioning benchmarks, like data partitioning ones, are also diverse in their use of system-wide atomics for computation and in load balancing. However, they all use system-wide atomics for inter-worker synchronization between fine-grain sub-tasks, which is a defining characteristic of this pattern.

For coarse-grain task partitioning, use of system-wide atomics for computation is also a differentiating aspect between benchmarks. However, all benchmarks use system-wide atomics for global synchronization between coarse-grain tasks. Load balance is a less relevant aspect because it is contained within a coarse-grain task, not across coarse-grain task partition boundaries. One aspect that is relevant to coarse-grain task partitioning is whether concurrency is supported (see Section III-C). Another aspect is whether different coarse-grain tasks are the same operation applied iteratively to different data, or if they are entirely different operations. These aspects are useful for studying task partitioning heuristics and scheduling policies.

VI. ALTERNATIVE IMPLEMENTATIONS

The implementations described in Section IV are written in OpenCL. These are the primary implementations contributed by the suite, and are referred to as OpenCL-U. The suffix '-U' denotes the use of a unified address space and system-wide atomics. However, to provide additional value for researchers, Chai also includes six other alternative implementations of each benchmark: OpenCL-D, C++ AMP, CUDA-U, CUDA-D, CUDA-U-Sim, and CUDA-D-Sim. The suffix '-D' denotes the use of discrete address spaces, and the suffix '-Sim' denotes that the benchmark is written for the gem5-gpu simulator [20]. These alternatives are described in this section.

A. OpenCL-D

For all benchmarks, we provide OpenCL implementations that run on systems without SVM, coherence, and system-wide atomics. These benchmarks use traditional techniques such as double allocation of buffers, copying of data between devices, and kernel launch/termination for cross-device synchronization. Comparing the OpenCL-U implementations to their OpenCL-D counterparts is useful for evaluating the impact of integrated heterogeneous architecture features.

For data partitioning, the OpenCL-D implementations have the additional burden of merging the final output results produced by the CPU and GPU after the copy. Moreover, dynamic partitioning is not possible without system-wide atomics if data partitioning is not coarse-grain so only static partitioning can be employed. Inter-worker synchronization between CPU and GPU workers is also not possible without system-wide atomics, so only the GPU is used in those benchmarks (PAD, SC, TRNS).

For fine-grain task partitioning, fine-grain communication is not possible without system-wide atomics. For this reason, the OpenCL-D implementations perform all instances of a fine-grain sub-tasks on its assigned device, store the intermediate results in a buffer and copy it to the other device, then perform all instances of the fine-grain sub-task on the other device.

For coarse-grain task partitioning, the OpenCL-D implementations copy data between devices whenever switching takes place. Kernel termination and relaunch is used instead of system-wide atomics for global synchronization.

B. C++ AMP

For all benchmarks, we provide a C++ AMP implementation that reproduces the OpenCL-U versions. The main differ-

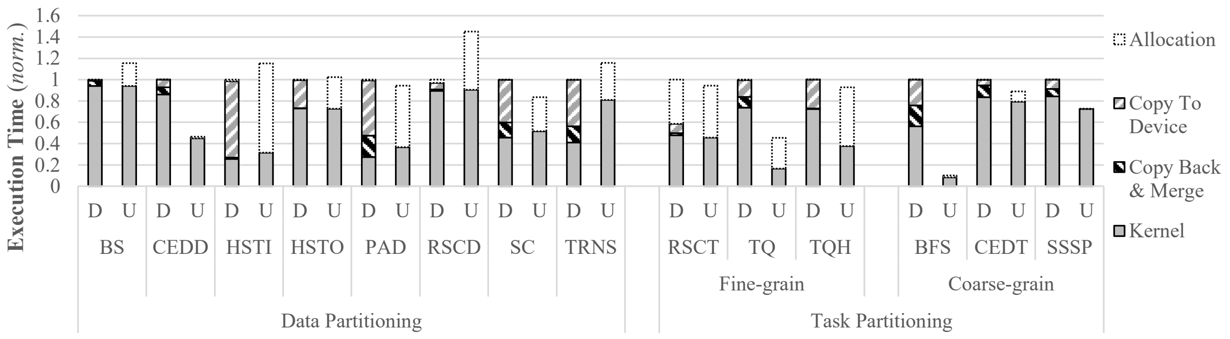


Fig. 2. Comparing OpenCL-D and OpenCL-U

Benchmark	Configuration (#wi, #wg)	Datasets
BS	16×16, 32	4x4 (default), 8x8, 12x12
CEDD [23]	16×16, framesize/#wi	Peppa (default), Maradona, Paw
HSTI	256, 16	256 bins (default), 32, 4096
HSTO	256, 16	256 bins (default), 32, 4096
PAD [15]	64, 16	1000×999 (default), 6000×5999, 12000×11999
RSCD	256, 64	2000 iterations (default), 4000, 8000
SC [15]	256, 32	50% (default), 0%, 100%
TRNS [17]	64, 64	tile=32 (default), 4, 128
RSCT	256, 64	2000 iterations (default)
TQ [18]	64, 320	50% (default)
TQH [18]	64, 160	Basket (default)
BFS [19]	256, 8	NY (default), NE, UT
CEDT [23]	16×16, framesize/#wi	Peppa (default), Maradona, Paw
SSSP [19]	64, 32	NY (default), NE, UT

TABLE III
BENCHMARK SOURCE, GPU CONFIGURATIONS, AND DATASETS.

ence is that system-wide atomics and coherent memory are implicit, not explicitly declared. Comparing the OpenCL-U implementations to their C++ AMP counterparts is useful for evaluating the tradeoffs between high-level and low-level implementations programming models.

C. CUDA

For all OpenCL-U and OpenCL-D implementations, we additionally provide equivalent CUDA implementations denoted as CUDA-U and CUDA-D respectively. We also provide CUDA implementations for the gem5-gpu simulator [20], denoted as CUDA-U-Sim and CUDA-D-Sim respectively.

VII. EVALUATION

A. Methodology

The OpenCL and C++ AMP experiments are performed on an AMD Kaveri A10-7850K APU with HSA features. This integrated platform includes 4 CPU cores, and 8 GPU compute units. They share DDR3 DRAM memory. The AMD APP SDK 3.0 is used for compiling OpenCL and ROCm 1.2 [21] is used for compiling C++ AMP. Timing measurements are obtained with regular Linux timers for uniformity across implementations. We run the programs 10 times. In each run, the kernel time is the average of 50 runs, after 5 warm-up runs. Profiling results are obtained using CodeXL [22]. CUDA-U-Sim experiments are simulated on gem5-gpu [20], a cycle-level simulator that merges gem5 and GPGPU-Sim.

Table III lists the benchmarks with the corresponding GPU configurations and datasets used. For benchmarks leveraging existing code, we cite the source; otherwise, the benchmark was implemented from scratch. For GPU configurations, we select the work-group and work-item count with the best performance on our system. For datasets, we provide a default dataset and two additional ones for the benchmarks used in the experiments in Section VII-C.

B. Impact of Heterogeneous Architecture Features

Figure 2 compares the execution time of OpenCL-D and OpenCL-U versions broken down into allocation, copy, and kernel time. The objective of this experiment is to evaluate the impact of using heterogeneous architecture features. Default datasets are used as well as the best GPU+CPU configuration for each benchmark (see Section VII-C). For data partitioning benchmarks, both OpenCL-D and OpenCL-U implementations use the best static partitioning for each for fair comparison.

All data partitioning benchmarks show improvement in execution time for OpenCL-U over OpenCL-D if allocation time is not considered. While the kernel execution times are generally comparable, the main source of improvement comes from the elimination of the copy and merge time. It is noteworthy, however, that SVM allocation in OpenCL-U takes longer than regular allocation in OpenCL-D, which makes OpenCL-U slower in total for a few cases. In real applications, this is less of an issue because buffers are allocated once then copied to and reused multiple times, but in general, this result is useful for researchers working on optimizing software and hardware support for memory allocation and management.

An interesting observation when comparing just kernel execution time is that the benchmarks where the OpenCL-U kernels take slightly longer than the OpenCL-D kernels are the same ones that use system-wide atomics for computation and synchronization as indicated in Table II. That is because system-wide atomics are more expensive than regular atomics. These results are useful for researchers working on software and hardware optimizations for system-wide atomics.

Another interesting observation when comparing just kernel execution time is that only in CEDD, OpenCL-U outperforms OpenCL-D. One possible reason is that OpenCL-U on Kaveri can potentially achieve higher effective memory bandwidth by using both the Radeon memory bus and the fusion bus [24], in contrast with OpenCL-D just using the Radeon memory bus. The fact that data partitioning in CEDD is coarse-grain means

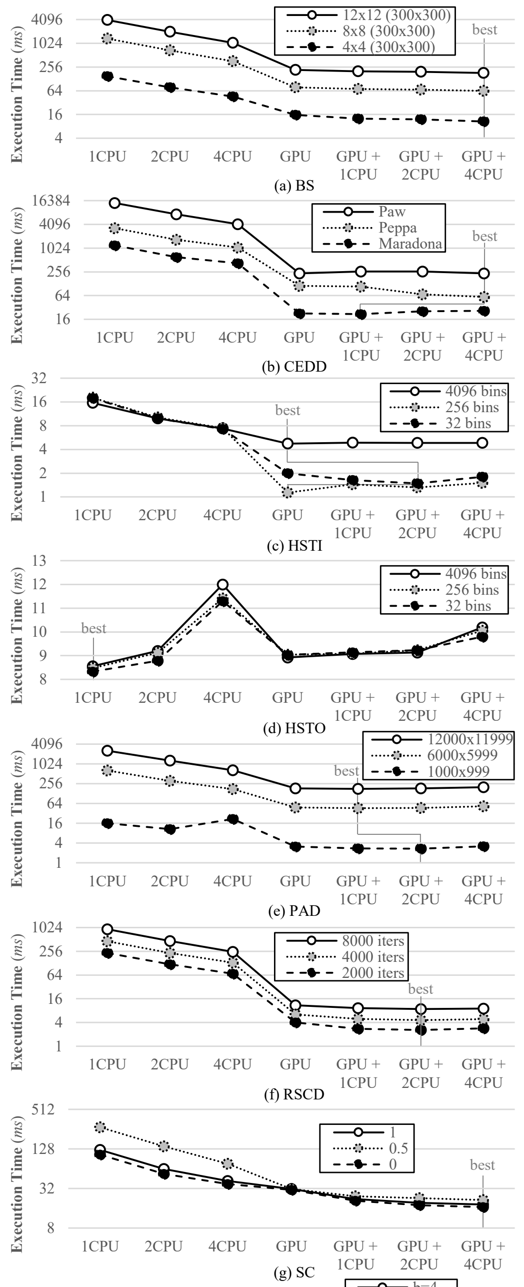


Fig. 3. Characterizing Data Partitioning Benchmarks with respect to Heterogeneity Combinations and Input Size

that there is no data sharing between the CPU and the GPU which process different frames, thus favoring the combined use of Radeon and fusion buses in OpenCL-U.

All fine-grain task partitioning benchmarks show performance improvement for OpenCL-U over OpenCL-D even with allocation time included. Like data partitioning benchmarks,

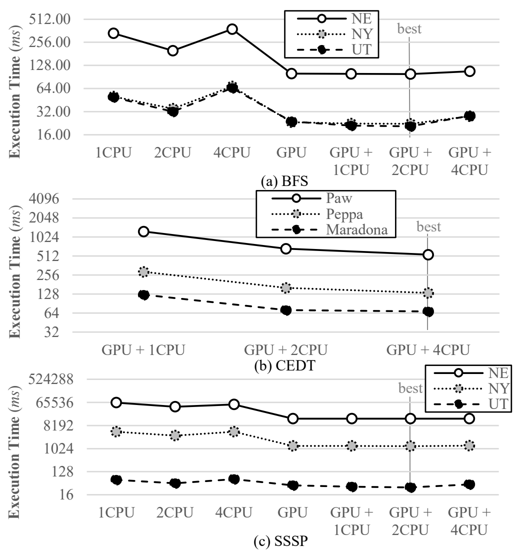


Fig. 4. Characterizing Coarse-grain Task Partitioning Benchmarks with respect to Heterogeneity Combinations and Input Size

part of the improvement comes from elimination of copy time. However, a significant improvement in kernel execution time is also observed. This comes from the fact that fine-grain communication enables concurrent execution of independent sub-tasks on different devices which OpenCL-D serializes.

All coarse-grain task partitioning benchmarks show performance improvement for OpenCL-U over OpenCL-D. Again, the elimination of the copy time contributes to this improvement. Moreover, kernel time in iterative benchmarks improves due to the use of system-wide atomics for global synchronization instead of kernel termination/relaunch. Note that SSSP is much less sensitive than BFS despite their similarity because it performs more computations.

Finally, we note that the benchmarks showing the most improvement for OpenCL-U over OpenCL-D (e.g., HSTI, TQ, BFS) have in common that they perform little computation relative to the data transfer and synchronization activities which new features of integrated heterogeneous architectures are intended to optimize.

C. Collaboration Combinations and Problem Size

The objective of the experiment in this section is to characterize each benchmark with respect to varying combinations of collaborative versus non-collaborative execution and problem size. OpenCL-U is used and all three datasets are tested.

Results for data partitioning benchmarks are shown in Figure 3. It is clear that heterogeneous collaboration improves performance over CPU-only and GPU-only execution. The best CPU+GPU version outperforms the GPU-only version by up to 47%, 91%, 34%, 16%, 55%, 82%, and 10% for BS, CEDD, HSTI, PAD, RSCD, SC, and TRNS respectively (the difference is hard to see in the figure). One interesting observation is that the improvement tends to be smaller for benchmarks that are more memory-bound such as PAD and TRNS. For example, PAD and SC implement similar algo-

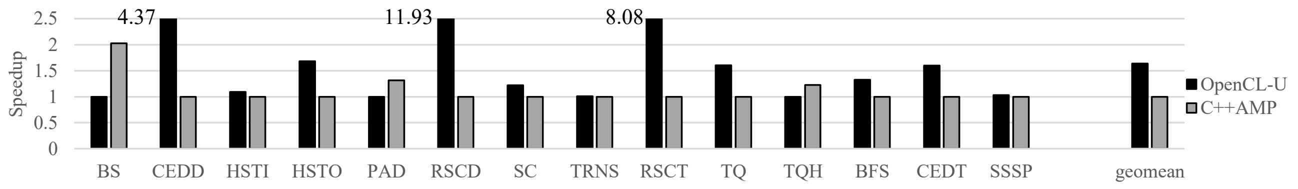


Fig. 5. Comparing OpenCL-U and C++ AMP Execution Time

rithms, but SC performs additional reductions and prefix-sum operations that make it more compute-bound, thus benefiting more from additional CPU threads. Moreover, adding all four CPU cores to assist the GPU is not always beneficial. There is usually a sweet spot for the number of CPU threads to add which varies for each benchmark and even each dataset in the same benchmark. These results are useful for researchers working on schedulers and tuners.

Comparing HSTI and HSTO, we note that HSTI is faster for the three datasets. That is because the image is large so HSTO is burdened by all CPU threads and GPU work-groups needing to load the entire image. However, HSTO is better for small images and large histograms. For example, it is 7× faster for a 4096-bin histogram of a 384×256 image.

Fine-grain task partitioning benchmarks are not evaluated in this experiment because they require both devices to execute simultaneously by definition. Since the objective of this experiment is to show the benefit of collaborative execution over non-collaborative execution, just showing the collaborative combinations is not interesting because there are no CPU-only and GPU-only versions to compare to.

Results for coarse-grain task partitioning benchmarks are shown in Figure 4. CEDT does not have CPU-only and GPU-only versions because the coarse-grain sub-tasks for different devices are represented by different kernels, some written for the CPU and some written for the GPU. On the other hand, BFS and SSSP are iterative and can be executed as CPU-only or GPU-only by biasing the condition that decides when to switch. Similar to data partitioning benchmarks, the best CPU+GPU version outperforms the GPU-only version by up to 15% and 22% for BFS and SSSP respectively, with the sweet spot varying for different benchmarks and datasets.

D. Comparison with C++ AMP

Figure 5 compares the performance of OpenCL-U and C++ AMP benchmarks normalized to the slower implementation. Note that C++ AMP implementations of RSCD and RSCT are disadvantaged because compilation fails when the outlier counter is placed in local memory, so global memory is used. This is a known issue [25].

The results show that the C++AMP implementations perform comparably for most benchmarks. C++ AMP versions perform slightly better in a few cases while OpenCL-U versions perform significantly better in other cases. Overall, OpenCL-U implementations have a geometric mean speedup of 1.64× over C++ AMP implementations (1.22× if we exclude RSCD and RSCT). These results are useful for the

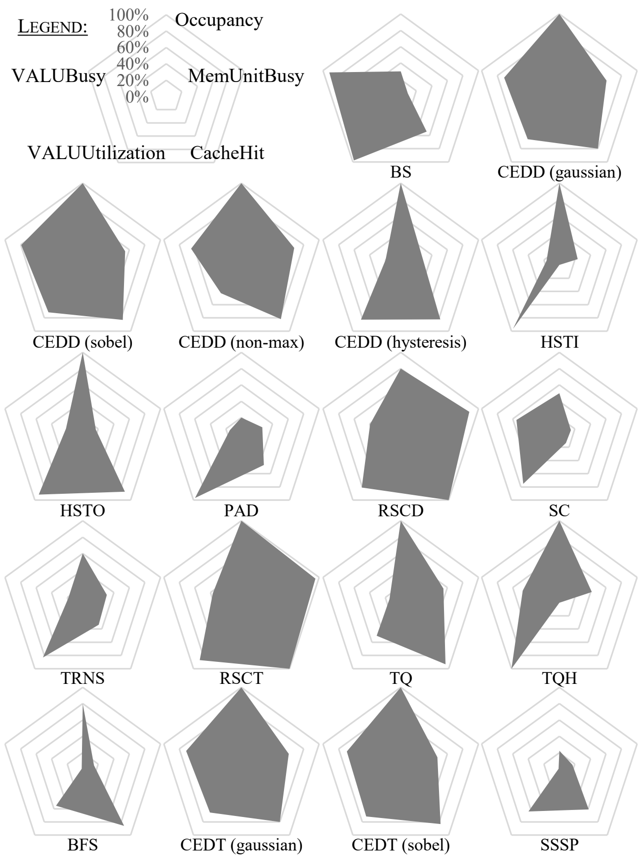


Fig. 6. Profiling Metrics for each Benchmark

evaluation of new programming models and tools compared to existing low- and high-level languages.

E. Benchmark Profiling

The objective of the experiments in this section is to further show the diversity of the benchmarks and validate some observations in Sections VII-B and VII-C.

Figure 6 shows radar charts for each benchmark for five profiling metrics, obtained by profiling the GPU kernels of CPU+GPU runs. We considered all metrics CodeXL provides and found these five to be the most useful for basic classification. VALUBusy and MemUnitBusy are the percentage of GPU time that vector ALUs and memory units are active respectively. CacheHit is the percentage of memory operations that hit the data cache. VALUBusy, MemUnitBusy, and CacheHit together help distinguish compute- and memory-bound benchmarks. Occupancy is the ratio of active wavefronts to the maximum, and VALUUtilization is the percentage

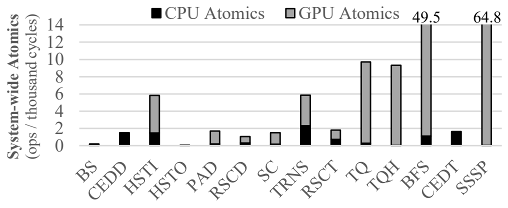


Fig. 7. Usage of System-wide Atomics

of active lanes in a wavefront. Occupancy and VALUUtilization show the quality of the GPU implementations.

Looking at MemUnitBusy and VALUBusy with reference to the results in Section VII-C gives multiple insights. BS and SC are more compute-bound which is why they benefit from increasing heterogeneity, while PAD and TRNS are more memory-bound and do not benefit as much. RSCD, despite keeping the memory unit busy, has a high L2 cache hit rate so it does not saturate the memory, which is why it does benefit from increased heterogeneity. HSTI, HSTO, and TQH have low MemUnitBusy and VALUBusy because most of the time is spent updating private histograms in local memory. Looking at VALUUtilization, most benchmarks show high utilization except CEDD (non-max) which contains many control flow statements that make work-items diverge, and BFS which is irregular by nature. As for Occupancy, most benchmarks have 100% occupancy because the configurations have already been selected to have the best performance. Exceptions are BS, SC, and PAD which use thread coarsening to exploit ILP and MLP.

Because CodeXL does not provide profiling information about system-wide atomics, we use the CUDA-U-Sim implementations to obtain this information from the simulator. The results are shown in Figure 7. These results further verify the classifications in Table II. All data partitioning benchmarks use system-wide atomics for dynamic partitioning. Beyond that, BS, CEDD, and HSTO do not use system-wide atomics; PAD, SC, and TRNS use them for inter-worker synchronization; HSTI and RSCD use them for computation but RSCD has more computations to hide them. For fine-grain task partitioning benchmarks, RSCT, TQ, and TQH all use system-wide atomics for fine-grain communication across devices, but RSCT does more work which hides them compared to the other two. Note that RSCT uses slightly more than RSCD because they both use them for computation, but RSCT also uses them for fine-grain communication. For coarse-grain task partitioning benchmarks, BFS and SSSP have very high usage because they use them for both computation and global synchronization, while CEDT only uses them for sycnhronication between CPU proxy threads.

VIII. Related Work

Many high-quality benchmark suites [4], [5], [6], [7], [8] are widely used for evaluating CPU and GPU computing, but they are not originally designed to capture true collaboration between devices. These suites could be modified to use new features of integrated heterogeneous systems for collaborative execution. However, the benchmarks in these suites may not be the best candidates for collaboration and may not collectively achieve the coverage we provide because the suites were not designed with these objectives in mind.

A few suites [9], [10] are designed with truly collaborative patterns. Valar [9] provides OpenCL benchmarks with closely-coupled execution on multiple OpenCL devices. However, it targets an old generation of AMD APUs without HSA and does not exercise latest features of heterogeneous architectures. Chai focuses on truly collaborative benchmarks using the latest heterogeneous architecture features. Hetero-Mark [10], [11], [12] provides collaborative benchmarks that exercise new features of heterogeneous architectures, including one data partitioning benchmark and three task partitioning benchmarks. However, it does not emphasize diversity within collaboration patterns and classifies task partitioning benchmarks by the order in which devices execute. Chai provides wider coverage within each pattern for more well-rounded performance analyses. It also considers partitioning granularity a more important sub-classification for task partitioning because it has a more fundamental impact on program structure.

MachSuite [26] is designed for accelerators and high-level synthesis which may or may not involve heterogeneous computing. Chai is designed for heterogeneous computing.

Multiple studies investigate the benefits and limitations of current heterogeneous architectures. Spafford *et al.* [27] evaluate performance, power, and programmability tradeoffs between integrated and discrete architectures. Lee *et al.* [28] characterize the performance of data-intensive kernels on integrated architectures. Zhu *et al.* [29] study co-run performance on APUs. Farooqui *et al.* [30] investigate optimizations for graph applications on APUs. Garcia-Flores *et al.* [31] propose a design of shared last-level cache for heterogeneous architectures. Erb *et al.* [32] address the problem of buffer overflow from GPU code, which is exacerbated by the sharing of CPU and GPU address spaces. Chai is a benchmark suite that would be useful to all such studies.

In addition to SVM and system-wide atomics, another emerging feature in heterogeneous systems is device-side kernel launch. This feature has stimulated much architecture and compiler research [33], [34], [35], [36], [37] but still lacks benchmark suite support, which is an important subject for future work.

IX. Conclusion

We present Chai, a suite of 14 collaborative heterogeneous benchmarks that leverage the latest features of heterogeneous architectures, cover a wide range of collaboration patterns, exhibit great diversity within each pattern, and have seven different implementations each: OpenCL-U, OpenCL-D, C++ AMP, CUDA-U, CUDA-D, CUDA-U-Sim, and CUDA-D-Sim. We use the benchmarks to show the impact of using heterogeneous architecture features and to demonstrate the potential of collaborative execution. Chai provides a suite that is much-needed for a well-rounded evaluation of heterogeneous architectures, programming models, and software stacks.

Acknowledgment

This work is supported by the Ministry of Education of Spain (TIN2013-42253P), the Junta de Andalucía of Spain (TIC-1692), Hewlett-Packard Enterprise Labs, the Starnet Center for Future Architecture Research (C-FAR), the Huawei Project on High Performance Algorithm Compilation for Heterogeneous System (YB2015120003). and the DoE National Nuclear Security Administration (DE-NA0002374).

References

[1] W.-m. W. Hwu, *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Morgan Kaufman, 2015.

[2] Khronos group, "The OpenCL specification," *Version 2.0*, 2015.

[3] NVIDIA, "CUDA C programming guide v. 8.0," September 2016.

[4] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *IMPACT Technical Report*, 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, IEEE International Symposium on*, pp. 44–54, 2009.

[6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, 2010.

[7] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "NUPAR: A benchmark suite for modern GPU architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015.

[8] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Workload Characterization, IEEE International Symposium on*, pp. 141–151, 2012.

[9] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli, "Valar: A benchmark suite to study the dynamic behavior of heterogeneous systems," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 54–65, 2013.

[10] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-Mark, a benchmark suite for CPU-GPU collaborative computing," in *Workload Characterization, IEEE International Symposium on*, 2016.

[11] S. Mukherjee, X. Gong, L. Yu, C. McCardwell, Y. Ukidave, T. Dao, F. N. Paravecino, and D. Kaeli, "Exploring the features of OpenCL 2.0," in *Proceedings of the 3rd International Workshop on OpenCL*, pp. 5:1–5:5, 2015.

[12] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli, "A comprehensive performance analysis of HSA and OpenCL 2.0," in *Performance Analysis of Systems and Software, IEEE International Symposium on*, pp. 183–193, 2016.

[13] M. Gupta, D. Das, P. Raghavendra, T. Tye, L. Lobachev, A. Agarwal, and R. Hegde, "Implementing cross-device atomics in heterogeneous processors," in *Parallel and Distributed Processing Symposium Workshop, IEEE International*, pp. 659–668, 2015.

[14] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 679–698, 1986.

[15] J. Gómez Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *Parallel Processing, 44th International Conference on*, pp. 210–219, 2015.

[16] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, pp. 381–395, June 1981.

[17] I.-J. Sung, G. Liu, and W.-M. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Computing*, pp. 1 –11, 2012.

[18] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic load balancing on single- and multi-GPU systems," in *Parallel Distributed Processing, IEEE International Symposium on*, pp. 1–12, 2010.

[19] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, pp. 52–55, 2010.

[20] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *Computer Architecture Letters*, vol. 13, Jan 2014.

[21] RadeonOpenCompute, "ROCm: Platform for GPU enabled HPC and ultrascale computing." https://github.com/RadeonOpenCompute/ROCm, 2016.

[22] AMD, "App profiler settings." http://developer.amd.com/tools-and-sdks/archive/compute/amd-app-profiler/user-guide/app-profiler-settings/.

[23] S. Kelley. https://github.com/smskelley/canny-opencl.

[24] AMD, "Memory system on Fusion APUs. The benefits of zero copy." http://developer.amd.com/wordpress/media/2013/06/1004_final.pdf, June 2011.

[25] bshaozi, "Compile problem." https://github.com/RadeonOpenCompute/hcc/issues/124, September 2016.

[26] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Workload Characterization, IEEE International Symposium on*, pp. 110–119, 2014.

[27] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," in *Proceedings of the 9th conference on Computing Frontiers*, pp. 103–112, 2012.

[28] K. Lee, H. Lin, and W.-c. Feng, "Performance characterization of data-intensive kernels on AMD fusion architectures," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 175–184, 2013.

[29] Q. Zhu, B. Wu, X. Shen, K. Shen, L. Shen, and Z. Wang, "Understanding co-run performance on CPU-GPU integrated processors: observations, insights, directions," *Frontiers of Computer Science*, pp. 1–17, 2016.

[30] N. Farooqui, I. Roy, Y. Chen, V. Talwar, and K. Schwan, "Accelerating graph applications on integrated GPU platforms via instrumentation-driven optimizations," in *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 19–28, 2016.

[31] V. Garcia-Flores, J. Gómez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications," in *Workload Characterization, IEEE International Symposium on*, pp. 1–10, 2016.

[32] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for gpgpus," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pp. 61–73, 2017.

[33] G. Chen and X. Shen, "Free launch: optimizing GPU dynamic kernel launches through thread reuse," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 407–419, 2015.

[34] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 528–540, 2015.

[35] H. Wu, D. Li, and M. Becchi, "Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 534–543, 2016.

[36] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[37] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, M. I. Sreepathi Pai, M. T. Kandemir, and C. R. Das, "Controlled kernel launch for dynamic parallelism in GPUs,"