# MOCHA: Multinode Cost Optimization in Heterogeneous Clouds with Accelerators

Peipei Zhou[1,2], Jiayi Sheng[3], Cody Hao Yu[2,4], Peng Wei[2,5], Jie Wang[2], Di Wu[2,6], Jason Cong[2] *
[1]University of Pittsburgh [2]University of California, Los Angeles
[3]Microsoft [4]Amazon Web Services [5]Google [6]Facebook
peipei.zhou@pitt.edu,{hyu,peng.wei.prc,jiewang,allwu,cong}@cs.ucla.edu,jisheng@microsoft.com

## ABSTRACT

FPGAs have been widely deployed in public clouds, e.g., Amazon Web Services (AWS) and Huawei Cloud. However, simply offloading accelerated kernels from CPU hosts to PCIe-based FPGAs does not guarantee out-of-pocket cost savings in a pay-as-you-go public cloud. Taking Genome Analysis Toolkit (GATK) applications as case studies, although the adoption of FPGAs reduces the overall execution time, it introduces 2.56× extra cost, due to insufficient application-level speedup by Amdahl's law. To optimize the out-of-pocket cost while keeping high speedup and throughput, we propose Mocha framework as a distributed runtime system to fully utilize the accelerator resource by accelerator sharing and CPU-FPGA partial task offloading. Evaluation results on HaplotypeCaller (HTC) and Mutect2 in GATK show that on AWS, Mocha saves on the application cost by 2.82× for HTC, 1.06× for Mutect2 and on Huawei Cloud by 1.22×, 1.52× respectively than straightforward CPU-FPGA integration solution with less than 5.1% performance overhead.

## KEYWORDS

CPU-FPGA Co-scheduling; Cost Optimization; Analytic Modeling; Distributed Runtime; Accelerators; Heterogeneous; Public Clouds

**ACM Reference Format:**
Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, Jason Cong. 2021. MOCHA: Multinode Cost Optimization in Heterogeneous Clouds with Accelerators . In Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3431920.3439304

## 1 INTRODUCTION

Field-programmable gate arrays (FPGAs) are gaining in popularity to accelerate a variety of applications in data centers for high performance and energy efficiency. Many public clouds, including AWS Elastic Compute Cloud (EC2), Huawei, Baidu and Alibaba [1, 3, 6, 23], have released their FPGA-powered instances. However, the overall system-level speedup could be limited due to CPU-FPGA data transfer inefficiency even if the FPGA accelerator is well designed. Although a number of approaches have been proposed to improve the system efficiency [11–13, 19], none of them paid enough attention to the out-of-pocket cost, one of the most important issues in the public cloud market.

To illustrate the cost issue in public clouds with FPGAs, we conduct case studies with FPGA accelerators in widely used genome variant calling programs in Genome Analysis Toolkit (GATK) [31]: HaplotypeCaller (HTC) [25] and Mutect2 [10]. GATK is one of the most popular toolsets in computational genome analysis. HTC and Mutect2 are the two most time-consuming applications in GATK that aim to find germline variants for pair-end sequence reads and tumor sequence reads. Both applications feature a high-complexity computation kernel called Pair Hidden Markov Models (PairHMM) [15] that can be accelerated by 40× on FPGA. This, however, leads to the result that HTC costs $6.35 on `f1.2xlarge` AWS EC2 instance with an FPGA when comparing to the cost as $2.46 on `m4.2xlarge` general purpose CPU instance, indicating prohibitive cost overhead, compared to a 1.6× end-to-end speedup. A key question is being raised: *How does using FPGA accelerators impact an application's out-of-pocket cost in public cloud services?*

The rationale is fairly straightforward: FPGA instances are priced higher than general purpose CPU instances, so applying FPGA accelerators has to bring high enough application-level speedup to achieve cost saving. Taking AWS EC2 as an example, Table 1[1] shows that the FPGA instance is priced at $1.65/Hour, which is **4.125**× over the price of CPU instance with the same type and number of virtual CPU cores (vCPU). In this case, if adopting FPGA in AWS does not achieve 4.125× application-level speedup, this solution is not as cost-efficient as pure CPU solutions. In fact, our initial CPU-FPGA integration for HTC, which lets all eight CPU cores send all PairHMM tasks to the FPGA, only demonstrates 1.6× application-level speedup for HTC, causing 2.56× of costs over the CPU-only system. Note that since the proportion of the PairHMM kernel in the whole application is 39%, the optimal application-level speedup of HTC is only 1.64× by Amdahl's law (see Section 2).

Although we cannot further improve the application-level speedup of HTC due to Amdahl's law, we can still achieve cost saving by improving the utilization of FPGA. Our PairHMM accelerator on FPGA is capable of achieving 40× speedup over a single-core CPU, the accelerator cannot be fully utilized by eight vCPU cores. As a result, we *borrow* more vCPU cores from other

*This research was performed while Peipei Zhou (intern), Jiayi Sheng, Cody Hao Yu (intern) and Di Wu were at Falcon Computing Solutions and Peipei Zhou, Cody Hao Yu and Peng Wei were graduate students at UCLA.

[1]The prices in Tables 1, 2 and 3 were collected in January 2019.


*FPGA '21, February 28-March 2, 2021, Virtual Event, USA*

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439304

**Table 1: Price comparison of CPU and FPGA instances on public cloud.**

| | CPU Instance | CPU-FPGA Instance |
|---|---|---|
| AWS EC2 [5] | 8 vCPU, $0.4/hr | 8 vCPU + 1 Xilinx VU9P, $1.65/hr (4.125×) |
| Huawei [22] | 32 vCPU, $1.64/hr | 32 vCPU + 1 Xilinx VU9P, $2.83/hr (1.725×) |

CPU instances via network to fully utilize the accelerator. In other words, we could achieve cost saving by performing many HTC tasks on a system with one FPGA and multiple CPU instances.

Based on the idea above, we design and implement Mocha framework to guarantee the cost saving for arbitrary applications with FPGA accelerators in public clouds. Mocha first profiles the given application with FPGA accelerators and identifies the performance bottleneck (CPU or FPGA). For the CPU-bottleneck applications such as HTC, Mocha improves FPGA utilization by sharing one FPGA among multiple CPU nodes through the network. For the FPGA-bottleneck applications like Mutect2, Mocha orchestrates CPU cores to execute some tasks instead of offloading all of them to the FPGA. As a result, *Mocha could improve the overall system resource utilization and thus reduce the application cost for any applications no matter how small the proportion of the kernel is as long as the FPGA kernel speedup is higher than the cost ratio.* To demonstrate the cost efficiency improvement from using Mocha, we give concrete and solid accelerator integration case studies on HTC and Mutect2 in GATK in Section 4. The evaluation is performed on both AWS and Huawei Cloud. In summary, the paper makes the following contributions:

- We analyze the out-of-pocket cost in using FPGAs on public clouds by using two cases where computation throughput of CPU and FPGA does not match, which explains how they result in extra cost over the CPU-only solution.
- We propose an end-to-end automation framework called Mocha that realizes FPGA sharing among multiple nodes through network and partial task offloading policy for CPUs in order to fully utilize FPGA and CPUs for any applications. Mocha guarantees that the cost efficiency of CPU-FPGA solution is higher than pure CPU solution as long as the FPGA kernel speedup is higher than the cost ratio.
- We provide model-driven cost optimization case studies with Mocha for two applications in GATK, HTC and Mutect2, on two different public cloud platforms AWS and Huawei Cloud. And we compare the results with two baseline solutions, pure CPU and straightforward CPU-FPGA integration.

Comparing to the straightforward CPU-FPGA integration, our optimal solution with Mocha achieves a state-of-the-art performance while saving on costs by 2.82× for HTC, 1.06× for Mutect2 on AWS, and 1.22×, 1.52×, respectively, on the Huawei Cloud.

## 2 ANALYSIS AND MODELING

We first analyze the performance and cost for an application that consists of independent parallel tasks on a single-node multiple-core CPU platform, where the execution timeline is shown in Figure 1a. For illustration purposes, we firstly assume the input size and

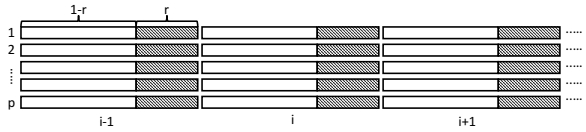

**Figure 1a: CPU-Only System**

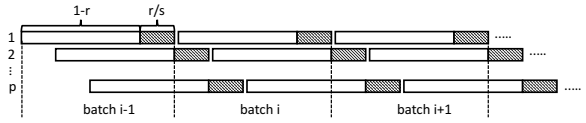

**Figure 1b: CPU-FPGA integration case a, CPU is bottleneck**

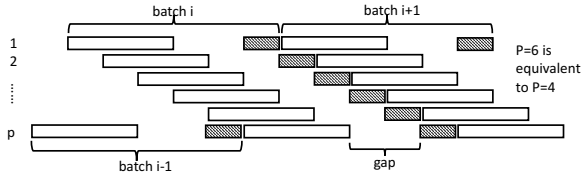

**Figure 1c: CPU-FPGA integration case b, FPGA is bottleneck**

execution time of each task are the same. Specifically, we consider the following factors in our analysis:

- $M$ is the total number of tasks.
- $P$ is the total number of CPU cores in a single node.
- $t$ is the time of each task on a CPU core.
- $r$ is the proportion of kernel that can be offloaded to a FPGA accelerator.
- $S$ is the end-to-end FPGA accelerator speedup compared to a single-core CPU. It includes the CPU-to-FPGA communication overhead.
- $c$ is the cost per unit time of a CPU core.
- $CR$ is the cost ratio of FPGA device compared to a single-core CPU. For example, on AWS `f1.2xlarge` instance, $CR$ = ($1.65/$0.4*8-8) = 25.
- $\widetilde{P}$ ( *Matching Core Number*) is the number of CPU cores in the CPU-FPGA integration system where both CPU and FPGA are 100% utilized.
- $T$ is the total runtime. $C$ is the total cost.
- $I$ is the cost index, that is, normalized cost of CPU-FPGA integrated solutions compared to pure CPU solutions. **Mocha optimization target is to achieve the lowest $I$ for CPU-FPGA integrated solutions**.

As shown in Figure 1a, in each batch, $P$ tasks are executed on $P$ CPU cores in parallel. There are $\frac{M}{P}$ batches of task in total, the total execution time $T_0$ and cost $C_0$ of a CPU-only system are:

$$T_0 = \frac{M}{P} \times t,\ C_0 = T_0 \times P \times c = M \times t \times c \tag{1}$$

Straightforward CPU-FPGA integrated runtime systems like Blaze [19] intuitively send all accelerable tasks to the FPGA, and leverage a task queue to deal with tasks requested from different workers. In this scenario, workers have to stay idle before their request can be fulfilled on the FPGA accelerator. In this subsection, we analyze the performance and cost of such systems.

Depending on $r$, $S$, and $P$, there are two cases when the computation throughput of CPU and FPGA are not balanced. Here we gradually increase $P$ to illustrate these two cases.

**Case A, $P < \widetilde{P}$: FPGA is underutilized.** When all CPU cores offload accelerable tasks to FPGA, those tasks need to be fulfilled on the FPGA sequentially. As shown in Figure 1b, cores 1 to $P$ offload tasks on FPGA in a pipeline fashion. After core 1 finishes batch $i$, it starts the non-accelerable part of batch $i+1$ on a new data partition [2]. When it is about to request accelerator in batch $i+1$, all the tasks in the previous batch have finished execution on FPGA already. Therefore, its kernel acceleration request can be fulfilled without waiting. Thus, core 1 has no idle cycles, nor do other cores. In this case, for any CPU core, it only needs to wait $\frac{r}{S} \times t$ when the FPGA is working on kernel part and there are no other idle cycles. Consequently, the execution time for each batch task is $t \times (1 - r + \frac{r}{S})$, and the total runtime $T_{1a}$ is :

$$T_{1a} = \frac{M}{P} \times t \times (1 - r + \frac{r}{S}), \tag{2}$$

For a platform with $P$ CPU cores and one FPGA, the total cost per unit time is $(P + CR) \times c$, so total cost to run the application $C_{1a}$ is:

$$\begin{aligned} C_{1a} &= T_{1a} \times (P + CR) \times c \\ &= M \times t \times c \times (1 - r + \frac{r}{S}) \times (1 + \frac{CR}{P}) \end{aligned} \tag{3}$$

Note that since the FPGA is not fully utilized, there are idle cycles between offloaded tasks. Comparing $C_{1a}$ in Equation 3 to $C_0$ in Equation 1, we can have *Cost Index* $I = \frac{C_{1a}}{C_0} = (1-r+\frac{r}{S})(1+\frac{CR}{P})$.

When we gradually increase $P$ to $\widetilde{P}$, FPGA reaches 100% utilization. This happens when total runtime of offloaded tasks from $\widetilde{P}$ cores equals to the runtime of a single batch task time. That is, $\widetilde{P} \times \frac{r}{S} \times t = (1 - r + \frac{r}{S}) \times t$. We refer to $\widetilde{P}$ as *Matching Core Number*, and it can be calculated as $\widetilde{P} = \frac{(1-r)\times S}{r} + 1$.

**Case B, $P > \widetilde{P}$: FPGA becomes bottleneck and CPU has idle cycles.** When $P$ is larger than $\widetilde{P}$, FPGA is fully utilized, and CPU cores have to wait more cycles in addition to $\frac{r}{S} \times t$. As shown in Figure 1c, after core 1 finishes non-accelerated part of batch $i$, it sends requests to the accelerator task queue. Since the offloaded task from core $P$ in batch $i-1$ has not finished yet, core 1 needs to wait until its task can be executed on FPGA. In this case, launching more than $\widetilde{P}$ CPU cores cannot further improve the application performance. Application runtime now equals to total runtime for FPGA to finish kernel execution from all $M$ tasks, as $T_{1b} = M \times \frac{r \times t}{S}$. Equivalently, application runtime equals to total runtime for CPU cores to finish $\frac{M}{\widetilde{P}}$ batches of task, which is $T_{1b} = \frac{M}{\widetilde{P}} \times (1 - r + \frac{r}{S}) \times t$. As $\widetilde{P} = (\frac{(1-r)\times S}{r} + 1)$, we can rewrite $T_{1b} = \frac{M}{\widetilde{P}} \times (1 - r + \frac{r}{S}) \times t = \frac{M}{\widetilde{P}} \times (1-r)(1+\frac{r}{(1-r)\times S}) \times t = \frac{M}{\widetilde{P}} \times (1-r)(1+\frac{1}{\widetilde{P}-1}) \times t = \frac{M}{\widetilde{P}-1} \times (1-r) \times t$. Using basic algebra rules, we have:

$$\begin{aligned} T_{1b} &= M \times \frac{r}{S} \times t = \frac{M}{\widetilde{P} - 1} \times (1 - r) \times t, \\ &= M \times t \times \frac{r + (1 - r)}{S + \widetilde{P} - 1} = \frac{M}{\widetilde{P} - 1 + S} \times t \end{aligned} \tag{4}$$

Actually, $T_{1b} = \frac{M}{\widetilde{P}-1+S} \times t$ is quite intuitive, as there are in total $M$ tasks, each with time $t$, and in the system there is a fully utilized FPGA that works as $S$ CPU cores, and $\widetilde{P}$ equivalent CPU cores. The minus 1 CPU core accounts for the penalty of CPU time that is spent waiting for FPGA kernel to be finished.

With $T_{1b}$, the total cost is:

$$C_{1b} = T_{1b} \times (P + CR) \times c = M \times t \times c \times (\frac{P + CR}{\widetilde{P} - 1 + S}) \tag{5}$$

Comparing $C_{1b}$ in Equation 5 to $C_0$ in Equation 1, *Cost Index* $I = \frac{C_{1b}}{C_0} = \frac{P+CR}{\widetilde{P}-1+S}$.

We summarize the above two cases and derive a generic model for *Cost Index* of straightforward CPU-FPGA system compared to CPU-only system as follows:

$$I = \begin{cases} (1 - r + \frac{r}{S})(1 + \frac{CR}{P}) & \text{if } P \le \widetilde{P} = (\frac{(1-r)\times S}{r} + 1) \\ \frac{P+CR}{\widetilde{P}-1+S} & \text{if } P > \widetilde{P} = (\frac{(1-r)\times S}{r} + 1) \end{cases} \tag{6}$$

**Table 2: Analysis of Cost Index I for HTC and Mutect2 on Amazon EC2 f1.2xlarge, S = 40, P = 8, CR = 25.**

| Application | $r$ | $S$ | $\widetilde{P}$ | Case A or B | $I$ |
|---|---|---|---|---|---|
| HTC | 39% | 40 | 64 | A ( 8 < 64) | 2.56× |
| Mutect2 | 89% | 40 | 6 | B ( 8 > 6) | 0.73× |

**Table 3: Analysis of Cost Index I for HTC and Mutect2 on Huawei fp.1c, S = 43, P = 32, CR = 23.**

| Application | $r$ | $S$ | $\widetilde{P}$ | Case A or B | $I$ |
|---|---|---|---|---|---|
| HTC | 39% | 43 | 68 | A ( 32 < 68) | 1.06× |
| Mutect2 | 89% | 43 | 6 | B ( 32 > 6) | 1.14× |

As shown in Table 2, both HTC and Mutect2 use PairHMM kernel, but kernel proportions are different. According to the profiling results using all the datasets shown in Table 6, pairHMM kernel takes only 39% in HTC and 89% in Mutect2. We implement a PairHMM kernel for Xilinx UltraScale FPGA on Amazon EC2 `f1.2xlarge` instance, and achieve 40× speedup over a single CPU core. To match the computation throughput of CPU cores with the PairHMM kernel in HTC, we need $\widetilde{P} = \frac{(1-39\%)}{39\%} \times 40 + 1 = 63.6$ cores, which is much larger than the number of CPU cores ($P$ = 8) on the AWS `f1.2xlarge`. In this case, FPGA board in HTC is severely underutilized. As shown in Table 2, *Cost Index I* in HTC with CPU and FPGA is about 2.56× than with only CPUs. On the other hand, for Mutect2, we only need six cores ($\widetilde{P}$ = 6), which means equivalently there are only six working CPU cores and two other cores are idle.

Similarly, Table 3 shows $I$ on Huawei `fp.1c` instance that has 32 CPUs and $CR$ = \$2.83/\$1.64*32-32 = 23. For HTC on this platform, when there are 32 CPU cores, FPGA utilization is better than that on AWS `f1.2xlarge` instance. However, it is still not fully utilized and $I$ is still larger than 1. For Mutect2, *Matching Core Number* $\widetilde{P}$ is 6, which leaves 26 CPU cores idle. Thus, $I$ is higher and now it is 1.14×, larger than 1.

To sum up, for straightforward CPU-FPGA integration, $I$ depends on $r$, $S$, $P$, $CR$ and is not guaranteed to be smaller than 1, which means the out-of-pocket cost is higher than CPU-only solution because either CPU or FPGA is underutilized.

## 3 MOCHA FRAMEWORK

In this section, we propose Mocha framework to optimize overall application cost on public clouds. We first present the key approach

[2]Here we plot the accelerated part at the end of the task to simplify the illustration. In real application, the accelerated kernel can be in anywhere in a task.

of Mocha framework in Section 3.1 that balances the throughput of CPU cores (by partially offloading kernel tasks) and FPGA (by sharing FPGA among multiple nodes) to achieve full computation resource utilization. It means Mocha guarantees the cost efficiency for any applications as long as FPGA speedup $S$ is larger than cost ratio $CR$.

According to the approach, Figure 2 depicts an overview of Mocha framework. By taking the user application and an instance list of public cloud, Mocha first launches its profiling application to obtain kernel proportion $r$, kernel speedup $S$, cost ratio $CR$ of CPU-FPGA instance and number of CPU cores $P$ on CPU instances. The information is used as the input of our cost model to achieve cost efficiency by determining the system configuration (e.g., number and type of instances to be launched, the percentage of total kernel tasks to be offloaded to FPGAs) to generate Mocha configuration in Section 3.2.

By taking the generated system configuration, Mocha runtime (Section 3.3) creates clients in the user application on each instance that needs to leverage the FPGA accelerator. The runtime, shown in the right part of Figure 2, includes client and node accelerator manager (NAM). The client is launched on CPU instances to offload partial tasks to the instance with an FPGA accelerator via network. NAM is launched on CPU-FPGA instances to receive tasks from multiple clients and schedule tasks on the accelerator.

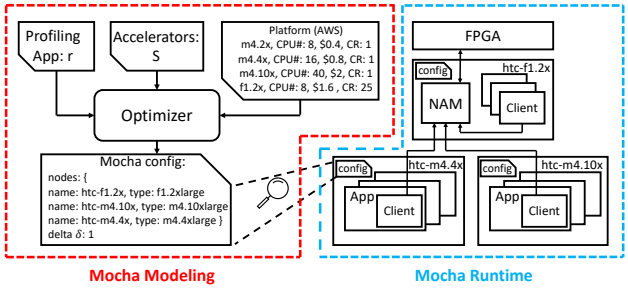


**Figure 2: Mocha framework overview**

## 3.1 CPU-FPGA Integration and Cost Modeling

In this subsection, we show that as long as $S - 1 > CR$, Mocha is **guaranteed to optimize that Cost Index I to be smaller than 1** for both Case A and Case B.

**FPGA Utilization Improvement**: For applications of Case A like HTC, one opportunity to reduce $I$ is to improve FPGA utilization by sharing FPGA among multiple nodes through network. In other words, when $\widetilde{P}$ is larger than the maximum number of CPU cores on a single node $P_0$ in a datacenter, we can launch more CPU node(s) to request the same FPGA. In Equation 6, if we set $P = \widetilde{P} = (\frac{(1-r)\times S}{r} + 1)$, we can achieve the optimized cost index $I'$ as:

$$I' = (1 - r + \frac{r}{S})(1 + \frac{CR}{\widetilde{P}}) = (1 - r + \frac{r}{S}) + (1 - r + \frac{r}{S}) \times \frac{CR}{\widetilde{P}}$$
$$= (1 - r + \frac{r}{S}) + (\frac{\widetilde{P} \times r}{S}) \times \frac{CR}{\widetilde{P}} = 1 - r + \frac{r \times (CR + 1)}{S} \quad (7)$$

We can see from the equation that when $S > CR + 1$, $I'$ is guaranteed to be smaller than 1. For HTC on AWS, if we set $\widetilde{P}$ as 64, $I'$ is 0.86 < 1, as opposed to $I$ as 2.56× in straightforward integration.

**CPU Utilization Improvement**: For Mutect2 on Huawei as shown in Table 3, PairHMM kernel dominates 89% of overall execution time, so its matching core number is $\widetilde{P} = \frac{(1-89\%)}{89\%} \times 43 + 1 = 6$. As there are in total 32 CPU cores on Huawei FP1 instance, CPU is severely underutilized (6 out of 32 are used) and $I$ is 1.14× over pure CPU solutions.

For applications of Case B like Mutect2, one opportunity to reduce $I$ is to improve CPU utilization by partial task offloading policy. As demonstrated by Figure 1c, for core 1 in batch $i$, instead of waiting extra cycles on the FPGA, core 1 can directly work on the shadow part (though using more time) to avoid waste of CPU resource. Intuitively, the most efficient way to utilize FPGA and CPU in this case is to schedule a part of the kernel tasks ($M_1$) on FPGA and the other tasks ($M_2$) on CPU, as shown in Figure 3. Thus, the overall application runtime $T'_{1b}$ and tasks number $M_1, M_2$ follow equations as:

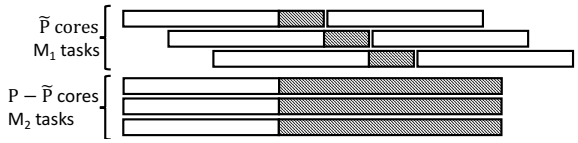


**Figure 3: Partial task offloading**

$$T'_{1b} = \frac{M_1}{\widetilde{P} - 1 + S} \times t = \frac{M_2}{P - \widetilde{P}} \times t, \quad M_1 + M_2 = M, \quad (8)$$

We can rewrite Equation 8 to $T'_{1b} = \frac{M1+M2}{P-1+S} \times t = \frac{M}{P-1+S} \times t$. As a result, the optimized cost index $I' = T'_{1b} \times (P+CR) \times c \times / C_0 = \frac{P+CR}{P-1+S}$, and $I' < 1$ when $CR < S - 1$. This is achieved when we offload $\delta = \frac{M_1}{M} = \frac{\widetilde{P}-1+S}{P-1+S}$ of the total kernel tasks to the FPGA and keep $1 - \delta = \frac{P-\widetilde{P}}{P-1+S}$ of the total kernel tasks on the CPU. For Mutect2 on Huawei, if we set $\delta = \frac{6-1+43}{32-1+43}$=0.65, $I'$ is 0.74 < 1, as opposed to $I$ as 1.14× in the straightforward integration.

To sum up, no matter $\widetilde{P}$ is larger or smaller than the number of CPU cores in a single node $P_0$, we can either choose to launch more CPU nodes or partially offload tasks to achieve *Optimized Cost Index* $I'$ as:

$$I' = \begin{cases} 1 - r + \frac{r \times (CR+1)}{S} & \text{if } \widetilde{P} > P_0, \text{ set } P = \widetilde{P} \text{ on multi-nodes} \\ \frac{P+CR}{P-1+S} = \frac{P_0+CR}{P_0-1+S} & \text{if } \widetilde{P} < P_0, \text{ set } \delta = \frac{\widetilde{P}-1+S}{P-1+S}, P = P_0 \end{cases} \quad (9)$$

Consequently, *as long as $S - 1 > CR$, $I'$ is guaranteed to be smaller than 1 in both cases.* This modeling gives quantitative support of CPU-FPGA integration for Mocha to set up a cluster with appropriate CPU-FPGA nodes and pure CPU nodes to achieve full resource utilization within the cluster.

## 3.2 Cost Model Realization

After profiling the application and FPGA accelerator to get $r$, $S$, Mocha calculates $\widetilde{P}$ as described in Equation 9. Specifically, with $r$, $s$ and the platform information which lists available CPU instances and number of CPU cores within an instance, we can obtain the number and type of CPU instances we should launch to optimize the cost efficiency. For example, on AWS EC2, `m4` series instances have `m4.x`, `m4.2x`, `m4.4x`, `m4.10x` and `m4.16x` which have 4, 8, 16, 40,

64 cores respectively. According to Equation 9, if $\widetilde{P}$ is larger than $P_0$, we set a cluster within the total $\widetilde{P}$ cores. For example, for HTC on AWS EC2, $\widetilde{P}$ is 64, which is larger than 8. We first select `f1.2x` instance, and then select other CPU nodes to get the remaining 64-8 = 56 CPU cores. We use a greedy algorithm to iteratively pick up the largest possible instance until all the remaining cores are allocated. In this example, we first pick up `m4.10x` which has 40 cores, and update the remaining cores as 56-40 = 16. Then we pick up `m4.4x` and the number of the remaining cores reaches zero. As a result, three instances including `f1.2xlarge`, `m4.10xlarge`, and `m4.4xlarge` with 8, 40, 16 cores are selected. If $\widetilde{P}$ is smaller than $P_0$ cores, we use only one CPU-FPGA instance, and set $\delta$ accordingly. For example, for Mutect2 on AWS EC2, $\widetilde{P}$ is 6, we can simply select `f1.2xlarge` and calculate $\delta$ = 95% based on Equation 9.

According to the determined system configuration, Mocha launches new instances and broadcasts the necessary information to them. In order to have a low-latency and high-throughput network among multiple nodes in AWS EC2, Mocha first creates an AWS EC2 placement group [4] and places all instances in the same group. In this case, all instances within a group have a network performance as high as 10 Gb/s (`m4.2x` and `m4.x` have 5 Gb/s network bandwidth as node limit). On Huawei cloud, all general computing instances have a 6 Gb/s network bandwidth.

### 3.3 Mocha Runtime

After the cluster has been launched, Mocha runtime starts executing the application. In Mocha runtime, there are two major components: CPU client and node accelerator manager (NAM). The CPU client is launched on all instances to communicate with the NAM for data sharing as well as task offloading to the FPGA accelerator (locally and remotely). The NAM in Mocha runtime is adapted from Blaze node manager [19, 37], an open source framework that enables FPGA accelerators as a service (FaaS). Mocha enhances the NAM by adding a feature that can divide a powerful FPGA accelerator into multiple logic accelerators.

In the rest of this section, we explain the communication mechanism among the CPU client, NAM, and the accelerator.

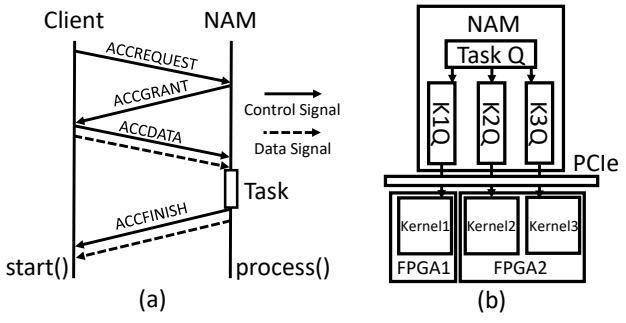


**Figure 4: (a) The communication protocol between Client and Node Accelerator Manager (NAM), (b) NAM enhanced by Mocha**

**Communication between CPU client and NAM**: As shown in Figure 4a, CPU client first connects to the instance with FPGA accelerators according to the system configuration broadcast by Mocha master. The client sends the message `ACCREQUEST` to NAM to ask for an accelerator with accelerator ID "PairHMM". If NAM has loaded the requested accelerator bitstream on FPGA, it sends `ACCGRANT` to acknowledge the client to send metadata and the input data block(s) of the tasks in `ACCDATA`. After all input data blocks are ready in NAM, NAM enqueues the task with input blocks to a task queue. After the task is finished, NAM sends back `ACCFINISH` with the metadata of output data block(s) to the client. When a client and NAM are on the same node, data are shared between a client and NAM directly through memory mapped files. When a client and NAM are on different nodes, data are shared through network by using Boost.Asio library [29]. Handling and dispatching requests in NAM are quite lightweight and impacts 4.7% overall application acceleration when using one NAM serving up to 64 software client threads.

**Communication between NAM and accelerator**: In the original Blaze, the task queue directly dispatches tasks to platform queues. Each platform queue is associated with a physical FPGA device. Mocha enhances the NAM by supporting multiple kernels in a single FPGA. As shown in Figure 4b, each kernel queue is associated with a FPGA kernel instead of a FPGA device. We configure one FPGA device to homogeneous smaller kernels for better routability. Nonetheless, our model could be extended to support heterogeneous kernels by modeling proportion r1, r2,... speedup S1, S2,... for n kernels. Then, FPGA resources will be allocated to different kernels to configure accelerator parallelism respectively so that S1 and S2 match the proportion as S1/S2 = r1/r2.

**Table 4: Time breakdown (secs) of a representative PairHMM task with 3MB input and 40KB output.**

| kernel on CPU | kernel on FPGA | PCIe | network |
|---|---|---|---|
| 91 | 2.29 | 0.0005 | 0.004 |

## 4 EXPERIMENTAL EVALUATIONS

To demonstrate Mocha, we first implement PairHMM (kernel from HTC and Mutect2) accelerator on Xilinx VCU1525 FPGAs, which is a common part on many public FPGA cloud instances. The experiments were carried out in January 2019. To the best of our knowledge, our accelerator achieves the best single-FPGA performance comparing with previous work [7, 21, 27, 32, 33]. As shown in Tables 2 and 3, The speedup $S$ is larger than $CR$+1, which guarantees improvement of cost efficiency of CPU-FPGA solutions after Mocha is used. To be noted here, $S$ is the end-to-end FPGA accelerator speedup, it includes PCIe data transfer and network communication. If an application is I/O intensive and speedup S is lower, our analytical model in Equation 9 reflects the trend that the cost ratio I' of adopting FPGA could be higher than the CPU-only solution. For PairHMM kernel on AWS EC2, we give one representative task breakdown as shown in Table 4. Here, PCIe bandwidth is assumed as 6GB/s and network bandwidth is 10Gb/s. After including network latency, we update $S$ from 39.7 to 39.6 and recalculate *Matching Core Number* in the modeling phase.

We have two baselines: a pure CPU solution and a straightforward CPU-FPGA integration solution using Blaze. For each dataset, we measure the time by calculating the average latency of 10 runs. On each platform, for each application, Mocha

**Table 5: Mocha system configuration for HTC and Mutect2 on AWS EC2 and Huawei. For example, eight `f1.2x:8` means we launch eight `f1.2x` instances, each with 8 CPU cores.**

| Application | AWS EC2 | | | | Huawei | | | |
|---|---|---|---|---|---|---|---|---|
| | $\widetilde{P}$ | pure CPU | Blaze [19] | Mocha | $\widetilde{P}$ | pure CPU | Blaze [19] | Mocha |
| HTC | 64 | `m4.16x :64` | eight `f1.2x: 8` | `f1.2x: 8, m4.10x: 40, m4.4x: 16` | 64 | `s2.16x: 64` | two `fp.1c: 32` | `fp.1c:32, s2.8x:32` |
| Mutect2 | 6 | `m4.2x:8` | `f1.2x: 8` | `f1.2x: 8` | 6 | `s2.8x: 32` | `fp.1c: 32` | `fp.1c: 32` |

**Table 6: Comparison of performance and cost of three solutions: pure CPU solution, Blaze and Mocha. A star ⋆ in normalized runtime and cost line represents that Mocha is the cheapest or fastest among the three solutions.**

| Application | SampleID | AWS EC2 | | | | | | Huawei | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | pure CPU | | Blaze [19] | | Mocha | | pure CPU | | Blaze [19] | | Mocha | |
| | | Time | Cost | Time | Cost | Time | Cost | Time | Cost | Time | Cost | Time | Cost |
| HTC | NA12878 [24] | 578 | $0.51 | 362 | $1.33 | 386 | $0.48 | 577 | $0.55 | 361 | $0.58 | 378 | $0.49 |
| HTC | NA12891 [24] | 592 | $0.53 | 381 | $1.40 | 404 | $0.50 | 591 | $0.57 | 369 | $0.60 | 373 | $0.48 |
| HTC | NA12892 [24] | 549 | $0.49 | 352 | $1.29 | 374 | $0.46 | 542 | $0.52 | 349 | $0.57 | 356 | $0.46 |
| HTC | NA12878Garvan [34] | 2767 | $2.46 | 1731 | $6.35 | 1767 | $2.18 | 2709 | $2.61 | 1710 | $2.79 | 1778 | $2.30 |
| **HTC** | **Normalized** | **1x** | **1x** | **0.63x** | **2.62x** | **0.66x** | **0.93x (⋆)** | **1x** | **1x** | **0.63x** | **1.07x** | **0.65x** | **0.87x (⋆)** |
| Mutect2 | TCRBOA1 [8] | 16784 | $1.86 | 3047 | $1.40 | 2885 | $1.32 | 4196 | $1.90 | 2807 | $2.21 | 1850 | $1.45 |
| **Mutect2** | **Normalized** | **1x** | **1x** | **0.18x** | **0.75x** | **0.17x (⋆)** | **0.70x (⋆)** | **1x** | **1x** | **0.67x** | **1.16x** | **0.44x (⋆)** | **0.76x (⋆)** |

generates a system configuration file which specifies the number and type of CPU nodes to be launched to fully utilize CPUs and FPGAs. To conduct a fair comparison of Mocha with two baselines, we launch instances for each baseline with the *same number of CPU cores* as $\widetilde{P}$ in Mocha modeling. We summarize the instances choices for the two baselines and Mocha in Table 5.

Table 6 gives comparison of performance in seconds and cost in dollars of the pure CPU solution, Blaze, and Mocha on four DNA sequences for HTC and one sequence for Mutect2 in AWS and Huawei. We also give normalized performance improvement and application cost of Blaze and Mocha as compared to the pure CPU solution. The average of normalized value of HTC and Mutect2 are shown in **bold** font to highlight the performance and cost difference of the two baselines and Mocha.

As shown in the row of averaged normalized value for HTC, Blaze incurs 2.62x extra cost than pure CPU, where inefficiency comes from underutilization of FPGA. Mocha improves cost efficiency by offloading tasks from multiple CPU instances to a single shared `f1.2x` FPGA instance. As a result, Mocha can spend less dollar per hour than Blaze (bring down 2.62x to 0.93x) while taking 0.66x of the original runtime for pure CPUs, which has 5.1% degradation compared to 0.63x of Blaze. For Mutect2, according to Table 2, $\widetilde{P}$ is 6, very close to 8 CPU cores on `f1.2x`, which implies a narrow optimization space for Mocha. In this case, by partially offloading kernel tasks from FPGA to CPU, Mocha further improves the performance of Blaze by $\frac{0.18x}{0.17x}$=1.06x. In Mutect2, as Blaze and Mocha use the same instance configuration, they spend the same amount of dollar per hour. The 1.06x performance improvement naturally translates to 1.06x (equivalently, $\frac{0.75x}{0.70x}$) cost savings. Similar analysis can be performed on Huawei Cloud. As compared to Blaze, Mocha improves cost efficiency of HTC by $\frac{1.07x}{0.87x}$ = 1.22x with 3% performance degradation. In Mutect2, Mocha improves performance by $\frac{0.67x}{0.44x}$= 1.52x, which equals to 1.52x cost savings. In summary, for HTC and Mutect2 on AWS and Huawei Cloud, Mocha provides close to the shortest (if not the shortest) and the cheapest solution among the three solutions.

## 5 RELATED WORK

**Cost Optimization on Cloud Systems.** There are a lot of existing work ( [30], HCloud [14], Paris [40], Tributary [18], Selecta [28], CherryPick [2], Doppio [41]) discussing about optimizing the cost on cloud systems. Mocha creates choices on virtual instances that are composed of CPU instances and accelerator instances and can be applied together with the existing work. Mocha creates choices on virtual instances that are composed of CPU instances and accelerator instances and can be applied together with the existing work.

**FPGA Sharing on Cloud Systems.** Researchers have devoted a lot of efforts to integrate FPGAs ([9, 16, 17, 20, 26, 35, 39]), GPUs ([36, 38]) into current cloud computing environment. Mocha provides analytical modeling and can be used as a guide in cost optimization when adopting existing CPU-Accelerator integration.

## 6 CONCLUSION

In this paper, we justify that in terms of the out-of-pocket cost FPGA is not a universal solution to accelerate all applications. It works best for applications that are rich in data parallelism, whose speedup S is higher than cost ratio CR+1. Thus, what users pay for FPGA brings throughput improvement in return. Mocha is the first work to model the cost: the very metric that customers care about when using FPGA in the cloud. From the modeling, we find the inefficiency of previous work where either FPGA or CPU resources are wasted and optimize the CPU-FPGA resources allocation by selecting proper instances provided by cloud vendors. Since CPU-FPGA ratio is set by cloud vendors and not customizable at this time, we compose "virtual" instances with the optimal CPU:FPGA ratio, which is the key to optimize the overall cost. We present performance comparison of our accelerator and provide model-driven cost optimization case studies for Genome Variant Calling applications, HTC and Mutect2, in two public cloud platforms Amazon EC2 and Huawei Cloud. On AWS, adopting Mocha gives 2.82x cost saving for HTC, 1.06x for Mutect2. While on Huawei it gives 1.22x, 1.52x cost savings respectively with less than 5.1% performance overhead.

## 7 ACKNOWLEDGMENT

We acknowledge the support from the Center for Domain-Specific Computing industrial partners. We also acknowledge the support from the University of Pittsburgh New Faculty Start-up Grant. We would like to thank all the reviewers for their valuable feedback. We thank Marci Baun for helping edit the paper and Amazon for the AWS credit donation.

# REFERENCES


[1] Alibaba. 2019. FPGA-based compute-optimized instance families. https://www.alibabacloud.com/help/doc-detail/108504.htm.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[3] Amazon. 2017. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[4] Amazon. 2019. Amazon EC2 Placement Groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[5] Amazon. 2019. AWS EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[6] Baidu. 2017. FPGA instances. https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html.

[7] S. S. Banerjee, M. el-Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer. 2017. On accelerating pair-HMM computations in programmable hardware. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[8] BCM. 2018. https://www.hgsc.bcm.edu/resources.

[9] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.

[10] Kristian Cibulskis, Michael S Lawrence, Scott L Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S Lander, and Gad Getz. 2013. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology* 31, 3 (2013), 213–219.

[11] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu. 2018. CPU-FPGA Coscheduling for Big Data Applications. *IEEE Design Test* 35, 1 (2018), 16–22.

[12] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. 2016. Heterogeneous Datacenters: Options and Opportunities. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[13] Jason Cong, Peng Wei, and Cody Hao Yu. 2018. From {JVM} to {FPGA}: Bridging Abstraction Hierarchy via Optimized Deep Pipelining. In *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[14] Christina Delimitrou and Christos Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. (2016), 473–488.

[15] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66.

[17] Gereon Führ, Seyit Halil Hamurcu, Diego Pala, Thomas Grass, Rainer Leupers, Gerd Ascheid, and Juan Fernando Eusse. 2019. Automatic Energy-Minimized HW/SW Partitioning for FPGA-Accelerated MPSoCs. *IEEE Embedded Systems Letters* 11, 3 (2019), 93–96.

[18] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R Ganger, and Phillip B Gibbons. 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 1–14.

[19] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. 2016. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 456–469.

[20] 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 *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 79–90.

[21] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W Hwu, and Deming Chen. 2017. 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*. 275–284.

[22] Huawei. 2019. Elastic Cloud Server Price Details. https://www.huaweicloud.com/en-us/price_detail.html#/ecs_detail.

[23] Huawei. 2019. Huawei FPGA-accelerated Cloud Server. https://www.huaweicloud.com/en-us/product/fcs.html.

[24] Illumina. 2019. https://support.illumina.com/downloads.html.

[25] Broad Institute. 2019. Genome Analysis Toolkit HaplotypeCaller. https://software.broadinstitute.org/gatk/documentation/tooldocs/3.8-0/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php.

[26] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F. Coutinho, and Mark Stillwell. 2016. High Performance in the Cloud with FPGA Groups. In *Proceedings of the 9th International Conference on Utility and Cloud Computing* (Shanghai, China) *(UCC '16)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[27] Megumi Ito and Moriyoshi Ohara. 2016. A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm. In *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*. IEEE, 1–3.

[28] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 759–773.

[29] Christopher Kohlhoff. 2016. Boost. asio. (2016).

[30] M. Mao and M. Humphrey. 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[31] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A DePristo. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research* 20, 9 (2010), 1297–1303.

[32] Chris Rauer and N Finamore. 2016. Accelerating Genomics Research with OpenCL and FPGAs. *Altera, Now Part of Intel, Tech. Rep* (2016).

[33] D. Sampietro, C. Crippa, L. Di Tucci, E. Del Sozzo, and M. D. Santambrogio. 2018. FPGA-based PairHMM Forward Algorithm for DNA Variant Calling. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 1–8.

[34] Standford. 2016. http://jimb.stanford.edu/giab-resources/.

[35] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 237–246.

[36] Prashanth Thinakaran, Jashwant Raj, Bikash Sharma, Mahmut T. Kandemir, and Chita R. Das. 2018. The Curious Case of Container Orchestration and Scheduling in GPU-Based Datacenters. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 524.

[37] UCLA-VAST. 2016. Blaze: Deploying Accelerators at Datacenter Scale. https://github.com/UCLA-VAST/blaze.

[38] Y. Ukidave, X. Li, and D. Kaeli. 2016. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 353–362.

[39] Z. Wang, S. Zhang, B. He, and W. Zhang. 2016. Melia: A MapReduce Framework on OpenCL-Based FPGAs. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3547–3560.

[40] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach. In *Proceedings of the 2017 Symposium on Cloud Computing*. 452–465.

[41] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. 2018. Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 22–32.