Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures

Sitao Huang
ECE, UIUC
shuang91@illinois.edu

Li-Wen Chang*
Microsoft
liwen.chang@microsoft.com

Izzat El Hajj
ECE, UIUC
elhajj2@illinois.edu

Simon Garcia De Gonzalo
CS, UIUC
grcdgnz2@illinois.edu

Juan Gómez-Luna
CS, ETH Zurich
juang@ethz.ch

Sai Rahul Chalamalasetti
Hewlett Packard Labs
sairahul.chalamalasetti@hpe.com

Mohamed El-Hadedy
ECE, Cal Poly Pomona
mealy@cpp.edu

Dejan Milojicic
Hewlett Packard Labs
dejan.milojicic@hpe.com

Onur Mutlu
CS, ETH Zurich
omutlu@ethz.ch

Deming Chen
ECE, UIUC
dchen@illinois.edu

Wen-mei Hwu
ECE, UIUC
w-hwu@illinois.edu

ABSTRACT

Heterogeneous CPU-FPGA systems are evolving towards tighter integration between CPUs and FPGAs for improved performance and energy efficiency. At the same time, programmability is also improving with High Level Synthesis tools (e.g., OpenCL Software Development Kits), which allow programmers to express their designs with high-level programming languages, and avoid time-consuming and error-prone register-transfer level (RTL) programming. In the traditional loosely-coupled accelerator mode, FPGAs work as offload accelerators, where an entire kernel runs on the FPGA while the CPU thread waits for the result. However, tighter integration of the CPUs and the FPGAs enables the possibility of *fine-grained collaborative execution*, i.e., having both devices working concurrently on the same workload. Such collaborative execution makes better use of the overall system resources by employing *both* CPU threads and FPGA concurrency, thereby achieving higher performance.

In this paper, we explore the potential of collaborative execution between CPUs and FPGAs using OpenCL High Level Synthesis. First, we compare various collaborative techniques (namely, *data partitioning* and *task partitioning*), and evaluate the tradeoffs between them. We observe that choosing the most suitable partitioning strategy can improve performance by up to 2×. Second, we study the impact of a common optimization technique, *kernel duplication*, in a collaborative CPU-FPGA context. We show that the general trend is that kernel duplication improves performance until the memory bandwidth saturates. Third, we provide new insights that application developers can use when designing CPU-FPGA collaborative applications to choose between different partitioning strategies. We find that different partitioning strategies pose different tradeoffs (e.g., task partitioning enables more kernel duplication, while data partitioning has lower communication overhead and better load balance), but they generally outperform execution on conventional CPU-FPGA systems where no collaborative execution strategies are used. Therefore, we advocate even more integration in future heterogeneous CPU-FPGA systems (e.g., OpenCL 2.0 features, such as fine-grained shared virtual memory).

KEYWORDS

CPU-FPGA architectures, Heterogeneous systems, OpenCL, Performance analysis

ACM Reference Format:
Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia De Gonzalo, Juan Gómez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, and Wen-mei Hwu. 2019. Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3310305

1 INTRODUCTION

The demand for processing larger amounts of data with higher performance under constrained power and energy budgets makes *heterogeneity* a fundamental feature of computing systems. Therefore, heterogeneous architectures (e.g., CPU-GPU, CPU-FPGA) are now ubiquitous in modern data centers and supercomputers [1–3]. In addition to powerful CPUs, current computing systems typically employ various types of specialized devices, such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Tensor Processing Units (TPUs), and other Application-Specific Integrated Circuits (ASICs). FPGAs are particularly interesting because they provide a tradeoff between performance and programmability,

*Li-Wen did this work when he was a PhD student at UIUC.

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310305

when programmed with High Level Synthesis (HLS) frameworks, like Intel FPGA SDK [4] for OpenCL and Xilinx SDAccel [5]. FPGAs are not only suitable for accelerating applications under stringent energy efficiency requirements [6–8], but they are also being increasingly adopted in cloud servers and data centers [9–14]. For example, Microsoft has built an Earth-scale FPGA-based network (the Catapult V2 [10, 11]), which enables network flows to be programmably transformed at line rate in the cloud, thereby accelerating both network functions and applications. Other major efforts using FPGAs in the cloud include IBM SuperVesselCloud [12], Amazon EC2 [9], Microsoft Brainwave [15], and the Intel CPU-FPGA deep learning inference accelerator card (DLIA) [13]. Intel estimates that FPGAs will run in 30% of data center servers in 2020 [14].

Traditionally, accelerators (including FPGAs) have been used as *offload engines*, where an entire kernel runs on the accelerator while the CPU remains idle, waiting for the result [16–19]. More recently, vendors provide interconnect technologies such as Intel QuickPath Interconnect (QPI) [20], Hyper Transport [21], Front Side Bus (FSB) [22], Accelerator Coherency Port (ACP) [23], AXI Coherency Extension (ACE) [24], ARM CoreLink Interconnect [25], IBM Coherent Accelerator Processor Interface (CAPI) [26], and Cache Coherent Interconnect for Accelerators (CCIX) [27]. In terms of functionality, these interconnects operate in a similar manner, but their details vary across CPU architectures, processor implementations, and silicon fabrication. These interconnects enable tighter integration between CPUs and FPGAs in SoC chips [28–30] and server-grade systems [31, 32].

The trend towards tighter integration of CPUs and FPGAs enables more *collaborative execution* between devices. Rather than executing an entire kernel on the FPGA while the CPU is idle, collaborative execution makes better use of the overall system resources by involving *both* CPU threads and FPGA in the execution. One of the key challenges of collaborative execution between CPUs and FPGAs is the identification of the best strategy for partitioning work between the CPU and the FPGA. There are two major approaches to partitioning of work. The first approach, called *data partitioning*, is to have the CPU and the FPGA perform the same task on different subsets of the data. The second approach, called *task partitioning*, is to have each device perform a different sub-task and communicate intermediate results between them. Each partitioning strategy entails its own tradeoffs, and different applications may benefit from different strategies. The factors that impact the suitability of each partitioning strategy encompass 1) the latency and bandwidth of inter-device communication, 2) the disparity in the workload's performance on the CPU versus the FPGA, 3) the diversity of computation phases within a task, and 4) the hardware resource constraints. Each strategy poses its own challenges, such as how much data to assign to each device or which sub-tasks to assign to which device. Our goals in this work are 1) to evaluate different collaborative execution strategies for CPU-FPGA systems by analyzing their effectiveness and their tradeoffs, and 2) to provide insights for designing future CPU-FPGA collaborative applications. Though our work focuses on integrated CPU-FPGA systems, it could be extended to collaborative execution using other types of accelerators in heterogeneous systems.

We make the following contributions:

- We carry out the first quantitative evaluation of collaborative execution strategies with OpenCL HLS on CPU-FPGA systems using benchmarks from diverse fields (e.g., image processing, graph processing, producer-consumer computing, computer graphics, etc.).
- We propose new analytical models for different collaborative execution strategies that assist us in estimating their performance of different strategies.
- We rigorously analyze the tradeoffs of different partitioning strategies for collaborative execution, and provide insights to help developers make informed decisions when designing collaborative programs for CPU-FPGA systems.

2 COLLABORATIVE EXECUTION STRATEGIES

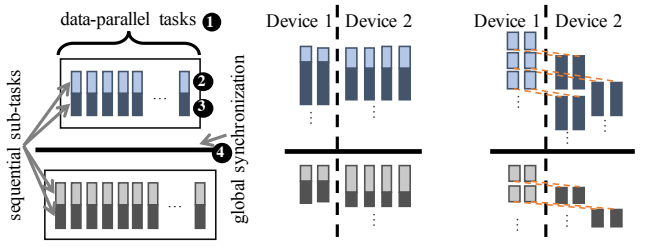
We first define collaborative execution and two main strategies for it, namely data partitioning and task partitioning. Then, we propose analytical models to estimate the performance of data partitioning and task partitioning.

Collaborative execution refers to an application execution structure where the CPU and the FPGA (or another accelerator) both participate in performing the computations required by the application, as opposed to the traditional offload accelerator model where the entire kernel is executed on the FPGA while the CPU thread waits for the result. The strategies for collaboratively executing a program on different types of devices can be classified into two main categories: data partitioning and task partitioning. We discuss these in Sections 2.1 and 2.2 respectively. Many programs are amenable to both data partitioning and task partitioning, and thus programmers need to choose between them. In this paper, we aim to provide insights to assist programmers in making the right decisions when writing collaborative programs for integrated CPU-FPGA systems.

Throughout this section, we illustrate the collaborative execution strategies using the simple example shown in Figure 1(a). In this example, a program consists of many data-parallel tasks ❶ that are applied to different data elements. Each data-parallel task consists of multiple types of sub-tasks (two in this case), and the result of the first sub-task ❷ is required for the execution of the second sub-task ❸. In some cases, a program may consist of multiple phases where there is a global synchronization point ❹ across all data-parallel tasks. In OpenCL, these phases are typically expressed as separate *kernels*, since global synchronization across the entire device is not supported in the programming model.

2.1 Data Partitioning

Data partitioning is a collaborative execution strategy wherein different devices perform the *same task on a different subset of the data*, i.e., the data-parallel tasks are distributed across devices. Figure 1(b) illustrates this strategy. The main challenge with data partitioning is determining the optimal partitioning, i.e., the distribution of data-parallel tasks across devices that results in the highest performance. One possibility is *static partitioning* where a fixed fraction of the data-parallel tasks is statically assigned to each device prior to execution. Another possibility is *dynamic partitioning* where the



(a) Program Structure (b) Data Partitioning (c) Task Partitioning

Figure 1: Program with Many Data-parallel Tasks (a) and Two Collaborative Execution Strategies: Data Partitioning (b) and Task Partitioning (c)

data-parallel tasks are dynamically assigned to different devices from a task pool during execution until all tasks are exhausted.

To better understand and analyze the collaborative execution patterns, we establish analytical models. We use the abstraction of *workers* to represent the processing units on a device. A thread on a CPU is a CPU worker. A processing element on an FPGA is considered an FPGA worker. We use the following notation to describe the application and system properties:

N – Number of data parallel tasks in the application
$t_{i,C}$ – Execution time of sub-task i by a CPU worker
$t_{i,F}$ – Execution time of sub-task i by an FPGA worker
w_C – Number of available CPU workers
w_F – Number of available FPGA workers

To define an analytical model for data partitioning, let α be the fraction of data-parallel tasks executed by the CPU, and let β_{data} be the factor of increase in the total execution time (i.e., overhead) due to distributing tasks and merging partial results. The total execution time of the application is expressed as:

$$t_{data,total} = \beta_{\text{data}} \cdot max\left(\frac{\alpha N \sum_i t_{i,C}}{w_C}, \frac{(1-\alpha) N \sum_i t_{i,F}}{w_F}\right) \quad (1)$$

The overall execution time is the maximum of the execution times on the CPU and the FPGA since the device that finishes first needs to wait for the other device.

To minimize the total execution time when performing data partitioning, α must be tuned such that the two terms in $max(\cdot,\cdot)$ are equal. This ensures load balance, and thus minimizes device idleness and the overall execution time.

The optimal α^* can be obtained as:

$$\alpha^* = \frac{\sum_i t_{i,F}}{w_F} \Big/ \left(\frac{\sum_i t_{i,C}}{w_C} + \frac{\sum_i t_{i,F}}{w_F}\right) \quad (2)$$

The optimal α^* is therefore determined by $t_{i,C}$, $t_{i,F}$, w_C, and w_F, which are specific to the application and the system. Statically determining the optimal α^* requires profiling the program and a performance model of the system.

Alternatively, to maximize the performance of data partitioning without the need for program profiling and a system performance model, *dynamic* data partitioning can be used. As opposed to static data partitioning where α is determined and fixed before execution, dynamic data partitioning does not partition the data using a fixed ratio. Instead, data is partitioned into fine-grained data blocks and those fine-grained data blocks are dynamically assigned to devices (and workers) from a task pool until all tasks are exhausted. The fraction of total data blocks that each device ends up processing is largely dependent on the relative performance difference of the different devices involved. With dynamic data partitioning, load balance between devices can be achieved. However, dynamic data partitioning might have a higher β_{data} due to the additional overheads caused by contention on the task queue.

2.2 Task Partitioning

Task partitioning is a collaborative execution strategy wherein different devices execute *different types of sub-tasks on the entire set of the data*, i.e, within each data-parallel task, different types of devices perform different types of sub-tasks. Figure 1(c) illustrates this strategy. The main challenge with task partitioning is to determine which type of sub-tasks within the data-parallel tasks is more suitable for each device. Even if one device is better at all types of sub-tasks, task partitioning may still be beneficial, if it makes better utilization of the devices that might be otherwise idle, thus improving parallelism. Since the sub-tasks within a data-parallel task are sequential, task partitioning creates a dependency between devices such that one device must wait for intermediate results from another device before executing its sub-task. However, with multiple tasks available, significant parallelism can still be achieved across devices via pipeline-style (i.e., pipeline-parallel) execution, as illustrated in Figure 1(c).

To define an analytical model for task partitioning, we use the same notation defined in Section 2.1. In addition, let S_C and S_F be the sets of indices of sub-tasks to be executed on the CPU and the FPGA respectively. Note that $S_C \cap S_F = \emptyset$ and $S_C \cup S_F$ is the set of indices of all sub-tasks. Also let β_{task} be the percentage of total execution time increase (i.e., overhead) due to communication overhead in task partitioning. In task partitioning, the execution of sub-tasks on the CPU and the FPGA may or may not be overlapped, depending on the granularity of task partitioning and pipelining. If the target platform supports fine-grained task partitioning, which allows the CPU processing and the FPGA processing to perfectly overlap with each other, the total execution time depends on the execution time of the device that takes longer to finish. The total execution time of the application in this case can be expressed as:

$$t_{task,total} = \beta_{\text{task}} N \cdot max\left(\frac{\sum\limits_{i \in S_C} t_{i,C}}{w_C}, \frac{\sum\limits_{i \in S_F} t_{i,F}}{w_F}\right) \quad (3)$$

If the target platform only supports coarse-grained task partitioning, where no overlap between CPU processing and accelerator processing is possible, the total execution time is the sum of the execution time of the CPU and the FPGA:

$$t_{task,total} = \beta_{\text{task}} N \cdot \left(\frac{\sum\limits_{i \in S_C} t_{i,C}}{w_C} + \frac{\sum\limits_{i \in S_F} t_{i,F}}{w_F}\right) \quad (4)$$

More generally, in cases where the CPU and the FPGA processing have some level of overlap, the total execution time must fall into the range given by Equations 3 and 4. Therefore, Equation 3 and

Equation 4 give the lower bound and upper bound, respectively, of task partitioning execution time.

Optimizing the performance of task partitioning increases in difficulty as the number of sub-tasks increases, since the number of combinations of S_C and S_F grows exponentially with respect to the number of sub-tasks. Thus, minimizing Equations 3 and 4 is not straightforward. It requires a performance model of the underlying hardware or manual effort, which are out of the scope of this work. This work mainly focuses on how collaborative execution patterns can be better used, and any automatic or manual optimization of Equations 3 and 4 is orthogonal to our work.

3 METHODOLOGY

We use OpenCL programs from the Chai benchmark suite [33], which is developed to evaluate collaborative execution. Table 1 shows the benchmarks we evaluate, along with the collaborative execution strategy used by each one. The benchmarks are compiled with the Intel FPGA SDK for OpenCL 16.0 [4]. For the comparative evaluation of the two collaborative execution strategies in Section 4, we use Canny Edge Detection and Random Sample Consensus, since these two benchmarks support both partitioning schemes.

Table 1: Evaluated Chai Benchmarks [33].

Benchmark	Description	Strategy
CED-D	Canny Edge Detection	Data Partitioning
CED-T	Canny Edge Detection	Task Partitioning
RSC-D	Random Sample Consensus	Data Partitioning
RSC-T	Random Sample Consensus	Task Partitioning
BS	Bézier Surface	Data Partitioning
HSTO	Image Histogram	Data Partitioning
SSSP	Single-Source Shortest Path	Task Partitioning
TQ	Task Queue System (Synthetic)	Task Partitioning
TQH	Task Queue System (Histogram)	Task Partitioning

We perform our evaluation on the two systems shown in Table 2. Note that the Nallatech 510T data center acceleration card hosts two Arria 10 1150 GX FPGAs, but the current OpenCL Board Support Package (BSP) from the vendor is a beta version that is limited to one FPGA and two DDR4 slots, so only one FPGA and 8GB device memory are used for the evaluation. The Arria 10 FPGA has more logic and DSP resources than the Stratix V FPGA does, and also features hard floating-point DSP blocks, which the Stratix V FPGA does not.

Table 2: System Specifications.

FPGA Board	Terasic DE5-Net [34]	Nallatech 510T [35]
FPGA	Stratix V GX [36]	Arria 10 GX [37]
Device Memory	4GB (DDR3)	8GB (DDR4)
Host CPU	Xeon E3-1240 v3 [38]	Xeon E5-2650 v3 [39]
Host Memory	8GB (DDR3)	96GB (DDR4)
Interface	PCIe gen3.0 ×8	PCIe gen3.0 ×8

In the experiments, we repeat the execution and measurement five times for each test. The reported execution time is the averaged execution time of five runs.

In Figures 3 and 4, we show results for both FPGAs, which summarize the comparison between data and task partitioning for CED and RSC. For the remaining results, we only show results for the Stratix V FPGA for brevity, but the trends are similar on both systems we evaluate. Unless otherwise specified, we report the results for the best performing CPU thread count (from among 1, 2, and 4 threads) and the best performing duplication factor. Besides, we use the built-in Intel FPGA dynamic profiler for OpenCL provided by the Intel OpenCL SDK to profile the execution of FPGA kernels and identify performance bottlenecks.

4 EVALUATION OF COLLABORATIVE EXECUTION STRATEGIES

In this section, we evaluate the performance of CPU-FPGA collaborative execution and analyze the sources of the performance improvements and bottlenecks.

4.1 Canny Edge Detection

Canny Edge Detection (CED) [40] is an edge detection algorithm that is commonly used for image processing. It consists of four stages: (1) a Gaussian filter, (2) a Sobel filter, (3) non-maximum suppression, and (4) hysteresis. We apply these four stages of the algorithm to a stream of video frames. In the data partitioning version of the benchmark, each device processes a different set of frames. In the task partitioning version, the first two stages are executed on the FPGA while the remaining two stages are executed on the CPU for all frames. We choose this style of task partitioning because Gaussian and Sobel filters are more regular whereas non-maximum suppression and hysteresis contain more control flow for which CPUs are well-optimized.

Data Partitioning (CED-D). Figure 2 shows the execution time of the CED-D benchmark for different data partitioning distributions. The values on the horizontal axis indicate the fraction of frames processed by the CPU in static partitioning swept in increments of 0.1, with the last pair of bars showing the results for dynamic partitioning. Here, static partitioning statically assigns a subset of frames to process to each device, while dynamic partitioning uses one CPU control (proxy) thread for each device (CPU or FPGA), fetching frames and sending them to the corresponding device, as soon as the device is available. In this benchmark, the overhead due to the control threads is negligible, since the granularity of partitioning is coarse (an entire frame). The execution time is broken down into compute time, copy time (for the FPGA only), and idle time (the time a device waits for the other device to finish).

From the results, we make three major observations. First, processing all frames on the FPGA (shown by the $\alpha = 0.0$ bars) achieves shorter execution time than processing all frames on the CPU (shown by the $\alpha = 1.0$ bars). Second, the data partitioning strategy outperforms both the CPU and the FPGA. For this particular workload, the sweet spot for static partitioning (among the tested distributions) is $\alpha = 0.4$ where the CPU processes 40% and the FPGA processes 60% of the video frames. Third, dynamic partitioning eliminates the idle time completely, thus outperforming the best static partitioning, and providing the lowest execution times.

Task Partitioning (CED-T). Figure 3 compares the execution time of collaborative execution with task partitioning to data partitioning

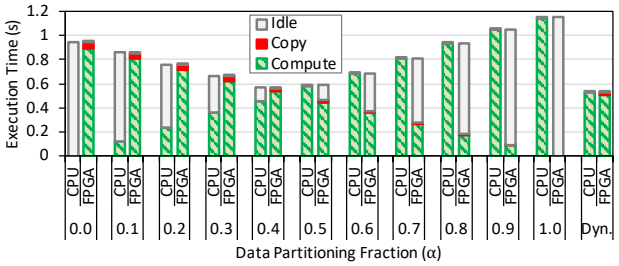


Figure 2: Execution Time of Canny Edge Detection (CED) with Different Data Partitioning Fractions (α) and with Dynamic Data Partitioning. α is the Fraction of Data-parallel Tasks Assigned to the CPU.

and to no collaboration. The execution time is broken down into compute time, copy time (for the FPGA only), and idle time (the time the device waits for the other device to finish).

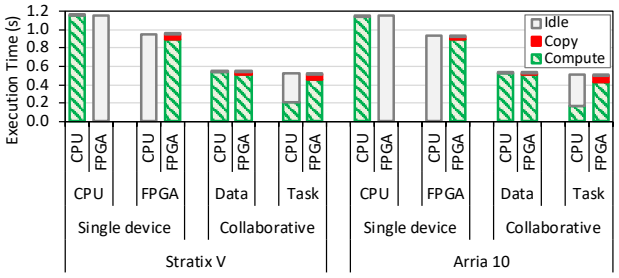


Figure 3: Execution Time of Canny Edge Detection (CED) across CPU-FPGA Systems and Collaborative Execution Strategies.

We make three major observations. First, the overall best performance of task partitioning is comparable to the overall best performance of data partitioning. Second, task partitioning incurs more communication overhead, with copy time accounting for 11.4% of overall execution time in task partitioning and only 4.4% in data partitioning. One reason is that with task partitioning, data from all data-parallel tasks must be copied to the FPGA, while with data partitioning, only a subset of the data needs to be copied. It is expected that the availability of coherent memory will make this communication overhead less of an issue. This is particularly important for workloads that are less compute-intensive, making copy-time a larger fraction of the total execution time. Third, in task partitioning, there is still some idle CPU time, whereas in data partitioning, this idle time is negligible due to dynamic data partitioning. Despite that, the performance of both strategies is comparable, as pointed out in our first observation. This fact represents a potential advantage of task partitioning over data partitioning, where different devices are more suitable or specialized for different workloads. If the sub-tasks assigned to the CPU in task partitioning were more time-consuming, the CPU could still use that fraction of idle time. However, the overall execution time of data partitioning would increase, since in data partitioning all sub-tasks run on *all* devices.

From the comparison of CED-D and CED-T, we derive three major conclusions. First, in data partitioning, finding the best load balance across devices is possible with static or dynamic partitioning. Second, task partitioning can greatly benefit from device specialization. Third, communication overhead in task partitioning is larger than that in data partitioning.

4.2 Random Sample Consensus

Random Sample Consensus (RSC) [41] is an algorithm for estimating the parameters of a model by taking random samples of an input iteratively until a successful model is found. A single iteration of RSC consists of two stages: (1) model fitting using random samples and (2) evaluating the model accuracy by computing outliers and error values. The iterations are independent and can be done in parallel [42]. In data partitioning, each device processes a different set of iterations. In task partitioning, the first stage is executed on the CPU while the second stage is executed on the FPGA. We do so because the first stage is inherently sequential while the second stage is massively parallel, and thus better suited for the FPGA.

Data Partitioning (RSC-D). Figure 4 shows the execution time of RSC for different static data partitioning fractions. We make two observations. First, the static partitioning sweet spot for RSC-D is at 50% for each device, which is different from CED, highlighting the need to optimize data partitioning strategies individually for different applications. Second, the performance of RSC-D is lower on the Arria 10 than on the Stratix V, mainly because the clock frequency of the FPGA implementation of this kernel is lower on the Arria 10. The reason for the lower clock frequency could come from either the beta BSP of Nallatech 510T (see Section 3) or better optimization on the BSP of Terasic DE5-Net.

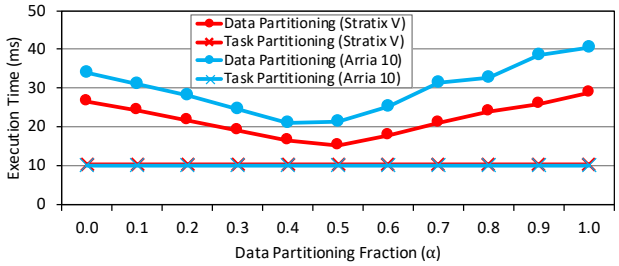


Figure 4: Execution Time of Random Sample Consensus (RSC) across CPU-FPGA Systems and Collaborative Execution Strategies. α is the Fraction of Data-parallel Tasks Assigned to the CPU in Data Partitioning.

The RSC implementation *cannot* perform dynamic data partitioning because the granularity of partitioning is smaller than that in CED. In CED, each data-parallel task is the processing of an independent frame with four OpenCL kernels launched by a CPU proxy thread. Thus, the CPU proxy threads perform dynamic partitioning by accessing a pool of data-parallel tasks via a shared atomic variable. However, in RSC, a single OpenCL kernel is launched to compute *all* data-parallel tasks. Within this kernel, an FPGA worker (an OpenCL *work-group*) computes each data-parallel task. Since current CPU-FPGA systems do *not* support OpenCL 2.0 shared virtual memory and system-wide atomic instructions, it is not possible for FPGA workers to access the same atomic variable as CPU

threads. Thus, dynamic data partitioning is not possible for RSC. OpenCL 2.0 shared virtual memory and system-wide atomic instructions are desirable architectural features in future CPU-FPGA systems for more effective collaborative execution.

Task Partitioning (RSC-T). Figure 4 shows that RSC with task partitioning noticeably outperforms RSC with the best data partitioning. The reason is that data partitioning exhausts the DSP blocks in the FPGA, since the first stage of RSC employs intensive floating-point computations. As task partitioning assigns the first stage to the CPU, the FPGA can utilize more resources for the second stage. This enables a higher degree of kernel duplication, a common optimization technique that duplicates the number of processing elements, which potentially translates into a significant performance improvement. We discuss kernel duplication in detail in Section 5.

4.3 Other Data Partitioning Benchmarks

Bézier Surface (BS). Bézier surfaces are parametric structures widely used in computer graphics and finite element modeling. This benchmark uses non-rational formulation of Bézier surfaces on a regular 2D surface. BS performs data partitioning on the output data by dividing the output surface into square tiles, which are assigned to different CPU threads and different OpenCL work-groups [43].

Figure 5 shows the execution time breakdown of BS running on the Arria 10 FPGA, with the data partitioning fraction α ranging from 0 to 1. We make two major observations. First, the kernel time varies as data partitioning fraction α changes, with $\alpha = 0.7$ minimizing the execution time. Second, for all the α's, most of the total execution time is spent on the kernel computation. The other parts of execution (e.g., data copy, allocation, and deallocation) account for a very small fraction of the total execution time.

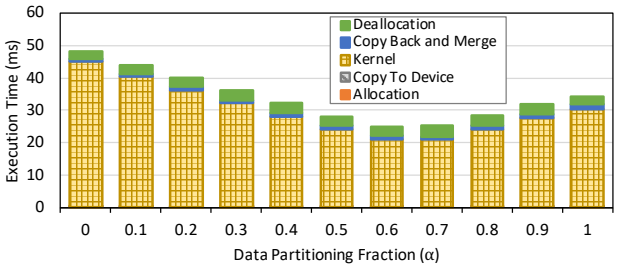


Figure 5: Execution Time of Bézier Surface (BS) with Different Data Partitioning Fractions (α). α is the Fraction of Data-parallel Tasks Assigned to the CPU.

Image Histogram (HSTO). A histogram describes the frequency of data falling into each of some predefined bins. Histogram computation is a frequently-used routine in many applications [44, 45], for example, image processing and pattern recognition. This benchmark implements the histogram computation of pixels of an image by binning pixels based on their value ranges. It uses data partitioning on the output bins, i.e., part of the bins are assigned to the CPU and the other part to the FPGA. Both the CPU and the FPGA process the entire set of input data, but they increase a bin counter only if an input data value falls into their assigned set of bins. This way, the CPU and the FPGA do not update the same bin counters.

Figure 6 shows the execution time breakdown of HSTO on the Arria 10 FPGA with various α values. A major observation is that the overall execution time of HSTO is almost *independent* of the data partitioning factor α. The reason is that in HSTO, both the CPU and the FPGA workers need to traverse through the large input data, the overhead of which overwhelms the benefit from data partitioning, leading to no performance improvement from any level of data partitioning.

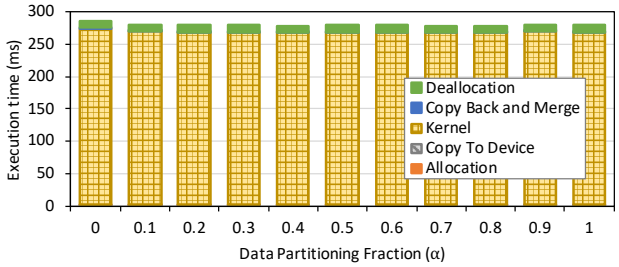


Figure 6: Execution Time of Histogram (HSTO) with Different Data Partitioning Fractions (α). α is the Fraction of Data-parallel Tasks Assigned to the CPU.

4.4 Other Task Partitioning Benchmarks

Single-source Shortest Path (SSSP). SSSP is a commonly-used graph algorithm that identifies the path between two vertices in a graph that has the minimal sum of weights of edges on the path. This benchmark has irregular memory access patterns and requires atomic instructions. SSSP performs coarse-grained task partitioning across a series of tasks, each of which is an iteration of the algorithm which constructs a frontier of vertices (i.e., the list of vertices to visit in the next iteration) and updates the shortest distance. The frontier of vertices is a queue data structure for communicating among tasks. Since graph structures are irregular, the size of vertex frontiers varies, which causes imbalance in processing times of different tasks. In SSSP, iterations are assigned to the CPU or the FPGA to process the frontier according to the *frontier size*. Small frontiers are assigned to the CPU while the large ones are assigned to the FPGA. This is based on the observation that the FPGA performs better for larger frontiers, where more parallelism can be exploited and the kernel launch overhead is less significant. On the CPU, CPU threads dequeue vertices from a vertex queue and process them sequentially. On the FPGA, different OpenCL *work-items* process different vertices and aggregate results with atomic instructions.

Task Queue Systems (TQ and TQH). Task queue systems (TQ and TQH) exemplify a type of the producer-consumer computing pattern, where the host enqueues tasks into some task queues in the device memory, while the device dequeues and processes the tasks. TQ works with synthetic data, while TQH generates the histograms of video frames (i.e., the input of each task is a video frame, and the output is its histogram). In both benchmarks, the CPU threads generate and enqueue tasks, while the FPGA processes the tasks. As soon as the FPGA finishes with one task, it dequeues a new task. This way, the workload across work-groups is balanced.

We evaluate the effect of kernel duplication for SSSP, TQ and TQH in Section 5.

5 EVALUATION OF KERNEL DUPLICATION

In this section, we evaluate the effect of a common optimization technique, kernel duplication [4], on the performance of the benchmarks described in Section 4. With kernel duplication, multiple identical hardware instances (processing elements) are instantiated on the FPGA from the same OpenCL kernel. The Intel OpenCL SDK for FPGAs provides a programming attribute to specify the *duplication factor* of OpenCL kernels, i.e., the number of identical processing elements on the FPGA. The OpenCL work-items are executed on these hardware processing elements in a SIMD manner, which can potentially improve performance. With kernel duplication, the utilization of the available configurable logic on the FPGA increases, since there are more processing elements. However, the higher resource utilization increases the complexity of place and route on the FPGA circuit, and therefore could potentially lower the operating frequency and increase the execution time. A higher number of processing elements can also lead to memory access contention in the memory system [46–52].

With kernel duplication, the number of FPGA workers w_F in the analytical model presented in Section 2 increases, while the execution time of each individual sub-task $t_{i,F}$ may increase due to changes in frequency and resources. Therefore, the tuning of the kernel duplication factor is dependent on the tradeoff between the the ability to exploit more parallelism and the overhead of having multiple processing elements on the FPGA.

5.1 Performance Effect of Kernel Duplication

Figure 7 shows the impact of possible kernel duplication factors on the overall performance of four data-partitioning benchmarks (CED-D, RSC-D, BS, and HSTO) with various data partitioning factors α. As shown in the figure, in the FPGA-only case ($\alpha = 0$), RSC-D, BS and HSTO benefit from duplicating kernels. For CED-D, kernel duplication does not change performance significantly. As discussed in Section 4, HSTO has a different behavior from the other benchmarks in terms of data partitioning – HSTO's performance is almost independent of the α value. The reason is that, in HSTO, partitioning happens only on output bins, and both CPU and FPGA need to traverse through a large amount of input data, the overhead of which hides the benefit of partitioning. In HSTO, kernel duplication is still beneficial, since the computation capacity of the FPGA and, thus, the whole system increases with kernel duplication.

We make three observations from Figure 7. First, a higher duplication factor does not necessarily lead to higher performance because of the tradeoffs we discuss above. Second, when kernel duplication is actually beneficial, the best α values tend to be smaller (i.e., more workload assigned to the FPGA) for higher duplication factors, since the computation capacity of the FPGA increases. Third, for larger values of α, kernel duplication has almost no impact on overall execution time, since most of the workload of the applications is assigned to the CPU.

Figure 8 shows the speedups from kernel duplication on three task-partitioning benchmarks (SSSP, TQH, and TQ). As shown in the figure, SSSP does not benefit much from kernel duplication because its irregular memory access pattern quickly saturates the

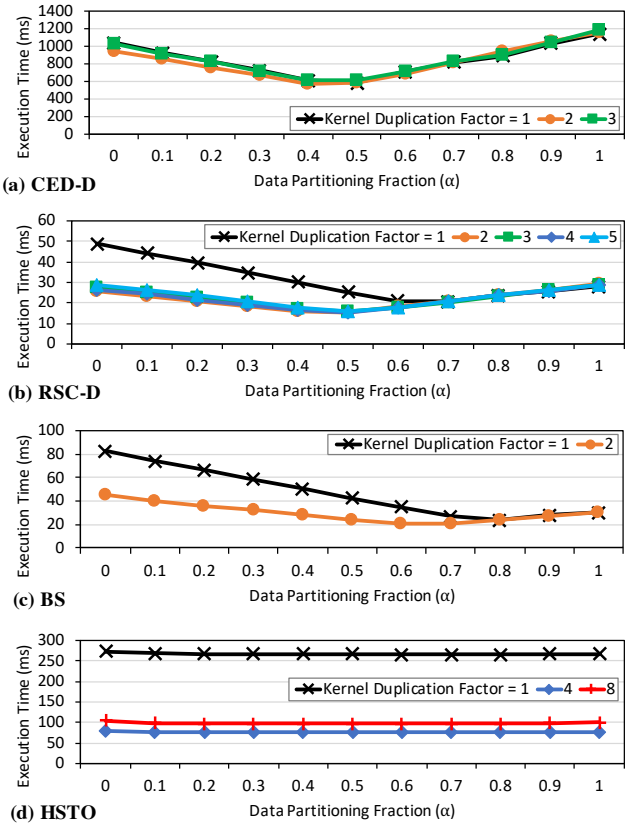


Figure 7: Execution Time of Data Partitioning Benchmarks (CED-D, RSC-D, BS, and HSTO) for Different Kernel Duplication Factors and Data Partitioning Fractions (α). α is the Fraction of Data-parallel Tasks Assigned to the CPU.

memory bandwidth. On the other hand, in TQH and TQ, performance scales well with the duplication factor due to regularity of their memory access patterns and load balancing across workgroups.

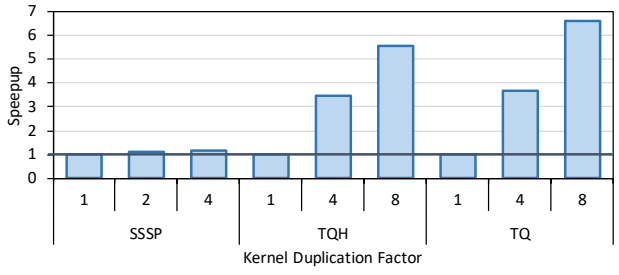


Figure 8: Speedup (Normalized to Kernel Duplication Factor 1) of Task Partitioning Benchmarks (SSSP, TQ, and TQH) for Different Kernel Duplication Factors.

Figures 9, 10 and 11 show the execution time breakdowns for SSSP, TQ, and TQH, respectively, on the Arria 10 FPGA. As the figures show, across all three benchmarks, kernel duplication reduces

the kernel execution time without affecting other portions of the total execution time.

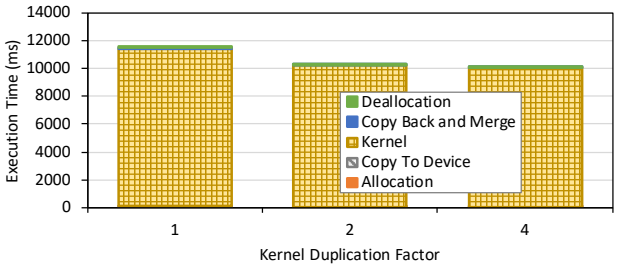


Figure 9: SSSP: Execution Time Breakdown for Different Duplication Factors.

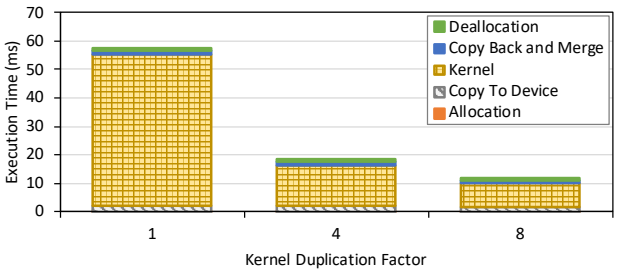


Figure 10: TQ: Execution Time Breakdown for Different Duplication Factors.

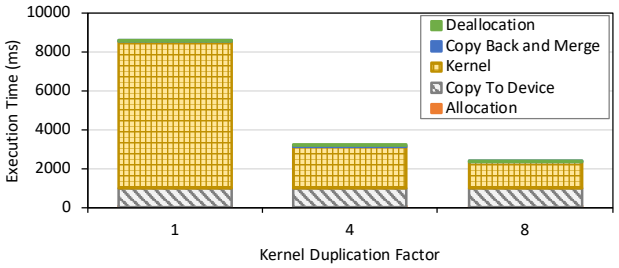


Figure 11: TQH: Execution Time Breakdown for Different Duplication Factors.

5.2 Analysis of Resource Utilization

We further analyze how kernel duplication changes FPGA resource utilization and how it impacts performance. For this analysis, we focus on Canny Edge Detection and Random Sample Consensus.

Canny Edge Detection (CED-D/CED-T). Figure 12 shows the effect of the kernel duplication factor on performance, as well as other utilization metrics. Mapping to the axis on the left, the light blue bar represents performance (higher is better) in terms of the speedup over data partitioning with a duplication factor of 1, and the black line represents the frequency of the FPGA processing elements also normalized to data partitioning with a duplication factor of 1. Mapping to the axis on the right, the different lines represent the utilization in terms of percentage of the logic, DSP blocks, and RAM blocks on the FPGA.

We make two major observations. First, the duplication factor has little impact on the performance of CED, and too much duplication may even slightly hurt its performance. Further profiling using the Intel FPGA OpenCL Profiler reveals that the main reason behind the performance effect of the duplication factor on CED is the saturation of the memory bandwidth. More processing elements on the FPGA exhaust the available bandwidth. Second, task partitioning tends to lead to less resource pressure and higher FPGA frequency for the same duplication factor. The reason is that, in task partitioning, only the sub-tasks that run on the FPGA need to be synthesized on the FPGA, while in data partitioning all sub-tasks need to be synthesized. As a result, the maximum duplication factor for task partitioning is higher than data partitioning.

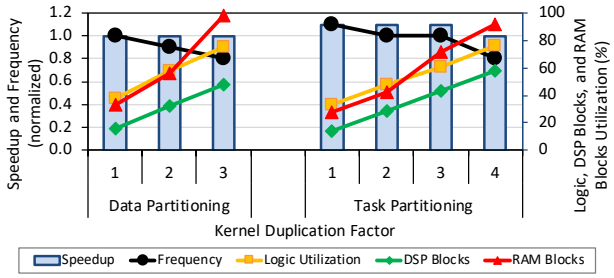


Figure 12: Canny Edge Detection: Speedup and Frequency (Normalized to Data Partitioning with Duplication Factor 1) and Resource Utilization for Different Duplication Factors.

Random Sample Consensus (RSC-D/RSC-T). Figure 13 shows the impact of the kernel duplication factor on the performance of RSC-D and RSC-T and FPGA utilization metrics. The bars, lines, and axes are set up in the same way as in Figure 12.

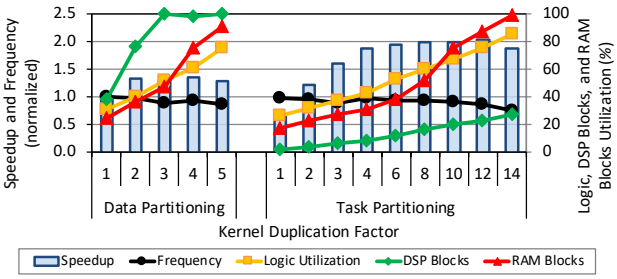


Figure 13: Random Sample Consensus: Speedup and Frequency (Normalized to Data Partitioning with Duplication Factor 1) and Resource Utilization for Different Duplication Factors.

In data partitioning, we observe reasonable performance improvement with a duplication factor of 2, but there is little improvement beyond that. Unlike all the other cases where the bounding resources are the RAM blocks, the bounding resources for RSC-D are the DSP blocks. This is due to the fact that the first stage of RSC-D performs a large amount of floating-point computations.

In the task partitioning strategy for RSC, because the first stage is offloaded to the CPU, the DSP block utilization drops significantly, enabling the kernel duplication factor to continue to increase, resulting in much better performance for task partitioning than for data partitioning. The performance improvement saturates around a kernel duplication factor value of 8, which results in a 1.6× speedup for task partitioning over data partitioning. Similar to CED, the profiler shows that the saturation of the memory bandwidth is the major reason why performance saturates at higher values of the kernel duplication factor.

6 KEY INSIGHTS

Based on the performance evaluations we present in Sections 4 and 5, we extract five key insights for developers who wish to write collaborative programs for CPU-FPGA architectures. These insights cover generic collaborative computing techniques for heterogeneous systems, as well as CPU-FPGA specific collaboration schemes.

The first insight is that collaborative execution is actually beneficial. We observe that with both data and task partitioning strategies, collaborative execution effectively reduces the execution time of almost all benchmarks we examine.

Second, data partitioning requires careful choice of partitions to provide the highest performance. We observe that the different data partitioning benchmarks (i.e., BS, CED-D, HSTO, and RSC-D) prefer different data partitioning fractions α that result in the best performance (Section 4). This observation emphasizes the need for application-specific heuristics or offline tuning for finding the best static data partitioning, or the use of dynamic data partitioning. We show that dynamic data partitioning is effective in minimizing idle time in CED-D (Section 4.1). Unfortunately, CED-D is the only evaluated benchmark that supports dynamic data partitioning. Dynamic data partitioning requires the use of shared memory variables to implement a task pool. CED-D uses two proxy CPU threads to control the execution on the CPU and the FPGA (i.e., launch CPU threads and FPGA OpenCL kernels for every task). The proxy threads can access the same task pool in the CPU memory and assign tasks to the CPU and the FPGA. In other data partitioning benchmarks, the CPU threads and the OpenCL kernel are launched only once at the beginning of the execution. Thus, in order to implement a shared task pool, CPU and FPGA workers would need to access the same shared variables, as CPU-GPU systems do [33]. However, shared virtual memory is not available in current CPU-FPGA systems. Future integration of shared coherent memory features and system-wide atomic instructions in FPGAs [53] will make dynamic partitioning more feasible.

Third, task partitioning generally enables more kernel duplication on the FPGA than data partitioning does, because task partitioning does not need to dedicate FPGA resources to all types of sub-tasks in an application, as it runs some computation stages entirely on the CPU. In RSC, there is a large difference between different sub-tasks. The first sub-task is sequential and much more floating-point intensive than the second sub-task. Thus, task partitioning saves FPGA resources that can be used for a higher kernel duplication factor than data partitioning. As a result, RSC-T outperforms RSC-D (Section 4.2). However, more kernel duplication does not always imply better performance. The potential benefit of kernel duplication is benchmark-specific (Section 5). In CED, the different sub-tasks are very similar to each other in terms of computation and resource requirements. They compete for the memory bandwidth. Hence, the higher kernel duplication factor of CED-T than CED-D does not provide performance benefits (Figure 12). Even if kernel duplication is effective, there can be diminishing returns from increasing the kernel duplication factor too much, if the memory bandwidth saturates, as we show for RSC in Figure 13. In summary, developers must carefully use kernel duplication, in order to make effective use of the FPGA resources and thus improve application performance.

Fourth, data partitioning inflicts less burden on programmers and has less communication overhead than task partitioning. In task partitioning, tasks can only be partitioned into sub-tasks at specific points in the code. Finding the partitioning points might be painful for the programmer and makes it more difficult to evenly balance the workload across devices. Moreover, task partitioning tends to require more communication and synchronization points between devices, because both devices participate in all tasks. Emerging shared coherent memory features [53] (e.g., fine-grained memory coherence and system-wide atomic instructions) are expected to be beneficial in making such communication and synchronization easier.

Fifth, the current OpenCL stack for FPGAs provides a convenient programming model for application programmers, but there is still room for better programmability and higher performance if new features are provided inside the OpenCL stack. We believe that incorporating more OpenCL 2.0 features, such as fine-grained shared virtual memory [53] and system-wide atomic instructions [54], to the OpenCL stack for FPGAs will greatly benefit programmability and performance.

7 RELATED WORK

To our knowledge, this is the first work to perform a thorough analysis of collaborative execution strategies on CPU-FPGA systems programmed with High Level Synthesis tools, like OpenCL. In this section, we first review recent works on collaborative execution on CPU-FPGA systems programmed with register-transfer level (RTL) code. Second, we review works on OpenCL programming for FPGAs. Third, we discuss recent efforts on collaborative execution for integrated CPU-GPU architectures.

7.1 CPU-FPGA Coherent Memory

CPU-FPGA platforms with shared coherent memory have recently captured great attention from both academia and industry. Choi et al. [55] conduct a quantitative study of modern CPU-FPGA platforms, including QPI-based and PCIe-based ones. This study focuses mainly on micro-benchmarking for memory systems and evaluates acceleration of *entire kernels* on FPGAs. Our work focuses on evaluating *collaborative* execution strategies on CPU-FPGA platforms.

Enabled by the tighter integration of the CPU and the FPGA in CPU-FPGA systems, collaborative execution has been analyzed in various studies that accelerate applications. Weisz et al. [56] present a task-partitioning collaborative strategy to accelerate linked-list traversals. Chang et al. [57] accelerate seeding in DNA sequence

alignment through a data-partitioning collaborative strategy. István et al. [58] adopt task-partitioning collaborative execution for regular expression operators for databases. Zhang et al. [59] present a task-partitioning collaborative algorithm to accelerate merge sort. Qiao et al. [60] accelerate the Deflate lossless compression algorithm on an FPGA. They apply a task-partitioning-based collaborative execution strategy for an entire compression service on a CPU-FPGA system which takes advantage of pipeline parallelism. Sidler et al. [61] accelerate pattern matching queries using a task-partitioning collaborative strategy. Schmit et al. [62] present a use case of a CPU-FPGA system, where the FPGA serves as a smart network transmitter/receiver and the CPU runs applications, using a task-partitioning collaborative execution strategy. These studies focus on accelerating specific applications by writing RTL code, while our work focuses on evaluating multiple collaborative patterns comparatively for each selected application using OpenCL HLS.

Several studies [63–66] focus on integration of different types of *accelerators* in heterogeneous systems with general purpose CPUs. Our work mainly focuses on collaborative execution strategies for CPU-FPGA platforms, but could be further extended to other accelerators in heterogeneous systems.

7.2 High-level Synthesis with OpenCL

High-level synthesis (HLS) with OpenCL has been widely adopted to accelerate FPGA design due to its programmability. Ndu et al. [67] present and evaluate a benchmark suite, CHO, for OpenCL FPGA accelerators. Verma et al. [68] evaluate OpenCL HLS using OpenDwarfs benchmarks and identify optimization techniques for OpenCL HLS. Ramanathan et al. [69] propose a work-stealing technique using OpenCL atomics on FPGAs. Wang et al. [70] present a performance analysis framework to identify bottlenecks of OpenCL kernels synthesized on FPGAs. Multiple recent studies accelerate applications using OpenCL HLS, including particle identification [71], relational queries [72], convolutional neural networks [6], etc. Most of these studies focus on accelerating or evaluating entire kernels on FPGAs. Our work evaluates collaborative execution patterns with OpenCL HLS on CPU-FPGA platforms.

7.3 Integrated CPU-GPU Architectures

Collaborative execution on heterogeneous PCIe-based CPU-GPU systems with discrete GPUs has been studied from various aspects. Shen et al. [73] propose a workload partitioning scheme for heterogeneous CPU-GPU systems. This work proposes modeling, profiling, and prediction techniques to predict the best workload partitioning. Luk et al. [74] propose adaptive mapping to automatically map computation to CPU-GPU systems. The proposed techniques adapt to changes in input problem size and system configuration. These works mainly focus on discrete CPU-GPU systems without much discussion on how tight integration of the CPU and the GPU on a single chip further enables acceleration opportunities.

Collaborative execution strategies have been studied for integrated CPU-GPU systems using benchmark suites such as Heteromark [75–77], Chai [33, 78], and HeteroSync [79]. We leverage benchmarks from Chai [33] to evaluate collaborative execution strategies on CPU-FPGA systems. Sun et al. [80] evaluate the Radeon Open Compute Platform using collaborative benchmarks. Gómez-Luna et al. [81] present three use cases of collaborative execution on a CPU-GPU system with the Heterogeneous System Architecture (HSA) [82]. Che et al. [83] study data partitioning between CPUs and GPUs specifically for betweenness centrality. Tang et al. [84] propose EMRF, a policy for balancing between fairness and efficiency in integrated CPU-GPU architectures. FinePar [85] and Cho et al. [86] automatically partition workloads to use both CPUs and GPUs in integrated CPU-GPU architectures. Airavat [87, 88] is a power management framework that improves the energy efficiency of collaborative CPU-GPU applications. HAShCache [89] adds a stacked DRAM as a shared last-level cache for integrated CPU-GPU processors to address the problem of disparity between the two devices in their demands on the memory system. Garcia-Flores et al. [90] evaluate integrated CPU-GPU systems with a shared last-level cache using collaborative benchmarks. Staged Memory Scheduling [91] is a multi-level QoS-aware memory scheduler for integrated CPU-GPU systems. Kayiran et al. [92] propose a concurrency management mechanism for integrated CPU-GPU systems to control the usage of memory and network by the CPU and the GPU. Garcia-Flores et al. [93] analyze the inefficiencies of demand paging in CPU-GPU systems when running collaborative workloads, and explore data sharing between the CPU and the GPU at finer granularity than a page (e.g., a cache line). Spandex [94] is a memory coherence interface specifically targeting integrated architectures. Vesely et al. [95, 96] enable system calls from GPUs which benefits for having shared virtual memory across CPUs and GPUs. These works represent the numerous research efforts on software and hardware approaches to collaborative execution on integrated CPU-GPU systems. Our work is the first step towards similar research lines for integrated CPU-FPGA systems with OpenCL.

8 CONCLUSION

In this paper, we present strategies for collaborative execution on CPU-FPGA architectures and evaluate these strategies using existing collaborative OpenCL applications with high-level synthesis. To our knowledge, this is the first paper to carry out a comprehensive analysis of collaborative execution on CPU-FPGA systems using the OpenCL programming framework. We show that collaborative execution outperforms the execution on conventional CPU-FPGA systems where no collaborative execution strategies are used. We describe the challenges that each collaborative execution strategy faces, providing insights for developers on how to use them. We find that 1) task partitioning enables more kernel duplication, a common optimization technique for FPGAs, than data partitioning, yet 2) data partitioning has lower communication overhead and achieves better load balance than task partitioning. We provide suggestions for emerging CPU-FPGA systems, where support for fine-grained shared coherent memory and system-wide atomic instructions would be beneficial. We believe and hope that our study will inspire FPGA developers to further explore collaborative execution on CPU-FPGA architectures to achieve the highest performance and efficiency. Our study could also be extended to other types of accelerators in heterogeneous systems.

ACKNOWLEDGMENTS

This work was supported by Hewlett Packard Labs and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. We also thank Intel, VMware, Huawei, AliBaba, and Google for their gift funding support.

REFERENCES

[1] Erich Strohmaier, Jack Dongarra, Simon Horst, and Martin Meuer. Top500 List June 2018.
[2] Feng Wu and Tom Scogland. Green500 List June 2018.
[3] RightScale. Rightscale 2018 state of the cloud report.
[4] Intel. Intel FPGA SDK for OpenCL. Programming Guide, October 2016.
[5] Xilinx. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[6] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.
[7] Sai Rahul Chalamalasetti, Martin Margala, Wim Vanderbauwhede, Mitch Wright, and Parthasarathy Ranganathan. Evaluating FPGA-acceleration for real-time unstructured search. In *ISPASS*, 2012.
[8] D. Chen, J. Cong, Y. Fan, and L. Wan. LOPASS: A Low-Power Architectural Synthesis System for FPGAs With Interconnect Estimation and Optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2010.
[9] Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/, 2018.
[10] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *MICRO*, 2016.
[11] 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 *ISCA*, 2014.
[12] New OpenPOWER cloud boosts ecosystem for innovation and development. http://www-03.ibm.com/press/us/en/pressrelease/47082.wss, 2015.
[13] Intel. Intel Deep Learning Inference Accelerator Product Specification and User's Guide. https://www.intel.com/content/dam/support/us/en/documents/server-products/server-accessories/Intel_DLIA_UserGuide_1.0.pdf, July 2017.
[14] The first chip from Intel's Altera buy will be out in 2016. http://fortune.com/2015/11/18/intel-xeon-fpga-chips/, 2015.
[15] Doug Burger. Microsoft unveils Project Brainwave for real-time AI. *Microsoft Research*, 2017.
[16] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *FPT*, 2011.
[17] S. Liu, A. Papakonstantinou, H. Wang, and D. Chen. Real-time object tracking system on fpgas. In *SAAHPC*, 2011.
[18] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W. Hwu, and Deming Chen. Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 275–284, New York, NY, USA, 2017. ACM.
[19] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *FPL*, 2017.
[20] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *HOTI*, 2010.
[21] HyperTransport Technology Consortium et al. Hypertransport i/o link specification. *Revision*, 1:111–118, 2008.
[22] Altera. Accelerating High-Performance Computing With FPGAs. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01029.pdf.
[23] Accelerator Coherency Port. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434a/BABGHDHD.html.
[24] AXI Coherency Extensions. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/BABIAFAJ.html.
[25] Arm CoreLink Interconnect. https://developer.arm.com/products/system-ip/corelink-interconnect.
[26] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.
[27] Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com, 2016.
[28] Xilinx. Zynq UltraScale+ MPSoCs. White Paper, June 2016.
[29] Altera. Altera's User-Customizable ARM-Based SoC, 2015.
[30] Mark Hummel, Mike Krause, and Douglas O'Flaherty. AMD and HP: Protocol enhancements for tightly coupled accelerators. 2007.
[31] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Convey HC-1 – the potential of FPGAs in numerical simulation. *Preprint Series of the Engineering Mathematics and Computing Lab*, (07), 2010.
[32] Convey Computer. The Convey HC-2 computer. Architectural overview, 2012.
[33] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *ISPASS*, 2017.
[34] Terasic. *DE5-Net User Manual*, 2018.
[35] Nallatech. *Nallatech 510T Product Brief*, 2018.
[36] Intel. Intel Stratix V FPGAs. https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-v.html.
[37] Intel. Intel Arria 10 FPGAs. https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.
[38] Intel. Intel Xeon Processor E3-1240 v3. https://ark.intel.com/products/75055/Intel-Xeon-Processor-E3-1240-v3-8M-Cache-3-40-GHz-.
[39] Intel. Intel Xeon Processor E5-2650 v3. https://ark.intel.com/products/81705/Intel-Xeon-Processor-E5-2650-v3-25M-Cache-2-30-GHz-.
[40] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.
[41] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 1981.
[42] Juan Gómez-Luna, Holger Endt, Walter Stechele, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Egomotion compensation and moving objects detection algorithm on GPU. In *PARCO*, 2011.
[43] Rafael Palomar, Juan Gómez-Luna, Faouzi A. Cheikh, Joaquín Olivares-Bueno, and Ole J. Elle. High-performance computation of bézier surfaces on parallel and heterogeneous platforms. *International Journal of Parallel Programming*, 2018.
[44] J. Gómez-Luna, J.M. González-Linares, J.I. Benavides, and N. Guil. An optimized approach to histogram computation on GPU. *Machine Vision and Applications*, 2013.
[45] J. Gómez-Luna, J.M. Gónzalez-Linares, J.I. Benavides, and N. Guil. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems*, 2013.
[46] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX SECURITY*, 2007.
[47] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[48] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.
[49] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The Application Slowdown Model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*, 2015.
[50] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.
[51] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.
[52] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.
[53] Khronos group. The OpenCL specification. *Version 2.0*, 2015.
[54] M. Gupta, D. Das, P. Raghavendra, T. Tye, L. Lobachev, A. Agarwal, and R. Hegde. Implementing cross-device atomics in heterogeneous processors. In *IPDPS Workshops*, 2015.
[55] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *DAC*, 2016.
[56] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *FPGA*, 2016.
[57] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu. The SMEM seeding acceleration for DNA sequence alignment. In *FCCM*, 2016.
[58] Z. István, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *FCCM*, 2016.
[59] Chi Zhang, Ren Chen, and Viktor Prasanna. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *IPDPS*, 2016.
[60] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *FPGA*, 2018.

[61] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD*, 2017.
[62] Herman Schmit and Randy Huang. Dissecting Xeon+FPGA: Why the integration of CPUs and FPGAs makes a power difference for the datacenter. In *ISLPED*, 2016.
[63] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. A. Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini. Exploring architectural heterogeneity in intelligent vision systems. In *HPCA*, 2015.
[64] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *ISLPED*, 2013.
[65] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *DAC*, 2015.
[66] H. Usui, L. Subramanian, K. Chang, and O. Mutlu. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM TACO*, 2016.
[67] Geoffrey Ndu, Javier Navaridas, and Mikel Luján. CHO: Towards a benchmark suite for OpenCL FPGA accelerators. In *IWOCL*, 2015.
[68] Verma Anshuman, Ahmed E Helal, Konstantinos Krommydas, and Wu-chun Feng. Accelerating workloads on FPGAs via OpenCL: A case study with OpenDwarfs. *Virginia Tech CS Tech. Rep.*, 2016.
[69] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A Constantinides. A case for work-stealing on FPGAs with OpenCL atomics. In *FPGA*, 2016.
[70] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*, 2016.
[71] S. Sridharan, P. Durante, C. Faerber, and N. Neufeld. Accelerating particle identification for high-speed data-filtering using OpenCL on FPGAs and other architectures. In *FPL*, 2016.
[72] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. In *FPL*, 2016.
[73] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload partitioning for accelerating applications on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2016.
[74] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.
[75] 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 *IISWC*, 2016.
[76] Saoni Mukherjee, Yifan Sun, Paul Blinzer, Amir Kavyan Ziabari, and David Kaeli. A comprehensive performance analysis of HSA and OpenCL 2.0. In *ISPASS*, 2016.
[77] Saoni Mukherjee, Xiang Gong, Leiming Yu, Carter McCardwell, Yash Ukidave, Tuan Dao, Fanny Nina Paravecino, and David Kaeli. Exploring the features of OpenCL 2.0. In *IWOCL*, 2015.
[78] Li-Wen Chang, Juan Gómez-Luna, Izzat El Hajj, Sitao Huang, Deming Chen, and Wen-mei Hwu. Collaborative computing for heterogeneous integrated systems. In *ICPE*, 2017.
[79] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs. In *IISWC*, 2017.
[80] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David Kaeli. Evaluating performance tradeoffs on the radeon open compute platform. In *ISPASS*, 2018.
[81] J. Gómez-Luna, I.-J. Sung, A.J. Lázaro-Muñoz, W.-H. Chung, J.M. González-Linares, and N. Guil. Chapter 8 - Application use cases: Platform atomics. In *Heterogeneous System Architecture*. 2016.
[82] Wen-mei W. Hwu. *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. 2015.
[83] Shuai Che, Marc Orr, and Jonathan Gallmeier. Work stealing in a shared virtual-memory heterogeneous environment: A case study with betweenness centrality. In *CF*, 2017.
[84] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures. In *SC*, 2016.
[85] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *CGO*, 2017.
[86] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R Gross. On-the-fly workload partitioning for integrated CPU/GPU architectures. In *PACT*, 2018.
[87] Trinayan Baruah, Yifan Sun, Shi Dong, David Kaeli, and Norm Rubin. Airavat: Improving energy efficiency of heterogeneous applications. In *DATE*, 2018.
[88] Trinayan Baruah. Energy efficient execution of heterogeneous applications. Master thesis. Northeastern University, 2017.
[89] Adarsh Patil and Ramaswamy Govindarajan. HAShCache: Heterogeneity-aware shared DRAMCache for integrated heterogeneous systems. *ACM TACO*, 2017.
[90] V. Garcia-Flores, J. Gómez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Peña. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *IISWC*, 2016.
[91] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012.
[92] O. Kayiran, N. Chidambaram Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. Managing GPU concurrency in heterogeneous architectures. In *MICRO*, 2014.
[93] V. Garcia-Flores, E. Ayguade, and A. J. Peña. Efficient data sharing on heterogeneous systems. In *ICPP*, 2017.
[94] Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. Spandex: a flexible interface for efficient heterogeneous coherence. In *ISCA*, 2018.
[95] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. Generic system calls for GPUs. In *ISCA*, 2018.
[96] Arkaprava Basu, Joseph L Greathouse, Guru Venkataramani, and Ján Veselỳ. Interference from GPU system service requests. In *IISWC*, 2018.