FLARE: Flexibly Sharing Commodity GPUs to Enforce QoS and Improve Utilization

Wei Han[1⋆], Daniel Mawhirter[1⋆], Bo Wu[1], Lin Ma[2], and Chen Tian[2]

Colorado School of Mines[1]
{whan,dmawhirt}@mymail.mines.edu, bwu@mines.edu
Huawei US R&D Center[2]
{lin.ma, chen.tian}@huawei.com

Abstract. A modern GPU integrates tens of streaming multi-processors (SMs) on the chip. When used in data centers, the GPUs often suffer from under-utilization for exclusive access reservations, hence demanding multitasking (i.e., co-running applications) to reduce the total cost of ownership. However, latency-critical applications may experience too much interference to meet Quality-of-Service (QoS) targets. In this paper, we propose a software system, FLARE, to spatially share commodity GPUs between latency-critical applications and best-effort applications to enforce QoS as well as maximize overall throughput. By transforming the kernels of best-effort applications, FLARE enables both SM partitioning and thread block partitioning within an SM for co-running applications. It uses a microbenchmark guided static configuration search combined with online dynamic search to locate the optimal (near-optimal) strategy to partition resources. Evaluated on 11 benchmarks and 2 real-world applications, FLARE improves hardware utilization by an average of 1.39X compared to the preemption-based approach.

1 Introduction

Datacenters are gaining increasing popularity as they significantly reduce the computation and storage cost for clients. However, the tremendous up-front investment in servers accounts for 50-70% of the total cost of ownership [4]. The problem is exacerbated by the wide adoption of expensive high-end GPUs to leverage the massive parallelism to accelerate various types of workloads, such as deep neural networks and graph analytics [28,11]. Unfortunately, while CPU utilization in servers is already low (ranging from 10% to 70% [18]), GPU under-utilization is more severe due to the complex dynamic behaviors of GPU applications [8].

A fundamental cause of hardware under-utilization is the strict QoS requirements of latency-critical (LC) applications (e.g., web services and deep learning inference). To meet the QoS target, a conservative scheduler will reserve the entire server for the LC application. A promising solution is multitasking, which co-locates best-effort (BE) applications together with the LC application to share

⋆ equal contribution

the same server and hence the GPUs. However, the BE application may interfere with the LC application, resulting in unacceptable performance degradation for LC requests. Notably, when both co-running applications heavily use the GPU, the slowdown of the LC requests could be over 10x [31].

As far as we know, Baymax [8] and Laius [32] are the only software systems that enforce QoS for shared GPU systems. Baymax assumes that the GPU is a non-preemptable processor and hence a long-running kernel reserves the entire GPU. However, a high-end GPU has tens of streaming multi-processors (SMs), which cannot be fully utilized by a single kernel. As we show in Section 2, GPU kernels may scale poorly in terms of SMs or threads within an SM. Laius takes advantage of the hardware-based partitioning capability but is limited to SM-level partitioning, therefore failing to addressing the scalability issues within SMs.

In this paper, we aim at improving GPU utilization by flexibly partitioning the abundant computational resource between co-running BE and LC applications. We assume that the source code of BE is available and an BE application is constantly running on the GPU when the LC application arrives. Instead of only coordinating GPU kernel executions, we allow a BE application to yield just enough resource to meet the QoS target of the LC kernel. To achieve this goal, we face multiple challenges. First, while one only needs to consider a 1-D resource space for CPU core allocation [20], the GPU has many SMs and each SM concurrently runs several groups of threads (i.e., thread blocks), thus forming a 2-D resource space. Second, since the GPU by default runs the launched kernels in an FIFO manner, a kernel from the BE application may use up all SMs, thus blocking the kernel of the LC application. We need to design a software mechanism to enable the two kernels to run simultaneously on different parts of the GPU. Third, the co-running kernels interfere with each other on a variety of hardware resources, including shared interconnect, L1 cache, L2 cache, streaming cores, and device memory. Therefore, quantifying the performance degradation given a partitioning configuration is difficult. Finally, we try to enforce QoS and maximize utilization which are two conflicting goals. Specifically, by allocating more resources to the LC application, we have a better chance to meet the QoS goal. But it probably reduces the overall throughput at the same time.

To overcome the challenges and improve utilization of *commodity* GPUs, we design and implement a software system, FLARE, which enables flexible GPU sharing, meets QoS goals for LC applications, and maximizes throughput for BE applications. FLARE transforms the kernel of the BE application to be able to yield k ($1 \leq k \leq MaxBlksPerSM$) thread blocks on a subset of n SMs ($1 \leq n \leq MaxSMs$). The pair n_k is called a configuration. The threads of the LC kernel can then be scheduled to run on the released hardware resource. The key novelty of FLARE is its intelligent runtime to quickly figure out the optimal GPU resource partitioning strategy by avoiding pitfalls from two popular existing approaches as follows. The performance model-based approach uses offline training to predict the best configuration [8,33], but its accuracy may suffer from input sensitivity and complicated hardware contention. On the other

hand, a pure dynamic approach (e.g., online profiling and adjusting [19,34]) may not be responsive enough. Worse, it may explore detrimental configurations that lead to hampered QoS or hardware under-utilization. FLARE employs a hybrid methodology. It uses microbenchmarks to characterize the co-run performance degradation space, so given two co-running kernels it quickly predicts an initial configuration to use. Then FLARE leverages the degradation space to dynamically search for the optimal configuration. We show in comprehensive experiments that FLARE outperforms the preemption-based approach while satisfying the QoS targets.

2 Background

Driven by the demand for high-throughput capabilities, the GPU has evolved to leverage massive parallelism with a many-core design to provide huge computational throughput and memory bandwidth. The cores of NVIDIA GPUs are called Streaming Multiprocessors, each of which can simultaneously host multiple active thread blocks (also known as Cooperative Thread Array) contexts. The number of active thread blocks that an SM can host depends on the hardware resource of the SM (i.e., register file size) and the resource requirement of the thread blocks. When a thread block runs on an SM, it is executed in a SIMD fashion with 32 threads (called a warp) at a time.

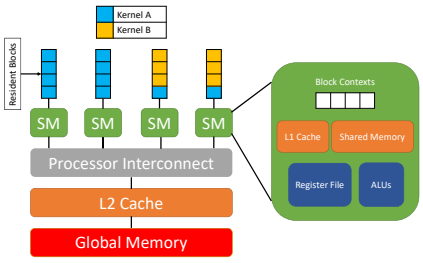

Fig. 1: Graphics Processing Unit (GPU)

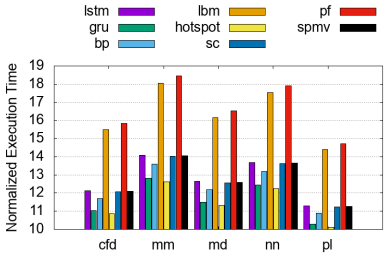

Fig. 2: QoS violation for co-runs when the GPU is unpreemptable.

Conceptually, all the thread blocks of the launched kernels wait in a queue. The hardware implements a FIFO thread block scheduler, which dispatches the waiting thread blocks to SMs as long as the available hardware resource can satisfy the resource demands. Hence, a kernel's thread blocks are guaranteed to be scheduled first before any other thread block of a later launched kernel.

Starting from the Fermi architecture, NVIDIA GPUs support concurrent kernel execution. Later, NVIDIA introduced the Multi-Process Service (MPS), which enables kernels from different applications to be executed simultaneously on the same GPU. However, due to significant context switch overhead, the GPU hardware does not support temporal core sharing. Consequently, the co-running kernels spatially share a GPU only when the earlier launched kernel cannot consume all the computational resources. Due to the organization of the hardware, Fig. 1 shows an interesting resource sharing scenario. The co-running thread blocks from both kernel A and B on the same SM compete to use the L1

cache and ALUs and all the currently running thread blocks contend for use of the interconnect, L2 cache and global memory bandwidth.

3 Motivation and Challenges

3.1 QoS Issues of Non-Preemptable Kernels

To understand the detrimental effect of non-preemptable kernel execution on QoS violation, we run 40 pairs of kernels (details in Section 5) on an NVIDIA Volta GPU. Fig. 2 shows the performance degradation of the LC kernels when they are immediately launched after the BE kernels shown on the X axis. Observe that QoS is violated for all the pairs even if the QoS target is as large as 10 times of the corresponding solo-run execution time when sharing is disabled. This is because the entire time the BE kernel is finishing normally, the LC kernel has to wait in queue, a clearly unacceptable solution.

3.2 Scalability Issues of Preemption-Based Solutions

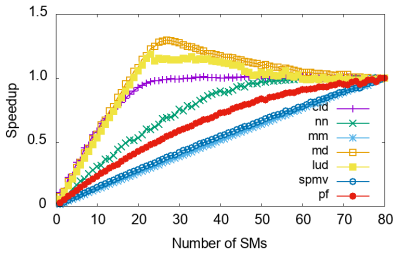

Fig. 3: Solo-run scalability with respect to the number of SMs

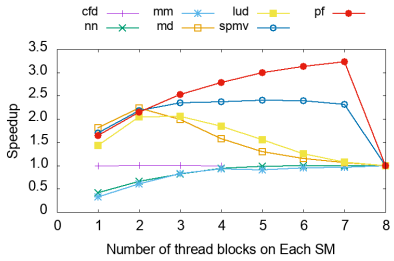

Fig. 4: Solo-run scalability with respect to the number of thread blocks on each SM

Recent work, such as FLEP [31] and Effisha [6], has proposed low-overhead software-based mechanisms to realize preemption on GPUs. With the capability of preemption, we can easily address the QoS issue by preempting the BE kernel whenever an LC arrives. However, the drawback of preemption is that LC kernels monopolize all the available resources regardless of how efficiently it will utilize them. We show the scalability of 7 benchmarks in Fig. 3 and Fig. 4 when we respectively increase the number of SMs and the number of thread blocks within each SM. Since the default scheduling uses up the SMs and thread blocks, the results show that less resource does not necessarily lead to worse performance. Moreover, different applications may have different scaling characteristics. For example, MM (matrix multiplication) prefers more computational resources, while MD's performance culminates with a small portion of the resources (i.e., 26 SMs or 2 thread blocks per SM). Therefore, we need an approach to appropriately partition the GPU to simultaneously corun kernels for the optimal utilization.

3.3 Spatial Co-Running and its Challenges

In order to run a pair of BE and LC application simultaneously, we need a mechanism for the BE application to be able to yield resources (entire SMs or thread block slots of SMs) to the LC application. Though the reduced resource availability and introduced contention cause slowdown for both kernels, we observe such a mechanism allows one to produce a better trade-off between QoS guarantees and overall GPU utilization.

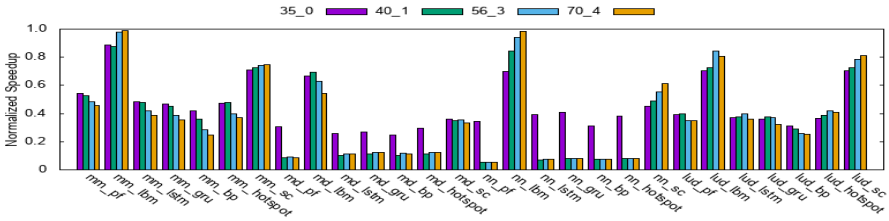

Fig. 5: LC kernel speedup with 280 thread blocks being allocated under different configurations

To understand the complexity of the interference due to co-running, we run 40 kernel pairs with 280 thread blocks allocated to each of the LC kernels. Fig. 5 shows the performance degradation of the LC kernels with four different configurations. On the X axis, the notation A_B represents a BE kernel A co-running with a LC kernel B. Observe that the slowdown varies significantly across LC kernels or even the co-runs of the same LC kernel with different BE kernels. Fig. 6 reports the overall throughput improvement (defined in Section 5) and demonstrates the difficulty of predicting the best configuration for the co-runs.

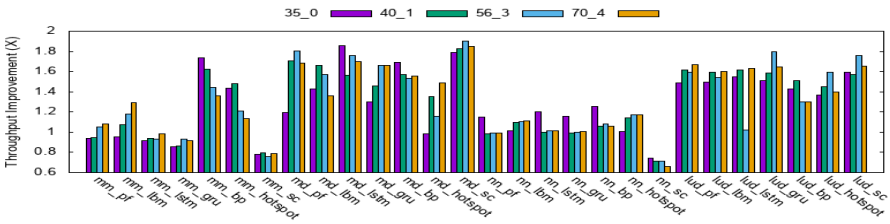

Fig. 6: Overall throughput improvement under different configurations

4 FLARE

4.1 System Overview

The goal of FLARE is to enable flexible sharing between LC and BE applications and optimize resource partitioning to enforce QoS as well as maximize overall throughput of co-run pairs. FLARE addresses the trade-off between latency and throughput based on offline and online dynamic search algorithms to quickly figure out an optimized co-running configuration. The system, as shown in Fig. 7,

consists of the following three components:

Kernel Transformation FLARE transforms the BE application to allow it to yield an arbitrary number of thread blocks on each SM. Note that we assume no access to the source code of the kernels of LC applications submitted by users, but the LC kernels can automatically use the yielded resources thanks to Nvidia's support of concurrent kernel executions.

Initial Configuration Selection To address the problem of unavailable LC applications for offline profiling, FLARE co-runs pairs of diverse microbenchmarks with many resource sharing configurations to characterize the performance degradation space. Based on the characterization, when the LC application arrives, FLARE only profiles its kernel invocation once to quickly model the performance degradation for both the LC and BE applications. FLARE then selects an initial configuration to spatially co-run the applications.

Online Refinement During the co-running, FLARE collects the performance degradation timing data as feedback to dynamically adapt the next configuration to use. By using the co-run degradation data of microbenchmarks, FLARE intelligently skips configurations and quickly reaches the optimal configuration to use for spatial co-running.

4.2 Kernel Transformation: Enabling Spatial Sharing

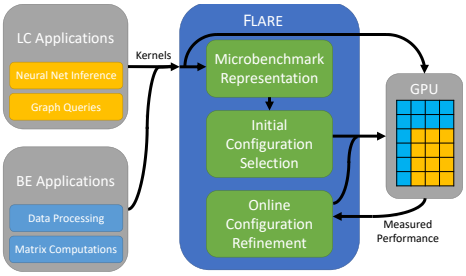

Fig. 7: The FLARE System

Kernel transformation enables the BE application to yield resources when a LC application is scheduled to the same GPU. The design, inspired by SM-centric transformation [30] and FLEP [31], runs just enough thread blocks to occupy the whole GPU. Specifically, given that a GPU has N SMs and each SM runs up to K thread blocks, FLARE schedules $N \times K$ thread blocks, each running the algorithm described in Fig. 8a. Every thread block first invokes *get_sm_id*() to obtain the ID of the host SM and then *atomic_get_blk_id*() to get its unique block ID on that SM, starting from 0. Each thread block stays in a while loop as long as there are tasks left to execute. At the beginning of each iteration, each thread block gets a unique ID. If a thread block ID is larger than the specified value $num\_blks[sm\_id]$, which is set by CPU, it means that the thread block needs to be yielded. To control the spinning overhead, we follow the approached proposed in [31] to control the granularity of the tasks. Once the resource of the yielded thread blocks is released, the kernel of the LC application can acquire the resource and start co-running. After the LC application is finished, the BE application is notified and launches the same number of thread blocks as yielded to fully occupy the GPU again.

Although the algorithm enables arbitrary ways to yield thread blocks, it requires the CPU and GPU to share the array num_blks, containing num_SMs elements, which may incur non-trivial communication overhead when num_SMs is large for high-end GPUs. To address this problem, FLARE uses the algorithm shown in Fig. 8b to sacrifice flexibility for reduced overhead. In this new design, FLARE asks the BE kernel to yield the same number of thread blocks (i.e., k) on a subset of SMs (i.e., n). Since thread blocks of a kernel have similar behaviors and the SMs are homogeneous, we expect this simplified design to perform as well as the more flexible one.

```
//Run by each thread block of BE kernel
//Global array num_blks[num_SMs]
BE_Kernel(...) {
   sm_id = get_sm_id();
   blk_id = atomic_get_blk_id();
   while(task_queue is non-empty) {
      if( blk_id > num_blks[sm_id]) quit;
      else {
         task = pull_task();
         execute(task);
}}}
```

(a) Arbitrary thread block yielding.

```
//n: number of SMs to yield blocks
//k: number of blocks to yield
BE_Kernel(...) {
   sm_id = get_sm_id();
   blk_id = atomic_get_blk_id();
   while(task_queue is non-empty) {
      if( sm_id < n && blk_id > num_blks[sm_id]) quit;
      else {
         task = pull_task();
         execute(task);
}}}
```

(b) Less flexible thread block yielding to reduce overhead.

Fig. 8: Transformed BE kernels to allow spatial co-run.

4.3 Initial Configuration Selection: Microbenchmark Driven

Due to the large spectrum of LC applications, FLARE cannot exhaustively profile BE-LC co-run pairs to find the optimal configuration. Instead, FLARE estimates the performance of BE-LC co-runs using microbenchmarks. Each kernel is matched with microbenchmark configurations that best represent its solo-run profiling statistics. Designing microbenchmarks that represent real-world applications is not easy, because the performance of a kernel is affected by many factors and the importance of each factor varies for different kernels. But the relevant features should be those related to resources for which the kernels contend when co-running. Out of the 120 performance counters of NVIDIA's *nvprof* profiler, we select the following 7 metrics which reflect or affect resource contention: L1, L2 cache hit rate, DRAM, L2, and L1 bandwidth utilization, arithmetic intensity, and total number of instructions.

To produce microbenchmark programs (also called microbenchmark instances in this paper) with varied features, we design a parameterized kernel with two parts. Here a instance is a microbenchmark with unique values on the 7 metrics. The first part loads a 1-D array and the second part contains a loop that performs pure arithmetic operations on the loaded data in each iteration. The microbenchmarks use the following 4 parameters to sample the configuration space: *stride length* to specify the distance between memory accesses from adjacent threads and hence control spatial locality of each thread block, *overlap ratio* to specifying the overlap between working sets of adjacent warps, *iterations* to control arithmetic intensity by specifying the number of iterations of the loop and *the number of threads* to run in total.

With 9 different stride lengths from 0 to 128 elements (L2 cache line size), 5 different overlap ratios ranging from 0 to 0.2, iteration counts from 1 to 4, and a fixed number 160K of threads, there are 180 different configurations of the microbenchmark. We find that the range of these metrics covers most real world applications by tuning these parameters.

Running all pairs of these microbenchmarks on all possible co-run configurations gives a large input dataset for training models to get a sense of the patterns that arise. Given 180 different instances of the microbenchmark, we co-run each pair of them in all possible ways to spatially share the GPU, resulting in a total of 640×640 co-runs. Based on these results, FLARE has the following two methods to select the initial configuration.

Linear Regression The linear model has 16 features: 14 profile features from two microbenchmark instances and the SM configuration (i.e., n and k in Fig. 8b). The linear regression maps that 16-element vector onto the 1-dimensional output space describing either estimated latency or throughput. FLARE builds the linear model with these 16 features and trains the model using all the data from the offline co-runs through the least squares method. FLARE profiles one iteration of a solo-run of the BE kernel and the LC kernel (when it becomes available) to obtain the 14 characterization features and combine them with the other two features to get the feature vector of co-run applications. Then FLARE uses the linear regression models to estimate the co-run performance degradation given each of the co-run configurations, and finally selects the one that satisfies QoS and maximizes throughput. Since the linear models are quite lightweight, the initial configuration-selection based on this method has a trivial overhead.

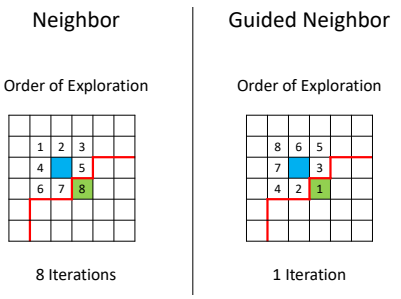

Fig. 9: Online search methods.

Nearest Neighbor Like the linear regression method, the nearest neighbor method also profiles one iteration of the BE and LC kernels to obtain their characterization features. For each kernel, it then searches for a profiling characterization feature vector among the microbenchmarks that is the most similar to that kernel in Euclidean distance. Specifically, each value of the 7-element feature vectors is normalized to the range (0,1), and this method searches for the nearest microbenchmark feature vector $\boldsymbol{b}$ to the feature vector of the real benchmark $\boldsymbol{m}$. The nearest neighbor method selects the microbenchmark for which the quantity $\|\boldsymbol{m} - \boldsymbol{b}\|_2$ is minimized. Each pair of representatives has offline-generated performance results on all of the co-run configurations available, so this gives another estimate of the performance degradation across the configuration space. Based on this, FLARE selects the configuration that satisfies QoS and maximizes

throughput. This method requires no training and incurs negligible runtime overhead.

4.4 Online Refinement: Dynamic Reconfiguration

The final configuration we select should be one with the highest possible throughput while still satisfying QoS. The initial configurations produced by the previously discussed methods are unlikely to match the globally optimal result every time. Therefore, configurations will need further refinement based on real performance feedback as shown in Fig. 7. FLARE starts at the initial configuration and gradually explores the neighborhood to finally reach the optimal configuration. FLARE includes two approaches to performing the search as follows.

Neighbor Search We demonstrate the idea of the first approach pictorially in Fig. 9 (left panel). The cells represent configurations. Towards the bottom left corner, the configurations give more resources to the LC kernel. The search process starts with an anchor cell (colored in blue), which should initially correspond to the configuration returned by the process described in Section 4.3. It then explores all the 8 neighbor cells (the numbers show the order of the exploration), and selects the best as the new anchor cell for the next round. Here the meaning of best is double-folded. When QoS is met, the best means that the overall throughput is optimal around neighbors. Otherwise, the best stands for the steepest decent of LC performance. This repeats until arriving at a cell where QoS is met and its throughput is the highest around its neighbors.

Guided Neighbor Search While the previous approach explores its neighborhood exhaustively, this approach searches first in the direction suggested by the microbenchmark data. Each neighbor cell has corresponding microbenchmark data, and therefore estimated QoS and throughput values associated with it. This gives some order to the neighbors in terms of their expected configuration performance, and we can simply explore the one with the best estimated performance. For example, Fig. 9 (right panel) shows that according to the microbenchmark data, the bottom right neighbor configuration (labeled by 1) should produce the highest performance for the LC kernel. We then explore that configuration and select it as the anchor for the next round. By leveraging the microbenchmark data, we substantially decrease the number of steps required to converge. This process continues until it reaches a configuration where the QoS requirement of LC is satisfied and microbenchmark throughput is maximized.

5 Evaluation

5.1 Experimental Setup

We evaluate FLARE using an NVIDIA TITAN V GPU with 12GB onboard memory hosted by a server with an Intel Xeon E3-1286 v3 CPU and 32GB main memory. The system runs Ubuntu 16.04 with kernel version 4.4.0-141, NVIDIA driver 410.48, and CUDA 10.0. We focus our evaluation on eleven benchmarks and two real-world applications. The benchmarks are from three

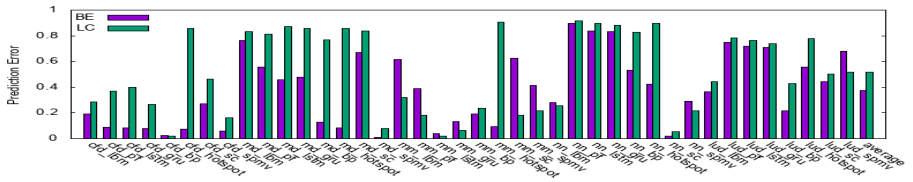

Fig. 10: Performance estimation error evaluation.

popular benchmark suites: Rodinia [5], SHOC [10], and NVIDIA's CUDA SDK. Two real applications, TC [1] and CN [23], represent deep learning inference workloads. TC uses an LSTM [12] model to classify documents and CN uses a GRU [9] model to predict the likely next character given an input string. Both of these inference applications heavily utilize the GPU, and are classified as LC applications. We also evaluate SPMV (SHOC), SC, PF, HOTSPOT, LBM and BP (Rodinia) as LC applications and MD (SHOC), MM (CUDA SDK), NN, LUD and CFD (Rodinia) as BE applications. Only the BE applications require adaptation to yield resources, while the LC applications can run unmodified.

5.2 Evaluation Strategy

We evaluate the performance of our approaches under the following scenario. The BE application runs continuously and consumes the entire GPU when no LC application is present. When a LC application arrives, the BE application yields part or all of the computation resources on the GPU to the LC application and two applications start to run simultaneously on the device. The LC application has a QoS deadline, and if its execution time exceeds this deadline the QoS is violated. As soon as the LC kernel completes, the BE application launches a kernel to reclaims all of the yielded hardware resources and resumes running exclusively. For this scenario, we always launch the BE kernel first and then start the LC kernel in a different CUDA stream. In order to compare with CUDA Multi-Process Service (MPS), we observe that simple SM-based partitioning gives the same performance results as MPS, and consider it as a possible configuration. Note that unlike FLARE, MPS does not support dynamic resource allocation. Once the application is launched, its allocation of the GPU resource cannot be changed. Thus, the MPS results in this section represent the best possible results MPS can produce.

In this paper, throughput refers to the number of instructions executed per microsecond. We define the overall throughput of a co-run pair as,

$$P^c = \left(INS_{LC} + INS^c_{BE}\right) / T^c_{LC} \tag{1}$$

where P^c is the overall throughput of co-run, INS_{LC} and INS^c_{BE} are the number of instructions of LC and BE applications during co-run, and T^c_{LC} is the performance of a co-run LC kernel. We are going to compare this overall throughput with the sequential throughput during T^c_{LC}. When a LC kernel arrives, the BE application will yield the GPU to the LC. Then the LC kernel starts to run and will be finished in T^s_{LC}. The BE resumes thereafter. But we only need to consider the number of BE instructions, INS^{tw}_{BE}, finished in time window $T^c_{LC} - T^s_{LC}$,

because our interest is to see throughput improvement of co-run. Therefore, the sequential throughput is given by

$$P^s = \left(INS_{LC} + INS_{BE}^{tw}\right) / T_{LC}^c \tag{2}$$

The ratio of P^c to the sequential throughput P^s gives us the throughput improvement.

We compare FLARE with the preemption-based approaches proposed in EffiSha [6] and FLEP [31]. Since the two approaches are similar, we only use FLEP as the baseline. FLARE proposes three ways to choose resource partitioning configurations: model-based, online search-based, and hybrid. The model-based approach incurs trivial runtime overhead but may choose a poor configuration where a QoS target could possibly be missed, while the online search-based approach may need to explore many configurations to find a desirable one. This evaluation will demonstrate that the flaws in these approaches prevent them from achieving the best performance. Section 3 notes that the performance of an application may not be linear in terms of allocated resource. Worse, the resource contention due to co-running makes it even more difficult to statically predict the optimal configuration. Since **NN** outperforms **Linear Regression** in all cases, we only show the results on the former. A hybrid approach uses the model-based approach to select an initial configuration followed by a online search approach to refine the configuration. FLARE supports two online search methods regardless of the initial configuration, namely neighbor search (**NS**) and guided neighbor search (**GNS**). It leads to two hybrid approaches: **NN_NS** and **NN_GNS**. Therefore, we evaluate 5 approaches included in FLARE: **NN**, **NS**, **GNS**, **NN_NS** and **NN_GNS**.

5.3 Results

Due to limited space, we only show the results for 1.5X QoS, that is, the co-run latency of a LC kernel cannot exceed 1.5X its solo-run time. Fig. 11 shows throughput improvement of FLARE with the best performing approach, **NN_GNS**, and binary search-based SM allocation with MPS. Observe that FLARE increases the average throughput improvement by 38.8% compared with FLEP. FLEP runs the LC application first to guarantee QoS and then the BE application after the LC application, thus missing co-running opportunities to improve throughput. As the figure shows, if MPS supports dynamic resource allocation, its performance could be close to FLARE. But FLARE still produces higher throughput because it not only considers SM allocation but also enables thread block allocation. We also measure the overheads of these 4 approaches. Fig. 12 shows the runtime overheads to find the configurations. The **NN** approach only needs to profile one iteration and then run a lightweight model. Hence it incurs negligible overheads. Observe that with the help of **NN** choosing an initial configuration, the hybrid approaches need substantially less time to find optimal configurations. The average iterations of the 4 algorithms are 48, 41, 28, and 24. With the guidance of microbenchmarks, the average overhead is about halved. For the

co-run pair NN_SC, the overheads of **NS** and **GNS** are 49 (**NN**) and 31 (**GNS**) iterations. These numbers are reduced to 9 and 14 using the microbenchmark guidance. The reason is that the initial configuration chosen by **NN** is closer to the optimal configuration of these benchmarks. This fact also indicates that our microbenchmarks capture crucial features of these benchmarks. It is important to point out that final chosen configurations by dynamic searching satisfy QoS, although the QoS may be violated along the way of the search process.

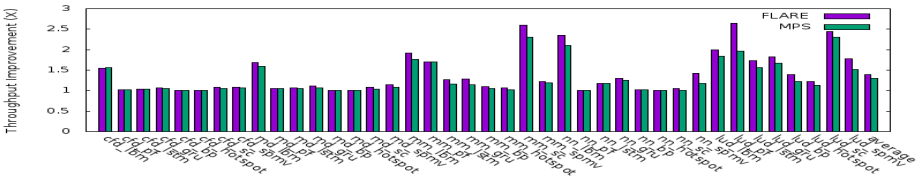

Fig. 11: Throughput Improvement at 1.5X QoS

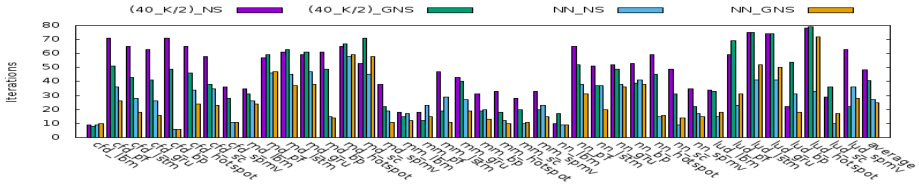

Fig. 12: Online searching overhead at 1.5X QoS

Micro-benchmark Prediction Error: Fig. 10 shows the relative error of the performance degradation of LC applications and the relative error of overall throughput predicted by the **NN** method for each of the BE and LC kernels. The relative error is defined as $|D' - D|/max(D, D')$. The results demonstrate the inefficiency of model-based approaches to predict performance degradation. For NN_PF, the prediction error for throughput and the LC degradation is 89% and 92%, respectively. They are 39% and 18% for MM_PF. The reason is that the applications have dramatically different properties, such as memory access pattern and branch divergence, which are difficult to accurately characterize using microbenchmarks. Fortunately, the microbenchmarks still capture important features relevant to co-running for a number of benchmarks. For instance, the prediction errors are as low as 3% (overall throughput) and 1% (LC latency) for CFD_BP. On average, the **NN** method produces 37% prediction error for the throughput and 51% error for LC latency. Therefore, it is reasonable to use the **NN** method to choose an initial configuration for online search. The NN_PF pair is an exception in all the pairs. No matter how the resources are allocated to PF, the degradation is 15X, which is why the prediction errors are so large.
Real Applications: Fig. 11 show the average throughput improvement across all the co-run pairs including real-world applications for different approaches. Fig. 12 illustrates overheads to find resource configurations for real-world applications. The pair enclosed in parentheses indicates the initial configuration

where MK is the number of maximum thread blocks an SM can host. Similar to the results on benchmarks, **NN** incurs minimum overhead. **NN** and **GNS** need a long search process evidenced by the substantial runtime overhead. **NN**, unfortunately, cannot find any configuration that satisfy QoS and hence results from the **NN** method are not included in these figures. Dynamic searching algorithms achieve the optimal overall throughput in all the 10 cases. The average throughput improvement of LSTM and GRU for 5 different pairs is 26% and 32%, respectively. The microbenchmark-based methods outperform **NN** and **GNS**. The overheads with microbenchmark guidance are about 60% of **NN** and **GNS**.

6 Related Work

Researchers have proposed architectural extensions to allow applications to co-run efficiently on the same GPU, with emphasis on cache sharing and bypassing [17], fine-grained sharing [29], preemption [21,26], dynamic resource management [22], and spatial multi-tasking [2,3]. The work [29] deals with spatial sharing through an enhanced scheduler (both thread block and warp level) to guarantee QoS. They use a quota to represent the QoS constraint. They assume that thread blocks are uniform in cost and the quota needs to reach zero at each epoch to satisfy QoS. To further improve the performance, they implement dynamic resource allocation by monitoring idle warps during each epoch. On the one hand, those techniques remain to be carefully evaluated for implementation in real GPUs. On the other hand, those studies do not systematically address reducing search overhead to find the best strategy for GPU sharing. Studies [16] have demonstrated that multi-tasking on GPUs can better utilize the hardware resource, but none of them predict performance degradation due to the co-running. Software systems, such as FLEP [31] and EffiSha [6], focus on lightweight preemption support but do not particularly study QoS enforcement. Baymax [8] and Prophet [7] predict GPU workload performance and use task re-ordering to handle QoS. Their approach to coordinate data transfers can be directly incorporated in FLARE to form a more general solution. Since they assume the GPUs are non-preemptable, they may use FLARE's methodology to further improve GPU utilization.

Another line of interesting work is practical GPU sharing in virtual environments, for which Hong et al. provide a comprehensive survey [14]. We briefly discuss several closely related studies. FairGV [13] achieves system-wide weighted fair sharing among GPU applications through collaborative scheduling and an accurate accounting mechanism. Gloop [25] proposes a new programming model to generate scheduling points in GPU kernels, which enables flexible suspending/resuming execution of GPU applications. Tian et al. propose a software system to virtualize Intel on-chip GPUs for graphics workloads [27]. None of these approaches have addressed fine-grained sharing or QoS of user-facing applications. To share the GPU memory GPUvm [24] partitions the GPU memory into regions and assign the regions to virtual machines. GPUswap [15] automatically coordinates GPU memory usage between applications even if the aggregate workload does not fit in GPU physical memory.

7 Conclusion

GPU sharing is a promising approach to improving hardware utilization, but resource contention may degrade the performance of the co-running latency-critical applications to violate QoS. In this paper, we demonstrated the complexities of partitioning GPU resources to enforce QoS and maximize throughput. To address the challenges, we proposed a software system named FLARE to enable and configure spatial GPU sharing between latency-critical and best-effort applications through kernel transformation, micro-benchmark guided partitioning and online configuration search. The experiment results showed 39% improvement on the overall throughput on 11 benchmarks and 2 real-world applications over existing systems. In the future, we plan to extend FLARE to address scenarios in which multiple latency-critical applications share the same GPU.

Acknowledgement

We would like to thank Akihiro Hayashi (our shepherd) and the anonymous reviewers for their constructive comments. This project was supported in part by NSF grant CCF-1823005 and an NSF CAREER Award (CNS-1750760).

References

1. Text classification in tensorflow. `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/learn#text-classification`. 2017.
2. Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *HPCA, 2012*.
3. T. Allen, X. Feng, and R. Ge. Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs. In *IPDPS*, 2019.
4. Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.
5. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
6. Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of GPU. In Vivek Sarkar and Lawrence Rauchwerger, editors, *PPoPP*, pages 3–16. ACM, 2017.
7. Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. ASPLOS, 2017.
8. Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax : QoS awareness and increased utilization of non-preemptive accelerators in warehouse scale computers. ASPLOS, 2016.
9. Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

10. Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, 2010.
11. Wei Han, Daniel Mawhirter, Matthew Buland, and Bo Wu. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *PACT*, 2017.
12. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
13. Cheol-Ho Hong, Ivor T. A. Spence, and Dimitrios S. Nikolopoulos. Fairgv: Fair and fast GPU virtualization. *IEEE Trans. Parallel Distrib. Syst.*, 28(12):3472–3485, 2017.
14. Cheol-Ho Hong, Ivor T. A. Spence, and Dimitrios S. Nikolopoulos. GPU virtualization and scheduling methods: A comprehensive survey. *ACM Comput. Surv.*, 50(3):35:1–35:37, 2017.
15. Jens Kehne, Jonathan Metter, and Frank Bellosa. GPUswap: Enabling oversubscription of GPU memory through transparent swapping. In *VEE 2015*.
16. Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. Efficient GPU spatial-temporal multitasking. *IEEE Trans. Parallel Distrib. Syst.*, 26(3):748–760, 2015.
17. Yun Liang, Xiuhong Li, and Xiaolong Xie. Exploring cache bypassing and partitioning for multi-tasking on GPUs. In *ICCAD 2017*.
18. David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. ISCA, 2014.
19. David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. ISCA, 2015.
20. Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. MICRO, 2011.
21. Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. ASPLOS, 2015.
22. Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Dynamic resource management for efficient utilization of multitasking GPUs. In *ASPLOS, 2017*.
23. Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In *ICML 2011*.
24. Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: Why not virtualizing gpus at the hypervisor? In *USENIX ATC 2014*.
25. Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. Gloop: an event-driven runtime for consolidating GPGPU applications. In *SoCC 2017*.
26. Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. ISCA, 2014.
27. Kun Tian, Yaozu Dong, and David Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In *USENIX ATC*, 2014.
28. Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. PPoPP, 2015.
29. Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on GPUs. ISCA, 2017.
30. Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey S. Vetter. Enabling and exploiting flexible task assignment on GPU through sm-centric program transformations. In *ICS 2015*.

31. Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. FLEP: Enabling flexible and efficient preemption on GPUs. In *ASPLOS*, 2017.
32. Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *ICS, 2019*.
33. Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. SMiTe: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *MICRO 2014*.
34. Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. ASPLOS, 2016.