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

Kelp: QoS for Accelerated Machine Learning Systems

Haishan Zhu[1]*‡, David Lo†, Liqun Cheng†, Rama Govindaraju†, Parthasarathy Ranganathan† and Mattan Erez‡
*Microsoft †Google ‡The University of Texas at Austin
haishanz@utexas.edu, {davidlo, liquncheng, govindaraju, parthas}@google.com, mattan.erez@utexas.edu

***Abstract*—Development and deployment of machine learning (ML) accelerators in Warehouse Scale Computers (WSCs) demand significant capital investments and engineering efforts. However, even though heavy computation can be offloaded to the accelerators, applications often depend on the host system for various supporting tasks. As a result, contention on host resources, such as memory bandwidth, can significantly discount the performance and efficiency gains of accelerators. The impact of performance interference is further amplified in distributed learning, which has become increasingly common as model sizes continue to grow.**

In this work, we study the performance of four production machine learning workloads on three accelerator platforms. Our experiments show that these workloads are highly sensitive to host memory bandwidth contention, which can cause 40% average performance degradation when left unmanaged. To tackle this problem, we design and implement Kelp, a software runtime that isolates high priority accelerated ML tasks from memory resource interference. We evaluate Kelp with both production and artificial aggressor workloads, and compare its effectiveness with previously proposed solutions. Our evaluation shows that Kelp is effective in mitigating performance degradation of the accelerated tasks, and improves performance by 24% on average. Compared to previous work, Kelp reduces performance degradation of ML tasks by 7% and improves system efficiency by 17%. Our results further expose opportunities in future architecture designs.

I. Introduction

Accelerators have started to drive much of the improvement in performance and cost-efficiency for mission-critical workloads in Warehouse Scale Computers (WSCs) [1], [2], [3], [4]. The increase in computational capacity enables various computation intensive applications at WSC scale, a notable example of which is machine learning (ML).

Previous work has observed that the size of the model has a strong impact on the effectiveness of many ML algorithms (e.g. Neural Networks or NN). Specifically, Canziani et al. survey various Deep Neural Network (DNN) models and show significant increase in models sizes and computation requirements as DNN architectures advance [5]. For example, Inception V-4 [6] requires more than 4x memory and computation beyond GoogLeNet [7] to improve the top-1 accuracy by about 15%. Future ML algorithms that target more challenging tasks are expected to demand even more computational resources. ML workloads are therefore often targets for acceleration.

Today, accelerators like GPUs and TPUs are widely used for ML training and inference [1], [8], [9], [10], [11], [12], [13], [14], while many more new accelerator architectures have been proposed (e.g., [15], [16], [17], [18], [19], [20], [21], [22]). With maturing accelerator architectures, service providers have started to deploy accelerators into production. Google's Cloud TPU, for example, delivers up to 180 TFLOPS and provides both

[1]This work was done when Haishan Zhu was an intern at Google.

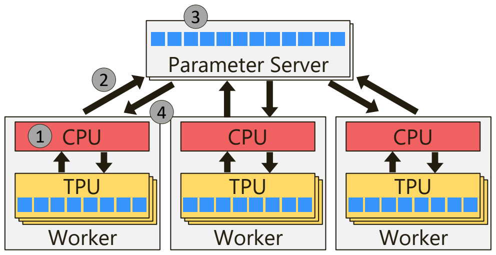


Figure 1: Example workflow of distributed TensorFlow training with parameter servers.

training and inference capabilities. Cloud TPU can also scale-out to a 64-device TPU-pod in order to accommodate larger and more complex models [2]. Microsoft has used FPGAs to accelerate Bing's search ranking efficiency [3] and distributed DNN applications [4]. Intel also released a PCI-e based FPGA card that targets WSC applications [23].

While accelerators carry out the heavy computation, the host system is often responsible for various supporting tasks to sustain high ML accelerator performance. One example of the CPU's supportive role is the *parameter server* in large-scale distributed machine learning. In this configuration, a cluster of worker tasks executes the TensorFlow graph using different training data, while the shared parameters are hosted by multiple parameter server instances spanning across nodes [24], [25]. Figure 1 shows an example workflow. Each worker first computes the gradients of the variables by gathering results from all local accelerators ① and sends them to the parameter servers ②. Each parameter server then aggregates the gradients and computes the updated training variables using a pre-defined optimizer ③ [26]. Finally, updated variables are copied to each worker at the end of the iteration ④. The parameter server tasks are memory intensive and can easily become the performance bottleneck of the entire service, as we show later in the paper. Furthermore, due to the distributed nature of the computation model, performance degradation on any parameter server instance is amplified at the service-level [27].

The supporting role of CPUs in accelerator platforms brings new challenges in system design and resource management. Specifically, host memory bandwidth (BW) interference can cause significant performance and Total Cost of Ownership (TCO) loss for the entire accelerated application. To understand the utilization of the memory BW resources in production environment, we plot the 99%-ile memory BW usage of a particular generation of servers over a day in Figure 2. Results show that 16% of the profiled servers experience peak BW higher than 70% of available BW, indicating wide presence of memory BW saturation.

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00036

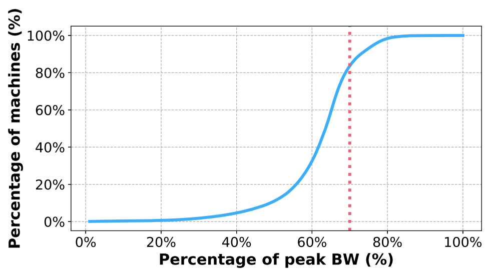


Figure 2: 99%-ile memory BW in production environment.

To demonstrate the impact of BW saturation, we show the execution timeline of a production RNN inference server running on the TPU platform [1]. Each query to the server in this workload is broken down into multiple iterations and Figure 3 shows one such iteration. We further break down the execution time into different phases, which include CPU-assist tasks, CPU-TPU communication, and TPU computation. We then show the execution timeline with and without a DRAM aggressor. Note that for this illustrative example, we generate requests serially to simplify the presentation of the trace.

The results show that, while the CPU-accelerator interaction (blue blocks in the figure) is not sensitive to the DRAM BW aggressor, the CPU-intensive phases are highly sensitive to memory BW interference. Execution time for CPU-intensive phases increases by up to 51%, while service-level tail latency increases by over 70%. In this work, we first use synthetic workloads to confirm memory BW interference to be the dominant factor that causes performance degradation. We further observe similar performance degradation with benchmarks across platforms. The resulting performance degradation can discount the efficiency gain of accelerators and cause significant loss of the capital investments in accelerator development and deployment.

Such contention also highlights the needs for robust and efficient performance isolation mechanisms. As shown in Figure 3, the interleaving among different steps in the execution timeline is on the order of sub-milliseconds to millisecond, which is too fine-grained for effective polling-based reactive core throttling, such as the approaches proposed in previous work [28], [29], [30]. Request pipelining (as used in production environments and evaluation in Section V) further exacerbates this issue because the timeline becomes more complicated due to phase interleaving and overlapping. Because of the above reasons, hardware-based solutions, such as fine-grained memory BW QoS (e.g., request-level prioritization), are much better suited to provide the much needed BW interference protection in future system architectures.

In this work, we use existing hardware features to demonstrate the capabilities of coarse-grained performance isolations techniques. We analyze the effectiveness of these features using production workloads on real accelerators and estimate the potential benefits of future memory performance isolation mechanisms. We further expose challenges in CPU designs for accelerated platforms and encourage system architects to rethink the balance between various system components in the accelerator era.

We present Kelp, a runtime system that coordinates allocation of system resources to mitigate performance interference

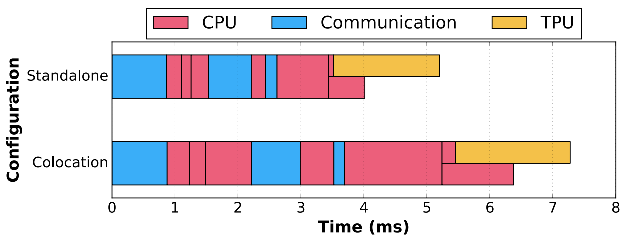


Figure 3: RNN inference server execution timeline on a TPU platform. Execution time for CPU-intensive phases increases by 51% under heavy contention. The interleaving among different phases in the execution timeline is on the order of sub-milliseconds to millisecond.

between low priority CPU and accelerated ML tasks. Kelp leverages existing hardware features such as cache partitioning, NUMA subdomains, and memory BW pressure management. We evaluate Kelp with production ML workloads on three accelerator platforms.

Our work on Kelp makes the following contributions:

1) We conduct a thorough performance study of various ML workloads that leverage different accelerated platforms. Through a detailed sensitivity study, we show that performance of these workloads can be significantly impacted by host memory BW pressure, and demonstrate the need for performance isolation mechanisms (Section III).
2) We explore the capabilities of existing CPU architectures to improve the efficiency of performance isolation (Section IV). Specifically, we use *NUMA Subdomains* to achieve channel partitioning on real systems. We show that by carefully managing memory saturation and avoiding global throttling, this mechanism enables higher system efficiency compared to previous work [28], [29], [30].
3) We present Kelp, a lightweight runtime system that leverages existing hardware features to mitigate performance interference between CPU and accelerated ML tasks (Section IV). We evaluate Kelp with production workloads on various accelerator platforms and compare its performance with competitive baseline solutions. Results show that Kelp is effective in isolating accelerator performance from memory bandwidth interference while sustaining high system throughput (Section V).
4) We show that high-performance accelerators pose new system architecture challenges. Specifically, our results motivate CPU designers to rethink the implication of existing micro-architectural features and designs, and motivate the need for fast and low-overhead fine-grained memory performance isolation mechanisms (Section VI).

While similar problems have been explored in previous work [31], [28], [29], [30], [32], [33], [34], to the best of our knowledge, this is the first paper to address the challenges in host-accelerator interactions in a production environment. In the process of exploring in this new direction, we leverage mechanisms in existing hardwares and demonstrate their capabilities as well as their limitations, motivating future work to continue investigating this issue.

II. Background

A. Target Accelerator Use Case

We focus on the use case in which the accelerator is used by a single application while the CPU is shared by multiple applications. This is in contrast to previous work that assumes multiple workloads share the same accelerator at the same time [33], [34]. While our assumption is different to that of previous work, we find this use case to be very common in production environments. Two main factors lead to our usage model. First, we observe in our measurements that performance of the accelerator is mostly bottlenecked by accelerator memory BW (similar conclusion as previous work [1]). As a result, time-multiplexing accelerators is unlikely to improve accelerator performance because the accelerator memory channels are already fully utilized. Second, given the large datasets of many production ML workloads, accelerator memory is often not large enough to fit the data of multiple workloads. Time-multiplexing is hence infeasible due to the large overhead of data spilling. However, we show in this work that, even in the absence of accelerator resource interference, colocation of accelerated tasks and CPU tasks can still lead to large performance degradation.

B. Task Colocation

We assume that each machine is shared by a *high priority ML task* and multiple *low priority CPU tasks*. Accelerated tasks typically have high priority because accelerators are often capable of higher computational throughput and efficiency compared to CPUs, and customers are usually charged more for using these resources ([35], [36]). While in principle performance interference may be avoided by blocking low priority jobs from colocating with high priority ones, it is often infeasible in production. For example, most computing nodes in production environments are dual-socketed with server-grade CPUs (e.g., two Xeons with > 100 hardware threads). With wide deployment of current and future accelerators, segregating accelerated tasks can cause significant under-utilization of not just the CPU resources, but also DRAM, network, and allocated node power budget. Furthermore, task colocation is often inevitable due to miscellaneous software behavior (system updates, garbage collection, load spikes of benign tasks, etc.). We have observed multiple occurrences of similar events in production in both internal and cloud servers.

C. Accelerator-CPU Interaction

Figure 4 shows the general architecture of an accelerated platform. While the ML workload offloads its heavy computation to the accelerator, part of the computation still runs on the CPU. At the same time, accelerator-CPU interactions are often inevitable. The accelerator achieves high peak throughput by accelerating computation with regular data access patterns and simple control flow, which constitute most of the operations in training and inference workloads. However, it is often expensive or infeasible to map all necessary operations to accelerators. Examples of these operations include irregular memory accesses (e.g., pointer chasing), complex numeric operations that are rarely used but expensive to implement (e.g., trigonometric functions), and communication between multiple workers in large-scale

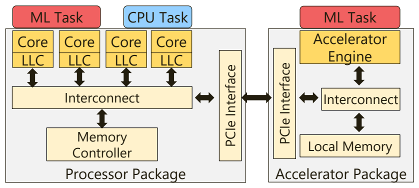


Figure 4: Architecture of an accelerated platform.

distributed training. When accelerators fall back to CPUs for these operations, there can be multiple interactions between accelerators and CPUs even within a single training step. As a result, under heavy memory BW contention, the CPU tasks in the accelerated ML workload can easily become the bottleneck of the entire ML application.

One example of such dependence on the CPU is the *parameter server* in distributed machine learning, as described in Section I. Another example of CPU assistance is the *in-feed operation*, in which the host processor is responsible for interpreting and reshaping the input data before it is consumed by the accelerators [37]. It is also common for CPUs to handle miscellaneous computation tasks, which take advantage of CPUs' capability to handle irregular and complex instruction streams. For example, *beam search* is a commonly used algorithm to reduce the search space in machine translation programs [38]. Instead of greedily following the best candidate in each iteration, beam search sorts partial solutions and expands on a subset of best candidates [38]. Slowdown of the above tasks can starve the accelerators and significantly degrade performance of the application. The machine learning workloads used in this work include all three types of interaction discussed above (Section III).

D. Managing Interference at WSC Scale

Contention for host resources between the ML tasks and low priority CPU tasks causes significant performance degradation and efficiency loss for the accelerated workload. The resource contention problem is further exacerbated when deploying accelerators at the WSC scale by the following factors.

1) Accelerated workloads can span multiple nodes and cross-node synchronization is often necessary for each iteration of variable computation [8]. As a result, service-level performance of distributed workloads is even more susceptible to interference due to "tail amplification" [27].
2) ASIC accelerators are largely programmable, and different accelerated workloads can have different levels of sensitivity to resource contention and different requirements on host resources. As a result, an ideal solution needs to handle different application behaviors (e.g., various compute and memory intensity, interaction time granularity, etc.) at runtime.
3) There is a large number of CPU workloads with drastically different performance characteristics in WSC production environments [39]. Performance of any colocated accelerated ML tasks can be severely impacted without effective and adaptive isolation mechanisms.

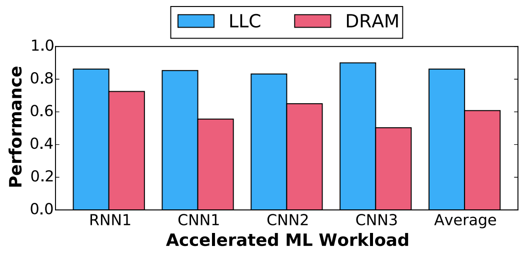


Figure 5: Workload sensitivity to shared resource interference. Performance is normalized to no interference.

III. Accelerated ML Workloads

We use four ML workloads that run on three accelerated platforms in this study (see Table I; details of the workloads are, unfortunately, confidential because they are used in production). In this section we first describe the platforms and workloads. We then analyze their sensitivity to different shared resources.

A. Platforms and Workloads

The TPU platform is equipped with the first generation *Tensor Processing Unit*. The TPU is a PCI-e based accelerator that targets inference workloads. The execution engine is a Matrix Accumulation unit with peak throughput of 92 TFLOPS [1]. We run an RNN-based natural language processing inference workload (**RNN1**) on the TPU platform. Requests are generated in a pipelined fashion to ensure high utilization of all computing resources. Specifically, we sweep the query throughput (measured in queries-per-second or QPS) and analyze the tail latency. The target throughput we use in the paper is at the knee of the tail latency curve. The sweep plot is omitted for brevity.

Cloud TPU is the second-generation *Tensor Processing Unit*. A Cloud TPU device has a peak throughput of 180 TFLOPS and 64 GB of high-bandwidth on-chip memory [2], and can also execute both training and inference workloads in the cloud. We include two CNN training benchmarks (**CNN1** and **CNN2**), which have different CPU and memory intensities in the workload mix.

Finally, we study GPU platforms which are widely used for training ML models (**CNN3**). While **CNN3** is based on a distributed TensorFlow architecture, we only use one GPU worker in the experiment in order to reduce noise caused by the network. The training steps of this benchmark are processed in lock-step among all distributed workers and parameter servers, and latency of the slowest parameter server can bottleneck the service-level throughput [27]. As a result, performance degradation caused by resource interference in scale-up environments has been measured to be similar.

B. Interference Sensitivity

As we discussed in Section II-A, we focus on the use case in which one high priority application has exclusive access to the accelerators. However, low priority CPU tasks can still interfere with the accelerated task by contending for shared resources, including in-pipeline resources and private caches shared through simultaneous multi-threading (SMT), the last-level cache, and main memory BW. To identify the performance bottlenecks and quantify sensitivity, we use the following two synthetic workloads (we will introduce more production workloads in Section V). **LLC** contends for the last-level cache, all caches closer to the CPUs, and in-pipeline resources (SMT is enabled in all experiments). Its dataset size is just small enough to fit in the LLC on the CPU of each platform. **DRAM** contends for DRAM BW. It traverses a large array that doesn't fit in the LLC. On our multi-socket platforms, we use core affinity and control the NUMA policy (`numactl`) to ensure that all threads and data reside in the same socket as the accelerated workload.

Figure 5 summarizes the results of the sensitivity study. On average, LLC resource contention causes a noticeable performance degradation of 14%. However, colocation with the DRAM aggressor causes a dramatic 40% performance loss on average. This result is surprising because the accelerators are already designed to minimize interactions with the host CPUs [1]. We also performed a sweep analysis of the ratio of computation and communication between accelerator and host CPU for CNN1 and CNN2. The same level of sensitivity is observed across the spectrum for both workloads. Figure for this analysis is omitted to conserve space.

In the rest of this paper, we focus on mitigating performance interference caused by DRAM BW contention because it dominates the performance degradation. We use a combination of production and synthetic batch workloads in our evaluation. LLC interference is addressed by dedicating an LLC partition to accelerated tasks using Intel's Cache Allocation Technology (CAT) [40]. We also identify another performance bottleneck in memory traffic that crosses socket boundaries. Related experiments are discussed in Section VI.

IV. Kelp: Mitigating Performance Interference

The goal of Kelp is to mitigate performance impact of DRAM BW interference as observed in the previous section. While much work has been done on various performance isolation mechanisms (Section VII), we are constrained to techniques that are already implemented today. A commonly used approach explored by previous work is core throttling [28], [29], [30]. It relies on a software runtime to detect performance interference and throttles tasks at core granularity. While effective, core throttling is not efficient to fully exploit the available bandwidth due to the coarse throttling granularity and time granularity.

To further improve system efficiency, we leverage NUMA Subdomain mechanisms to separate each socket into two NUMA subdomains, enabling low priority tasks to take full advantage of memory BW resources. We also discover and address challenges in shared memory backpressure [41] and further enhance system throughput with subdomain backfilling. The rest of this section describes the design of Kelp in more detail. Our exploration of existing CPU capabilities also exposes challenges and opportunities for future system architectures, as detailed in Section VI.

A. NUMA Subdomain Performance Isolation

One of the key CPU mechanisms that Kelp relies on is *NUMA subdomain* performance isolation. Figure 6 provides a high-level overview of this feature. The technique splits a physical socket (cores, LLC, interconnect, and memory controllers) into two *NUMA subdomains*, which are exposed to the operating system

Workload	Platform	Description	CPU-Accelerator Interaction	CPU Intensity	Host Memory Intensity
RNN1 Inference	TPU	Natural language processing	Beam search	Medium	Low
CNN1 Training	Cloud TPU	Image recognition	Data in-feed	Low	Low
CNN2 Training	Cloud TPU	Image recognition	Data in-feed	High	Medium
CNN3 Training	GPU	Image recognition	Parameter server	Low	High

Table I: Accelerated ML platforms and production workloads. Detailed measurements are not publishable due to confidentiality concerns.

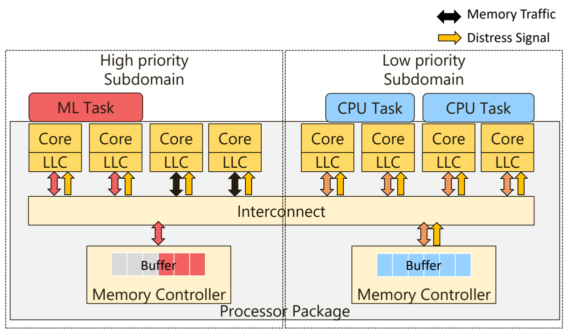


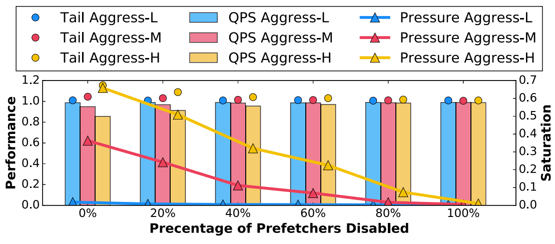
Figure 6: NUMA subdomain and memory backpressure.

as two NUMA domains. Memory requests within each NUMA subdomain are handled by the corresponding memory controller. While channel partitioning has been discussed before for CPU workloads [32], we evaluate it on real accelerated platforms with Intel processors that implement this techniques as *sub-NUMA Clustering* (SNC) [42] and *Cluster-on-Die* (CoD) [43].

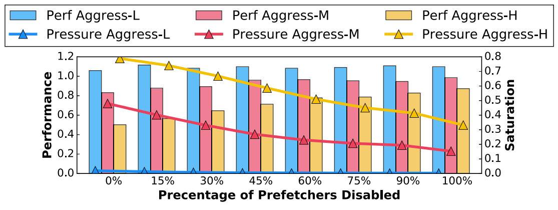
As a side benefit of this partition, local memory requests enjoy both lower LLC and memory latency compared to when NUMA subdomain is disabled, although latency for memory accesses to the remote NUMA subdomain will be slightly higher. More importantly, because memory traffic for each NUMA subdomain is handled by a different memory controller, we can largely isolate contention for memory BW between the high priority accelerated tasks and low priority CPU tasks by assigning them to two different NUMA subdomains.

B. Shared Memory Backpressure

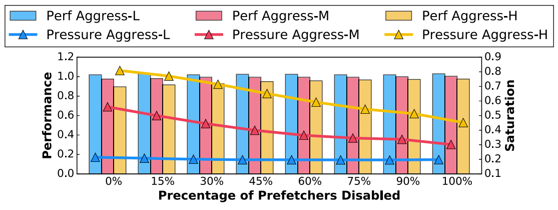
While the NUMA subdomain ideally should provide almost perfect memory isolation, in our experiments with synthetic benchmarks, we still observe noticeable performance degradation. After further investigation, we identify the source of cross subdomain interference to be the shared backpressure mechanism, which is also shown in Figure 6. When CPU tasks in the low priority subdomain generate a large amount of memory traffic, requests queue up at the corresponding memory controller and saturate its bandwidth. In such cases, that memory controller broadcasts a distress signal to all CPU cores across the entire socket. After receiving the distress signal, the CPU cores are throttled in order to avoid congesting the interconnection network. Such throttling is essential in most cases to prevent unnecessary delay of other communication over the network. Unfortunately, this mechanism is detrimental in our use case because most of the memory traffic is routed within each NUMA subdomain. Instead, this mechanism actually reduces the effectiveness of the memory interference protection that NUMA subdomains can potentially provide.



(a) RNN1



(b) CNN1



(c) CNN2

Figure 7: Performance impact of shared memory backpressure and effectiveness of backpressure management with prefetchers toggling. Three levels of aggressiveness of the antagonists (L, M, and H) are experimented with.

Fortunately, system software can measure the level of memory saturation on our hardware using existing hardware performance monitoring infrastructure. Specifically, we use measurements from the performance event `FAST_ASSERTED` from the Intel uncore LLC coherence engine [41]. This event reports the number of cycles in which the distress signal is asserted. We can then quantify *memory saturation* by dividing this cycle count by the number of elapsed cycles between two measurements. To control the memory pressure generated by the low priority CPU cores, we resort to disabling L2 prefetchers for the low priority CPU tasks [44], which significantly reduces memory traffic at the cost of performance loss of low priority CPU tasks.

To demonstrate the impact of the global throttling caused by memory backpressure and the effectiveness of toggling prefetchers, we run three accelerated workloads (RNN1, CNN1,

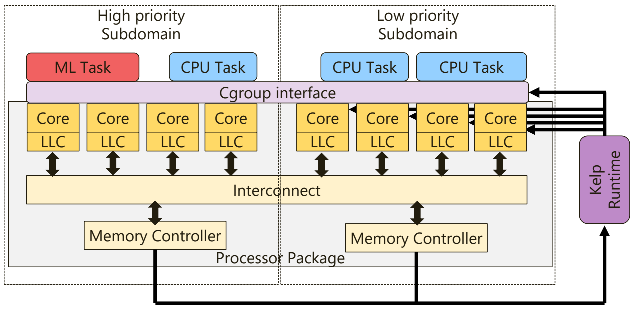


Figure 8: Kelp architecture.

and CNN2) with synthetic DRAM aggressors. The aggressors are configured to produce three levels of memory pressure (low, medium, and high). Accelerated tasks and CPU tasks run in separate NUMA subdomains. Performance of the accelerated tasks is normalized to that of a standalone configuration. For each workload mix, we gradually change the number of prefetchers disabled and plot the performance of the accelerated task and measured memory saturation. For the RNN1 inference workload, we further plot the 95%-ile tail latency. Results are summarized in Figure 7. Each cluster of bars plots ML task performance (and tail latency for RNN1) for the given prefetcher configuration across three levels of memory pressure. Measured memory pressure is plotted as lines using the right axes.

Several observations can be made from the results. First, NUMA subdomains alone cannot provide enough protection. When no prefetchers are turned off, RNN1 QPS decreases by 14% and tail latency increases by 16%. CNN1 and CNN2 suffer a performance degradation of 50% and 10% respectively. Second, turning off prefetchers reduces the performance degradation caused by shared backpressure for most of the workload mixes. Finally, when memory pressure is low, performance of the accelerated tasks may be slightly better than standalone due to the lower LLC and memory access latency associated with enabling NUMA subdomains (e.g., CNN1 and CNN2 performance is 9% and 2% higher than standalone in best cases).

C. *Improving System Throughput*

One significant limitation of using NUMA subdomains alone to provide performance isolation is that this degrades total system throughput. Due to the coarse granularity of NUMA subdomains (SNC and CoD), we can only achieve memory BW interference isolation between two subdomains. As a result, this approach can suffer from significant fragmentation of resources (core, cache, and memory). We quantify this performance loss in Section V. To regain the lost throughput due to fragmentation, we backfill the high priority subdomain with CPU tasks. We show in Section V that combining backfilling with subdomains can improve system efficiency by 17%.

With task backfilling, it is crucial to tightly manage the BW interference within the high priority subdomain. Since the DRAM BW is lower than when NUMA subdomains are not enabled, a similar level of BW interference can potentially cause even higher performance degradation. In this work, we measure the BW consumed by the memory channels that correspond to the high priority subdomain, and throttle CPU tasks when necessary by reducing the number of cores available to the low priority tasks using CPU masks.

Algorithm 1 Kelp Resource Management Algorithm

```
1:  procedure KelpResourceMngt
2:      $bw_s, lat_s, sat_s$ = MeasureSocket()
3:      $bw_h$ = MeasureHiPriority()
4:
5:      if HiBW$_h(bw_h)$ or HiLat$_s(lat_s)$ then
6:          $action_h$ = THROTTLE
7:      else if LoBW$_h(bw_h)$ and LoLat$_s(lat_s)$ then
8:          $action_h$ = BOOST
9:      else
10:         $action_h$ = NOP
11:     if HiBW$_s(bw_s)$ or HiLat$_s(lat_s)$ or HiSat$_s(sat_s)$ then
12:         $action_l$ = THROTTLE
13:     else if LoBW$_s(bw_s)$ and LoLat$_s(lat_s)$ and LoSat$_s(sat_s)$
    then
14:         $action_l$ = BOOST
15:     else
16:         $action_l$ = NOP
17:
18:     ConfigHiPriority($action_h$)
19:     ConfigLoPriority($action_l$)
20:     EnforceConfig()
```

D. *Kelp Workflow and Implementation*

To put everything together, Figure 8 summarizes the architecture of Kelp. Kelp is designed to run with the node-level scheduler runtime (e.g. Borglet [45]) in order to gather necessary task information such as job priority and profile. When applications are first scheduled onto the server, the corresponding profile is loaded by Kelp, which includes high and low watermarks for each measurement. Kelp assigns both accelerated ML tasks and low priority CPU tasks to the designated subdomains. CPU tasks are prioritized to be assigned to the low priority subdomain. At runtime, Kelp makes four types of measurements from the processor: socket-level memory bandwidth, memory latency, memory saturation (discussed in detail in Section IV-B), and high-priority subdomain bandwidth. Kelp samples system performance every 10 seconds and has negligible performance overhead. The effectiveness of Kelp is not sensitive to the sampling frequency.

Algorithm 1 describes the node level resource management algorithm used by Kelp. Subscripts denote the scope (subdomain or socket) that the corresponding value describes. By comparing measurements from performance counters with the watermarks specified in the application profile, Kelp chooses to boost, throttle, or keep the resource configuration for low priority CPU tasks in each subdomain. Algorithm 2 details the approach we use to configure resources within each subdomain. When throttling the low priority subdomain, we are more aggressive in disabling prefetchers in order to prioritize ML task performance.

Note that NUMA subdomain partitioning is a passive mechanism that redirects memory requests to the corresponding memory controllers. Specifically, Kelp enforces priorities for memory requests across NUMA subdomains with little additional la-

Algorithm 2 Resource Configuration Algorithms

```
1:  procedure ConfigHiPriority($action_h$)
2:      if $action_h$ = THROTTLE then
3:          if $coreNum_h > minCoreNum_h$ then
4:              $coreNum_h$ -=1
5:      else if $action_h$ = BOOST then
6:          if $coreNum_h < maxCoreNum_h$ then
7:              $coreNum_h$ +=1
8:
9:  procedure ConfigLoPriority($action_l$)
10:     if $action_l$ = THROTTLE then
11:         if $prefetcherNum_l > 0$ then
12:             $prefetcherNum_l$ /=2
13:         else if $coreNum_l > minCoreNum_l$ then
14:             $coreNum_l$ -=1
15:     else if $action_l$ = BOOST then
16:         if $prefetcherNum_l < coreNum_l$ then
17:             $prefetcherNum_l$ +=1
18:         else if $coreNum_l < maxCoreNum_l$ then
19:             $coreNum_l$ +=1
```

tency. The sampling is necessary only to avoid throttling caused by backpressure and opportunistically improve low priority task performance. Thresholds for throttling are configured conservatively to prioritize accelerated tasks. The combination of these techniques enables Kelp to handle fast phase changes (Figure 3) without having to react at the same time granularity.

V. Evaluation

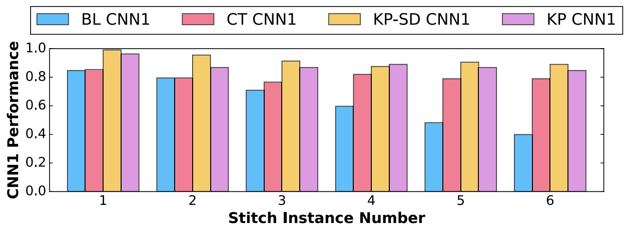
A. Methodology

We evaluate Kelp with four production ML workloads across three accelerator platforms as listed in Table I. For the colocated CPU tasks, we use a combination of synthetic and production workloads as follows:

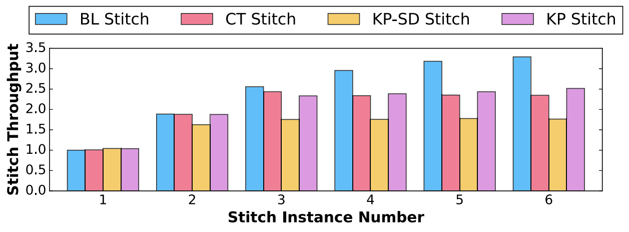
1) **Stream** traverses a large array that does not fit in the last-level cache of any of the platforms.
2) **Stitch** is a production batch job that stitches images to form the panoramas for Google Street View.
3) **CPUML** is a production CNN training workload based on TensorFlow-Slim [46]. Low priority CPU-based training task is common because of high resource availability.

To better demonstrate the effectiveness of Kelp, we compare evaluation results of four system configurations:

1) **Baseline (BL)**: Task priority is specified through the Borg [45] interface; resource contention is unmanaged.
2) **CoreThrottle (CT)**: A competitive resource management configuration that closely mimics mechanisms from previous work[28], [29], [30]. Memory BW interference is managed by limiting the number of cores available to the low priority CPU tasks through CPU masks, while LLC interference is managed by using dedicated LLC partitions to the accelerated tasks through Intel Cache Allocation Technology (CAT) [40].
3) **Kelp Subdomain (KP-SD)**: A simplified Kelp implementation that uses only NUMA subdomains (SNC and CoD) and manages global throttling due to memory backpressure by toggling L2 prefetchers for CPU tasks [44].
4) **Kelp (KP)**: The full Kelp implementation which further improves system throughput by backfilling CPU tasks.



(a) CNN1 performance normalized to standalone performance.



(b) Stitch throughput normalized to Baseline with one instance.

Figure 9: Memory pressure sweep CNN1 + Stitch.

We evaluate the workload mixes on real hardware. Experiment for each configuration and workload mix combination is repeated multiple times and the median is reported. Performance results are captured from the applications using application-specific metrics. Hardware measurements are made through performance counters. Requests for RNN1 are generated in a parallel and pipelined fashion. While results not included in the paper, we sweep load generation configurations for RNN1 and choose a target rate at the knee of the throughput-latency curve.

B. Benchmark Case Studies

To understand the effectiveness of Kelp, we perform a configuration sweep analysis that compares the performance of the four configurations for all workload mixes. In this section, we discuss the results for two representative cases.

In the first mix, we run CNN1 with Stitch. This workload mix is interesting because CNN1 is highly sensitive to BW contention and Stitch aggressively contends for BW resources. Figure 9a plots CNN1 performance and Figure 9b plots Stitch throughput. As the number of Stitch instance increases, Baseline performance of CNN1 decreases by up to 60%, while throughput of Stitch keeps increasing. CoreThrottle improves the average performance of CNN1 by 16% while decreasing the harmonic mean of throughput of the low priority Stitch by 11%. Subdomain further improves CNN1 average performance from CoreThrottle by 12%, but Stitch suffers from a significant 25% average throughput degradation. In comparison, Kelp improves the average performance of CNN1 from CoreThrottle by 8%, while only reduces Stitch throughput by 9%. We conclude that Kelp achieves higher efficiency for this challenging workload mix compared to previous work because it achieves 8% higher CNN1 performance *and* 2% higher Stitch throughput.

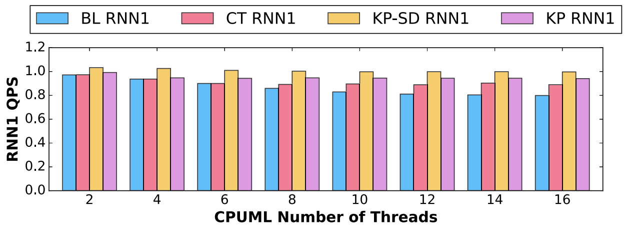
For the second workload mix, we run RNN1 with CPUML. Compared to the previous workload mix, RNN1 is less sensitive to host memory BW interference and CPUML is also less aggressive. Figure 10a plots the QPS of RNN1, Figure 10b plots the tail latency of RNN1, and Figure 10c plots the normalized throughput of CPUML. In this workload mix, Baseline performance of RNN1 gradually plateaus as more CPUML instances are added and system resources saturate. Comparing with the other three configurations, on average CoreThrottle manages 9% average QPS loss, 13% average tail latency increase, and 5% decrease in CPUML throughput. Subdomain has almost no performance degradation for RNN1 at the cost of 33% average CPUML throughput degradation. In comparison, Kelp achieves the best of both worlds with 5% QPS loss, 8% tail latency increase, and 13% average CPUML throughput degradation.

Figure 11 and Figure 12 show the key parameters that each of the three performance isolation mechanisms use at runtime for the two workload mixes. As a general trend, each mechanism becomes more aggressive in throttling CPU tasks to enforce performance isolation as more memory BW contention is introduced to the system. Overall, the second workload mix exerts less stress on memory BW and the system is throttled less; in Figure 12b the vanilla Subdomain configuration is able to achieve enough isolation without having to toggle any prefetchers off. However, Kelp is able to consistently achieve better performance isolation despite different levels of BW sensitivity and interference. Comparing the number of cores allocated for CPU tasks between CoreThrottle and Kelp (Figure 11a and Figure 11c, Figure 12a and Figure 12c), Kelp enables the CPU tasks to use more resources and achieves higher system efficiency.

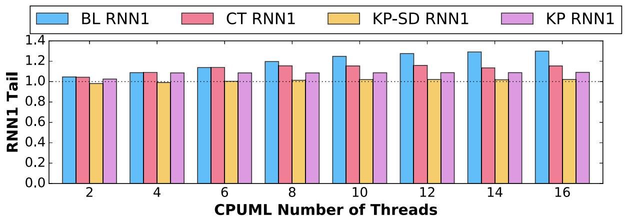
C. Overall Results

Figure 13 summarizes the evaluation results for all workload mixes. We plot ML workload slowdown on the left axis (average computed as arithmetic mean) and CPU workload slowdown on the right axis (average computed as harmonic mean). Compared to Baseline, Kelp reduces the slowdown of accelerated ML tasks by 43%, at the cost of 24% CPU task throughput loss. Compared to previous work, as represented by CoreThrottle, Kelp reduces the slowdown of ML tasks by 7% while achieving the same CPU throughput. Compared to Subdomain, Kelp increases ML task slowdown by 4%, but achieves 19% higher CPU task throughput.

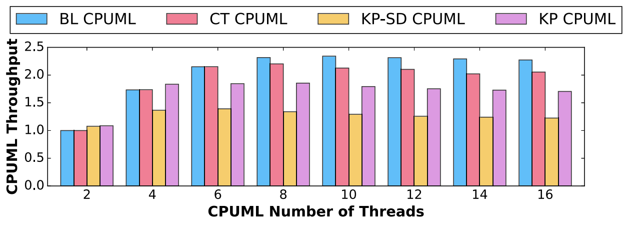
To quantify the efficiency achieved by each runtime solution, we define a new metric that represents the tradeoff between ML and CPU task performance. Specifically, we define the efficiency of a runtime to be the ratio of *performance gain of high priority ML tasks compared to Baseline*, and *throughput loss of CPU tasks compared to Baseline*. This metric can also be interpreted as the ML task performance gain per unit of CPU task throughput loss (so higher is better). Note that this metric does not account for tail latency changes (e.g., RNN1 + CPUML as shown above). Figure 14 summarizes the results for all workload mixes. Overall, Subdomain has the lowest efficiency because of the fragmentation of resources at coarse granularity.



(a) RNN1 QPS normalized to standalone performance.



(b) RNN1 95%-ile tail latency normalized to standalone.



(c) CPUML throughput normalized to Baseline with two threads.

Figure 10: Memory pressure sweep RNN1 + CPUML.

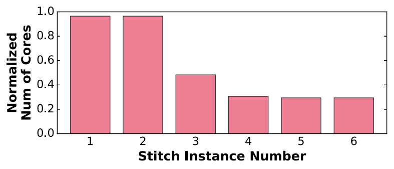
Kelp has higher efficiency compared to CoreThrottle for almost all the workload mixes we tested. While Kelp is less efficient than CoreThrottle for two RNN1 workload mixes, Kelp reduces the tail latency of RNN1 in both cases as we demonstrated with the second case study above. On average, Kelp achieves 17% higher efficiency compared to CoreThrottle, and 37% higher efficiency compared to Subdomain.

VI. CPU Design Challenges and Opportunities

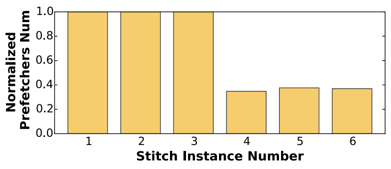
Our exploration of the capabilities of existing hardware also exposes several challenges in CPU designs for accelerated platforms. We summarize these challenges in this section and provide suggestions for future architectures.

A. Remote Memory Interference

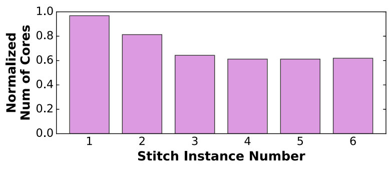
So far we have studied memory BW interference when both accelerated ML tasks and low priority CPU tasks reside on the same socket. However, on some of the platforms we tested, we notice remote memory traffic that crosses socket boundaries causing exceptionally large performance degradation. To focus on this issue, we use an additional synthetic workload **Remote DRAM**. **Remote DRAM** is similar to **DRAM** with the exception that only some of the data and threads are resident on the local memory socket (where the accelerated ML task resides), while the rest reside in the remote socket. This exercises the inter-processor interface (i.e., UPI [47] and QPI [48]). We observe that, compared to TPU



(a) CT cores allocated for CPU tasks.

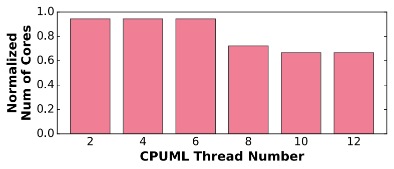


(b) KP-SD prefetchers for CPU tasks.

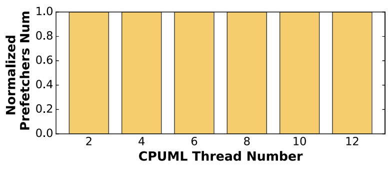


(c) KP cores allocated for CPU tasks.

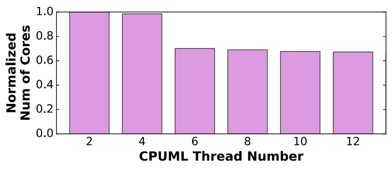
Figure 11: Parameters for three performance isolation configurations for CNN1 + Stitch.



(a) CT cores allocated for CPU tasks.



(b) KP-SD prefetchers for CPU tasks.



(c) KP cores allocated for CPU tasks.

Figure 12: Parameters for three performance isolation configurations for RNN1 + CPUML.

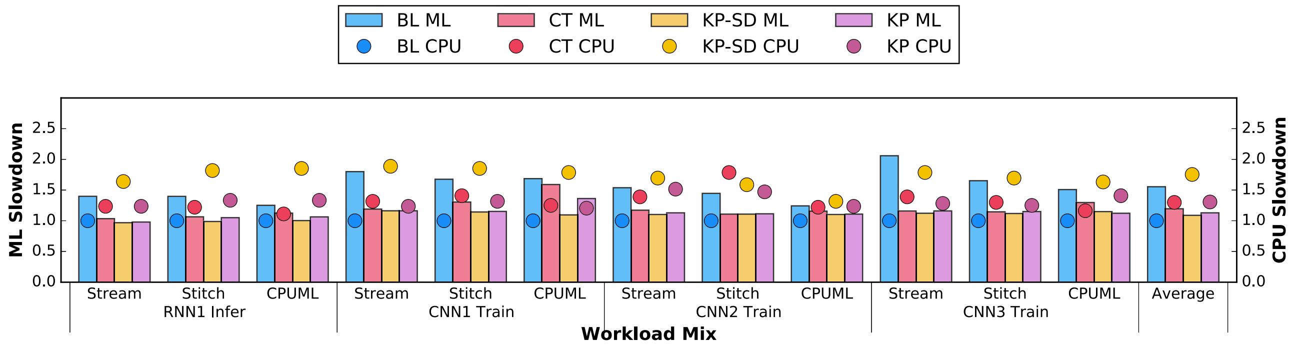


Figure 13: ML and CPU task performance results.

and GPU platforms, Cloud TPU platform (CNN1 and CNN2) are more sensitive to **Remote DRAM** traffic that crosses socket boundaries. Figure 15 summarizes the results. Compared to the performance degradation caused by **DRAM**, **Remote DRAM** causes an additional 16% and 27% performance loss for CNN1 and CNN2.

To further understand the performance impact of remote memory traffic on the Cloud TPU platform, we perform a sweep analysis in which we gradually change the percentage of the aggressor's dataset on the socket local to the ML tasks. Within each dataset percentage configuration, we sweep the percentage of threads that reside on the ML tasks' local socket. Results of this experiment are shown in Figure 16. The figure shows that remote memory traffic causes even higher slowdown than local memory interference. While the high sensitivity can be an artifact caused by processor-specific implementation choices (e.g., overhead associated with the coherence protocol), future scaling of multi-processor multi-core systems is likely to encounter similar issues in the memory subsystem. These architectural and micro-architectural decisions can have significant system-level performance and utilization implications.

B. QoS-Aware Prefetching

We show in Section IV-B that prefetching requests can cause high memory pressure. This is shown by the restored accelerated task performance when L2 prefetchers for low priority CPU task cores are partially turned off. While it is well understood that prefetcher requests should not impact the performance of demand memory requests [49], this problem still exists in our scenario because it involves interactions between the memory subsystem and NUMA subdomains. Although Kelp solves the issue by managing prefetcher pressure in system software, this functionality can be integrated into hardware. A hardware-based solution has the advantage of being able to adapt to fast-changing system behavior with little performance overhead. It can also guide the aggressiveness of prefetchers based on the immediately-available information of memory resources [50].

C. Global Memory BW Backpressure

We show in Figure 7 that NUMA subdomains alone cannot provide performance isolation because of the global memory backpressure mechanism. The slowdown caused by global memory backpressure is an example of how the on-chip interconnect and memory subsystems can together cause unexpected QoS

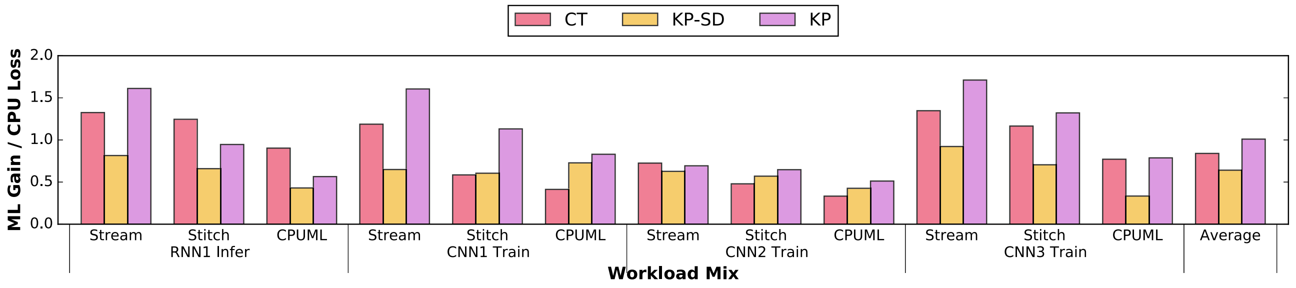


Figure 14: Performance tradeoff comparison between CT, KP-SD, and KP.

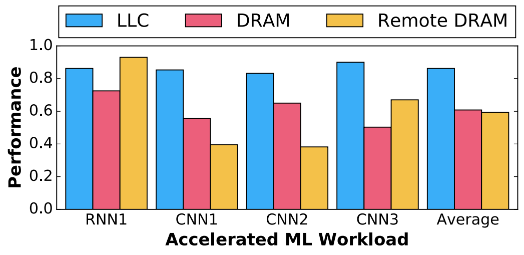
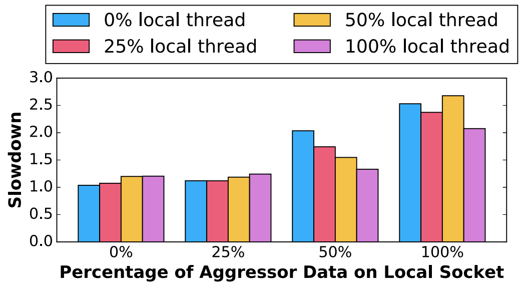
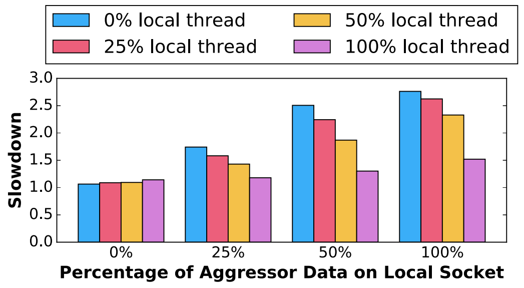


Figure 15: Workload sensitivity to remote memory interference compared to LLC and local DRAM.



(a) CNN1 Sensitivity to Remote Memory Traffic



(b) CNN2 Sensitivity to Remote Memory Traffic

Figure 16: Cloud TPU Platform Remote Memory Sweep.

issues. Specifically, the backpressure-based throttling mechanisms do not differentiate requests coming from different subdomains. Ideally, memory backpressure should be sent to the offending hardware thread in order to avoid unnecessary performance loss. Exposing the capability of throttling individual threads to system software (through interfaces such as machine specific registers [40]) can help further improve system utilization. One example is to let users annotate priority of hardware threads so that low priority offending tasks can be throttled first in cases of memory BW contention.

D. Fine-Grained Memory Isolation

While we show in Section V that Kelp is able to successfully isolate performance interference and improve system efficiency compared to previous work, it has the limitation of depending on SNC and CoD to achieve most of the benefits. Therefore, Kelp operates on a relatively coarse granularity. Industry has made significant progress in enhancing the QoS capabilities of server products. For example, Intel introduced the Memory Bandwidth Allocation (MBA) feature in recent architectures that uses a hardware request rate controller [40]. Users can de-prioritize memory-intensive jobs by throttling these jobs' memory requests [40]. Unfortunately, this rate controller appears to throttle traffic from the core to the interconnect, last-level cache, and memory controllers. As a result, throttling decisions also impact last-level cache BW in addition to main memory BW.

In future work, we will explore the possibility to further improve system efficiency when enforcing QoS for accelerated tasks by hardware-based fine-grained memory performance isolation [51], [52], [53], [54]. Compared to software solutions (e.g., Kelp), hardware techniques can differentiate memory requests from different tasks and handle them with different policies. By exposing the hardware QoS capability to software, system software can further control the tradeoff between ML-task QoS and system throughput.

To estimate the effectiveness of such fine-grained mechanisms, our evaluation results on Subdomain and Kelp in Figure 13 approximate an upper bound of what can potentially be achieved. Specifically, fine-grained isolation can achieve ML performance better than Subdomain (at least 4% higher than Kelp), because Subdomain methods increase memory access latency at high BW due to decreased channel interleaving. In the meantime, low priority performance can still be higher than CoreThrottle or Kelp because the hardware mechanism can achieve higher total memory BW utilization. More importantly, hardware solutions can provide more robust performance by adapting to program behavior changes faster (we demonstrate examples of such opportunities in Figure 3), which is critical for high priority tasks that have short execution time deadlines.

VII. Related Work

A. Wide Adoption of Accelerators

Recent years have seen the increasingly wide adoption of accelerators, especially for ML applications due to their inherently high computation and data intensity. Notably, GPUs are frequently used in various ML frameworks (e.g., [8], [9], [10], [11]) and applications (e.g., [12], [13], [14]). Jouppi et al. describe the first generation TensorFlow Processing Unit (TPU) that targets inference for neural network applications [1]. The recent release of the Cloud TPU further expands the capability of Google's ML accelerator to training and to scale out using high-speed interconnection [2]. Many other ASIC neural network accelerator designs (e.g., [15], [16], [17], [18]) and optimizations (e.g., [19], [20], [21], [22]) have been proposed while other work explores the option of using FPGA-based solutions (e.g., [55], [56], [57], [58]).

Many other disciplines have adopted accelerators targeting mission-critical tasks. Putnam et al. explore the option of using FPGAs to accelerate Bing's ranking stack [3]. Khazraee et al. study designs of ASIC-based acceleration schemes for video transcoding and cryptocurrency systems [59], [60]. Baidu and Xilinx develop FPGA acceleration services to accelerate encryption for public Cloud services [61]. Intel also announced FPGA-based products that targets WSC workloads [23].

B. Accelerator QoS and Utilization

Accelerator QoS and utilization have been studied with different assumptions in the past. Chen et al. study QoS for accelerators, assuming that the accelerators are time-multiplexed between several tasks that are categorized into two priority groups [33], [34]. *Baymax* focuses on performance interference caused by queuing delay and PCI-e BW contention [33]. It predicts task durations using linear regression and KNN, and re-orders tasks based on the prediction results to enforce QoS targets. *Prophet* further considers intra-accelerator memory BW and profiles each application with testing inputs to estimate runtime memory BW usage [34]. Shen et al. focus on the utilization of arithmetic units for FPGA-based CNN accelerators and propose to partition FPGA resources to improve utilization [62].

However, we assume in this work that each accelerator device can be subscribed by only one application at a given time. As shown in the roofline model analysis in [1], performance for production workloads is almost always bound by accelerator memory BW instead of computational throughput, so there is little motivation to enable the time sharing of accelerators across applications. Doing so will also increase the complexity of on-chip memory management and potentially data management overhead. Also, while not discussed in the evaluation, we did not observe PCI-e BW constraining the performance of the profiled workloads. On the other hand, we demonstrate that resource interference of host memory can cause significant performance degradation across various production workloads and accelerator types.

C. System Performance Isolation

Kambadur et al. conduct a study of application interference using Google's production datacenter workloads [63]. Mars et al. use microbenchmarks to measure workload sensitivity and stress for the memory subsystem and schedule tasks accordingly [64], [65]. Zhang et al. propose to use CPI data to identify interference issue and throttled the interfering tasks with CPU capping [30]. Heracles is a feedback-based controller that uses architectural techniques to guarantee that high priority tasks meet their latency targets [28]. Zhu et al. propose to convert latency headroom for high priority tasks to improved system performance [29]. Jacob et al. and Hsu at al. study the tail latency of memcached and address interference problems using an improved kernel scheduler [66] and fine-grained voltage boosting [67]. Kasture et al. propose a cache partitioning technique to balance the tail latency of high priority tasks and system throughput [31].

Kelp builds on many of the ideas proposed by previous work (e.g., cache partitioning [31], core throttling [28], [29], [30], and channel partitioning [32]). However, we identify the new problem of performance interference in accelerated machine learning platforms due to fine-grained interaction between CPUs and accelerators. We successfully apply a combination of the above techniques in this new context. We show that these techniques, when enabled by hardware, can effectively tackle the problem of accelerator QoS which was not present before. Our detailed profiling of production ML workloads also shows additional opportunities to further improve system utilization and QoS for accelerated platforms through hardware-based performance isolation.

VIII. Conclusion

In this work, we study the performance interference between high priority accelerated ML tasks and low priority CPU tasks. We experiment with four production ML workloads on three accelerated platforms. Our experiments with synthetic workloads show that these ML workloads are highly sensitive to host memory BW contention. Specifically, while core resource contention causes a noticeable performance degradation of 14%, resource contention for DRAM BW causes a 40% performance loss on average. To address this issue, we design and implement Kelp, a software runtime that isolates high priority accelerated ML tasks from memory resource interference. Kelp uses existing hardware features such as NUMA subdomains and memory pressure management by toggling prefetchers. We evaluate Kelp with both production workloads and synthetic aggressors and compare effectiveness with previously proposed solutions. Results show that Kelp is effective in mitigating performance degradation of the accelerated tasks and improves their performance by 24% on average. Compared to previous work, Kelp reduces performance degradation of ML tasks by 7% while achieving the same throughput from low priority CPU tasks, and increases system efficiency by 17%.

The wide adoption of accelerators creates exciting opportunities to evolve traditional system architectures. Our work focuses on node-level runtime mechanisms and demonstrates multiple challenges posed by high-performance accelerators. Specifically, we show that further exploring the design space of fine-grained memory performance isolation can potentially enable better tradeoff between performance and QoS of accelerated tasks and total system throughput.

IX. Acknowledgements

We recieved much help and insights from friends and colleagues while working on Kelp. In no particular order, we would like to thank Luiz Barroso, Nishant Patil, Robert Hundt, Norm Jouppi, Andy Swing, Doe Hyun Yoon, Zach Rowinski, Dan Hurt, Jichuan Chang, David Ruiz, Rohit Jnagal, Konstantinos Menychtas, Stephane Eranian, Milad Hashemi, Minkyu Jeong, Alex Ramirez, and Junwhan Ahn. We also thank the annonymous reviewers for their feedback.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[2] J. Dean and U. Hölzle, "Google cloud tpus." https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, August 2017.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, 2014.

[4] "Microsoft unveils project brainwave for real-time ai." https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[5] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[6] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning.," in *AAAI*, pp. 4278–4284, 2017.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[11] T. A. S. Foundation, "Apache mahout." http://mahout.apache.org/, 2017.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[13] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, 2012.

[14] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[15] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, 2014.

[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[18] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[20] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015.

[21] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, 2016.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[23] Intel, "Intel programmable acceleration card with intel arria 10 gx FPGA." https://www.altera.com/pac, October 2017.

[24] Google, "Distributed tensorflow." https://www.tensorflow.org/deploy/distributed, August 2017.

[25] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, 2012.

[26] Google, "High-performance models." https://www.tensorflow.org/performance/performance_models, August 2017.

[27] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[28] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[29] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[30] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), ACM, 2013.

[31] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," *ACM SIGARCH Computer Architecture News*, 2014.

[32] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[33] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[34] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[35] Google, "Google cloud platform pricing calculator." https://cloud.google.com/products/calculator/, 2017.

[36] Amazon, "Amazon EC2 pricing." https://aws.amazon.com/ec2/pricing/on-demand/, 2017.

[37] Google, "Xla: The tensorflow compiler framework." https://www.tensorflow.org/versions/r0.12/resources/xla_prerelease, February 2017.

[38] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, Association for Computational Linguistics, 2003.

[39] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[40] Intel, "Intel 64 and ia-32 architectures software developer's manual," October 2017.

[41] Intel, "Intel Xeon processor e5 and e7 v3 family uncore performance monitoring reference manual," June 2015.

[42] D. Mulnix, "Intel Xeon processor scalable family technical overview." https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, September 2017.

[43] D. Mulnix, "Intel Xeon processor e5-2600 v4 product family technical overview." https://software.intel.com/en-us/articles/intel-xeon-processor-e5-2600-v4-product-family-technical-overview, April 2016.

[44] V. Viswanathan, "Disclousure of hardware prefetcher control on some intel processors." https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014.

[45] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[46] S. Guadarrama and N. Silberman, "Tensorflow-slim." https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim.

[47] Intel, "Intel Xeon processor scalable family datasheet, volumn one: Electrical," July 2017.

[48] Intel, "An introduction to the intel quickpath interconnect," January 2009.

[49] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," *ACM SIGARCH Computer Architecture News*, 2011.

[50] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[51] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "SQUASH: Simple QoS-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *arXiv preprint arXiv:1505.07502*, 2015.

[52] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012.

[53] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, 2013.

[54] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture. ISCA*, 2016.

[55] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, 2015.

[56] K. Ovtcharov, O. Ruwase, J. Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Toward accelerating deep learning at scale using specialized hardware in the datacenter," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015.

[57] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009.

[58] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale FPGA-based convolutional networks," *Scaling up Machine Learning: Parallel and Distributed Approaches*, 2011.

[59] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "ASIC clouds: specializing the datacenter," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[60] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor, "Moonwalk: Nre optimization in ASIC clouds," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[61] S. Gianelli, "Baidu deploys xilinx FPGAs in new public cloud acceleration services." https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html, 2017.

[62] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[63] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[64] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011.

[65] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, ACM, 2013.

[66] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[67] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015.