# G-Space: Sharing Commodity GPUs Spatially to Enforce QoS and Improve Utilization

## ABSTRACT

Graphics Processing Units (GPUs) ~~see~~ gain ever-increasing popularity in platforms from data centers to high-performance clusters to accelerate various workloads due to plenty of thread level parallelism (TLP) GPUs can support. A modern GPU has tens or even more than one hundred of cores manufactured on the chip. Although massive TLP can be provided by a GPU, an application's performance may not be proportional to the amount of resources being allocted to it. We observe that more computation resources could lead to slowdown for some applications. This fact indicates that oversubsribing is detrimental to the utilization of resources. This underutilization issues is exacerbated on clusters at a data center. Multitasking is the key to solve the underutilization issue. It is quite chanllenging to share a GPU in a fine-grained way among different applications in a software system. ~~Therefore, different applications may share the GPU to improve hardware utilization.~~ Recent studies show that spatial-temporal sharing is more efficient than spatial or temporal sharing alone on GPUs. However, due to resource contention, latency-critical applications may suffer too much performance degradation to meet Quality-of-Service (QoS) targets. In this paper, we propose a software system, G-Space, to spatially and temporally share commodity GPUs between latency-critical applications and best-effort applications to enforce QoS as well as maximize ~~overall~~ throughput of best-effort applications. By transforming the kernels of best-effort applications, G-Space enables both streaming multi-processor (SM) partitioning and thread block partitioning within an SM for co-running applications. It uses a micro-benchmark guided static configuration search combined with online dynamic search to locate the optimal (near-optimal) strategy to partition resources. Evaluated on 10 benchmarks and 3 real-world applications, G-Space improves hardware utilization by up to 1.9× compared to existing approaches of yielding the whole GPU for LC applications while enforcing QoS restrictions.

## 1. INTRODUCTION

Datacenters are gaining increasing popularity as ~~the infrastructure sharing~~ they ~~enable~~ are able to significantly reduces 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 (TCO) [3]. The problem is exacerbated by the wide adoption of expensive high-end GPUs to leverage the massive parallelism for acceleration of various types of workloads, such as deep neural networks [8] and graph analytics [37]. Unfortunately, while CPU utilization in servers is already low (ranging from 10% to 70% [11, 24]), GPU underutilization is more severe due to the complex dynamic behaviors of GPU applications [6, 7].

A fundamental cause of the hardware utilization problem ~~is the strict quality-of-service requirements of latency-critical (LC) applications (e.g., web services)~~ results from scalabilities of some applications whose performance is bounded by memory operations. To meet the QoS target of an application, a conservative scheduler ~~may~~ usually will reserve ~~a~~ the whole server for ~~only LC requests~~ the LC application. Generally speaking, the performance of the LC application may not be beneficial from utilizing the whole computation unit. Conservative reservation statergy most likely leads to resource underutilization as shown in Fig. TODO: . A promising solution is multitasking, that is, ~~to co-locate~~ simultaneously co-locating a best-effort (BE) application together with the LC application to share ~~the same server~~ a GPU device. ~~However,~~ Since a BE application is going to share both memory and computation resouces on the same device the performance of a LC application will be interfered by the BE application ~~may interfere with the LC application,~~ In many cases, multitasking could ~~resulting~~ result 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 10× [6, 40].

Many efforts ~~seek~~ have been devoted to predict whether co-locating a pair of BE and LC applications violates QoS [26, 44, 42, 31]. A common theme is to characterize ~~the~~ resource contentions among applications, based on which a runtime builds a model for performance degradation prediction. The global scheduler ~~only~~ ~~identifies safe co-locating~~ co-locates applications if the prediction shows that doing so would not violate the QoS requirement. Another line of works proposes controllers to dynamically partition resources between co-running applications to meet the QoS goal [46, 25, 12]. Although existing multitasking research on CPU ~~considers~~ well studied contention on various shared resources, including memory bandwidth, CPU cores, and network bandwidth, ~~but~~ the findings do not directly apply to GPUs because of ~~their~~ unique architectural features of GPUs (details in Section 2).

As far as we know, Baymax [7] and Prophet [6] are the only two software systems that enforce QoS for shared GPU systems. They leverage performance models for both kernel execution and data transfers to coordinate GPU requests from co-running applications. But both studies assume that the GPU is a non-preemptable processor and hence a long-running kernel could possibly reserve~~s~~ all the GPU computational resources and a time sensitive kernel has a long wait till it get served by the 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 and threads in an SM. Recent research [5, 40] shows the possibility of using software transformation to support preemption on a GPU, which inspires us to revisit the assumption and exploit the new opportunity to address the hidden GPU underutilization problem.

In this paper, we aim at improving GPU utilization by spatially partitioning the abundant resource between co-running BE and LC applications. Instead of only coordinating GPU kernel executions, we allow ~~the~~ a BE application to yield just enough resource to the LC kernel so that~~meet~~ the QoS target of the LC kernel is satisfied. To achieve this goal, we face multiple challenges. ~~First~~ First of all, while one only needs to consider a 1-D resource space for CPU core allocation [26, 25], ~~the~~ a GPU has ~~many~~ tens of SMs and there could be several groups of threads (i.e. cocurrently thread array) running on each SM concurrently. Thus, it is a resource allocation problem in 2-D space. ~~Second~~ Secondly, since ~~the~~ a GPU by default runs the launched kernels in a FIFO manner, a kernel from ~~the~~ a BE application may use up all the SMs on the device, ~~so LC appliations have to wait in the Queue until the BE application is done~~ . ~~Hence the BE application blocking blocks the kernel of the LC application from running.~~ In such circumstances, the QoS goal of the LC application probably will be violated. ~~We need~~ Therefore, it is necessary to design a software mechanism ~~to~~ that enable the two kernels to run simultaneously ~~run~~ on different parts of the GPU. ~~Third~~ Thirdly, the co-running kernels will content for ~~interfere with each other on~~ a variety of hardware resources, including shared interconnect, L1 cache, L2 cache, streaming cores, and device memory. Such resource contention leads to performance interference between two applications. ~~Therefore~~ However, quantifying the performance degradation given a partitioning configuration is difficult. Finally, we try to enforce the QoS of one application and maximize ~~utilization~~ the throughput of the other simultaenously, which are two ~~conflicting goals~~ contradictory conditions. Specifically, by allocating more resources to the LC application, we have a better chance to meet the QoS goal but it will ~~decrease~~ hurt the throughput of the BE application and the ~~overall~~ resource utilization.

To overcome the challenges and improve utilization of *commodity* GPUs, we design and implement a software system, G-Space, which enables flexible spatial GPU sharing, meets QoS goals for LC applications, and maximizes throughput for BE applications. G-Space transforms the kernel of the BE application ~~to~~ such that the BE application is ~~be~~ able to yield a specific number of thread blocks on a subset of SMs. G-Space is more flexsible on resource partition than other systems mentioned above because it supports both inter and intra SM thread block parition. The threads of the LC kernel can then be scheduled ~~to run on the released hardware resource~~ on the released hardware resources by the BE application. The key novelty of G-Space 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 resource configuration [7, 44], but its accuracy ~~may suffer~~ suffers from input sensitivity and complicated hardware contention. On the other hand, a pure dynamic approach (e.g., online profiling and adjusting [25, 46]) may not be responsive enough. Even Worsely, it ~~may~~ will explore ~~detrimental~~ unfriendly configurations that lead to either hampered QoS or hardware under-utilization. G-Space employs a hybrid methodology. It uses carefully designed micro benchmarks to characterize the ~~co-run~~ multitasking performance degradation space so that ~~given~~ with two co-running kernels given, it can quickly predicts an initial thread block configuration ~~to use~~ between two kernels. Then G-Space leverages the degradation space to dynamically search for the optimal configuration.

To the best of our knowledge, this is the first work that systematically studies spatial computational resource sharing to enforce QoS of a LC application and maximize ~~utilization~~ throughput of a BE application simultaneously when both of them are corunning on a commodity ~~GPUs~~ GPU. We make the following primary contributions:

- We propose a code transformation technique which allows BE and LC applications to both temporally and spatially share a GPU in ~~arbitrary configurations~~ a find-grained fashion, thread block level partition.
- We propose a micro-benchmark based approach to quickly ~~identifying~~ identify a good configuration for GPU sharing once the BE and LC applications are ~~ready~~ provided.
- We design and implement ~~effective~~ efficient online configuration ~~search~~ searching ~~methods~~ algorithms to find the optimal configuration for ~~spatial GPU sharing~~ a corun pair sharing GPU spatially.
- We ~~implement~~ integrate the proposed techniques in a software system, G-Space, which substantially improves hardware utilization and enforces QoS requirements. ~~shown~~ This improvement is justified by experiments on 10 benchmarks and 3 real-world applications on a commodity GPU.

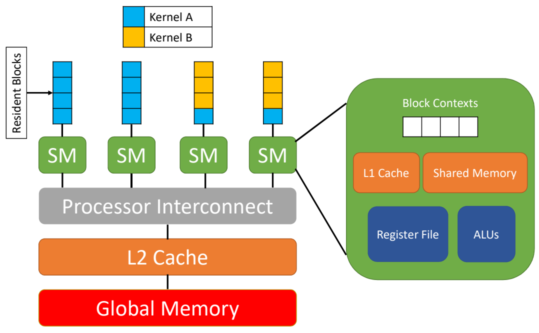

Figure 1: Graphics Processing Unit (GPU)

## 2. BACKGROUND

### 2.1 GPU Architecture and Resource Sharing

Driven by 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 ~~known as~~ called Streaming Multiprocessors (SM), each of which can simultaneously host multiple active thread block (~~sometimes~~ also known as Cooperative Thread Arrays) 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 32 threads (called a warp) at a time.

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~~ provided a solution to support concurrent kernel execution on their GPUs. This technique is named as Hyper-Q. The benefit of Hyper-Q is that it partially removes false dependency of kernels. Later, NVIDIA introduced the Multi-Process Service MPS, which enables kernels from different applications to ~~share the same GPU~~ be executed simultaneouly on one GPU deive. However, due to ~~huge~~ 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 ~~occupy the whole~~ consume all the compuation resources on a GPU. ~~, which happens in two cases~~ This situation happens in the following two cases.

- The earlier launched kernel only subscribes a small number of thread blocks. So other kernels could be scheduled on the GPU by using the leftover compuation resource.
- The earlier launched kernel is about to finish its execution. As seen in Figure 1, the overall resource requirement of kernel A's remaining thread blocks is so small that the greedy scheduler dispatches kernel B's thread blocks on the device such that kernel B's thread blocks will co-run with those of kernel A on the last two SMs. Due to the organization of the hardware, Figure 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.

### 2.2 QoS Issues in Co-Located GPU Applications

Most cloud service platforms nowadays employ GPUs because of their high throughput and low cost. But as pointed out by Chen et al. [7], the GPUs may experience very low utilization due to the diurnal pattern and the dynamic behaviors of GPU applications. A natural solution similar to what exists in CPU-based platforms is ~~to co-locate applications to share the same GPU~~ multitasking where multiapplications share the hardware resources. There are two alternatives for multitasking on GPUs, spatial and temporal sharing. While the co-location indeed improves hardware utilization, uncoordinated co-running applications may introduce~~s~~ severe performance interference to othersand such interference will leads to violat~~es~~ion of the QoS requirement of user-facing applications.

Baymax models the interference on data transfer between the CPU and the GPU and the contention on GPU computational resources [7]. Based on a runtime system, it carefully manages data transfers and reschedules kernel executions to mitigate the QoS violation issue. However, Baymax assumes the GPU cannot be preempted, so once a kernel is launched, it may monopolize the whole GPU for a long time, leading to unacceptable performance degradation for the waiting kernels. In this paper, we focus on the complicated problem of GPU core sharing between co-running GPU applications and assume that the contention on data transfer is handled by the model proposed in Baymax.

## 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 35 pairs of kernels on an NVIDIA ~~Pascal~~ Volta GPU (see details of the kernels and GPU specifications in Section 5). In the paper, We assume that Hyper-Q is used by default without mentioning it explicitly. Figure 2 shows the performance degradation of the LC kernels when they are immediately launched after the BE kernels. Observe that QoS is violated for ~~23~~ all the pairs of the co-runs ~~when~~ if the QoS target is ~~4~~ set less than 10 times of the corresponding solo-run execution time. It is because ~~the~~ a LC kernel has to wait for the BE kernel to ~~finish~~

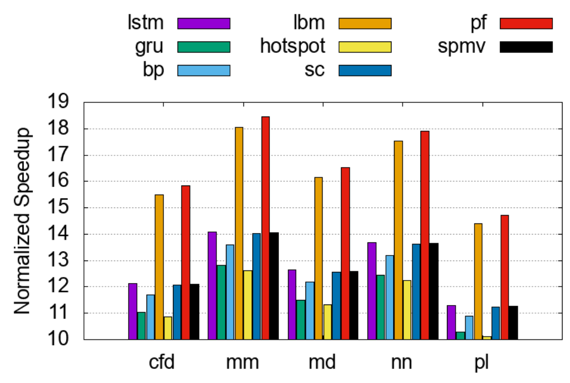

Figure 2: QoS violation for co-runs when the GPU is unpreemptable.

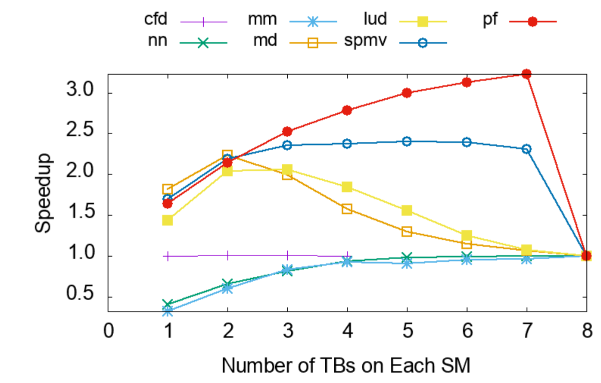

Figure 4: Solo-Run Scalability with respect to the number of thread blocks on each SM

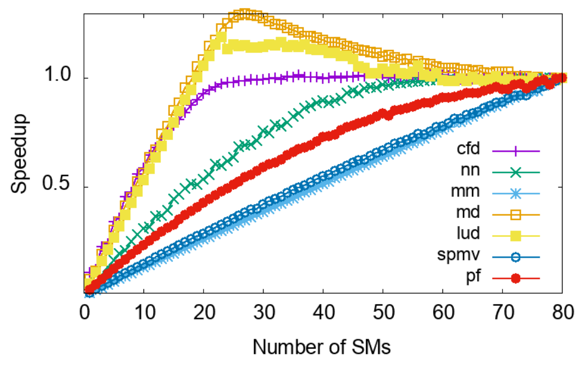

Figure 3: Solo-Run Scalability with respect to the number of SMs

~~execution~~ be completed. ~~Note that rescheduling kernels, no matter how intelligent the algorithm is, cannot completely solve this problem.~~ This phenomena reflects the fact that rescheduling strategy, no mattere how intelligent the algorithm is, cannot completely solve the problem of concern. Since it is ~~hard~~ already quite difficult, if not impossible, for us to predict the arrival time of the LC kernels, launching a BE kernel without any control on its resource usage may ~~render the whole GPU unusable for LC applications.~~ prevent LC applications from being executed because all the resources has been claimed by the BE kernel. According to the scheduling policy, LC applications have to wait until the BE kernel is done. The QoS target is most likely violated in this case.

### 3.2 Scalability Issues of Preemption-Based Solutions

Recent work, such as FLEP [40] and Effisha [5], has proposed low-overhead software-based mechanisms to ~~preempt~~ 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 ~~then the~~ LC kernels monopolizes all the available resources regardless of how efficiently it ~~can~~ will utilize them. When all of the running thread blocks come from the same kernel, they tend to have the same performance characteristics and bottlenecks. Such a homogeneous environment causes contention on the bottleneck resources leading to poor scalability. ~~We leverage the idea of SM-centric transformation [39], which flexibly controls how many SMs to use and how many thread blocks to run on each used SM, to modify the kernels (details in Section 4) from 4 benchmarks to run on a real GPU with 20 SMs to understand how serious the issue is.~~ We adopt the idea of SM-centric transformation [39] to modify the kernels (details in Section 4). The benefit is that we have the ability of fine tuning the thread blocks being yielded to an LC kernel. We are able to control how many SMs to use and how many thread blocks to run on each used SM. ~~The results in Figure ?? demonstrate that while some kernels show near-linear performance scaling with increased numbers of SMs, some reach performance saturation with far less resources. For example, by using only 10 SMs, NN can reach around 90% of the performance of its full-GPU execution. The low-utilization problem is especially evident in the thread block plot, for which going beyond 50% occupancy gives almost no performance benefit for two benchmarks. Interestingly, while LBM experiences linear scalability in terms of SMs, it poorly utilizes intra-SM resources.~~ We show scalability of some benchmarks in Figures 3 and 4. These two figures tell us the following important things regarding to resource allocation.

1. Less resources could have positive speedup, that is, more resources don't guarantee better performance. In Figure 3, the performance of md and lud is linearly proportional to the amount of resources at the beginning. The speedup will be larger than 1 around 20 SMs and then the performance will start to decrease as more resources are given to the applications. The same phenomena occurs in the scalability with respect to the number of TBs on each SM as well.

2. The performance could be saturated when partial resouces are allocated. This fact is refected by the cfd and nn in Figure 3 and by nn and mm in Figure 4. The cfd kernel's performance is saturated when

20 SMs are allocated, while nn saturates around 50 SMs.

3. Given the same amount resources, the performance of an application could have significant difference for different configurations. Take the pf application as an expample. Assume that the application is given a half of the resources. The configuration (40, 8) will have speedup less than 1 while the configuration (80, 4) provides more than 2X speedup. Our notation is that the first component of the configuration tuple is the number of SM being used and the second one is the TBs on those SMs.

4. The scalability is case dependent. Obviously, it could be super-linear, linear and sub-linear based upon the observation on all the curves in two figures.

5. The hardware is generally underutilized. An application could possibly oversubscribe resources. The oversubsribing provides either no speedup or slowdown. The excessive resources should be freed up for better hardware utilization without hurting the performance.

## 3.3 Spatial Co-Running and its Challenges

The rescheduling approach and the preemption-based approach are just two extreme points in a rich design space of ~~spatially sharing the GPU~~ GPU spatial sharing between co-running applications. ~~When running~~ In order to run one pair of a BE and an LC applications simultaneously, we need a mechanism ~~to~~ for the BE application to ~~flexibly~~ be able to yield resources (entire SMs or thread block slots of SMs) to the LC application. ~~Through the reduced resource availability and introduced contention cause of slowdown for both kernels, we observe that only QoS matters for the LC applications and only throughput matters for the BE applications.~~ Although this mechanism will introduce performance interference between two applications, the objective of the two kernels are independent. We care about only the throughput of the BE kernel and the slowdown of the LC kernel. Therefore, such a mechanism allows one to produce a better trade-off between QoS guarantees and overall GPU utilization. This leaves the following goals for this paper:

- Better saturate GPU resources by spatially co-running two kernels rather than one
- Keep the run time of the Latency Critical (LC) kernel under its QoS deadline
- Maximize throughput for Best Effort (BE) kernel subject to latency constraint

To show the complexity of the interference due to spatial co-run, we show the performance of LC kernels and the throughput of BE kernels with 280 TBs allocated to LC kernels in Figures 5 and 6. The notation used in the paper is the following. The first component in the corun pair is BE application, while the second is LC kernel. The first number and the second one in SM configuration is the number of SM being yield to LC kernel and the number of TBs on these SMs, respectively. These two figures shows that different pairs have different preference toward resource allocation. The resource allocation becomes even more complicated when the QoS and throughput are taken into account. For example, the pair nn_pf preference inter-SM sharing while the throughput favors intra-SM sharing. The pairs mm_blm and nn_lbm vote for intra-SM sharing for both QoS and throughput. Mm_pf likes intra-SM for throughput but inter-SM for QoS performance. Note that there is nothing specail for 280 TBs for LC kernel. The same phenomena could be observed for difference total number of TBs. The rule of thumb is that a number should have as many as possible divisors between 1 and 8 to see such difference. Therefore, different pair prefers different resource allocation to maximize the throughput even if the QoS is satisfied. This is the focus of our paper.

In summary, to achieve the goals, this paper addresses the following challenges.

- Estimate the co-running performance of kernels without extensive benchmarking
- Search for the configuration whose throughput is maximized. In the meantime, QoS requirement is satisfied.
- Minimize the time taken to complete the configuration search

## 4. G-SPACE

In this section we introduce G-Space, a system to enable spatial sharing of GPUs between Latency-Critical and Best-Effort applications. Spatial sharing among applications helps to improve GPU resource utilization. ~~, which dynamically selects configurations to improve resource utilization.~~ Furthermore, G-Space dynamically explores the configuration space and searches for the best configuration, which satisfies the QoS requirment and where the throughput of BE is maximized.

### 4.1 Goals

The primary design goal of G-Space is to maximize the throughput of BE applications subject to QoS constraints ~~from~~ on LC applications. Datacenters often co-locate applications on shared servers to improve resource utilization. But GPUs lack the fine-grained context switching and preemption features of CPUs, making sharing them more difficult. On shared GPUs, G-Space improves hardware utilization by safely sharing the GPU between BE and LC applications.

G-Space is designed to consider the trade-offs between latency and throughput caused by different co-run configurations. The system must configure co-runs to maximize the available throughput while respecting QoS requirements. This needs to be done quickly, with

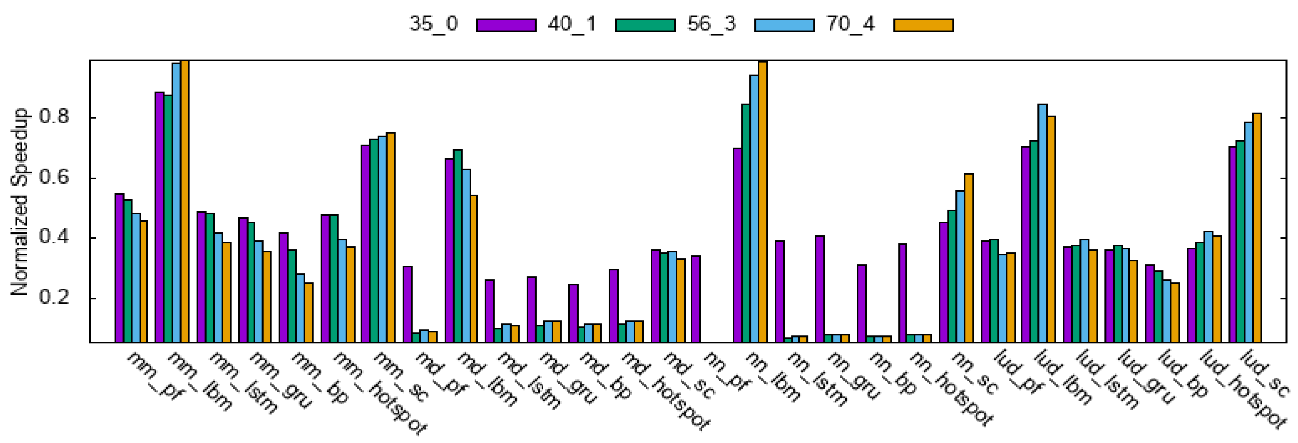

Figure 5: QoS Kernel Speedup with 280 TBs Being Allocated

as few QoS violations along the way as possible. Resource allocation occurs at the kernel level, so the dynamic reconfiguration can only ~~occur~~ happen between runs.

## 4.2 G-Space Overview

The goal of G-Space is to enable spatial sharing between LC and BE applications on ~~the~~ GPU and partition the resources on the GPU between them to enforce QoS as well as maximize ~~overall~~ the throughput of BE application. G-Space is designed to address the trade-off between latency and throughput when co-running GPU applications. In order to accomplish this, the system as shown in Figure 7 consists of three components: Kernel Transformation, Initial Configuration Selection, and Online Refinement.

***Kernel Transformation*** G-Space transforms the BE application to allow it to yield an arbitrary number of thread blocks on ~~each~~ each SM. Note that we assume there is no access to the kernels of LC applications submitted by users, so G-Space does not transform the kernels of LC applications.

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

***Online Refinement*** It is pointed out in the paper [41] that relationship between performance and resource is usually non-linear and non-convex for some cases. This would make building models to predict resource allocation inefficiently. Therefore, dynamical resource allocation is prefered. During the co-running, G-Space collects the performance degradation timing data as feedback to dynamically adapt the next configuration

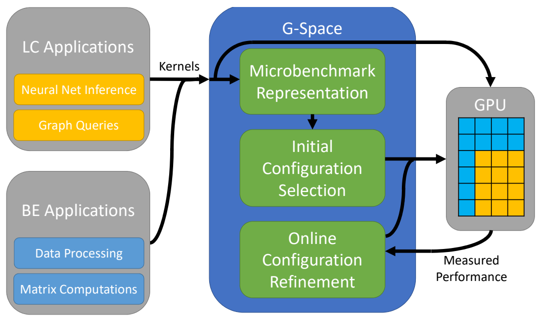

Figure 7: The G-Space System

to use. By using the co-run degradation data of micro-benchmarks, G-Space intelligently skips configurations and quickly reaches the optimal configuration to use for spatial co-running.

## 4.3 Kernel Transformation: Enabling Spatial Sharing

**Algorithm 1:** Arbitrary thread block yielding.

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

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 [39] and FLEP [40], runs ~~just enough~~ ex-

**Algorithm 2:** Less exible thread block yielding to reduce overhead.

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

act amount of thread blocks to fully occupy the ~~whole~~ GPU. Specifically, Given that a GPU has $N$ SMs and each SM runs up to $K$ thread blocks, G-Space schedules $N \times K$ thread blocks, each of them running the algorithm described in Figure 0. The default size of a thread block is 256 to hide memory access latency. Every thread block first invokes *get_sm_id*() to obtain the ID of the host SM and then calls *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 be executeid. At the beginning of each iteration, the thread block checks whether its ID is larger than the specified number of thread blocks, *num_blks*[*sm_id*], whose initial value is $K$ and set to a different value by the CPU to inform the GPU to yield thread blocks. Once the resource of the yielded thread blocks is released, the kernel of the LC application can acquire the resource and start to co-run~~ning~~ with the BE kernel.

Although the algorithm is quite flexible and 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, G-Space uses the algorithm shown in Figure 0 to sacrifice flexibility for reduced overhead. In this new design, G-Space 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 and tasks of the same kernel have similar behaviors and the SMs are homogeneous, we expect this simplified design to perform as well as the more flexible one.

The result of this mechanism is the following workflow:

1. BE application ~~occupies the full GPU~~ claims all the resources on a GPU using exactly the same thread blocks supported on the GPU
2. LC application arrives, BE kernel yields resources
3. LC kernel greedily uses all yielded resources
4. BE application reclaims resources once LC kernel completes

## 4.4 Initial Configuration Selection: Micro-benchmark Driven

Due to the unavailability of LC applications offline, G-Space cannot exhaustively profile BE-LC co-run pairs to find the optimal configuration. Instead, G-Space estimates the performance of BE-LS co-runs using micro-benchmarks. Micro-benchmarks play two important roles in G-Space. Firstly, micro-benchmarks provide an estimated performance degradation of real world applications. Secondly, they serve as maps or dictionary during dynamical search for the best configuration. Each kernel is ~~matched~~ proximated with micro-benchmark configurations that best represent its solo-run profiling statistics. Designing micro-benchmarks to represent real-world applications is not easy, because kernels can have wildly different performance properties. 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 and remarkably affect the performance of applications.

- L1 and L2 Cache Hit Rate
- DRAM, L2, and L1 Bandwidth Utilization
- Arithmetic intensity
- Total Number of Instructions

To produce micro-benchmark instances with varied features, we design a parameterized kernel with two parts. The first part loads a 1-D array and the second part contains a loop that performs arithmetic operations on the loaded data in each iteration. Note that it is quite difficult to decouple the parameters used in micro-benchmarks to simulate metrics related to $L1$ and $L2$ cache. Hence, the micro-benchmark ~~has 4 parameters in total:~~ use the following 4 parameters to sample the configuration space.

- Stride Length – specifies the distance between memory accesses from adjacent threads and hence control spatial locality of each thread block
- Overlap – specifies the overlap between working sets of adjacent ~~thread blocks~~ warps
- Iterations – controls arithmetic intensity by specifying the number of iterations of the loop
- Threads – number of threads to run in total

With 9 different stride lengths, 5 different overlap ratios ranging from 0 to 0.2, iteration counts from 1 to 4, and a fixed number 160K of threads~~to 1 million~~ , there are 180 different configurations of the micro-benchmark. A stride length play the role of putting pressure on $L1$, $L2$, and DRAM throughput. The bigger the stride length, the more pressure on throughputs. The maximum pressure happens when the stride is equal to the L2 cache line size. As mentioned above, it is difficult to decouple the $L1$ and $L2$ hit rates. Therefore, we use overlap ratio between adjacent warps to characterize cache hit rates. We observe that the range we picked cover wide variaties of cahce hit rates through profiling. The range of $L1$ hit rate of microbenchmarks is $[4\%, 60\%]$ and $L1$ hit rate of real world application is from 0.22% to 64%. This rate can be further increased if we tune up the overalp ratio. The current value is large enought for our experiments. Microbenchmark $L1$ hit rate almost cover the range of real world application. The profiling data also shows that the stride length also affects the hit rates. The number of iterations varies the number of instructions and arithmetic intensity. We find that mirobenchmark arithmetic intenty falls between $[50\%, 98\%]$. The minimum value of arithmetic intensity is 50% where all the intructions in the kernel are memory operation. Although the theoretical maximum value is 100%, this kind of application is useless. Our microbenchmark maximum value covers most real world applications. Each one of these was run under *nvprof* to characterize its performance according to the 7 parameters above. Running all pairs of ~~this~~ these micro-benchmarks on all possible co-run configurations gives a large input dataset for training models to get a sense of the patterns that arise. We evaluate two low-latency techniques for utilizing this dataset to predict performance degradation ~~in~~ of application kernels.

**Algorithm 3:** Microbenchmark pseudocode

```
function MICROBENCH(data, stride, overlap, arithmetic)
    get_block_id();
    get_warp_id();
    calculate_warp_offset();
    i = 0;
    while i less than arithmetic do
        pure_arithmetic_operations;
        i++;
```

Given 180 different instances of the micro-benchmark, we co-run each pair of them in all possible ways to spatially share the GPU, resulting in a total of $640 \times 640$ co-runs. Note that the total is not half of that, because we need to use each instance to represent a BE workload and a LC workload. Each co-run has two sets of characterization features for the two micro-benchmark instances, respectively, a co-run configuration for resource partitioning, and the corresponding performance degradation data. Based on these results, G-Space has the following two methods to select the initial configuration.

***Linear Regression*** The linear regression method builds a pair of models to predict the performance degradation for each of the two co-running kernels. Both models have the same set of features: 14 profiled features from the two co-running micro-benchmark instances and the co-run configuration (i.e., $n$ and $k$ in Algorithm **??**). The model, therefore, concatenates the two 7-element feature vectors and 2-element configuration

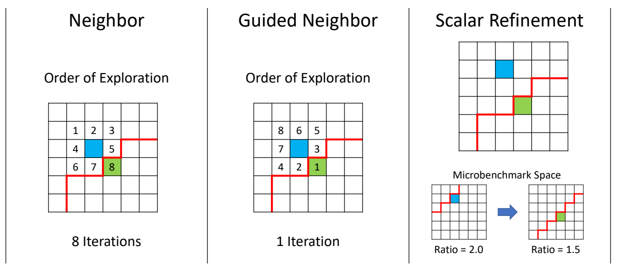

Figure 8: Online search methods.

vector into a single 16-element vector. The linear regression ~~them~~ maps that 16-element vector onto the 1-dimensional output space describing either estimated latency or throughput. G-Space uses all the data from the offline co-runs as training data to build the linear regression models (i.e., training the weights). After deployment, G-Space profiles one iteration of ~~the solo-GPU runs of the BE kernel~~ a solorun of the BE kernel and the LC kernel (when it becomes available) to obtain the 14 characterization features and ~~generates all possible co-run configurations to obtain~~ combine with the other two features to get the feature vector of corun applications. Then G-Space 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, initial configuration selection based on this method has trivial overhead.

***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 micro-benchmarks that is the most similar to that kernel in Euclidean distance. Specifically, the each value of the 7-element feature vectors are normalized to the range (0,1), and this method searches for the nearest micro-benchmark feature vector $\vec{m}$ to the feature vector ~~for~~ of the real benchmark $\vec{m}$. The ~~selected~~ nearest neighbor method selects the micro-benchmark for which the quantity $\|\vec{m} - \vec{b}\|_2$ 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 G-Space selects the configuration that satisfies QoS and maximizes throughput. This method requires no training and incurs negligible runtime overhead.

## 4.5 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 Figure 7. In other words, G-Space needs to try a subset of configurations (i.e., execute the co-running kernels given those configurations) to figure out the optimal one. When G-Space tries a configuration, we say that configuration is explored. Note that due to the smooth performance degradation shown in Figure **??**, it may seem that a binary search-based approach can minimize the number of explored configurations. However, it does not directly translate to minimized search time. For example, exploring the extreme configuration of allocating most resources to the BE kernel may lead to unacceptable QoS violation for the LC kernel. So not only should these approaches find the optimal configuration, they should do it quickly and with as few QoS violations as possible. To achieve this goal, G-Space starts at the initial configuration and gradually explores the neighborhood to finally reach the optimal configuration. G-Space includes three approaches to performing the search as follows.

***Neighbor Search*** We demonstrate the idea of the first approach pictorially in Figure 8 (left panel). The cells represent configurations. Towards the bottom left corner, the configurations give more resources to the LC kernel. Therefore, there should exist a continuous region of cells (included by the red line) that can satisfy the QoS requirement. 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 4.4. It then explores all the 8 neighbor cells (the numbers show the order of the exploration), which have not been tested before, and selects the best as the new anchor cell for the next round. This repeats until arriving at a cell which corresponds to a better configuration than any of its neighbors. The "best" neighbor configuration has different meanings when the anchor cell falls in the two different regions. In the regions of configuration that cannot satisfy the QoS, the best neighbor configuration is the one that minimizes the execution time of the LC kernel. Otherwise, the best neighbor configuration should minimize the execution time of the BE kernel. Therefore, when the initial configuration cannot satisfy the QoS, this approach tries to quickly enter the region that can satisfy the QoS and then maximize overall GPU utilization. This process works best on relatively low-noise data with a smooth performance degradation due to fewer allocated resources. It almost always eventually finds a configuration which satisfies QoS with better throughput than its neighbors, but may take many iterations to do so. To improve on this, it is important to maintain the same properties, but cut down on the total time taken.

***Guided Neighbor Search*** With access to representative data from the micro-benchmarks, the patterns of the micro-benchmark and real kernel performance should be similar, even if not identical. This affords the opportunity to give some direction to the naive approach of the basic neighbor search. While the previous approach explores its neighborhood exhaustively, this approach searches first in the direction suggested by the micro-benchmark data. Each neighbor cell has corresponding micro-benchmark 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, Figure 8 (middle panel) shows that according to the micro-benchmark 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 micro-benchmark data, we substantially decrease the number of steps required to converge, but have no guarantee of finding a higher quality configuration than the plain neighbor search, because the characterized performance degradation based on micro-benchmarks may not match that of the real applications. Hence, once the guided neighbor search terminates, we start a local refinement process to use the neighbor search process to improve QoS of the finally selected configuration.

***Scalar Refinement Search*** The neighbor searches described above perform well when the initial configuration is close to the optimal configuration. Additionally, the guided search requires that the performance degradation pattern of the micro-benchmark exactly matches that of the application kernels. While we observe that the patterns are similar, the rate of degradation can differ significantly from the true pattern. We propose a hypothesis: that the micro-benchmark and application kernel performance degradation differences are equivalent to a multiplicative scalar difference in the QoS target. The red lines in Figure **??** which separate configurations satisfying QoS from those that don't has an equivalent in the micro-benchmark space. The corresponding line in micro-benchmark space is typically substantially offset from the line that appears in that figure. For example, in the mm and lbm co-run there are 48 configurations which satisfy QoS with a QoS ratio of 2.0 (i.e., the QoS target is twice its solo execution time). Their representative micro-benchmark pair has 134 configurations which satisfy QoS for the same ratio, so the line is substantially offset toward the bottom-left. A similar line could be found by decreasing the QoS ratio used for the micro-benchmark. Scalar Refinement method can automatically perform this correction for the offset in an online fashion.

This approach also starts at the configuration that was optimal for the micro-benchmark. Based on the performance data for that single run, we can adjust the micro-benchmark QoS ratio to better fit that result. If the result fails to satisfy QoS, the ratio needs to decrease to make it more restrictive as shown in Figure 8 (right panel). If it satisfies QoS by a wide margin, the ratio should increase to allow for a more aggressive configuration. Recomputing the micro-benchmark optimal configuration may give a new configuration, which we explore in the next iteration. This continues until the

Table 1: System Setup

| | |
|---|---|
| CPU | 2S Intel Xeon E7-4830 v3 @ 2.1 GHz |
| GPU | NVIDIA TITAN V100 w/ 12GB GDDR5X |
| OS | Ubuntu 16.04 x64 with kernel 4.4.0-128 |
| CUDA | Driver 418.56 CUDA SDK 10.1 |

new and old configurations match, which happens very quickly. Figure 8 illustrates this process in terms of the line adjustment it implies. More precisely, scalar refinement adjusts the scalar $\gamma$ on the target QoS ratio $q$ for the micro-benchmark, such that the micro-benchmark search looks for configurations that satisfy a QoS ratio of $q \times \gamma$, with $\gamma$ initially 1. In subsequent iterations, if the real performance result exceeds the QoS target by a factor of $c$, then for the next iteration $\gamma$ will be decreased by an amount proportional to $c$. Conversely if the result over-achieves the QoS target by a factor of $c$, then $\gamma$ will be increased proportionally to $c$. The scalar $\gamma$, therefore, represents the increased or decreased restrictiveness of the QoS target assumed for the micro-benchmark to allow it to closely match the real boundary. We determine the proportional adjustments empirically to smooth the adjustment of $\gamma$ and maintain a small number of steps to convergence. While this method is not guaranteed to reach a configuration which satisfies QoS, it should be very close to configurations that do, and will likely have found a border configuration. So we also add a local refinement process the same as in guided neighbor search to enforce QoS of the selected configuration.

## 5. EVALUATION

### 5.1 Experimental Setup

We evaluate G-Space using an NVIDIA ~~GTX 1080~~ TITAN V100 GPU. The detailed setup is summarized in Table 1. ~~As listed in~~ Table 2 shows ~~we benchmark using~~ ten kernels and three real-world applicationswe use throughout the paper. The benchmarks are from three popular benchmark suites, Rodinia [4], SHOC [10], and NVIDIA's CUDA SDK, and cover various domains. Two real applications, TC and CN represent deep learning inference workloads. TC uses an LSTM [16] model to classify documents. CN uses a GRU [9] model to predict the likely next character given an input string. CC is a popular graph application to compute connected components. All of the three applications heavily use the GPU. The table also identifies which applications are tested as BE and LC applications shown by the "Role" column. Every co-run presented in this section is composed of one BE application and one LC application.

### 5.2 Compared Approaches

Section 3.1 shows that rescheduling-based methods may seriously violate QoS when the BE application has non-trivial workloads. Thus, we compare G-Space with preemption-based approaches proposed in EffiSha [5] and FLEP [40]. Since these two approaches are similar, we only use FLEP as a comparison point.

Table 2: Benchmarks

| App. | Source | Description | Role |
|---|---|---|---|
| CFD | Rodinia | finite volume solver | BE |
| MD | SHOC | molecular dynamics | BE |
| MM | CUDA SDK | dense matrix multiplication | BE |
| NN | Rodinia | nearest neighbor | BE |
| BP | Rodinia | backpropagation | LC |
| LBM | Rodinia | fluid dynamics | LC |
| HOTSPOT | Rodinia | thermo dynamics | LC |
| PF | Rodinia | dynamic programming | LC |
| SC | Rodinia | data mining | LC |
| SPMV | SHOC | sparse matrix multiplication | LC |
| TC | [1] | text classification | LC |
| CN | [32] | text generation | LC |
| CC | [15] | connected component | LC |

G-Space has three ways to choose co-run configurations: model based, online search based, and hybrid. The model-based approach incurs trivial runtime overhead but may choose a poor configuration, while the online search based approach may need to try many configurations to find a desirable one. G-Space supports both linear regression and nearest neighbor methods for model-based configuration selection. Both model based approaches have flaws. See section 4 for why model based approach does not provide a better solution. It is pointed out in section 3 that the performance of an application may not be linear in term of allocated resource. And the nearest search implicitly assumes that configuration space is flat and each dimension in the configuration space has equal weight. In reality, the configuration space is a non-trivial topological space. Due to space limitations, we evaluate the prediction accuracy of both methods but only use the nearest neighbor method in the model-based approach. A hybrid approach uses the model-based approach to select an initial configuration ~~and uses~~ followed by the online search based approach to refine the configuration, which has the potential to perform the best among all the three approaches. Since G-Space supports two online search methods regardless of the initial configuration, namely neighbor search (NS), and guided neighbor search (GNS), as well as scalar refinement search (SRS), we have 3 hybrid approaches: NN_NS, NN_GNS, and NN_SRS, where the nearest neighbor method selects the initial configuration, based on which the following dynamic search methods refine the configuration. SRS requires representative data from NN to run, so while it fundamentally differs from the neighbor methods, it cannot run alone. Therefore, in total we evaluate 6 approaches included in G-Space: NN, NS, GNS, NN_NS, NN_GNS, and NN_SRS. To evaluate NS and GNS, we allocate the whole GPU to the LC application as the initial configuration.

### 5.3 Evaluation Strategy

This section evaluates the performance of our approaches under the following scenario:

- The BE application runs continuously using the whole 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 for it to run depending on the evaluated approach.
- The LC application has a QoS deadline, and if the kernel exceeds this deadline the QoS is violated.
- As soon as the LC kernel completes, the BE kernel reclaims all of the hardware resources and resumes running exclusively.

In order to emulate this scenario we always run the BE application first and then start the LC application. A popular metric for measuring the co-running performance for BE and LC applications is simply the throughput of the BE kernel during the period where both kernels run simultaneously. Here we define the throughput of BE application as

$$P^c_{BE} = (T^c_{LC} - T^s_{LC})/T^c_{LC} \times P^s_{BE} \qquad (1)$$

where $P^{c(s)}_{BE}$ is the co-run or solo-run throughput of a BE application and $T^{c(s)}_{LC}$ is the co-run or solo-run performance of a LC kernel.

## 5.4 Micro-benchmark Based Performance Degradation Prediction

Figure 9 shows the relative error of the performance degradation predicted by the nearest neighbor method. Given the performance degradation of an application as $D$ and the predicted performance degradation due to a co-run as $D'$, the relative error is defined as $|D' - D|/max(D, D')$. The results demonstrate the challenge of relying on model-based approaches to predict performance degradation. For PL_SPMV, the prediction error for the BE application (i.e., PL) is 88%, and for MM_PF, the prediction error for the LC application (i.e., PF) is 73%. 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 micro-benchmarks. Fortunately, the micro-benchmarks still capture important features relevant to co-running for a number of benchmarks. For instance, the prediction errors are as low as 27% and 17% for the LBM_CFD co-run. On average, the NN method produces 35.7% prediction error for BE applications and 35.1% error for LC applications. Therefore, it is reasonable to use the NN method to choose an initial configuration for online search. Due to space limitations, we do not present detailed data for the linear regression method, which produces ~~average errors of 32.8% and 62% for BE and LC applications, respectively~~ much larger (about $2X$ to $5X$) error than nearest neighbor method.

## 5.5 Results on Benchmarks

Figure 10 shows the throughput of the final selected co-run configuration from all 7 approaches normalized to the throughput of the globally optimal configuration. Note that the selected configurations of all the approaches ~~except NN~~ satisfy the QoS requirement ~~thanks to the~~ after dynamic searchs. The NN-only method relies purely on the prediction and chooses configurations that achieve 59.3% of the optimal throughput. By leveraging online search, all the other approaches perform significantly better. Notably, NN_SRS exploits 83.6% of the throughput potential, and even the simple NS approach could achieve 75.1% of the optimal throughput.

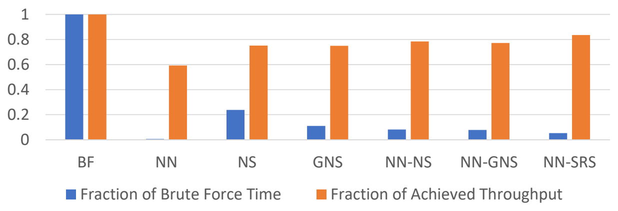

Figure 10: Achieved throughput and incurred runtime overhead.

Since all the approaches incur runtime overhead, we also measure the time needed for all approaches to choose the final configuration to use. Figure 10 shows the runtime overhead to find the co-run configuration normalized to the overhead of brute-force search. The NN approach only needs to profile one iteration and then running a lightweight model, hence incurring negligible overhead. Observe that with the help of NN to choose an initial configuration, the hybrid approaches need substantially less time to find the configuration compared to the online search approaches. For instance, NN_NS incurs 66% less overhead than NS and NN_GNS incurs 29% less overhead than gns. The reason is that the initial configuration chosen by NN is closer to the optimal configuration, which helps the neighbor searches to avoid local maxima in some cases. Although the final chosen configuration satisfies QoS as long as the approach uses dynamic search, the QoS may be violated during the search process.

Although the final chosen configuration satisfies QoS as long as the approach uses dynamic search, the QoS may be violated during the search process. ~~Figure ?? shows the fraction of iterations which satisfy QoS, where the remainder violate QoS due to choosing detrimental configurations. NN only explores one configuration, so it either satisfies the QoS (the bar is of height 1) or misses the QoS target (the bar is of height 0). We see that 11 out of 25 co-runs violate QoS, which makes the NN-only approach unacceptable. Also observe that the hybrid approaches satisfy QoS more frequently than the dynamic-only approaches because NN chooses a "good" initial configuration and avoids the exploration of configurations far from the optimal one. NN_SRS outperforms NN_NS and NN_GNS for most co-runs but explores 42% configurations that violate QoS during the search for co-runs involving CFD. Our investigation indicates that the micro-benchmark data fails to enable SRS to find a good configuration for local refinement. However, we note that the results do not necessarily mean the NN_SRS approach is worse than NN_NS or NN_GNS, because as previously shown NN_SRS converges~~

~~to the final configuration that satisfies the QoS requirement much faster than the other two approaches.~~

FLEP, as well as other preemption-based approaches, runs the LC application first to guarantee QoS and then the BE application in the remainder of the QoS window to improve throughput. We compare G-Space with NN_SRS to this approach to understand the improved hardware utilization from spatial sharing. Figure **??** shows the improvement of overall thoughput from G-Space over FLEP. G-Space produces higher performance for 16 of the 25 benchmark pairs. Higher throughput indicates that the benchmarks have poorer scalability and hence can spatially share the resource to improve overall utilization. In the other 9 cases G-Space chooses the same configuration as FLEP and hence provides the same performance. The results demonstrate the benefits from spatial co-runs enabled by G-Space, which is a key distinction from previous systems.

## 5.6 Results on Real Applications

We run the three real-world applications as LC applications paired with 5 benchmarks as BE applications on G-Space. Figure **??** shows the average throughput across all co-run pairs for different approaches to choose the co-run configuration that satisfies QoS, as well as the runtime overhead to find the configurations. Similar to the results on benchmarks, NN incurs minimum overhead while the pure dynamic approaches need a long search process evidenced by the substantial runtime overhead. NN, unfortunately, cannot find any configuration that satisfy QoS and hence its throughput is not included in the figure. NN_NS achieves the optimal throughput in all the cases, but it is ?? $\times$ slower to find the corresponding configurations than NN_SRS, which achieves 90.5% of the optimal throughput. Figure **??** shows the improved overall throughput over FLEP for 13 of the 15 co-runs. Compared to always preempting the whole GPU for the real applications, G-Space also enforces QoS but improves the hardware utilization by 30.8% on average.

## 6. RELATED WORK

GPU workload characterization has been studied for over a decade. Reference [14] proposed a set of characterizing a workload to evaluate the performance of a GPU microarchitecture. However, our concern is the impact of resource contentions on the performance of multiple applicationons. In G-Space, the performance affected by a specific GPU architecture is simulted by the data of those carefully designed microbenchmarks run offline.

Performance prediction and resource partitioning for co-located applications on CPU platforms have attracted significant attention. Bubble-Up [26], Octopus-Man [30], and SMiTe [44] use profiling data of the target applications to characterize performance degradation on real CPU systems, but the LC applications may be submitted by data center users and hence not available for detailed profiling and characterization. In comparison, we do not assume the availability of the LC applications and purely rely on micro benchmarks and online configuration search to guarantee QoS. Multiple systems use online profiling and adaptive resource partitioning to provide flexibility [46, 25], but unlike G-Space they do not leverage micro benchmarks to reduce search overhead. Moreover, researchers have proposed cache partitioning approaches to guaranteeing QoS of LC applications [19, 23, 43, 13] on CPUs. A linear model is assumed in SMiTe. However, our experiments have shown that linear model doesn't work well for GPU.

Researchers have proposed architectural extensions to allow ~~efficient application~~ applications to co-run~~s~~ efficiently on the same GPU. The emphasis was on cache sharing and bypassing [22], fine-grained sharing [38], preemption [28, 35], dynamic resource management [29], and spatial multi-tasking [2]. The work [38] deals with spatial sharing through enhanced scheduler (both thread block and warp level) to guarantee QoS. They use quota to represent the goal of QoS. They assume that thread block are uniform and quota needs to be zero at each epoch to satisfy QoS. To further improve the performance, they implement dynamical 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 address systematically reducing search overhead to find the best way for GPU sharing. Several studies [21, 45, 27] 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 [40] and EffiSha [5], focus on lightweight preemption support but do not particularly study QoS enforcement. Baymax [7] and Prophet [6] predict GPU workload performance and use task re-ordering to guarantee high QoS. Their approach to coordinating data transfers can be directly incorporated in G-Space to form a more general solution.Since they assume the GPUs are non-preemptable co-processors, they may use G-Space'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 [18]. We briefly discuss several closely related studies. FairGV [17] achieves system-wide weighted fair sharing among GPU applications through collaborative scheduling and an accurate accounting mechanism. Gloop [34] 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 workload [36]. None of these approaches have addressed fine-grained sharing or QoS of user-facing applications. To share the GPU memory between applications, GPUvm [33] partitions the GPU memory into regions and assign the regions to virtual machines. GPUswap [20] automatically coordinates GPU memory usage between applications even if the aggregate workload does not fit in the GPU physical memory. Although GPU virturalization solve the corun to a certain extent, the purpose of the virturalation is to separate GPU environment so that one application would not experience the existence of other applicaitons. Though G-Space is focused on the sharing of computational resources, there is a potential to integrate these approaches in G-Space into hypervisor of GPU virtualization to provide a more comprehensive system.

## 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 G-Space 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 substantial benefit in improving QoS and overall throughput on 10 benchmarks and 3 real-world applications over existing systems.

## 8. REFERENCES

[1] Text classification in tensorflow. `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/learn#text-classification`, 2017 (accessed March 3, 2018).

[2] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 79–90, 2012.

[3] 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.

[4] 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.

[5] 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, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 3–16. ACM, 2017.

[6] 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. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 17–32, New York, NY, USA, 2017. ACM.

[7] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax : Qos awareness and increased utilization of non-preemptive accelerators in warehouse scale computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[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] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[12] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[13] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sánchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 104–117, 2018.

[14] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10, Dec 2010.

[15] Wei Han, Daniel Mawhirter, Matthew Buland, and Bo Wu. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, Oregon, USA, September 9-13*, 2017.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[17] 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.

[18] 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.

[19] Harshad Kasture and Daniel Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 729–742, 2014.

[20] Jens Kehne, Jonathan Metter, and Frank Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*, pages 65–77, Istanbul, Turkey, March 14–15 2015.

[21] 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.

[22] Yun Liang, Xiuhong Li, and Xiaolong Xie. Exploring cache bypassing and partitioning for multi-tasking on gpus. In *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, pages 9–16, 2017.

[23] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA*, pages 367–378, 2008.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy

proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 301–312, Piscataway, NJ, USA, 2014. IEEE Press.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[26] 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. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[27] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013.

[28] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[29] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 527–540, 2017.

[30] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 246–258, 2015.

[31] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 62–75, New York, NY, USA, 2015. ACM.

[32] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 1017–1024, 2011.

[33] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[34] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. Gloop: an event-driven runtime for consolidating GPGPU applications. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, pages 80–93, 2017.

[35] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.

[36] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA, 2015. ACM.

[38] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 269–281, New York, NY, USA, 2017. ACM.

[39] 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 *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 119–130, 2015.

[40] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[41] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242, June 2016.

[42] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[43] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: a dynamic cache partitioning system using page coloring. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 381–392, 2014.

[44] 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 *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 406–418, 2014.

[45] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *CoRR*, abs/1303.5164, 2013.

[46] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

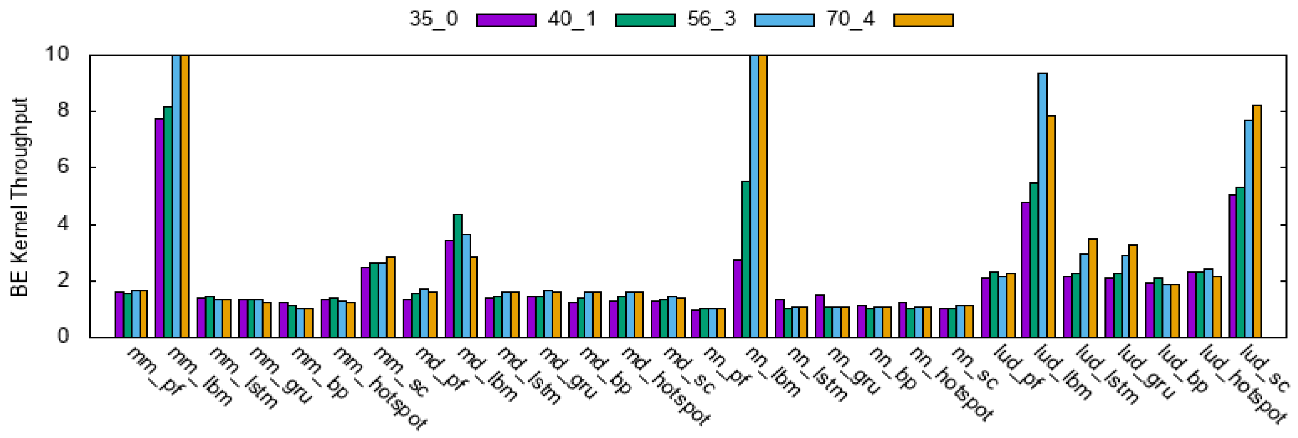

Figure 6: Throughput of BE applications with 280 TBs Allocated to LC applications

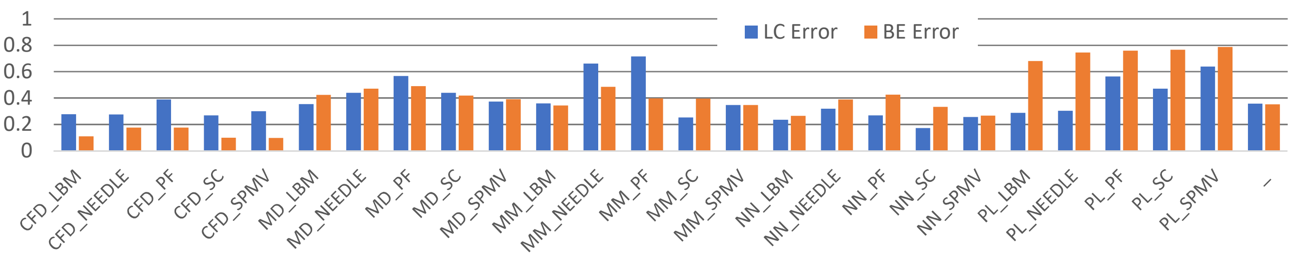

Figure 9: Performance degradation prediction error for NN.