2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

Poly: Efficient Heterogeneous System and Application Management for Interactive Applications

Shuo Wang, Yun Liang[‡]
CECA, School of EECS, Peking University, China
{shvowang, ericlyun}@pku.edu.cn

Wei Zhang
Hong Kong University of Science and Technology
{wei.zhang}@ust.hk

***Abstract*—QoS-sensitive workloads, common in warehouse-scale datacenters, require a guaranteed stable tail latency percentile response latency) of the service. Unfortunately, the system load (e.g., RPS) fluctuates drastically during daily datacenter operations. In order to meet the maximum system RPS requirement, datacenter tends to overprovision the hardware accelerators, which makes the datacenter underutilized.Therefore, the throughput and energy efficiency scaling of the current accelerator-outfitted datacenter are very expensive for QoS-sensitive workloads. To overcome this challenge, this work introduces Poly, an OpenCL based heterogeneous system optimization framework that targets to improve the overall throughput scalability and energy proportionality while guaranteeing the QoS by efficiently utilizing GPUs and FPGAs based accelerators within datacenter. Poly is mainly composed of two phases. At compile-time, Poly automatically captures the parallel patterns in the applications and explores a comprehensive design space within and across parallel patterns. At runtime, Poly relies on a runtime kernel scheduler to judiciously make the scheduling decisions to accommodate the dynamic latency and throughput requirements. Experiments using a variety of cloud QoS-sensitive applications show that Poly improves the energy proportionality by 23%(17%) without sacrificing the QoS compared to the state-of-the-art GPU (FPGA) solution, respectively.**

***Keywords*-Heterogeneous; GPU; FPGA; Performance Optimization;**

I. Introduction

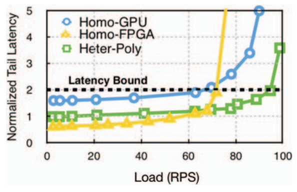
As the cloud infrastructure evolves into warehouse scale, thousands of servers are outfitted into datacenters. The quality of service (QoS) becomes a major concern for the applications running on the cloud. Most of the QoS-sensitive applications use *tail latency*, not average latency, as the latency constraint. For example, web search leaf nodes must provide 99^{th} percentile latencies of a few milliseconds [1]. As a result, servers running these QoS-sensitive workloads are often kept lightly loaded to meet the latency constraint, leading to low system utilization between 5% and 30% [2, 3]. This low utilization wastes billions of dollars in the infrastructure and energy consumption [2]. Moreover, the increasing demands from various applications have been pushing the datacenters to provide better support for improving the system throughput and energy efficiency while guaranteeing the latency constraint [4, 5].

Recently, cloud infrastructure begins to routinely use heterogeneous system outfitted with GPUs and FPGAs to service the QoS-sensitive workloads as these specialized hardware accelerators provide orders of magnitude performance and power benefits over CPUs. For example, the heterogeneous system has been used for online gaming [6], quantitative trading [7], and web search [8], and to support the needs of the next generation applications like cognitive computing [9], and intelligent personal assistant services [4]. This trend is mirrored by GPU and FPGA outfitted server offerings by cloud providers like Microsoft Cloud [10], IBM SuperVessel [9], Amazon EC2 [11], Baidu Cloud [12], and Nimbix HPC Cloud [13].

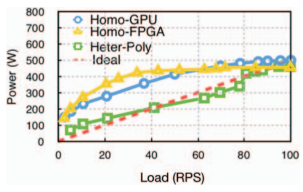
While the benefits of heterogeneous system is clear, unfortunately, the accelerators such as GPUs [14–18] and FPGAs [8, 10, 19–28] are difficult to optimize and the prior techniques designed for the CPU-based servers [5, 29–31] cannot be directly applied to GPUs and FPGAs due to the distinct architecture, programming model, and runtime environments. First, although we have a unified cross-platform programming model such as OpenCL for both GPUs and FPGAs, it only enables the function portability. The poor performance portability prevents its adoption for the heterogeneous system in cloud computing. Second, prior heterogeneous system optimization flow [4] using coarse-grained optimizations forgoes fine-grained optimization opportunities, such as different implementation possibilities with different performance trade-offs. At last, since GPUs and FPGAs exhibit quite distinct characteristics from each other with respect to performance and power, hard mapping of application kernels that depends on a single implementation solution is a fragile heuristic [4]. A promising approach is to dynamically tune a kernel implementation and allocation at runtime under the latency constraint.

In this paper, we introduce *Poly*, an optimization framework for efficient heterogeneous system and application management for cloud computing. The key enabling idea of Poly is to expose a fine-grained design space of the applications on the heterogeneous system statically and provide the flexibility to match the application to the best performing hardware accelerators (FPGA or GPU) at runtime. In order to achieve this goal, Poly combines static kernel analysis and optimization with dynamic kernel scheduling. At compile-time, to simplify and automate the kernel optimization process, Poly defines a concise set of parallel patterns using OpenCL programming model. Parallel patterns that express parallelizable computation in an abstract way have been successfully used in performance optimization for various applications [32]. More importantly, we can enable efficient

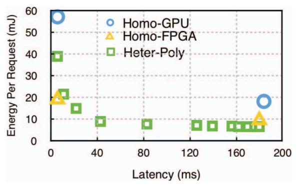
[‡]Corresponding Author



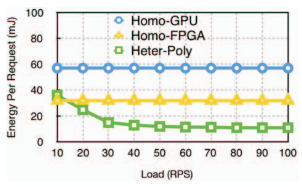
(a) tail latency v.s. load



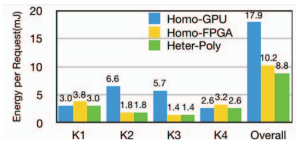
(b) power v.s. load



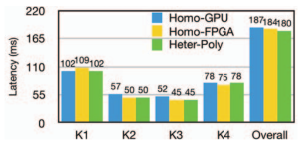
(c) design space exploration



(d) scheduling decisions



(e) energy breakdown



(f) latency breakdown

Figure 1: Comparison of Poly with other techniques.

optimization through a unified parallel pattern interface on both FPGAs and GPUs. Then, Poly explores different implementation and optimization possibilities within and across patterns for both GPUs and FPGAs. This provides a comprehensive design space that exposes different latency, throughput, and power trade-offs for each kernel. At runtime, Poly constantly monitors the performance metrics of the cloud computing system and employs a kernel scheduler to examine the various kernel implementation and allocation schemes and judiciously selects the best strategy. The optimization goal of Poly is to meet the latency constraint of QoS-sensitive workloads while improving the system throughput and energy efficiency.

Overall, this work makes the following contributions:

- **Poly Framework.** We present Poly framework to manage heterogeneous systems in datacenter with both GPUs and FPGAs, reaping the benefits from both accelerators.
- **Kernel Analysis.** We expose a rich design space with the trade-off between latency, throughput, and power for each kernel on both GPUs and FPGAs through static pattern analysis and optimization.
- **Kernel Scheduler.** We propose an efficient runtime kernel scheduler to choose the best kernel implementation and allocated hardware accelerator to meet the optimization objectives.

We use six interactive applications and conduct comprehensive experiments to evaluate Poly framework on both Xilinx and Intel FPGAs, and AMD and NVIDIA GPUs. The experimental results show that Poly not only enables fine-grained trade-offs among latency, throughput, and energy under certain QoS constraint but also improves the energy proportionality by 23% (17%) without sacrificing the QoS compared to the state-of-the-art GPU (FPGA) solution, respectively.

II. Background and Motivation

In this section, we first introduce the heterogeneous system for cloud computing, and then we present a motivation study.

A. Heterogeneous System

As cloud infrastructure increasingly relies on heterogeneity to improve performance, scalability and energy efficiency, the application execution context becomes more and more complicated with different programming abstractions and runtimes [33]. For the heterogeneous system composed of GPUs and FPGAs, a unified programming model is paramount for both programming and performance portability. In this work, we adopt the OpenCL programming model, which is an open-source parallel programming model designed for the heterogeneous system. Moreover, OpenCL has shown very promising performance speedup for the mainstream heterogeneous devices, such as GPUs [34–36] and FPGAs [8, 26].

An OpenCL program consists of a host program, which is in charge of one or more OpenCL devices, and a kernel program. Host program dynamically invokes the OpenCL kernel executed on the accelerators. In OpenCL, the basic unit of execution is a work-item and a group of work-items is bundled together to form a work-group. Multiple work-groups are combined to form a unit of execution called NDRange.

We evaluate three different heterogeneous system architectures, which uses GPUs and FPGAs differently. The first two architectures use either GPUs or FPGAs solely. We name them as *Homo-GPU* and *Homo-FPGA*, respectively. The third architecture uses both GPUs and FPGAs and we name it as *Heter-Poly*. For all the three architectures, we set a power constraint (500W for the leaf node in datacenter) and determine the number accelerators for each architecture capped by this constraint. Finally, for all these three architectures, CPUs are used to receive service requests and coordinate the workload allocation between GPUs and FPGAs.

We use *energy proportionality* (EP) as the metric to quantify the energy efficiency of cloud computing systems [29, 37]. Fig. 1(b) presents an example of the energy proportionality curves. The red dotted line represents the power scaling trend of the ideal energy proportional system whose power is linearly proportional to the system throughput. The EP is defined as follows:

$$EP = 1 - \frac{Area_{actual} - Area_{ideal}}{Area_{ideal}} \tag{1}$$

where $Area_{actual}$ and $Area_{ideal}$ is the area under the system's actual and ideal energy proportionality curve, respectively. It is necessary to note that the closer the energy proportionality curve of a system is to the ideal curve, the better its energy proportionality would be.

B. Motivation

Most of the QoS-sensitive applications running on the cloud are interactive applications. These applications run correctly as long as the latency is kept under constraints [38]. This provides us the opportunity to exploit the latency slack for throughput and energy efficiency optimization by exploring different kernel implementation with latency, throughput, and energy trade-off and dynamic kernel allocation strategy.

Here, we use an automatic speech recognition (ASR) service [39] deployed in the Google cloud computing system to illustrate the benefits of our Poly framework. In practice, ASR service is fed with a set of speech vectors and returns predictions for each feature vector that is post-processed to find the most likely sequence of text to produce the final result [40]. The core part of ASR is a Long Short-Term Memory (LSTM) network. In our experiment, the ASR service is maintained by the CPU and the requests for the ASR service are sent to it in a constant interval which is varied from 100ms to 1ms. Then, the ASR computation workloads are offloaded from CPU to GPUs and FPGAs in the heterogeneous system. We use the widely used 99^{th} percentile tail latency as the latency bound, which is set at 200ms.

ASR service is implemented using four OpenCL kernels as shown in Fig. 6. We compare our Poly framework *Heter-Poly* (GPUx1 and FPGAx5) with the *Homo-GPU* (GPUx2) and *Homo-FPGA* (FPGAx10) systems in [4] under the same total power constraint (500W) using the hardware Setting-I listed in Table III. In this setting, *Heter-Poly* uses half of the GPUs of *Homo-GPU* and half of the FPGAs of *Homo-FPGA*. In Section VI, we will evaluate *Heter-Poly* using different splits between heterogeneous devices. The prior work [4] only tries to minimize either the latency or power and predefines a hard mapping of all the kernels onto one accelerator (FPGA or GPU). However, our Poly framework explores a large design space for each kernel in ASR service and enables flexible kernel allocation at runtime.

We plot the trend of tail latency with respect to different request throughput values as shown in Fig. 1 (a). It shows that the tail latency of each system first grows slowly and then increases exponentially when the request throughput reaches certain values. Comparing *Homo-GPU* with *Homo-FPGA* systems, we can find GPU-based system could obtain higher system throughput (85 RPS) but higher tail latency while the FPGA-based system exhibits lower throughput (78 RPS) but lower latency. Under the 200ms latency constraint, the maximum system throughput of *Homo-GPU*, *Homo-FPGA*, and *Heter-Poly* systems are 68, 74, and 96 RPS, respectively. It is very promising to find that *Heter-Poly* largely increases the maximum system throughput over the *Homo-GPU* and *Homo-FPGA* by 41% and 30%, respectively under the same power constraint. As for the EP results shown in Fig 1(b), we observe that the curves of *Homo-GPU* and *Homo-FPGA* are far above the ideal one while the *Heter-Poly*'s curve is close to it. Moreover, the idle power of *Heter-Poly* is much less than the other two systems when RPS is 0. The EP values of *Homo-GPU*, *Homo-FPGA*, and *Heter-Poly* systems are 0.68, 0.63, and 0.92, respectively.

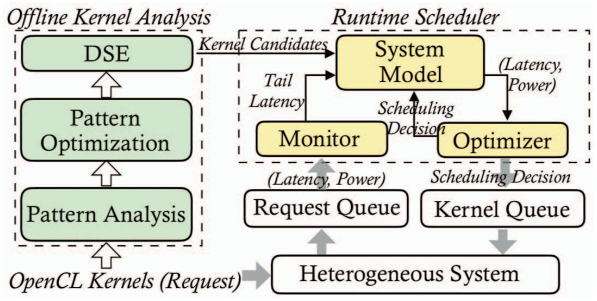


Figure 2: The Poly framework overview.

The reason for this improvement is the fine-grained design space explored by Poly provides flexible kernel implementations with different trade-offs between performance and power. As shown in Fig. 1 (c), Poly forms a design space consisting of different kernel implementations of LSTM on the Pareto-frontier with respect to energy efficiency and latency, while the prior techniques only resort to the implementation with the minimum latency or maximum energy efficiency [4]. Moreover, when the system utilization varies, Poly can dynamically adjust the kernel implementation and allocation to improve the energy efficiency, while the prior techniques do not support such flexibility as shown in Fig. 1 (d).

We also analyze the most energy efficient designs in Fig. 1(c) for three systems in details by breaking down the ASR benchmark into kernels. Fig. 1 (e) and (f) give detailed kernel-by-kernel energy and latency results for the most energy efficient designs. As we can see from Fig. 6, ASR has four kernels, and two independent kernel execution paths merge at $K4$ which are $K1 \Rightarrow K4$ and $K2 \Rightarrow K3 \Rightarrow K4$. The four kernels behave differently with respect to energy and latency on GPUs and FPGAs. The overall latency is determined by the critical path, which may vary on different systems. Fig. 1 (f) shows that *Homo-GPU* ($102 + 78 < 57 + 52 + 78ms$) is bounded by the latter path while *Homo-FPGA* ($109 + 75 > 50 + 45 + 75ms$) is bounded by the former. *Heter-Poly* schedules $K1$ and $K4$ on GPUs and $K2$ and $K3$ on FPGAs. By doing this, *Heter-Poly* improves the energy efficiency and satisfies the latency constraint.

III. Poly Framework

Fig. 2 presents an overview of Poly framework. It is mainly composed of two components which are *offline kernel analysis* and *runtime kernel scheduler*. Both components take an application written in OpenCL as input. The distinction is that the former focuses on design space exploration of a single kernel at offline while the latter schedules all the kernels of an application at runtime.

The offline kernel analysis component analyzes each kernel hierarchically from parallel pattern level to operator level. Poly first introduces a concise set of parallel patterns as shown in Fig. 3 to capture different parallel paradigm in an OpenCL kernel. In general, since a kernel may involve multiple patterns, Poly first builds a parallel pattern graph where

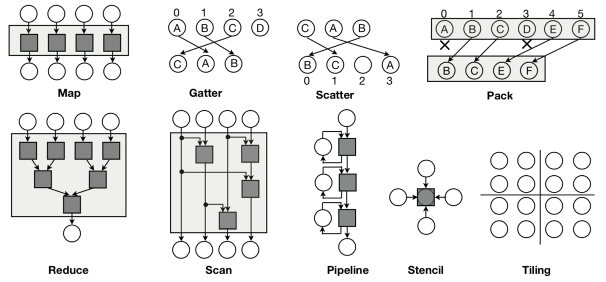


Figure 3: Overview of parallel patterns. (Circle represents the data element and the square is an operator applied to the associated input data)

each node is a parallel pattern and every edge represents the data dependency between the patterns. At the operator level, each parallel pattern is transformed into a control data flow graph (CDFG) to analyze the pattern computation in details. Then, Poly enables both local and global optimization for the parallel patterns. More clearly, for local optimization, we apply different optimization within a pattern by transforming the operators in CDFG, e.g. loop unrolling, memory partition, pipelining and etc. For global optimization, we enable a large design space exploration considering the optimization opportunities among different parallel patterns, such as reducing the data communication overhead by fusing or reordering the patterns. The output of the offline kernel analysis is an OpenCL kernel design space consisting of Pareto-optimal designs with respect to power and latency on each platform (GPU and FPGA) as shown in Fig. 1 (c).

The kernel scheduler component makes the kernel scheduling decisions at runtime aiming to improve the system throughput and energy efficiency while meeting the latency constraint. The runtime kernel scheduler is mainly composed of three parts, which are system monitor, model, and optimizer as shown in Fig. 2. The system monitor constantly monitors the fluctuating workload received by the heterogeneous system. There often exists latency slack for the interactive applications. Poly exploits the latency slack by selecting the right kernel implementation from the design space based on the information provided by the system model. Then, the selected kernel is scheduled to the right platform according to the decisions made by the system optimizer. In general, the kernel or task scheduling problem is NP-complete [41], which is computationally intractable. To make the scheduling practical, we propose an efficient algorithm by prioritizing the ready kernels with shorter latency to approximate the optimal latency of the application. Then, based on the latency slack, we conservatively relax the latency slack by adjusting the scheduling plan to improve the energy efficiency at each time interval.

IV. Offline Kernel Analysis

In this section, we present the details of the offline kernel analysis component in Poly framework.

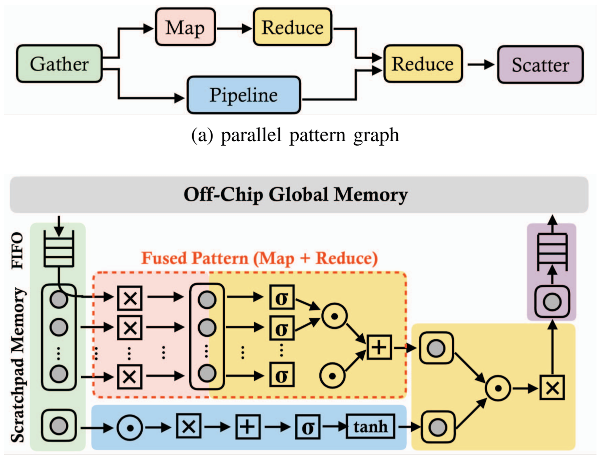
A. Parallel Pattern Analysis

Recent research on parallel programming has shown that parallel patterns are succinct and powerful abstractions for parallelizable computation by capturing the thread parallelism, synchronization, data locality, and memory access pattern for a wide range of applications [42] on both GPUs [14] and FPGAs [19, 43, 44]. We enable parallel patterns for OpenCL model and automatically characterize each pattern. Poly achieves this through parallel pattern annotation and automatic pattern analysis. In parallel pattern annotation, we define nine parallel patterns as shown in Fig. 3 based on the OpenCL programming model and use them as the unified programming and optimization interface for both FPGAs and GPUs. These patterns are selected because they are general enough to cover a wide range of cloud computing applications [45] and amenable to hardware acceleration. Then, we perform automatic pattern analysis to characterize each pattern.

Parallel Pattern Annotation. We abstract the complex parallel patterns as function-level annotations listed in the annotation method column of Table I. Programmers can compose an OpenCL program by assembling different parallel patterns. We use `inputs` to represent input data, which could be a vector or a multi-dimensional array. The `func` is used to represent the operation function applied to the input data, and this function could be as simple as addition/multiplication or a complicated customized function such as the encoding/decoding functions used in the WebP transcoding benchmarks in Table II.

`Map` replicates a function over a set of independent elements of input data, which is a natural mapping to the SIMD architecture of GPUs and the parallel compute units embedded in FPGAs. The `Reduce` pattern combines every element in a collection into a single element using an associative combiner function, such as addition or multiplication. The `Scan` pattern is fundamentally similar to `Reduce` but unlike reduce, returns all the intermediate accumulation function values. `Stencil` is a generalization of `Map` pattern in which an elemental function can access not only a single element in an input collection but also a set of neighboring elements.

Table I provides the `Stencil` annotation methods by introducing the `list` set to index the input data collection. The `Pipeline` pattern connects operators in a producer-consumer relationship, where all stages of the pipeline are active at once and each stage can sustain states that can be updated as data flows through them. The OpenCL programmers could directly add any functions to the fucntion list in the *Pipeline* annotation method. The `Gather` pattern reads a collection of data from another data collection, given a collection of indices. `Gather` can be considered as a combination of `Map` and random serial read operations. The `Scatter` is the inverse process of `Gather`. The `Tiling` pattern decomposes a data collection into a set of subcollections and is usually combined with other patterns, such as `Stencil`, `Map` and etc. The `[x,y,z]` and `[X,Y,Z]` used in the `Tiling` annotation method represent the tile



(a) parallel pattern graph

(b) local and global optimizations

Figure 4: Parallel pattern optimization for ASR.

size and tile number, respectively.

Automatic Pattern Analysis. In order to automate the pattern analysis for OpenCL kernels, we build a tool in Poly using LLVM Clang [46] as the frontend. The tool hierarchically analyzes the kernels by first identifying the parallel patterns according to the parallel pattern annotations and then construct a parallel pattern graph (PPG) based on the data dependencies among patterns. Fig 4 (a) gives an example of PPG of LSTM kernel containing six parallel patterns. Based on the PPG, the tool analyzes the data communication intensity between each neighboring pattern pair. To be more specific, we estimate the data communication overhead under different data transfer strategies including off-chip global memory and on-chip scratchpad memory based data transfer. This information is critical to the global optimization in pattern optimization (Section IV-B).

For each pattern, the tool first transforms it into a control-data-flow-graph (CDFG) [23], where each node is an operator and the edge indicates the data dependency between two nodes. For example, Fig. 4 (b) shows the CDFG for each parallel pattern of the LSTM kernel in ASR benchmark, where the gray circle represents the on-chip data buffers and the remaining of circles and squares are arithmetic operators. It is necessary to note that the operators could be as simple as multiplication, addition, and sigmoid. They could also be highly customized and optimized libraries, such as the convolution or encoding/decoding IP core.

Based on the CDFG, the tool characterizes the data-parallelism and compute-parallelism for each pattern. For the patterns including `Gather`, `Map`, `Reduce`, and `Scatter`, the tool estimates the data-parallelism based on the capacity of the data buffer, data type, and the associated access patterns. The compute-parallelism is estimated in a similar way but based on the independent operators. The obtained parallelism within each kernel is the key to local optimization during pattern optimization.

B. *Parallel Pattern Optimization*

Poly enables parallel pattern optimization in two steps: each parallel pattern is first locally optimized and then all the parallel patterns within a kernel are considered as a whole to be optimized globally.

Table I: Annotations and Optimization for parallel patterns.

Parallel Patterns : `Annotation Method`	**Optimization on Hardware Platforms**	
	GPU	**FPGA**
Map : `Map(inputs, func)`	work-group size TLP	work-group size # compute units # loop unrolling # BRAM ports
Reduce : `Reduce(inputs, func)`	serial/tree algorithm software pipeline # loop unrolling	serial/tree architecture hardware pipeline # BRAM ports
Scan : `Scan(inputs, func)`	scratchpad memory memory coalescing	# loop unrolling # BRAM ports
Stencil : `Stencil(inputs, func, list)`	scratchpad memory work-group size # loop unrolling	double buffers work-group size # compute units # loop unrolling
Pipeline : `Pipeline(inputs, func0, func1, ...)`	register reuse software pipeline pipes	hardware pipeline pipes
Gather : `Gather(inputs, list)`	scratchpad memory memory coalescing	double buffers memory burst accesses
Scatter : `Scatter(inputs, list)`	scratchpad memory memory coalescing	double buffers memory burst accesses
Tiling : `Tiling(inputs, [x,y,z], [X,Y,Z])`	work-group size	work-group size

Local Optimization. For each parallel pattern, Poly integrates a local optimization flow to enable automatic optimization on heterogeneous accelerators. To be more specific, Poly first prepares a suite of potential optimization options for each parallel pattern on both GPU and FPGA based platforms, which are listed in Table I. The optimization is enabled through either compiler directives/pragma or code restructure.

For GPUs, the optimizations mainly fall into two categories: memory access and computation optimization. In the former category, memory coalescing can be used in `Gather` and `Scatter`. The memory coalescing is realized in OpenCL by remapping the memory access indices to be physically continuous in global memory as shown in the Line 2-3 of the code snippet in Fig. 5 (a). The register reuse and scratchpad memory are used to improve the memory access efficiency for small-size but frequent memory accesses by taking advantage of the on-chip register file and scratchpad memory, respectively. For example, the code snippet (2-3) indicates using the *__local* OpenCL primitive to explicitly allocate the scratchpad memory on GPUs. In the latter category, due to the explicit data-parallelism embodied in the `Map`, `Reduce`, and `Stencil` patterns, the parallel operations inside these patterns can be naturally mapped to the SIMD architecture of GPUs by using persistent kernel code structure [47] and loop unrolling to tune the parallelism in a fine-grained manner. This is illustrated in the code snippet (Line 5-11) in Fig. 5 (a).

For FPGAs, the optimizations mainly fall into three categories: (1) pipeline, (2) parallel processing elements (PE), and (3) BARM ports optimization. On FPGAs, highly customized datapath can be implemented using the pipeline related optimization directives including hardware pipeline, pipe and, double buffer. For example, the pipeline pragma used in Fig. 5

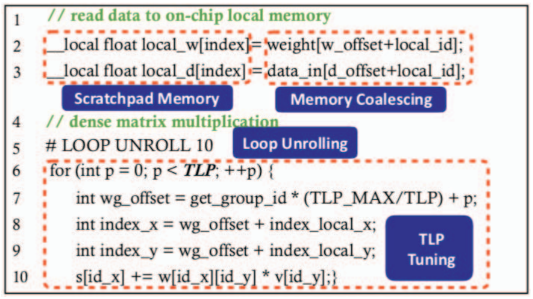
(b) (Line 6) could be used to optimize the datapath of the `Pipeline` pattern to be a fine-grained deep pipeline. The pipe and double buffer optimization directives could be used to synthesize coarse-grained pipelines by buffering data using FIFOs and double buffers, respectively. This is very useful for `Gather` and `Scatter` patterns to hide the off-chip global memory access latency by overlapping loading/storing data with other patterns in a pipeline manner. Moreover, a single PE cannot take full advantage of all the on-chip resources of an FPGA, and thus the loop unrolling and multiple compute units pragmas are used to increase the parallelism for patterns such as `Map`, `Reduce`, and `Stencil`. However, the increased PE parallelism also requires more BRAM memory bandwidth to sustain pipeline throughput, and therefore the BRAM partition method is used to increase the number of simultaneous access ports as to match the PE parallelism.

Global Optimization. In the second step of pattern optimization, Poly explores the optimization opportunities across patterns by sharing the analysis of all the patterns globally. For example, the local optimization step shows that the scratchpad memory optimization of the `Gather` pattern in Fig. 4 (b) must be delayed to global optimization step due to the missing data-parallelism of the adjacent patterns. As long as the parallelism information is provided with `Gather` pattern, we can immediately determine the size of the scratchpad memory and thus resolve this pending optimization. Similarly, the `Map`, `Reduce`, and `Pipeline` patterns of the LSTM kernel in Fig. 4 (b) could be resolved.

We also consider the fusion technique by merging the neighboring patterns. This helps to reduce the communication overhead to off-chip global memory by saving the intermediate data between patterns to on-chip memory. This is enabled in Poly by first analyzing the data transfer between each adjacent pair of patterns and then determining the number of adjacent patterns can be fused under the on-chip memory capacity constraint. As shown in Table I, if the patterns are fused, the on-chip data transfers are implemented using the scratchpad memory or pipe optimization methods on GPUs or BRAM allocation and partition techniques on FPGAs. For example, Fig. 4 (b) shows that the `Map` pattern in the local optimization design has to first write the data back to global memory and apply the `Reduce` pattern to load the data back from the global memory to continue. In this case, memory transfer overhead is saved by fusing the `Map` and `Reduce` patterns together.

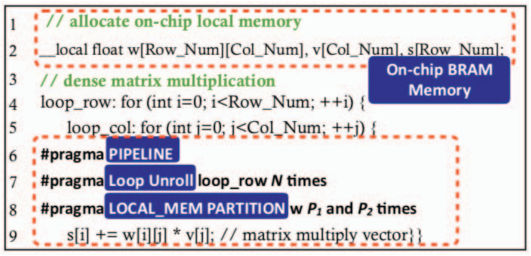
C. Design Space Exploration

Poly will first use local optimization for each pattern and then apply global optimization. By exploring different optimization parameters, we will generate a large design space for both GPU and FPGA platforms. However, we are only interested in the Pareto-optimal points with trade-offs between latency, throughput, and power. It will take extremely long if we explore the design space exhaustively especially the FPGA implementation involves lengthy placement and routing cycle [44, 48]. In order to find these Pareto-optimal points from the design space efficiently, Poly employs the state-of-the-art analytical models to navigate the exploration



```
1  // read data to on-chip local memory
2  __local float local_w[index]= weight[w_offset+local_id];
3  __local float local_d[index] = data_in[d_offset+local_id];
4  // dense matrix multiplication
5  # LOOP UNROLL 10
6  for (int p = 0; p < TLP; ++p) {
7      int wg_offset = get_group_id * (TLP_MAX/TLP) + p;
8      int index_x = wg_offset + index_local_x;
9      int index_y = wg_offset + index_local_y;
10     s[id_x] += w[id_x][id_y] * v[id_y];}
```

(a) GPU



```
1  // allocate on-chip local memory
2  __local float w[Row_Num][Col_Num], v[Col_Num], s[Row_Num];
3  // dense matrix multiplication
4  loop_row: for (int i=0; i<Row_Num; ++i) {
5      loop_col: for (int j=0; j<Col_Num; ++j) {
6  #pragma PIPELINE
7  #pragma Loop Unroll loop_row N times
8  #pragma LOCAL_MEM PARTITION w P1 and P2 times
9      s[i] += w[i][j] * v[j]; // matrix multiply vector}}
```

(b) FPGA

Figure 5: OpenCL code snippet optimized by Poly. (TLP refers to thread-level parallelism.)

process. Specifically, Poly employs the performance and power models proposed in [18, 49] for GPUs. As for FPGAs, the performance and resource models proposed in [26, 48, 50] are used. The power consumption is roughly proportional to the resource utilization in FPGA which makes it accurate enough to characterize the power scaling trend and guide the design space exploration of FPGAs [51]. Overall, the analytical model based exploration reduces the exploration time from tens of hours to seconds.

V. Runtime Kernel Scheduler

In this section, we present the details of the runtime kernel scheduler component in Poly framework. Before making runtime scheduling decisions, Poly builds a directed acyclic kernel graph $G = (K, E)$ by analyzing the input OpenCL code. In G, each node ($k_i \in K$) and edge ($e_{ij} \in E$) correspond to a kernel and the corresponding data dependency between kernel i and j, respectively. Then, the runtime scheduling is determined in two steps as shown in Fig. 6. The first step is latency optimization, where the scheduler tries to minimize the total execution latency by selecting the kernel implementation with shorter latency on the corresponding accelerator. The second step is energy efficiency optimization, where the scheduler improves the energy efficiency by exploiting the latency slack. The details of these two steps are explained in the following.

Step 1: Latency Optimization. For interactive applications, exceeding the latency requirement of interactive applications is often considered a failure, which may lead to a system malfunction. In order to preclude these cases as many as possible, we propose to optimize the latency first during the runtime scheduling. However, given the kernel graph G,

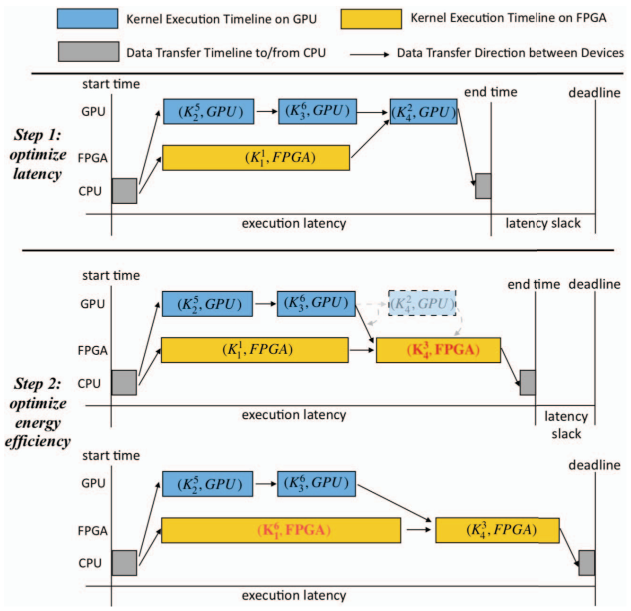


Figure 6: Scheduling result of ASR benchmark on the heterogeneous system. (K_i^r, Device) represents that the r^{th} implementation of i^{th} kernel is scheduled to the associated device (GPU/FPGA).

it is computational intractable to compute the optimal kernel scheduling with the minimum latency [41]. We propose a lightweight scheduling algorithm based on latency priority list. The key idea behind this algorithm is to construct a latency priority list W_L for kernels and then schedule based on their priorities. The latency priority list W_L is built in a way similar to the HEFT [41] and MKMD [52] algorithms used for real-time scheduling. To be more specific, Poly firstly traverses the kernel graph G in a bottom-up manner and computes the latency priority value $W_L(k_i)$ of kernel k_i using the following equation,

$$\begin{aligned} W_L(k_i) \quad &= \max_{k_j \in Succ(k_i)} (T(e_{ij}) + W_L(k_j)), \\ &+T_{min}(k_i) \qquad k_i \neq k_{sink} \end{aligned} \tag{2}$$

where $T(e_{ij})$ is the data transfer time from kernel k_i to k_j through PCIe, and $T_{min}(k_i)$ is defined as the minimum latency of kernel k_i across all the and is calculated as follows,

$$T_{min}(k_i) = \min_{r,n} T(k_i^r, d_n) \tag{3}$$

where $T(k_i^r, d_n)$ is the execution latency of r^{th} implementation of kernel k_i on the n^{th} device. The data transfer time $T(e_{ij})$ depends on the amount of data transferred between kernels k_i to kernel k_j and available PCIe bandwidth.

$W_L(k_i)$ accumulates the maximum priority from its successors. It indicates the longest path from the kernel k_i to the sink kernel. Poly picks the kernel one by one in the descending order of the priority. At each time slot, when a kernel is selected, Poly needs to check the availability of accelerator before determining where the kernel is assigned to. Since a kernel cannot be scheduled if the target device is occupied or its input data are not ready, we derive the earliest starting time (EST) for each pair of device and kernel before making scheduling decisions. We use $EST(k_i, d_n)$ to represent the earliest starting time of kernel k_i on device d_n and it is calculated as

$$EST(k_i, d_n) = \max_{k_j \in Pred(k_i)} T_{end}(k_j) + T_{queue}(d_n) \tag{4}$$

where $T_{queue}(d_n)$ is the network queueing time of device d_n and $T_{end}(k_j)$ is the finishing time of kernel k_j. The EST table is calculated in a top-down manner. Poly will schedule the selected the kernel k_i onto the n^{th} device only if $EST(k_i, d_n)$ is earlier than the launching time of kernel k_i. When a new kernel is dispatched to the n^{th} device, then earliest starting time (EST) of the rest kernels will be updated for this device. This process is repeated until all the kernels are scheduled.

The upper part of Fig. 6 shows the scheduling result of optimization Step 1 for the ASR benchmark. ASR has four kernels from K_1 to K_4. K_1 and K_2 that are generated from the `map` patterns can be executed in parallel. Since K_2 has the highest priority, it (implementation K_2^5) is scheduled to GPU first. In the following, kernel K_1 (implementation K_1^1) is assigned to FPGA because GPU is occupied by K_2^5. Kernel K_3 is derived from `reduce` pattern and dependent on kernel K_2. After K_2^5 is finished, kernel K_3 could be scheduled. It is necessary to note that although we find that K_3 incurs the minimum latency on FPGA, it is eventually scheduled to GPU because the kernel K_1^1 executed on the FPGA is not finished according to the EST table. In the last, kernel implementation K_4^2 is scheduled to GPU. In such a schedule, the latency is highly optimized, leading to a large slack which can be further exploited for energy efficiency optimization in the next step.

Step 2: Energy Efficiency Optimization. For each kernel, its design space exhibits a Pareto-optimal power-latency scaling trend as shown in Fig. 1(c). The latency slack is $LB - L$, where the L is the total execution latency of kernel graph G from the previous scheduling step and LB is the latency bound of the application. Similar to latency priority list W_L, Poly builds an energy priority list W_E to guide the energy efficiency optimization. $W_E(k_i)$ is calculated as

$$W_E(k_i) = \max_r (P(k_{i_0}^{r_0}) - P(k_i^r))(T(k_{i_0}^{r_0}) - T(k_i^r)) \tag{5}$$

where $P(k_i^r)$ is the average power consumption of the r^{th} implementation of i^{th} kernel. We use kernel $k_{i_0}^{r_0}$ to represent the initial implementation determined by the latency optimization step, and thus $W_E(k_i)$ indicates the maximum energy reduction we could achieve by using the new kernel implementation k_i^r.

$W_E(k_i)$ implies the potential of kernel k_i to trade latency for power reduction on device d_n. Next, Poly selects the kernel one by one in the descending order with respect to $W_E(k_i)$. When a kernel is selected, Poly updates the kernel implementation using the one with the highest energy efficiency improvement. It is necessary to note that the latency constraint has to be met if a new implementation is chosen.

Table II: Summary of QoS-sensitive benchmarks.

Benchmarks	Kernels Name	Parallel Patterns	# Designs	
			GPU	FPGA
Automatic Speech Recognition (ASR) [39]	LSTM	Map, Reduce, Pipeline, Tiling	164	256
	Fully Connected	Map, Pipeline, Pack	148	192
Finance Quantitative Trading (FQT)	PRNG	Map, Pipeline	64	128
	Black Scholes	Map, Pipeline	64	128
	Reduce	Reduce, Pack	16	64
Image Recognition (IR) [53]	Convolution	Gather, Map, Pipeline, Stencil, Tiling, Scatter	192	256
	Pooling	Map, Stencil, Tiling	128	256
	Fully Connected	Map, Pipeline, Pack,Tiling	92	128
Cloud Storage (CS) [54]	RS Encoder	Gather, Map, Pipeline, Scatter, Tiling	108	128
	RS Decoder	Gather, Map, Pipeline, Scatter, Tiling	108	128
Online Matrix Factorization (MF) [17]	Read Data	Gather, Pack, Tiling	16	16
	RS Decoder	Gather, Map, Pipeline, Scatter, Tiling	108	128
WebP Transcoding (WT) [55]	Intra-prediction	Gather, Map, Pipeline, Tiling	128	256
	Probability Counting	Map, Pipeline, Reduce, Pack	64	128
	Arithmetic Coding	Scatter, Map, Pipeline, Stencil	92	128

Poly iteratively updates the kernels' implementations until the latency slack cannot be further reduced.

The lower part of Fig. 6 illustrates how Poly iteratively optimizes the energy efficiency. Firstly, Poly finds that reallocation of kernel K_4 from GPU to FPGA using implementation K_4^3 only increases the latency by 12% but reduces its power consumption by 45%. This gives the highest energy efficiency improvement without violating the latency constraint. Next, Poly updates the latency slack based on the latest scheduling plan and further exploits the remaining latency slack by replacing the 1^{st} implementation of kernel K_1 with the 6^{th} on FPGA which further improves the energy efficiency by 6% and fully utilizes the remaining latency slack. This also terminates the energy efficiency optimization step.

VI. Experimental Results

A. Experiment Setup

We assemble three different heterogeneous systems presented in Table III using different GPU and FPGA accelerators in Table IV and Table V. The heterogeneous server is prototyped based on an Intel i7-4790k CPU, 16GB DRAM and the associated GPU and FPGA accelerators are attached to the server through the PCI-e interface. The OpenCL-oriented high-level synthesis (HLS) toolchains used in this work are Xilinx SDAccel 2017.1 and Intel OpenCL SDK. The performance and power reported in this section are all measured on real hardware. The power of FPGA is measured through the power monitor interface of TI fusion toolkit and the power of GPU is measured through the profiling tool CodeXL.

We use six typical QoS-sensitive cloud computing applications to evaluate the proposed Poly framework as shown in Table II. For each benchmark, it consists of multiple kernels and each kernel involves multiple parallel patterns. On both GPU and FPGA platforms, we form a design space for the application through a unified parallel pattern optimization interface. The design space ranges from 16 to 256 as shown

Table III: Configurations of three heterogeneous system settings (Power split is 50%-50% for Heter-Poly).

Setting Number	System Codename	Accelerator Type (x #Accelerators)	
		GPU	FPGA
Setting-I	Homo-GPU	AMD W9100 (x2)	-
	Homo-FPGA	-	Xilinx 7V3 (x10)
	Heter-Poly	AMD W9100 (x1)	Xilinx 7V3 (x5)
Setting-II	Homo-GPU	NVIDIA K20 (x2)	-
	Homo-FPGA	-	Xilinx ZCU102 (x16)
	Heter-Poly	NVIDIA K20 (x1)	Xilinx ZCU102 (x8)
Setting-III	Homo-GPU	NVIDIA K20 (x2)	-
	Homo-FPGA	-	Intel Arria 10 (x8)
	Heter-Poly	NVIDIA K20 (x1)	Intel Arria 10 (x4)

Table IV: GPU Platform Specifications.

	Cores	Peak Frequency	Memory	Peak Power	Manufacturing Process	Price ($)
AMD FirePro W9100	2,816	930MHz	32GB	270W	TSMC 28nm	4,999
NVIDIA Tesla K20	2,496	706MHz	5GB	225W	TSMC 28nm	2,999

Table V: FPGA Platform Specifications.

	Xilinx Zynq UltraScale+ ZCU102	Xilinx Virtex7-690t ADM-PCIE-7V3	Intel Arria 10 GX115
Peak Frequency	333MHz	470MHz	800MHz
Peak Power	30W	45W	65W
Logic Cells	600K	693K	43K
BRAMs	4.0MB	6.5MB	8.2MB
DSP Slices	2,520	3,600	1,518
Manufacturing Process	TSMC 16nm	TSMC 28nm	TSMC 20nm
Price ($)	2,495	3,200	4,495

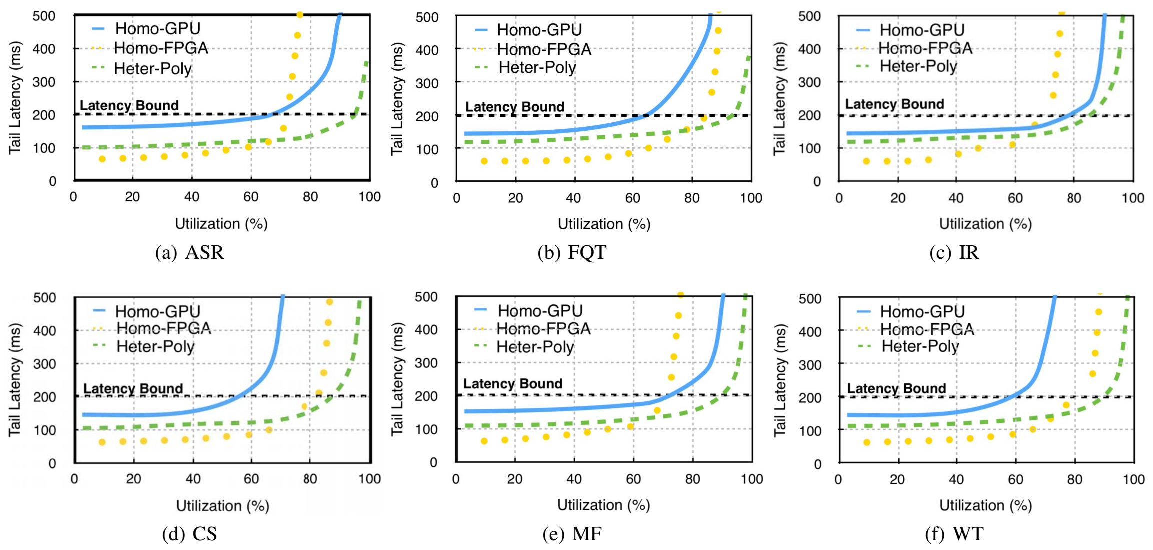
in Table II. The target latency constraint is set to be 200ms. The baseline used throughout this section are *Homo-GPU* and *Homo-FPGA* solution [4] whose hardware platform settings are shown in Table III, IV, and V. The allocation scheme of *Homo-GPU* or *Homo-FPGA* is fixed across different load intensities by using only one implementation with the maximum energy efficiency or minimum latency depending on the latency constraint.

We perform three sets of experiments to evaluate Poly. First, we present the static analysis including latency, throughput, and energy proportionality improvements of Poly by fixing the load intensity at different levels. Secondly, we use trace-driven experiments to further evaluate Poly and discuss the power saving and latency violations. Lastly, we evaluate Poly using three different heterogeneous system settings to demonstrate its scalability to new architectures shown in Table III.

B. Static Load Evaluation

We evaluate Poly by using the different levels of load ranging from 10% to 100% with an interval of 10%. The setting-I shown in Table III is used as the leaf node prototype configuration of a cloud computing system.

Tail Latency. According to the tail latency results shown in Fig. 7, we find that all these three systems could satisfy the tail latency constraints when the request throughput is low. However, if the request throughput continues to grow, the tail latencies are increased drastically and the latency constraints are violated. We also find that although the tail latency of *Heter-Poly* is not always the lowest, its tail latency scaling trend is rather flat compared with the other two systems.



(a) ASR (b) FQT (c) IR

(d) CS (e) MF (f) WT

Figure 7: Tail latency comparison.

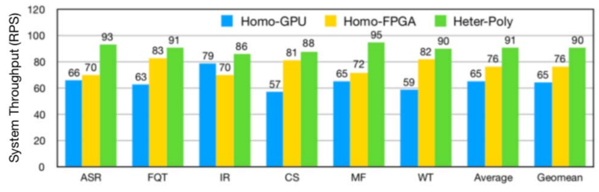


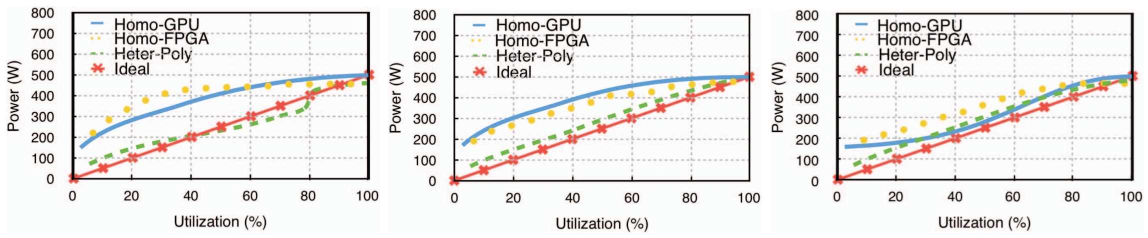
Figure 8: System throughput comparison.

The reason behind the better latency-throughput scaling of *Heter-Poly* is that Poly explores a larger design space of the kernel allocation scheme among the heterogeneous devices. Moreover, during the runtime, the allocation scheme of each kernel is not fixed but determined by the Poly scheduler based on the latency constraint and system states such as the number of available accelerators, request throughput and etc. For example, the `IR` is widely used in datacenter to process uploaded images and thus highly latency-sensitive. Fig. 7 (c) shows that when the load is below 60%, the tail latency of *Homo-FPGA* is much less than *Homo-GPU*. The reason is that the highly customized pipeline is designed for `IR` and there is no need for FPGA device to batch a few images to improve the resource utilization as GPU does. If the load is set to be even higher, *Homo-FPGA* cannot sustain the associated load and thus the tail latency of *Homo-FPGA* increases dramatically due to the long queueing time. However, *Homo-GPU* could guarantee the tail latency under higher load intensity. In order to integrate the benefits from both GPUs and FPGAs, *Heter-Poly* first statically estimates the kernel performance under different load levels and then dynamically allocates the most of latency-critical requests to FPGAs when the load is light or shift the workload to GPU when the load is much heavier.

Throughput. The maximum system throughput of a server indicates the maximum RPS the system could sustain without violating the QoS (tail latency) constraint. The higher the maximum system throughput is, the less the system is under-utilized. According to the summary of the maximum system throughput results shown in Fig. 8, the proposed *Heter-Poly* achieves 40% and 20% maximum system throughput improvement over *Homo-GPU* and *Homo-FPGA*, respectively. As we can see from the figure, the proposed *Heter-Poly* consistently performs better than both the *Homo-GPU* and *Homo-FPGA* designs in all six applications. Moreover, both the average and geometric mean of the maximum system throughput are more than 90% and much higher than the other two designs, which would obviously improve the system utilization and improve energy proportionality of cloud computing systems. We also observe that *Homo-GPU* and *Homo-FPGA* systems perform quite differently across the six applications. For example, in the `FQT` application, *Homo-FPGA* achieves 83% while *Homo-GPU* reaches only 64% maximum system throughput resulting from the fact that the maximum system throughput is constrained by the pseudo-number generator kernel of `FQT` which requires large batch size to enable high throughput. However, this kernel is naturally amenable to be implemented as a customized pipeline on FPGAs with both relatively high throughput and low latency. Therefore, in the *Heter-Poly* system, the GPU-amenable kernels (`Black Scholes` and `Reduce`) of `FQT` are allocated to GPUs while the PRNG kernel is assigned to FPGA during runtime.

Energy Proportionality. Fig. 9 shows the power scaling trends with respect to load intensity of three benchmarks[1]. Overall, the proposed *Heter-Poly* design improves the energy proportionality by 23% and 17% on average over *Homo-GPU* and *Homo-FPGA*, respectively. We can also find that the proposed *Heter-Poly* system is much closer to the ideal power scaling trend than the other two systems.

[1]Due to space limitation, we only show three typical benchmarks in this paper and the other three benchmarks exhibits similar scaling trends.



(a) ASR (b) FQT (c) IR

Figure 9: Power scaling trends comparison.

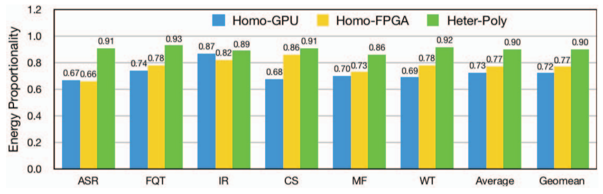


Figure 10: Energy proportionality comparison.

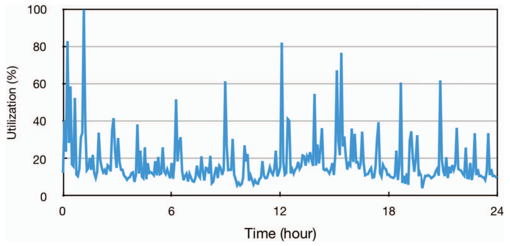


Figure 11: Server utilization trace of Google datacenter.

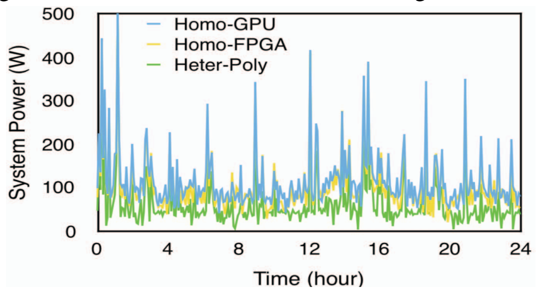


Figure 12: Power savings comparison for three designs.

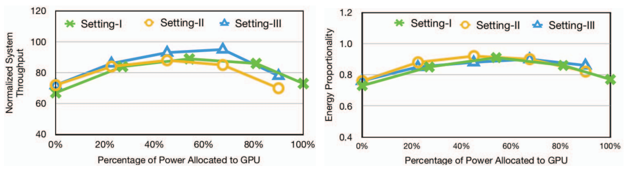
C. Datacenter Trace-based Evaluation

We use the workload trace (24-hour) obtained from trace data set of Google cloud computing cluster with 12.5 thousand servers over about a month-long period in May 2011 [56]. Since the trace records the CPU utilization of each server, we directly use the same utilization value for *Homo-GPU*, *Homo-FPGA*, and *Heter-Poly* platforms. The *Homo-GPU* and *Homo-FPGA* is used as the baseline platforms both configured with static scheduling scheme. For each application, the target tail latency is set to be the 200ms. The configuration of the hardware platform we use for evaluation is the Setting-I shown in Table III.

Power Savings. A 24-hour server utilization trace is presented in Fig. 12. We can see from the figure that *Homo-GPU* generally consumes the highest power for almost every time interval especially when there is a burst increase in throughput requirement. The reasons are two folds. On one hand, the GPU accelerator incurs higher idle power compared with FPGAs. On the other hand, the operating frequencies of GPU cores and global memory are boosted when the request throughput is high. As for the *Homo-FPGA* system, we observe that the power consumption values are generally less than *Homo-GPU* but higher than *Heter-Poly*. The saved energy and higher energy efficiency achieved by *Heter-Poly* could be attributed to three aspects. First, Poly not only keeps the high-throughput feature of GPUs to high-intensity loads but also uses FPGAs to largely reduce power consumption by enabling very few on-chip logics when the load is very low. Second, Poly is configured with a flexible runtime scheduler to dynamically select the best kernel candidates and then allocate it to the associated the heterogeneous accelerators. For example, when the load intensity is very high, it will boost operating frequency of GPUs and FPGAs to increase performance. While the load is very low, the runtime scheduler of Poly will try to reduce the idle power as much as possible by reducing the GPU operating frequency and reconfiguring FPGA with a low-power kernel.

QoS Violations. In addition to power savings, the *Heter-Poly* also achieves lower latency violation ratios than the baseline counterparts, and the 99^{th} percentile tail latency is consistently maintained to be less than 200ms. This could be attributed to two aspects of reasons. On one hand, the high-intensity loads immediately translate to longer queues, and *Heter-Poly* reacts to changes in queue length immediately. Note that the target tail latency depends on the service-time distributions, not on queue lengths. Thus, a sudden change in load makes *Heter-Poly* immediately shift to higher performance mode by the runtime scheduler. By contrast, the *Homo-GPU* and *Homo-FPGA* respond very slowly as shown in Fig. 12.

Poly employs analytical performance models [18, 26, 48, 49] to navigate the design space as discussed in Section 4.3. For all the benchmarks, the absolute performance prediction error of the performance model is within 6% on both FPGA and GPU platforms. The performance estimation error is mainly caused by the dynamic workload which may change drastically during a 24-hour duration. Poly tolerates the wrong prediction by making self-correction through the feedback loop as shown in Figure 2. More clearly, the system monitor periodically updates the kernel and accelerator related information and the system model



(a) system throughput (b) energy proportionality

Figure 13: Architecture scalability of Heter-Poly.

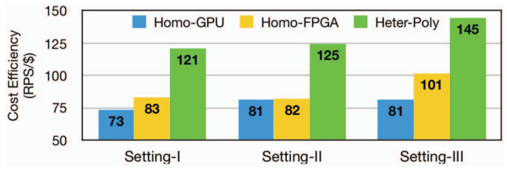


Figure 14: Cost efficiency analysis under three settings.

explores and evaluates different scheduling decisions. If the scheduling decision is not optimal due to wrong prediction or dynamic workload, the optimizer will make an adjustment using the latest feedback (latency, power, throughput) from system monitor accordingly in the next iteration.

D. *Architecture Scalability*

We evaluate the architecture scalability by varying the number of GPUs and FPGAs configured in the heterogeneous system. This is done by using different power split ratios between GPUs and FPGAs from 0% to 100% at the granularity of 20% under the 1000W total power cap shown in Fig. 13. The *Homo-GPU* setting is the rightmost point (100%) and *Homo-FPGA* is the leftmost point (0%). For example, when the power split between GPUs and FPGAs is 80%-20%, the Setting-I contains three GPUs and four FPGAs. As we can see from Fig. 13, the scaling trends are similar for different system settings and the *Heter-Poly* are consistently better than the *Homo-GPU* and *Homo-FPGA*.

E. *Cost Efficiency*

Cost efficiency is a very important metric for large datacenters which is defined by the value of the maximum throughput divided by TCO. We perform our cost efficiency analysis using the TCO model recently proposed by Google [57]. The parameters used in our TCO model are the same as [4]. We use three different hardware settings listed in Table III to validate the cost efficiency results shown in Fig 14. We can find that the Poly is consistently much better than the homogeneous baseline designs with respect to cost efficiency. The better cost efficiency of Poly comes from its better energy efficiency which leads to lower operational cost. Since the operational cost is the dominant cost of a long-term operating datacenter [2], the higher infrastructure cost of the Poly can be amortized over a relatively long time horizon.

VII. Related Work

Latency, throughput and energy efficiency are three major factors that are considered when managing modern warehouse-scale datacenters [2, 38, 58, 59]. Due to the huge amount of computation parallelism and bandwidth of GPUs, they are widely deployed in modern cloud computing systems. The key problem of GPU accelerated workloads in cloud computing system is to balance the throughput speedup and latency degradation. DjiNN [60] proposes to accelerate the DNN based workloads on warehouse scale datacenter by properly determine the batch size and co-locate multiple DNN services on a single GPU, which achieves promising throughput and energy efficiency speedup subject to certain latency constraints. Baymax [61] finds that the GPU accelerators deployed in cloud computing systems are usually underutilized. In order to improve the utilization of GPUs, Baymax [61] proposes a dynamic scheduling algorithm to co-locate the long-term latency-insensitive batch workloads with short-term latency-sensitive services. Dysel [16] proposes a low-overhead runtime profiling method to dynamically choose the best kernel for GPUs as to achieve satisfying trade-offs between throughput and latency.

Thanks to the customized architecture and reconfigurability, FPGAs have been more and more popular in cloud computing systems to improve the energy efficiency and reduce response latency. LINQits [21] provides a low-power solution to offload the light loaded workloads to FPGA based SoC platforms. Since the customized accelerator implemented on FPGAs, it achieves promising energy land latency reductions. The search engines deployed in cloud computing systems are highly latency-sensitive and power-hungry, and thus Catapult [8] proposes an FPGA-based cluster for Bing search to resolve the problem. Sirius [4] evaluates the DNN based workloads on different platforms and concludes that FPGAs are much more energy and latency friendly compared with GPU counterparts.

VIII. Conclusion

QoS-sensitive workloads, common in warehouse-scale datacenters, require a guaranteed stable tail latency of a few milliseconds. Since the system load dynamically fluctuates during daily datacenter operations, the servers outfitted with accelerators (e.g. GPUs and FPGAs) are overprovisioned to meet these stringent latency targets. In order to improve the overall throughput scalability while guaranteeing the QoS for the current datacenter, we propose to take advantage of the heterogeneous architecture of leaf node within a datacenter, e.g. a server tightly coupled with both GPUs and FPGAs. In this work, we present *Poly*, a framework that enables fine-grained design space exploration offline and efficient kernel scheduling at runtime for heterogeneous cloud computing system. Experimental results show that the proposed Poly framework enables a fine-grained trade-offs among latency, throughput, and power and improves the energy proportionality.

Acknowledgment

This work is supported by National Natural Science Foundation China (No. 61672048), Beijing Natural Science Foundation (No. L172004) and Municipal Science and Technology Program Grant (Z181100008918015). We thank all the anonymous reviewers for their feedback.

References

[1] J. Dean and L. A. Barroso, "The Tail at Scale," *Communication of ACM*, vol. 56, pp. 74–80, Feb. 2013.

[2] L. A. Barroso and U. Hlzle, "The Case for Energy-Proportional Computing," *IEEE TC*, vol. 40, pp. 33–37, Dec 2007.

[3] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *ASPLOS'14*.

[4] J. Hauswald and et al., "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers," in *ASPLOS'15*.

[5] H. Kasture and et al., "Rubik: Fast Analytical Power Management for Latency-critical Systems," in *MICRO'15*.

[6] NVIDIA cloud gaming. http://www.nvidia.com/object/cloud-gaming.

[7] Kinetica. http://https://www.kinetica.com/solutions/finance/.

[8] A. Putnam, , and et al., "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *ISCA'14*.

[9] IBM SuperVessel. https://openpowerfoundation.org/.

[10] A. M. Caulfield and et al., "A Cloud-scale Acceleration Architecture," in *MICRO'16*.

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

[12] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "SDA: Software-Defined Accelerator for Large-Scale DNN Systems," in *HotChips'14*.

[13] Nimbix HPC. https://www.nimbix.net/.

[14] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware Mapping of Nested Parallel Patterns on GPUs," in *MICRO'14*.

[15] X. Li, Y. Liang, W. Zhang, T. Liu, H. Li, G. Luo, and M. Jiang, "cuMBIR: An Efficient Framework for Low-dose X-ray CT Image Reconstruction on GPUs," in *ICS'18*.

[16] L.-W. Chang, H.-S. Kim, and W.-m. W. Hwu, "Dysel: Lightweight Dynamic Selection for Kernel-based Data-parallel Programming Model," in *ASPLOS'16*.

[17] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs," in *HPDC'17*.

[18] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing Compute and Memory Power in High-Performance GPUs," in *ISCA'15*.

[19] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware System Synthesis from Domain-specific Languages," in *FPL'14*.

[20] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs," in *DAC'17*.

[21] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big Data on Little Clients," in *ISCA'13*.

[22] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, "C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs," in *FPGA'18*.

[23] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE TCAD*, vol. 30, no. 4, pp. 473–491, 2011.

[24] L. Lu and Y. Liang, "SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs," in *DAC'18*.

[25] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *FCCM'17*.

[26] Z. Wang, B. He, W. Zhang, and S. Jiang, "A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs," in *HPCA'16*.

[27] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "TGPA: Tile-grained Pipeline Architecture for Low Latency CNN Inference," in *ICCAD'18*.

[28] Y. Liang et al., "High-level Synthesis: Productivity, Performance, and Software Constraints," *JECE'12*.

[29] D. Wong and M. Annavaram, "Knightshift: Scaling the Energy Proportionality Wall Through Server-level Heterogeneity," in *MICRO'12*.

[30] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven Task Management for Heterogeneous Multicores in Warehouse-scale Computers," in *HPCA'15*.

[31] B. Vamanan, H. B. Sohail, J. Hasan, and T. Vijaykumar, "Timetrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search," in *MICRO'15*.

[32] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in A Scala Embedded Language," in *DAC'12*.

[33] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A Compiler and Runtime for Heterogeneous Systems," in *SOSP'13*.

[34] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IISWC'10*.

[35] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *HPCA'15*.

[36] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs," in *MICRO'15*.

[37] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Trends in Server Energy Proportionality," *IEEE TC*, vol. 44, no. 9, 2011.

[38] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference," in *ISCA'16*.

[39] H. Sak, A. Senior, and F. Beaufays, "Long Short-term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling," in *InterSpeech'14*.

[40] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *FPGA'17*.

[41] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing," *IEEE TPDS*, vol. 13, no. 3, pp. 260–274, 2002.

[42] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A Reconfigurable Architecture For Parallel Paterns," in *ISCA'17*.

[43] S. Wang and Y. Liang, "A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model," in *DAC'17*.

[44] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware," in *ISCA'16*.

[45] M. D. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[46] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO'04*.

[47] K. Gupta, J. A. Stuart, and J. D. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," in *IEEE InPar'12*.

[48] S. Wang, Y. Liang, and W. Zhang, "FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs," in *DAC'17*.

[49] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 280–289, ACM, 2010.

[50] Y. Liang, S. Wang, and W. Zhang, "FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs," *IEEE TC*, 2018.

[51] F. Li, Y. Lin, L. He, D. Chen, and J. Cong, "Power Modeling and Characteristics of Field Programmable Gate Arrays," *IEEE TCAD*, 2005.

[52] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating Multiple Data-parallel Kernels on Multiple Devices," in *PACT'15*.

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *NIPS'12*.

[54] G. Chen, H. Zhou, X. Shen, J. Gahm, N. Venkat, S. Booth, and J. Marshall, "OpenCL-based Erasure Coding on Heterogeneous Architectures," in *ASAP'16*.

[55] Google. https://developers.google.com/speed/webp/.

[56] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google Cluster-usage Traces: Format + Schema," technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[57] D. A. Patterson, "The Data Center is the Computer," *Communications of the ACM*, vol. 51, no. 1, pp. 105–105, 2008.

[58] D. Wong, "Peak Efficiency Aware Scheduling for Highly Energy Proportional Servers," in *ISCA'16*.

[59] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *ASPLOS'09*.

[60] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers," in *ISCA'15*.

[61] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," in *ASPLOS'16*.