2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00064

Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS

Han Zhao*, Weihao Cui*, Quan Chen*◇, Youtao Zhang†, Yanchao Lu‡, Chao Li*◇, Jingwen Leng*◇, Minyi Guo*◇
{zhaohan_miven,weihao,chen-quan,chaol,leng-jw,myguo}@sjtu.edu.cn, zhangyt@cs.pitt.edu, yanchaol@nvidia.com
*Shanghai Jiao Tong University ◇Shanghai Qi Zhi Institute †University of Pittsburgh ‡NVIDIA

***Abstract*—The proliferation of machine learning applications has promoted both CUDA Cores and Tensor Cores' integration to meet their acceleration demands. While studies have shown that co-locating multiple tasks on the same GPU can effectively improve system throughput and resource utilization, existing schemes focus on scheduling the resources of traditional CUDA Cores and thus lack the ability to exploit the parallelism between Tensor Cores and CUDA Cores.**

In this paper, we propose Tacker, a static kernel fusion and scheduling approach to improve GPU utilization of both types of cores while ensuring the QoS (Quality-of-Service) of co-located tasks. Tacker consists of a Tensor-CUDA Core kernel fuser, a duration predictor for fused kernels, and a runtime QoS-aware kernel manager. The kernel fuser enables the flexible fusion of kernels that use Tensor Cores and CUDA Cores, respectively. The duration predictor precisely predicts the duration of the fused kernels. Finally, the kernel manager invokes the fused kernel or the original kernel based on the QoS headroom of latency-critical tasks to improve the system throughput. Our experimental results show that Tacker improves the throughput of best-effort applications compared with state-of-the-art solutions by 18.6% on average, while ensuring the QoS of latency-critical tasks.

***Keywords*—Tensor Core, GPU Utilization, QoS.**

I. Introduction

GPUs have been widely adopted as a flexible acceleration solution for a wide range of modern applications, in particular, image processing applications [27], [47]. With fast technology advances, modern GPUs have become increasingly more powerful, which integrate a large number of CUDA Cores for improved parallelism, e.g., 4352 CUDA Cores in Nvidia RTX2080Ti. Furthermore, to meet the acceleration demands from proliferated AI/ML (artificial intelligence and machine learning) applications, recently released commodity GPUs, e.g., Nvidia Volta [9] and later architectures, integrated Tensor Cores in streaming multiprocessors (SM) for speeding up general matrix multiplication (GEMM), the main type of operations in AI/ML applications.

Given the abundant computing resources in modern GPUs, studies have proposed co-locating multiple applications onto the same GPU, which can effectively improve resource utilization and reduce system energy consumption, particularly for computing servers in data centers. Based on QoS (Quality-of-Service) demands, we may categorize two types of data center applications: latency-critical (LC) applications/services and best-effort (BE) applications. The former refers to those that have stringent latency constraints, e.g., to recognize interesting objects from a live video stream without glitches, the object detection algorithm needs to finish within 50ms [25], [41], [46]. The latter refers to those that have no or very loose constraints, e.g., to breadth-first search a node in a graph without setting a deadline. It is more cost-efficient to leverage the under-utilized GPU resources to run some BE applications, while guaranteeing the QoS in servicing an LC application.

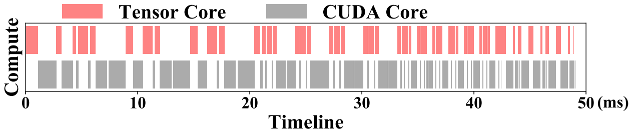


Fig. 1: The active timeline of Tensor Cores and CUDA Cores when Baymax is used to run *Resnet50* and *sgemm*.

To co-locate LC and BE applications, there are two types of strategies: non-preemptive methods and preemptive methods. Non-preemptive methods, e.g., Baymax [19] and Laius [59], launch the kernels from BE applications only when the latencies of LC applications are within the QoS target. Preemptive methods, e.g., SMK [54] and Rollover [53], can preempt the execution of BE kernels to ensure the QoS of LC applications. However, off-the-shelf GPUs currently do not support preemption due to the context switch overhead and large hardware cost [18], [28]. This paper focuses on developing novel non-preemptive co-locating strategies, which are ready to deploy for existing commodity GPUs [3], [8], [9].

Since existing co-locating schemes were developed mainly to schedule resources of traditional CUDA Cores, they tend to produce suboptimal results when being adopted for GPUs with both Tensor Cores and CUDA Cores. Figure 1 illustrates the active timeline of two units on the SM when Baymax [19] co-locates LC services and BE applications on an Nvidia 2080Ti GPU. In this experiment, we use *Resnet50* [29] as the LC service and *sgemm* from Parboil [49] as the BE application. From the figure, we observe that, while the GPU can be identified as computation-busy, either Tensor Cores or CUDA Cores are idle at any given time. This problem is referred to as *the false high utilization* problem in this paper.

The main reason behind this problem is that existing schemes fail to recognize Tensor Cores and CUDA Cores are independent resources. By scheduling a single kernel at

Quan Chen and Minyi Guo are corresponding authors.

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00064

any given time, they lack the ability to exploit both types of cores in parallel. While commodity GPUs place their warp scheduling in the black box, we test various scheduling policies and find out Tensor Cores and CUDA Cores may be used in parallel if different warps in a thread block of a kernel use the two units simultaneously. This is because multiple warps of a thread block are active at the same time. The inter-warp divergence does not incur unnecessary computation in the intra-warp divergence, as each warp has a deterministic execution branch.We also observe that both LC applications and BE applications have Tensor Core kernels and CUDA Core kernels. By fusing the Tensor Core kernel and the CUDA Core kernel from different applications, we could use the two types of cores in parallel.

In this paper, we propose **Tacker**, a kernel fusion and scheduling approach for resolving the false high utilization problem. The approach enables parallel usage of both types of cores by fusing CUDA Core kernel and Tensor Core kernel. In order to ensure the required QoS of LC applications when fusing kernels, Tacker is comprised of a *Tensor-CUDA Core kernel fuser*, a *duration predictor for fused kernels*, and a *runtime QoS-aware kernel manager*. Tacker introduces no extra security vulnerability compared with Nvidia's co-running interface MPS [1]. In both Tacker and MPS, the original programs launch the kernels, and a server process manages the actual execution. To the best of our knowledge, this is the first approach that can exploit the parallelism between Tensor Cores and CUDA Cores. Our code is also released on the Github[1]. We summarize our contributions as follows.

- We propose a static kernel fusion method to utilize two hardware's parallelism without online latency overhead. The method prepares the static fused kernels for potential LC and BE kernel pairs offline. These fused kernels use persistent thread block to deal with dynamic inputs, thus avoid online fusion overhead.
- We propose accurate prediction models for fused kernels to ensure QoS of LC applications. As a fused kernel runs longer than the original LC kernel, we adopt a model-driven predictor to predict the fused kernel's duration.
- We propose an online kernel management method to execute fused kernels. It determines to invoke the original kernel or the fused kernel accordingly, based on QoS constraints, the requirements of maximizing the throughput, and the runtime inputs of the kernels.

We evaluate the proposed approach on real hardware (Nvidia 2080Ti and V100 GPUs). Our experimental results show that Tacker not only ensures the required QoS but also improves the throughput of the co-located BE applications by 18.6% compared with Baymax on average (up to 41.1%).

II. Related Works

In recent years, several schemes have been developed to improve GPU throughput [22], [54]. To achieve better GPU scheduling, Wang *et al.* proposed SMK to exploit block preemption for block-level scheduling [54]. Based on block-level scheduling, SMK improves the throughput of the system by dividing the resources carefully. Park *et al.* proposed Maestro to change the multitasking mode in GPU for achieving better performance at runtime [44]. Wang *et al.* proposed to scale memory resources to manage memory bandwidth [52] so that an application-aware memory scheduler may be developed [35]. Optimizing SM management in multitasking GPUs also helps to improve GPU throughput [15], [16], [30], [37], [62]. These approaches infer the performance impact of SM allocation based on related metrics. Besides, there are works [21], [61] on multi-task scheduling, which are orthogonal to Tacker.

It is important for ensure QoS (quality of service) in GPU scheduling [23]. Baymax [19] and Prophet [18] exploited MPS scheduling to predict performance interference among co-located GPU applications for a temporally shared GPU. Laius *et al.* [59] proposed to predict the kernel duration and reorder the kernels on spatial multitasking GPUs. TimeGraph [36] and GPUSync [24] adopted priority-based scheduling to guarantee the performance of real-time kernels. Since these works rely on MPS [1] scheduling at the kernel level, they cannot exploit the parallelism from two types of computing cores. Wang *et al.* [53], [55] proposed to employ fine-grained sharing of SM-internal resources to improve QoS. Jain *et al.* [33] proposed careful memory isolation to ensure the performance of applications. When one of co-located kernels has high priority, it would be scheduled first. These works ensure the high-priority application's performance, and they are not applicable for the throughput problem in this paper.

HSM [65] and GDP [32] predicted the slowdown of co-located GPU applications. Compared with Tacker, many existing schemes [32], [50], [63]–[65] rely on simulation to validate the effectiveness and thus are not applicable to commodity GPUs. In addition, these schemes do not consider two types of computing cores and thus cannot explore the parallelism between Tensor Cores and CUDA Cores. Besides the above researches, there are also researches [39], [40], [56], [60] for microbenchmark-based performance model development for NVIDIA GPUs. These research works only model the applications' performance in different hardware, and could not be adapted for the fused kernel's duration prediction. They are orthogonal to Tacker.

III. Motivation

In this section, we first elaborate on the false high utilization problem. We then discuss the potential in leveraging the two types of computing cores for improving GPU utilization, and summarize the challenges in exploiting this parallelism with QoS requirements.

A. The False High Utilization Problem

To elaborate on the false high utilization problem, we conduct an experiment to study the GPU utilization when co-locating the kernels of an LC service and a BE application onto a modern GPU that has both Tensor Cores and CUDA Cores.

[1] https://github.com/sjtu-epcc/Tacker.git

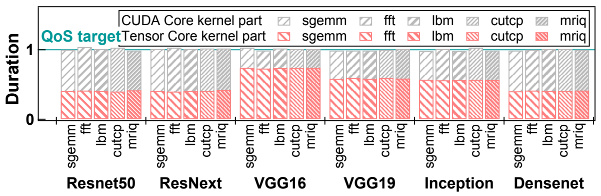


Fig. 2: The active time of the kernels with Baymax.

We choose a non-preemptive co-locating strategy Baymax [19] to exploit the idle GPU cycles between the runs of LC kernels for BE kernels and ensure the QoS of LC service at the same time. We use six DNN models (*Resnet50*, *ResNext*, *VGG16*, *VGG19*, *Inception*, and *Densenet*) as the LC services, and five tasks (*sgemm*, *cutcp*, *lbm*, *fft*, *mriq*) from Parboil benchmark suite [49] as the BE applications in the experiment. Each kernel's duration is collected to compute the duration of all the Tensor Core kernels and all the CUDA Core kernels. These two duration values are normalized to the QoS target.

Figure 2 shows the duration results of different co-located kernel pairs. The red portion indicates the duration of all Tensor Core kernels, while the gray portion indicates the duration of all CUDA Core kernels. We stack the results to show the overall active time of two hardware. From the figure, we observe that the computing units' overall active time equals the QoS target for all the kernel pairs. This is because the two types of cores are not active simultaneously.

From this experiment, we conclude that current co-locating strategies generate sequential and interleaving execution of co-located LC service and BE application, which leaves either Tensor Cores or CUDA Cores in an idle state. This is referred to as the **false high utilization** problem in this paper. Our study shows that the false high utilization problem exists widely when co-running LC services and BE applications.

B. Potential Parallelism Opportunity

We next study the potentials of utilizing CUDA Cores and Tensor Cores in parallel. We construct several micro-kernels, in which a thread block has both warps for CUDA Cores and Tensor Cores. The micro-kernels are devised in this way because the warp scheduler supports the warp scheduling for the block with different warps since Tesla [14].

For example, we implement a micro-benchmark (referred to as "Bench-A") that fuses a Tensor Core kernel K_t and a CUDA Core kernel K_c into one kernel. K_t and K_c have the same duration. K_t uses the Nvidia official GEMM implementation [4], [11]. K_c has the same grid dimension and block dimension as the Tensor Core kernel. Each thread in K_c performs pure computation using registers and has negligible memory operations. In Bench-A, the first half threads of each block are responsible for running K_t, while the other half for K_c. We also implement two more benchmarks, "Bench-B" and "Bench-C", as shown in Table I. These two benchmarks run two K_t and two K_c, respectively.

TABLE I: The normalized duration of the three benchmarks.

	1st half	2nd half	*Duration*
Bench-A	K_t	K_c	1.03
Bench-B	K_t	K_t	2
Bench-C	K_c	K_c	2

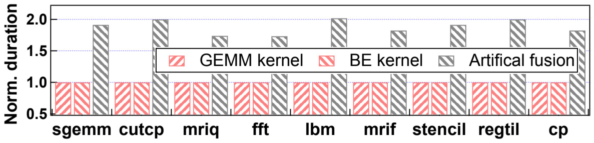


Fig. 3: The duration of the fused kernels with artificial fusion.

Table I also shows the processing time of different micro-benchmarks. The time is normalized to the duration of K_t. From the table, the normalized processing time of Bench-A is only 1.03, while that time of either Bench-B or Bench-C is 2. In this experiment, K_t and K_c already occupy all the Tensor Cores and CUDA Cores, respectively, so that their normalized execution time is 2. This further indicates that the improved execution time of "Bench-A" comes mainly from the parallel execution on both types of computing cores. To conclude, *exploiting the Tensor Cores and CUDA Cores in parallel can effectively improve the overall system throughput.*

C. Challenges in Parallelizing Tensor/CUDA Cores

However, directly fusing a Tensor Core kernel and a CUDA Core kernel does not always bring throughput improvement. For example, we choose K_t as the Tensor Core kernel and kernels from Parboil [49] benchmark suite as the CUDA Core kernel. Figure 3 shows the processing time of the fused kernels. The performance of the independent execution of each kernel is normalized to 1. From the figure, the performance of most fused kernels is around 2, indicating low parallel utilization of both types of computing cores.

Direct kernel fusion's inefficiency comes from the contention for SM resources. If both component kernels use a large amount of explicit resources (e.g., thread slot and shared memory), the fused kernel could launch fewer blocks on an SM. Both components are slowed down. Moreover, the implicit resource contention (e.g., L1/L2 caches) also slows down the execution of each component (Section V-C). Besides, kernel fusion is very likely to introduce a longer return time. Thus, inappropriate fusion may result in QoS violations.

To summarize, there exist three challenges in parallelizing the Tensor cores and CUDA cores.

- **The kernel fusion has to adapt to dynamic inputs.** While online fusion methods bring high overhead, a static fusion method needs to adapt to dynamic inputs at runtime. Besides, the static fusion should explore two kernels' maximum parallelism under limited resources.
- **The kernel fusion has to quickly and precisely predict the performance of the fused kernel.** While the fused kernel has a longer duration than the LC kernel's original duration, blindly fusing the active kernels may result in

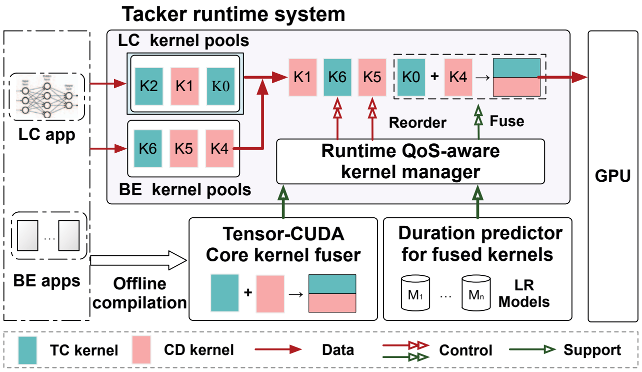


Fig. 4: The design overview of Tacker.

the QoS violation. It is challenging to make an accurate prediction, as the fused kernel's warps run different codes.

- **The kernel fusion demands QoS-aware online kernel management.** When multiple kernels from LC services and BE applications are available, Tacker should identify the fusion decision that maximizes the throughput while ensuring the QoS of LC services.

IV. The Tacker Design

In this section, we present the Tacker design to alleviate the false high utilization problem and guarantee the QoS of LC service at the same time in modern GPUs.

As shown in Figure 4, Tacker is a kernel fusion and scheduling approach that consists of a *Tensor-CUDA Core kernel fuser*, a *duration predictor for fused kernels*, and a *runtime QoS-aware kernel manager*. The kernel fuser statically combines the kernel using the Tensor Cores with the kernel using CUDA Cores. The duration predictor exploits a two-stage LR (linear regression) model to predict the duration of fused kernels. Finally, the QoS-aware kernel manager determines the appropriate kernels (either the original kernel or fused kernel) to invoke at runtime.

To efficiently fuse a kernel from a Tensor Core kernel and a CUDA Core kernel, Tacker transforms the dynamic grid dimensions of the to-be-fused kernels to static grid dimensions using Persistent Thread Block (PTB). The transformation eliminates the need to percept the grid dimension online. Since the to-be-fused kernels use a different amount of resources (e.g., registers, shared memory, and global memory bandwidth), another challenge is how to design an efficient mechanism to maximize the throughput of fused kernels. (Section V).

As the fused kernel tends to finish in a longer time (comparing to original runs), we need to predict their duration for ensuring the QoS of LC services. The challenge here is that the widely-used linear regression for predicting a kernel's latency [18] is not applicable for fused kernels, as the warps of a thread block run different codes in a fused kernel. In this paper, we analyze the warp scheduling in a block of a fused kernel, and predict the execution of a fused kernel using a two-stage linear regression model (Section VI).

```
① dim3 mix_grid, mix_block;
  mix_grid.x = (block_num_tc > block_num_cd) \
             ? block_num_tc : block_num_cd;
  mix_block.x = thread_num_tc + thread_num_cd;
  __global__ void fused_kernel(...) {
   ② if ( threadIdx.x < thread_num_tc
         && blockIdx.x < block_num_tc) {
          TC_kernel(...);
     } else if (threadIdx.x < thread_num_cd
         && blockIdx.x < block_num_cd) {
      ③ int thread_step = thread_num_tc;
         int thread_id = threadIdx.x - thread_step;
         CD_kernel(params, thread_id);
     }
  }
```

Fig. 5: Implementation of direct kernel fusion.

The kernel manager may decide to invoke the fused kernels or reorder the original kernels according to QoS headroom of the LC query at runtime. When an LC query is received, Tacker predicts the duration of each kernel in the query. The duration of all the possible fusion pairs from LC kernels and BE kernels are also predicted. The LC kernels and BE kernels are not limited to a specified type. We prioritize the selection of the fused pair that can ensure the QoS of LC service and maximize the throughput of BE applications at the same time. If such a fused kernel cannot be found, the LC kernels and BE kernels are reordered, the same as that in Baymax [19]. To invoke a fused kernel, the kernel manager obtains the parameters of the original kernels through shared memory (Section VII).

Tacker can be used to manage long-running LC services in private data centers where all the workloads are known, and Tacker has access to the applications' codes. This is similar to those in prior works [18], [19], [43], [53], [58]. To achieve long-term throughput improvement, it is acceptable to profile the LC services and BE applications and then statically fuse kernels. Moreover, kernel fusion can also be done on the clouds based on an application's occurrence if the code is available. If an application's occurrence exceeds a threshold, Tacker prepares fused kernels for its kernels. The threshold is adjustable. Tacker can also be implemented at the cluster level. For instance, at the cluster level, we can identify the long-running applications and prepare the fused kernels. The fused kernels are then distributed to GPUs based on the BE applications' location.

V. Tensor-CUDA Core Kernel Fusion

In this section, we describe the direct kernel fusion, its limitations, and present our method to address these limitations.

A. Direct Kernel Fusion

A simple fusion strategy is to fuse the thread blocks of a CUDA Core kernel (CD kernel for short) and a Tensor Core kernel (TC kernel for short) into a new block. Figure 5 shows the general implementation. As shown in the figure, the first half of threads are responsible for the TC kernel while the second half are for the CD kernel (step ② in Figure 5). The CD kernel's threads compute their original thread id using

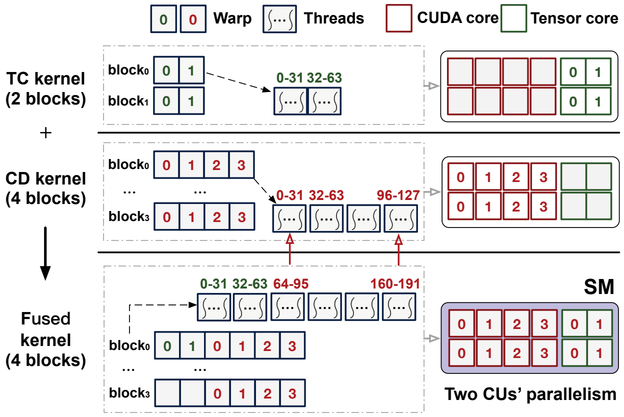


Fig. 6: An example process of direct kernel fusion.

```
__global__ void CD_kernel(...) {
    int i = blockIdx.x;
    ...
}
__global__ void ptb_CD_kernel(...) {
    for (int block_pos = blockIdx.x;
             block_pos <= original_block_num;
                 block_pos += issued_block_num) {
         int i = block_pos;
         ...
    }
}
```

Fig. 7: The original and PTB versions of a kernel.

```
dim3 mix_grid, mix_block;
mix_grid.x = SM_NUM;
mix_block.x = thread_num_tc * 2 + thread_num_cd;

__global__ void new_fused_kernel(...) {
    if ( threadIdx.x < thread_num_tc * 1) {
        thread_step = thread_num_tc * 0;
        ptb_TC_block0(...);
    } else if ( threadIdx.x < thread_num_tc * 2) {
        thread_step = thread_num_tc * 1;
        thread_id = threadIdx.x - thread_step;
        ptb_TC_block1(params, thread_id);
    } else if (threadIdx.x <
       thread_num_tc * 2 + thread_num_cd ) {
        thread_step = thread_num_tc * 2;
        thread_id = threadIdx.x - thread_step;
        ptb_CD_block0(params, thread_id);
    }
}
```

LC Blocks

BE Blocks

Fig. 8: A fused kernel's construct example.

the thread step, the TC kernel's block dimension (step ③ in Figure 5). Besides, all threads decide whether they perform computation based on the block id.

Figure 6 shows an example process of direct kernel fusion. In the figure, the TC kernel has 2 blocks, each block has 2 warps, and thread id ranges from 0 to 63. The CD kernel has 4 blocks, each block has 4 warps, and thread id ranges from 0 to 127. After kernel fusion, the fused kernel has 4 blocks, each block has 6 warps, and thread id ranges from 0 to 191. For each block in the fused kernel, threads 0-63 are responsible for TC kernel part while threads 64-191 are for CD kernel part. Since each thread in the block determines its computation based on its block id and thread id, Thread 64-191 needs to be converted to thread 0-127 using the thread step. Besides, each warp in the first two blocks performs computation while two warps are idle for the last two blocks.

Since the block number and block dimension have changed after kernel fusion, each thread's block id and thread id need to be converted to locate the original computation. In this case, the direct kernel fusion method requires two kernels' block numbers and block dimensions in advance. However, the block number is determined by the task's input that is only known online. If we fuse kernels dynamically in the runtime, we need to transform source code and generate binary code online (like JIT in JVM). The process takes almost 900 milliseconds (Section VIII-I), and introduces serious QoS violations.

The direct kernel fusion method is inappropriate for LC services that have unstable inputs.

B. PTB-based Kernel Fusion

To eliminate the impact of the block number and block dimension, we fix the block number of each kernel using Persistent Thread Block (PTB) technique [26], [42], [48]. PTB's idea is to treat each issued block as a worker on SM. With PTB, each persistent block is assigned some tasks that correspond to the original thread blocks. A persistent thread block exits while it completes its assigned tasks.

Figure 7 shows the PTB version of a kernel *CD_kernel* (named by *ptb_CD_kernel*). In the figure, *issued_block_num* is the number of persistent blocks on all SMs in *ptb_CD_kernel*, and *original_block_num* is the number of blocks of the original kernel *CD_kernel*. Specifically, the thread blocks in *ptb_CD_kernel* perform computation based on *block_pos* that is calculated from the new *blockIdx* and *issued_block_num*.

We can use the source-to-source compilation to create the PTB version of a kernel *CD_kernel* (named by *ptb_CD_kernel*). The compilation idea is to add one for loop inside the original kernel, and recompute the block id in each iteration. The original block number becomes a parameter of the PTB version kernel. In this way, *ptb_CD_kernel* has the fixed block number, though the original version has a dynamic block number that depends on the inputs. *With the fixed block number, the PTB-based kernels can be fused offline.*

C. Flexible Kernel Fusion

The naive PTB-based method fuses two kernels' blocks at a 1:1 ratio. However, this ratio is likely to slow down one component kernel. For instance, to achieve the original performance, the TC kernel needs 2 persistent blocks per SM, and each block uses 16KB shared memory; the CD kernel needs 1 persistent block per SM, and each block uses 32KB shared memory. When the CD and TC kernels are fused, a block of the fused kernel uses 48KB shared memory. In this case, only a single block of the fused kernel could be issued when an SM only has 64KB shared memory, and the performance of the TC kernel part drops seriously.

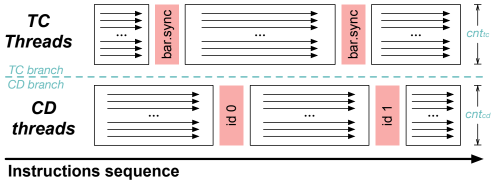


Fig. 9: Partial thread synchronization in a fused kernel.

We therefore enhance the fusion by enabling flexible fusion ratio. The key idea here is to add enough TC blocks into a fused block first, for ensuring the performance of TC kernel part. We prioritize TC blocks because Tensor Cores are more powerful than CUDA Cores in an SM. Higher throughput could be achieved with this option. Later, according to the remaining resources, the CD blocks are added accordingly. Figure 8 shows an example of fusing two blocks of a TC kernel and one block of a CD kernel into a fused block. It is done automatically with Tacker.

Even with the flexible kernel fusion, there is still implicit memory subsystem contention. It is not always optimal to put all the CD blocks on the SM. Therefore, we create all possible fused kernels for two kernels, measure these candidates' performance and two kernels' sequential performance, and choose the best one among them. If the sequential case shows the best performance, we do not fuse the two kernels.

D. Synchronization and Power

In a fused kernel, heterogeneous warps run in different branches of a block. While original kernels generally use $\_\_syncthreads()$ to synchronize all threads in a block, Tacker has to perform explicit synchronization for the warps from the same branch. This is because invoking $\_\_syncthreads()$ in different branches may lead to dead-lock, performance and correctness problems in fused kernels.

As shown in Figure 9, we therefore propose an adaptive thread synchronization interface based on the low-level PTX code $bar.sync$ [7], a thread-level barrier inside a block. Specifically, we replace the line of $\_\_syncthreads()$ in original code to *asm volatile("bar.sync id, cnt;")*, when generating the fused kernel. The parameter *id* indicates the barrier id, and it is allocated to avoid deadlock. The parameter *cnt* indicates how many threads need to arrive at the specific barrier before passing it. It is calculated based on the block dimensions.

In terms of power consumption, the power of a GPU (2080Ti and V100) already achieves the peak power limit when the GPU runs a single TC kernel. When the CUDA Cores and Tensor Cores are active simultaneously, the power stays at the peak. The power is measured using *nvidia-smi*.

VI. Modeling Fused Kernels

In this section, we propose a model-based performance prediction approach that can accurately estimate the fused kernel's duration. This helps to avoid possible QoS violations.

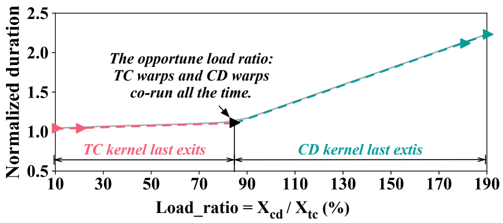


Fig. 10: The duration of a fused kernel with different load ratios, when the TC component has fixed original time.

A. Analyzing The Duration of Fused Kernel

Tacker does not predict the performance of a fused kernel based on the counters, as performance counters in GPU are not available at runtime. Therefore, Tacker does not need to separate the counters of the CUDA Core and Tensor Core parts. To construct a model for predicting the duration of fused kernels, we study the fused kernel's duration through extension profiling. Since the fused kernel's block setup is static, its duration could only be affected by the computation size of TC part and CD part. These two parts correspond to the original time of TC kernel and CD kernel, and we use X_{ori_tc} and X_{ori_cd} to represent them. To model the fused kernel's duration from two variables, we then define a metric *load ratio* in Equation 1 to simplify the process. Based on that, our profiling experiments could be divided into two parts: changing load ratio with fixed TC kernel's original time, and changing TC kernel's original time with fixed load ratio.

$$Load\_ratio = X_{ori\_cd} \ / \ X_{ori\_tc} \tag{1}$$

For the first experiment, we fix the TC kernel's workload, i.e., with static X_{ori_tc}, and model the fused kernel's duration with different workloads of the CD kernel, i.e., a changing X_{ori_cd}. Figure 10 shows the fused kernel's duration of the *GEMM-fft* pair. In the figure, the x-axis is the load ratio; the y-axis is the duration of the fused kernel that is normalized to the fixed X_{ori_tc}. From the figure, the duration curve fits a two-stage linear regression model. In particular, there exists an inflection point before the line exhibits a sharper slope, and the sharper slope is 1. This means that the duration growth of the CD kernel is converted to the duration growth of the fused kernel after the inflection point.

Therefore, we may divide the duration prediction of a fused kernel into two stages: the co-running of two kernels and the solo-running of one kernel. There is an *opportune* load ratio that the two kernels always co-run and finish at the same time. For the stage before the inflection point, the solo-run kernel in the fused kernel becomes the TC kernel. The smaller slope is decided by the increasing co-running time of two kernels.

For the second experiment, we fix the load ratio, i.e., with static $Load\_ratio$, and model the fused kernel's duration with different workloads of the TC kernel, i.e., a changing X_{ori_tc}. We choose several load ratios randomly to show the experimental results better. Figure 11 shows the duration curves with different load ratios. Each curve in Figure 11

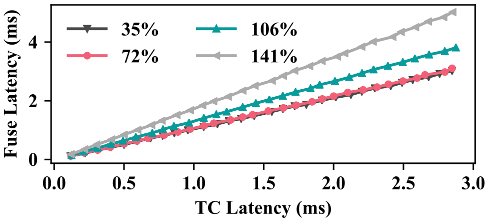


Fig. 11: The duration of a fused kernel with fixed load ratios, when the TC component has different original time.

corresponds to one fixed load ratio. The x-axis is the TC kernel's original time, and the y-axis is the fused kernel's duration. As shown in the figure, the fused kernel's duration has a linear relationship with the TC kernel's original duration while the load ratio is fixed.

We have two observations from the above analysis. *First, the fused kernel's duration shows a two-stage linear regression model, if the TC kernel's original duration is fixed. Second, when the load ratio is fixed, the fused kernel's duration has a linear relationship with the TC kernel's original time.*

Therefore, we could predict the fused kernel's duration in three steps. 1) we predict the TC kernel and CD kernel's original time using LR models, which are X_{ori_tc} and X_{ori_cd}. 2) we compute the $Load\_ratio$ based on Equation 1. 3) we predict the fused kernel's duration using the two-stage linear regression model in Figure 10.

B. The Two-stage Linear Regression Model

We infer the two-stage linear regression model through warp scheduling. For modern GPUs [6], [10], warps are switched on the SM to hide the computation gap and the switching strategy is deterministic [5], [17], [38]. When multiple warps perform computation alternately, warp switching is triggered by memory access or synchronization.

Figure 12 (a) and (b) show the warp execution timeline of PTB-based TC kernel and CD kernel, respectively. These persistent warps process the original warps' computation in a loop. With the deterministic warp switching strategy and the warps' instruction loop, PTB-based warp execution exhibits a repetitive pattern. Recent studies have shown that an LR-(linear regression) based model can precisely predict the duration of PTB-based kernels (if the kernel demands either Tensor cores or CUDA cores, but not both) [18], [32], [65].

Though the block of a fused kernel contains both TC warps and CD warps, they are scheduled with the same strategy. As shown in Figure 12(c), TC warps and CD warps run at the same time since they use different computing cores. Due to memory contention, the execution behaviors of TC warps and CD warps are different from original execution. Nonetheless, while both TC warps and CD warps have instruction loops, the warp execution of the fused kernel still exhibits a repetitive pattern when they co-run. *Therefore, LR is applicable for the fused kernel when the two component warps co-run.*

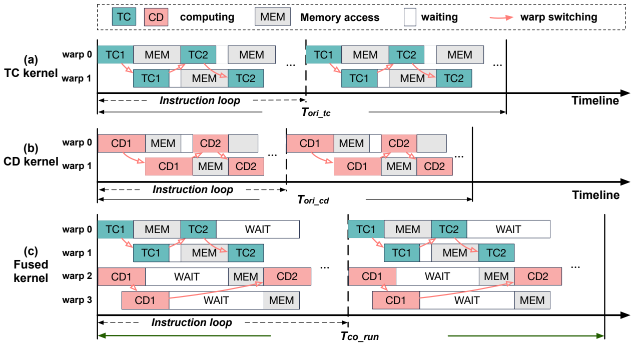


Fig. 12: The warp execution timelines of different kernels.

As discussed, the execution of a fused kernel can be divided into: the co-run of TC warps and CD warps, and the solo-run of either the remaining TC warps or CD warps. In Equation 2, T_{fuse}, T_{co_run}, and T_{solo_run} indicate the duration of the fused kernel, the duration of the co-run stage, and the duration of the solo-run stage, respectively. *While both stages could be predicted using LR, the fused kernel's duration have a linear relationship with TC kernel's original duration if the two kernels have static load ratio.* This corresponds to the second observation in Section VI-A.

$$T_{fuse} = T_{co\_run} + T_{solo\_run} \tag{2}$$

As observed from Equation 2, the load ratio determines the solo-run warp type and two stages' duration. While the fused kernel's scheduling have two cases: the TC warps solo-run after co-run and the CD warps solo-run after co-run. These two cases all could utilize LR to predict the solo-run duration. Therefore, we could infer that *the performance of the fused kernel can be predicted using a two-stage linear regression model based on the two component kernels' load ratio.*

Let $Load\_ratio_{opportune}$ represent the very load ratio point that divides the two stages in Figure 10. We use T_{ori_cd} and T_{ori_tc} to represent the original time of the CUDA Core part and the Tensor Core part when $Load\_ratio_{opportune}$ is achieved, and use $T_{opportune}$ to represent the duration of the fused kernel in this case. According to Equation 1, $Load\_ratio_{opportune} = T_{ori\_cd}/T_{ori\_tc}$.

When scheduling a TC kernel or a CD kernel with dynamic inputs at runtime, we predict their original duration as X_{ori_tc} and X_{ori_cd}, respectively. If the two component kernels' load ratio equals to $Load\_ratio_{opportune}$, the two kernels co-run and finish at roughly the same time. Given the duration of the co-run could be predicted using LR, we can get the relationship in Equation 3. In this case, Equation 4 predicts the duration of the fused kernel when the original time of the two kernels are X_{ori_tc} and X_{ori_cd}.

$$\frac{T_{co\_run}}{T_{opportune}} = \frac{X_{ori\_tc}}{T_{ori\_tc}} = \frac{X_{ori\_cd}}{T_{ori\_cd}} \tag{3}$$

$$\begin{aligned} T_{fuse} &= T_{co\_run} + T_{solo\_run} = T_{co\_run} + 0 \\ &= T_{opportune} \times \frac{X_{ori\_tc}}{T_{ori\_tc}} \end{aligned} \tag{4}$$

If we fix the TC kernel's original time and increase the CD kernel's original time by Y_{ori_cd}, the fused kernel changes its execution from Figure 12(c) to Figure 12(b) after the co-run. Therefore, the duration of the new computation stage is added to the duration of the co-run directly. As shown in Equation 5, the duration of the fused kernel and the load ratio exhibits a linear relationship over Y_{ori_cd}. Therefore, T_{fuse} has a linear relationship with *Load_ratio*.

$$\begin{aligned} T_{fuse} &= T_{co\_run} + T_{solo\_run} \\ &= T_{opportune} \times \frac{X_{ori\_tc}}{T_{ori\_tc}} + Y_{ori\_cd} \\ &\propto Y_{ori\_cd} \\ Load\_ratio &= \frac{X_{ori\_cd} + Y_{ori\_cd}}{X_{ori\_tc}} \propto Y_{ori\_cd} \\ T_{fuse} &\propto Load\_ratio \end{aligned} \tag{5}$$

On the contrary, if we reduce the duration of the CD kernel by Y_{ori_cd}, we can infer that TC warps must perform some computation on the GPU alone. The execution changes from Figure 12(c) to Figure 12(a) after the co-run stage. Assume Y_{solo_tc} indicates the duration of the solo-run of TC warps. As shown in Equation 6, the fused kernel's duration and the load ratio exhibit a linear relationship with $X_{ori\_cd} - Y_{ori\_cd}$. Therefore, T_{fuse} has a linear relationship with *Load_ratio*.

$$\begin{aligned} Y_{solo\_tc} &= X_{ori\_tc} - X_{ori\_tc} \times \frac{X_{ori\_cd} - Y_{ori\_cd}}{X_{ori\_cd}} \\ T_{fuse} &= T_{co\_run} + T_{solo\_run} \\ &= T_{opportune} \times \frac{X_{ori\_cd} - Y_{ori\_cd}}{X_{ori\_cd}} + Y_{solo\_tc} \\ &\propto (X_{ori\_cd} - Y_{ori\_cd}) \\ Load\_ratio &= \frac{X_{ori\_cd} - Y_{ori\_cd}}{X_{ori\_tc}} \propto (X_{ori\_cd} - Y_{ori\_cd}) \\ T_{fuse} &\propto Load\_ratio \end{aligned} \tag{6}$$

To conclude, the performance of the fused kernel can be predicted using a two-stage linear regression model based on the two component kernels' load ratio.

C. Building Duration Models

The fused kernel duration prediction relies on the two kernels' load ratio, which requires the original kernels' duration prediction. We choose LR to predict each GPU kernel's duration, and the input is the block number in non-PTB mode, and the output is the kernel's duration, as prior researches [18], [32], [65]. Each GPU kernel needs its own LR model, and this model characterization only needs to collect a few points. In this paper, we use historical data to train the LR model, which introduces ignorant overhead.

As for the available TC-CD kernel pairs, we train the corresponding prediction models. For a TC-CD pair, we collect the fused kernel's duration in four load ratios: 10%, 20%, 180%, 190%, and build the initial duration model, and use online co-running data to update the model. Whenever the prediction error of one model exceeds 10%, Tacker updates the model using online data.

Note that, since a fused kernel runs stable benefited from the PTB implementation, its duration model does not change with the kernel's inputs [32], [65]. Meanwhile, the memory system contention is already captured, as the samples use the actual duration of a fused kernel as the baselines.

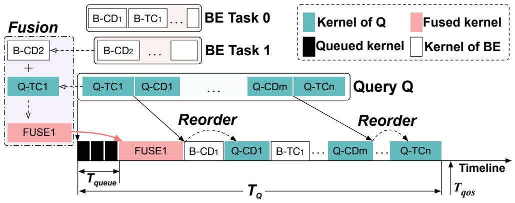


Fig. 13: The online scheduling an LC query Q with Tacker.

VII. Online Kernel Scheduling

In this section, we describe the mechanism used to schedule the kernels of LC services and BE applications.

A. End-to-End Latency Breakdown

A query's duration is the time interval between when the first kernel is issued and when the last kernel ends. As shown in Figure 13, Q's end-to-end latency (T_Q) comprises four parts. They are: (1) the running time of queued kernels (T_{queue}); (2) the running time of the kernels of Q (T_{lc}), i.e., the aggregated time of its TC kernels (Q-TC$_1$, ..., Q-TC$_n$ in Figure 13), and CD kernels (Q-CD$_1$, ..., Q-CD$_m$ in Figure 13); (3) the running time of fused kernels (T_{fuse}); and (4) the running time of kernels of BE tasks (T_{be}), which could be selected from the kernels (B-CD$_i$ and B-TC$_j$ in Figure 13).

B. Scheduling Policy

Tacker uses both kernel fusion and kernel reorder to maximize the system throughput. Figure 13 presents the end-to-end scheduling procedure of an LC query Q colocated with BE applications. Let T_{qos} represents the QoS target of a query Q, and T_Q represents Q's end-to-end latency. Q's QoS is satisfied only when Equation 7 is satisfied.

$$T_Q = T_{queue} + T_{lc} + T_{fuse} + T_{be} \leq T_{qos} \tag{7}$$

The runtime kernel scheduler of Tacker decides to perform kernel reorder or fusion for each LC kernel and BE kernel based on Equation 7 as follows.

1) Calculating QoS Headroom: As discussed above, T_{queue} is known and cannot be reduced when the query Q is launched. Tacker first predicts the original solo-run duration of Q (denoted by T_{ori_solo}) for calculating its QoS headroom (denoted by T_{hr}). T_{ori_solo} is known ahead of the execution based on the prediction models. T_{hr} reveals the free GPU time left for kernels from BE applications while co-running with Q. When the first kernel of Q is issued, $T_{hr} = T_{qos} - T_{ori\_solo} - T_{queue}$. Based on T_{hr}, each time a kernel of Q is launched, Tacker iterates over the ready BE kernels to check whether there are potential opportunities of kernel fusion and kernel reorder.

Suppose the current kernel of Q is a TC kernel and its predicted duration is T_{tc}, and there is a ready CD kernel from BE applications with duration T_{cd}. Tacker predicts the duration of the kernel fused from the two kernels (denoted by T_{k_fuse}). If Equation 8 is satisfied, Tacker actually fuses the two kernels and launches the fused kernel. Equation 8 states that the two kernels' fusion could take advantage of two computing units' parallelism, and the fused kernel's duration

TABLE II: Experimental specifications.

CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
GPU	NVIDIA RTX 2080Ti (68 SMs, 544 Tensor Cores)
Software	CUDA Version: 10.0, CUDNN Version: 7.5
LC Services	Resnet50 (batch size: 32), ResNext (24), VGG16 (24) VGG19 (16), Inception (32), Densenet (16)
BE Apps [49]	mriq, fft, mrif, cutcp, cp, sgemm, lbm, tpacf, Resnet50-T (Res-T), VGG16-T (VGG-T), Inception-T (Incep-T), Densenet-T (Dense-T)

is within the QoS headroom. More specifically, the kernel fusion spends $T_{k\_fuse} - T_{tc}$ to complete the CD kernel, which originally takes T_{cd}. After the kernel launch, Tacker updates T_{hr} to be $T_{hr} - (T_{k\_fuse} - T_{tc})$.

$$T_{tc} + T_{cd} > T_{k\_fuse} \ \&\& \ T_{k\_fuse} - T_{tc} \quad < T_{hr} \tag{8}$$

If all the ready BE kernels may not be fused with the current kernel of Q, Tacker checks whether a BE kernel can be launched directly. For a BE kernel with prediction duration T_{tmp}, if T_{tmp} is smaller than T_{hr}, it is launched directly and T_{hr} reduces by T_{tmp}. Otherwise, the kernel is not launched.

Note that, if multiple BE applications are active, Tacker fuses the kernels with the highest throughput gain. The throughput gain can be calculated to be $T_{gain} = T_{cd} - (T_{k\_fuse} - T_{tc})$. In this equation, $T_{k\_fuse} - T_{tc}$ is the time for Tacker to finish the CD kernel, which has original time T_{cd}. Tacker fuse the kernel of Q with the BE kernel with the largest T_{gain} to maximize the system throughput.

2) Multiple active LC queries: It is possible that multiple LC queries are active. In this case, in order to ensure the QoS of all the LC queries, we choose to complete the early queries, and only perform kernel reorder and kernel fusion for the last arrived query. For instance, if an LC query Q_i is still active when Q arrives, the kernels of Q_i must complete the computation first. Otherwise, the long processing time of Q_i may already result in the QoS violation of Q.

When we calculate the QoS headroom of Q, the GPU time reserved for Q_i's unexecuted kernels needs to be subtracted. Therefore, we monitor the remaining GPU time that each query needs to complete the computation. For a specific query, such as Q_i, we calculate its remaining GPU time by subtracting the time of its completed kernels from its predicted overall time (T_{lc} of Q_i).

Suppose there are n active LC queries when Q is launched. Let T_{lc_1}, ..., T_{lc_n} represent each query's remaining GPU time. Equation 9 calculates Q's QoS headroom when it is issued. If the T_{hr} of the new query is close to 0, Tacker directly launches all the kernels to the GPU.

$$T_{hr} = T_{qos} - T_{queue} - T_{ori\_solo} - \sum_{i=1}^{n} T_{lc\_i} \tag{9}$$

VIII. Evaluation

In this section, we describe the implementation of Tacker, and evaluate it in improving the throughput of BE applications while ensuring the QoS of LC services.

A. Implementation of Tacker

To evaluate Tacker method, we implement the kernel fuser and the runtime kernel manager. We implement the kernel fuser to be a source-to-source compiler. The fuser first converts all the to-be-fused kernels to PTB mode, collects each kernel's per-block resource usage (number of registers and shared memory size). After that, the kernel fuser profiles each kernel's persistent block number, which has the optimal performance. Lastly, a fused kernel is generated following Section V-C, and a dynamic-link library is created for online invocation.

We implement the kernel manager based on Caffe [34]. The manager determines to invoke the original kernels or the fused kernels through the dynamic libraries. We implement shared memory-based parameter passing to pass parameters from the original kernels to the fused kernel. Specifically, CD kernels store their data objects into the shared memory, and the fused kernel obtains the required data objects from the shared memory. Besides, for each application, Tacker maintains a kernel queue that contains the kernel type, ready state, and scheduling interface of each of its kernels. When a memory-copy kernel completes, the next kernel is set to be ready.

To implement Tacker method in the python-based frameworks like TensorFlow, the fused kernels are compiled into customized operators through *custom-op* [13]. At runtime, Tensorflow invokes either the customized or original operators.

B. Experiment Setup

Table II shows the detailed experimental setup. We use six commonly used DNN models, *Resnet50* [29], *ResNext* [57], *VGG16*, *VGG19* [47], *Inception* [51], and *Densenet* [31], as LC applications; use eight applications from Parboil [49] and four DNN training tasks, *Resnet50-T*, *VGG16-T*, *Inception-T* and *Densenet-T* ("-T" is used for distinguishing with inference models) as BE applications. The BE applications are categorized into compute-intensive (*mriq*, *fft*, *mrif*, *cutcp*, *cp*) and memory-intensive (*sgemm*, *lbm*, *tpacf*). DNN training jobs are also treated as memory-intensive. We use 50ms to be the QoS target, and LC queries arrive in Poisson distribution [45]. The batch sizes of the LC services are set based on the QoS target. The load of each LC service is configured to be 80% of its peak supported load without causing QoS violation, to emulate a real datacenter scenario. **All the benchmarks in Parboil use CUDA Cores, and the kernels in training tasks use either Tensor Cores or CUDA Cores.**

The experiments are carried out on a server equipped with an Nvidia RTX 2080Ti GPU. Tacker does not rely on any particular hardware features of 2080Ti and is easy to be set up on other GPUs that integrate Tensor Cores. We also evaluate Tacker on an Nvidia V100 GPU in Section VIII-F.

C. Improving Throughput

In this subsection, we compare Tacker with Baymax [19], a system that improves accelerator utilization while guaranteeing the QoS by reordering kernels. Equation 10 calculates the throughput improvement [19], [41], [43], [58] of Tacker compared with Baymax. In the equation, T_{Baymax} and T_{Tacker}

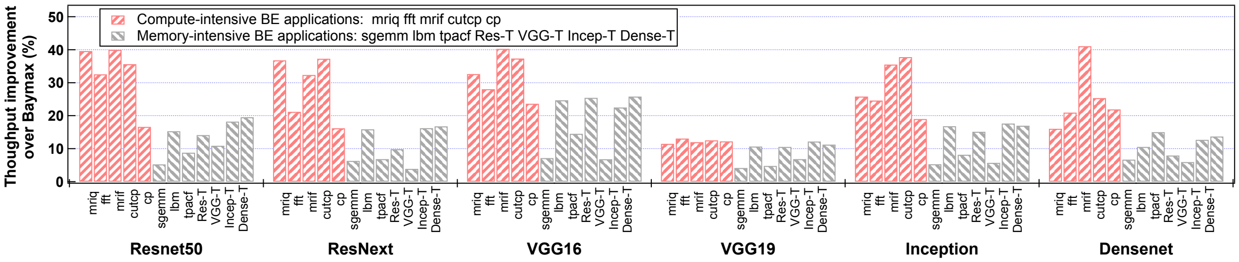


Fig. 14: The throughput improvement of BE applications at co-location with Tacker.

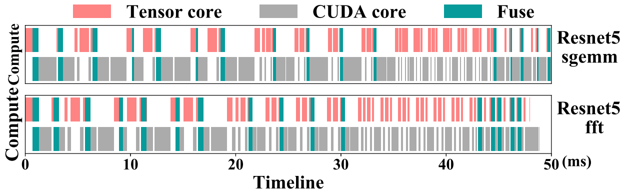


Fig. 15: The active timelines of the two types of cores.

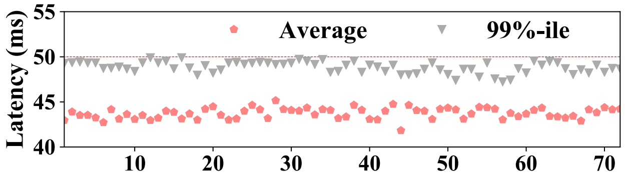


Fig. 16: The average and 99%-ile latencies of the LC services in all the 72 co-location pairs with Tacker.

represent the processing time of BE applications using Baymax and Tacker, respectively. The throughput improvements only include the results from BE applications as ensuring QoS is sufficient for LC services [19], [41], [43], [58].

$$Throughput\ improvement = \frac{T_{Tacker} - T_{Baymax}}{T_{Baymax}} \tag{10}$$

Figure 14 compares the throughput of the BE applications when adopting Tacker and Baymax. From the figure, Tacker achieves an average of 18.6% (and up to 41.1%) improvement over Baymax. Tacker improves the throughput for all 72 (=6×12) co-location pairs because it exploits both adaptive kernel fusion and kernel reorder, which help to explore not only the parallelism from two types of computing cores but also the idle GPU time in the QoS headroom. As a comparison, Baymax only utilizes the idle GPU cycles with kernel reorder.

We also observe that Tacker achieves higher throughput improvements for compute-intensive BE applications. This is because memory-intensive applications require more memory resources and their co-runs face more resource contention, leading to lower throughput improvements.

Figure 15 presents the execution traces of LC application *Resnet50* and two BE applications (*sgemm* and *fft*) with Tacker, which help to clarify the reason why Tacker performs better than Baymax. In the figure, the two rows represent the active time of the CUDA core and Tensor cores, respectively. We use blue bars to represent the co-run with Tacker.

From Figure 15, Tacker successfully exploits the parallelism from the two types of cores. By comparing the timelines for *Resnet50+sgemm* and *Resnet50+fft*, we find that the total time of both types of cores being active from *Resnet50+fft* is longer than that from *Resnet50+sgemm*. Given the BE applications in both pairs are compute-intensive, the longer time the two types of cores stay active, the more parallelism Tacker explores, and thus the higher throughput the fused kernel can achieve.

We then evaluate the efficiency of Tacker by analyzing its improvement upper bound on throughput. For simplicity, we assume the query only has a single TC kernel and a single CD kernel. The throughput improvement is maximized when the BE application has an opportune CD kernel and an opportune TC kernel. Let T_{lc} represent the duration of TC kernel in an query, and T_{be} represent the duration of CD kernel in the BE application. Let T_{fuse} represents the duration of the fused kernel, and T_{hr} represents the QoS headroom. Based on that, the squeezed time with kernel fusion that can be used to run BE kernels is $T_{squeezed\_tc} = T_{be} - (T_{fuse} - T_{be})$. As for CD kernel part, the squeezed time for CD kernel part could also be calculated in the same way as $T_{squeezed\_cd}$. Besides, both Tacker and Baymax are able to run BE kernels in the QoS headroom period T_{hr} through kernel reorder. The time upper bounds of Tacker and Baymax that can be used to run BE kernels are $T_{squeezed\_tc} + T_{squeezed\_cd} + T_{hr}$, and T_{hr} respectively. The throughput improvement of Tacker over Baymax is $(T_{squeezed\_tc} + T_{squeezed\_cd})/T_{hr}$.

For instance, the optimal throughput improvement of Tacker in *Resnet50+sgemm* and *Resnet50+fft* are 11% and 66%, respectively. In our experiment, Tacker improves the throughput by 5.5% and 32.9%. This is because we only use 55.4% of the TC kernels for fusion. Since the TC kernels in CuDNN [20] is black-boxed, we use an open-source Nvidia implementation [4], [11] with the similar performance to replace the TC kernels in CuDNN. In addition, the opportune load ratio may not always be achieved due to the random inputs of BE tasks. Higher improvement could be achieved if we can use the CuDNN implementation.

When the batch size of the LC application is smaller, the co-located BE application gets higher throughput. The experimental results show that the BE application has a further

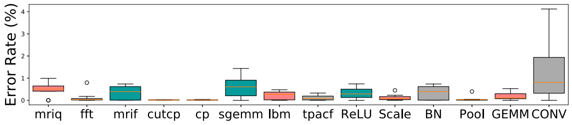


Fig. 17: The duration prediction errors of the PTB kernels.

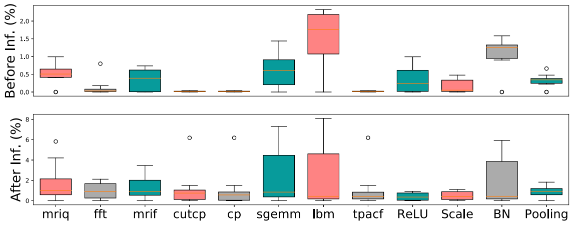


Fig. 18: The duration prediction errors of the fused kernels.

17.4% throughput improvement when the batch size of the LC application is 1. Meanwhile, the throughput gain from the kernel fusion technique is smaller, because the LC application's duration determines the fusion potential. The throughput of the benchmarks with Tacker is 5.5% more compared with Baymax, when the batch size is 1.

D. Guaranteeing QoS

Figure 16 presents the 99%-ile and average latencies of the LC applications in the 72 co-location pairs. As shown in the figure, Tacker ensures the QoS for LC applications under all the co-locations. This is because Tacker determines whether to perform kernel fusion based on the queries' QoS headroom in real-time. If there is a possible QoS violation, Tacker launches the kernels of the LC application directly.

Moreover, LC applications in all the co-locations achieve similar average latency, because the queries arrive in the same distribution. The same arrival distribution and scheduling strategy bring similar average latency. Besides, LC applications have similar 99%-ile latencies in all the co-locations. This is because the QoS targets of the LC applications are all 50*ms* in our experiment. Tacker effectively uses the QoS headroom in all the co-locations to run the BE kernels, the 99%-ile latencies of the LC applications are close to the QoS target.

E. Accuracy of The Duration Predictor

In this subsection, we evaluate the duration prediction accuracy for fused kernels. As presented in Section VI-A, Tacker first predicts the duration of each kernel before fusing, and then predicts the duration of the fused kernel based on the predicted duration of the to-be-fused kernels.

In this experiment, we first investigate the prediction accuracy of the linear regression models on a single PTB kernel. These LR models accept the basic runtime configuration (input parameters) of kernels and predict their running time. Figure 17 shows the prediction error of these single kernels prediction error. Besides the kernels from Parboil, we choose four representative kernels from DNN training tasks, which are *ReLU, Scale, BN*, and *Pooling*. The predicted running time

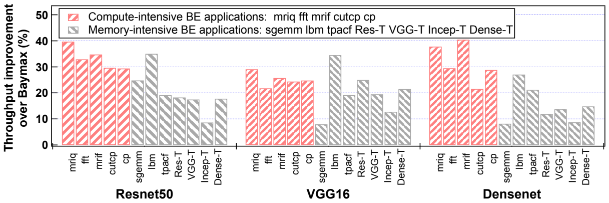


Fig. 19: Throughput improvement on an Nvidia V100.

differs from the actual value by at most 3%, and the average prediction error is less than 2%. Therefore, Tacker is able to use linear regression to predict the duration of PTB kernels.

We also evaluate the two-stage LR model's prediction accuracy for the fused kernels. As discussed in Section VI-B, the duration prediction is based upon two kernels' load ratio. We fit these two stages (denoted as "Before Inflection" and "After Inflection") separately with two LR models. Figure 18 shows the prediction accuracy for these two stages. These LR models achieve an error rate lower than 8%.

The two-stage LR modeling technique is accurate for predicting the duration of fused kernels.

F. Adapting to Other GPU Generations

Besides 2080Ti, Figure 19 shows the throughput improvement of BE applications with Tacker on a V100 GPU [9]. Results of three LC services are shown due to the tight space. As observed, Tacker increases the throughput of BE applications by 23.3% on average (up to 40.4%). By comparing Figure 19 and Figure 14, Tacker improves the throughput of memory-intensive BE applications more on V100 than on 2080Ti. This is because V100 has larger shared memory in each SM (96KB) than 2080Ti (64KB). In this case, there are higher possibilities for memory-intensive BE kernels to co-run with TC kernels.

We need to update the prediction models to deploy Tacker on other GPUs, as kernels show different performance on different GPUs. No other update is required.

G. Comparing with Co-running Interfaces

MPS [1] and CUDA Stream [2] cannot use the two units simultaneously. We find that they can partially achieve the purpose by implementing extra synchronization between co-running kernels and integrating with the PTB techniques. In this experiment, we compare the kernel fusion technique in Tacker with MPS+PTB, and Stream+PTB. Specifically, we use two Nvidia *GEMM* [11], [12] implementation as the TC kernel, co-run it with the CD kernels in the BE applications.

Equation 11 defines the metric, *overlap rate*, to evaluate the achieved performance with Tacker, MPS+PTB, and Stream+PTB. We have tuned the solo-run time of the TC kernel and the CD kernels to be the same to show the highest overlap rate. In the equation, $T_{TC\ kernel}$, $T_{CD\ kernel}$, and T_{corun} are the solo-run time of the TC kernel, solo-run time of the CD kernel, and the co-running time of the two kernels,

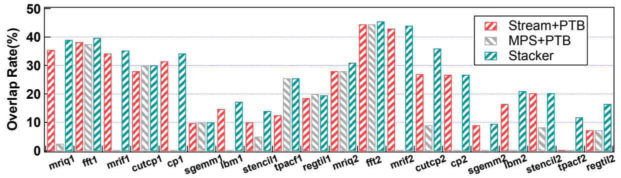


Fig. 20: Comparison with other co-running interfaces.

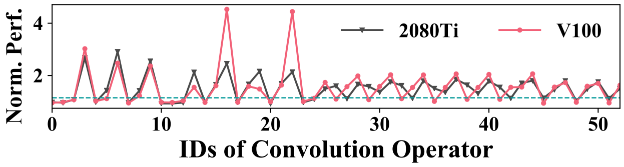


Fig. 21: Normalized performance of im2col+GEMM method over cuDNN convolution kernel.

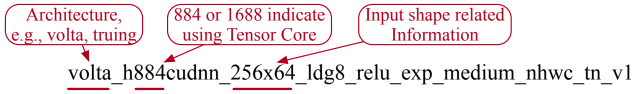
respectively. The overlap rate ranges from 0 to 50%, the larger the better.

$$Overlap\ Rate = \frac{T_{TC\ kernel} + T_{CD\ kernel} - T_{corun}}{T_{TC\ kernel} + T_{CD\ kernel}} \quad (11)$$

Figure 20 shows the overlap rates of all the kernel co-run pairs. The label of x-axis represents BE applications co-located with different GEMM kernel. The kernel fusion technique in Tacker brings the highest overlap rate in all the co-run cases. This is because Tacker guarantees that two kernels can run on an SM at the same time. For MPS and Stream, since their scheduling logic is black-boxed, it is difficult to know how they schedule the kernels that use the two types of hardware. The overlap performance of MPS is pretty poor in many cases, and Stream's performance on several cases is also unsatisfying, which is the colocation with $tpacf$, $cutcp$, and $stencil$. Even though they achieve similar overlap rates with Tacker in some cases, they are not suitable to be used for runtime scheduling due to the unstable performance.

H. Comparing with cuDNN Implementation

DL frameworks generally rely on cuDNN to provide high performance, and the implementation of cuDNN kernels is black-box. While kernel fusion needs the kernel's source code, we convert the cuDNN convolution kernel $cudnnConvolutionForward()$ to $cudnnIm2col()$ kernel and GEMM kernel [4], [11]. Figure 21 shows the normalized performance of $cudnnIm2col()$ + GEMM implementation over $cudnnConvolutionForward()$ implementation in Resnet50. As shown, the performance gap between the two implementations is less than 15% for 39.6% of the convolution kernels. By only transform the kernels with low performance gap, the entire application has less than 2% performance loss after the transformation. Specifically, 36.5% of convolution kernels in two VGG models and 55.4% of convolution kernels in the other four models are transformed to $cudnnIm2col()$ kernel and GEMM kernel.



Architecture, e.g., volta, truing
884 or 1688 indicate using Tensor Core
Input shape related Information
volta_h884cudnn_256x64_ldg8_relu_exp_medium_nhwc_tn_v1

Fig. 22: The definition of convolution implementation names

TABLE III: The resource usage of cuDNN kernels.

CONV TYPE	T1	T2	T3	T4	T5	T6
Register (%)	69.5	79.3	79.3	67.2	82.8	73.4
Shared memory (%)	64.0	100	64.0	64.0	100	76.8
Max DRAM bandwidth (%)	32.5	64.1	42.8	70.3	50.2	41.9
FP32 utilization (%)	0	0.31	0	0.19	0	0
CONV TYPE	**T7**	**V1**	**V2**	**V3**	**V4**	**V5**
Register (%)	76.9	88.6	88.6	88.6	88.6	88.6
Shared memory (%)	76.8	86.4	51.2	86.4	86.4	51.2
Max DRAM bandwidth (%)	32.2	53.4	63.9	59.1	38.5	30.2
FP32 utilization (%)	0	0	0	0.25	0	0

For the 53 convolution kernels in Resnet50, there are 7 internal implementations on 2080Ti and 5 internal implementations on V100 in cuDNN. Figure 22 presents the name of an example implementation and its definition rules. Table III shows the resource usage of the 12 cuDNN implementations. In the table, $T1$ to $T7$ and $V1$ to $V5$ are the implementations used on 2080Ti and V100 respectively. As observed, 6 implementations on 2080Ti have register usage below 80%, and 2 implementations on V100 have shared memory usage as 51.2%. All the implementations have the DRAM bandwidth usage lower than 71%, and do not use FP32 cores. **Therefore, there are unused resources in the cuDNN implementations, even though they are highly-optimized.**

I. Overhead

Tacker brings slight offline overhead and online overhead. As for the online scheduling overhead, Tacker only considers fusing the first kernel in each application's kernel queue each time. Suppose 10 LC services and 50 BE applications co-run on a GPU. Since Tacker only schedules one kernel at a time, Tacker only considers 50 kernel pairs for fusion. The operation takes 1.2 milliseconds. In the same case, we also measure the overhead of the static scheduling by forcing Tacker not to fuse the kernels. The overhead of the static scheduling is 0.5 milliseconds on average. Therefore, the online scheduling overhead of Tacker is acceptable.

Tacker's offline overhead comes from the kernel fusion process and the model training process. For a BE task in Parboil, compiling a fused kernel and generating the shared library takes 0.9 seconds, and the size of the shared library is 62KB on average. Besides, we also create a shared library for 10 DNN operators to support DNN training tasks. The operation completes in 0.7 seconds, and the size of the library is 463KB. While there are tens of operators for mainstream DL frameworks, Tacker only needs a few megabytes. In terms of training duration models, training the duration model for a fused kernel completes in 20 milliseconds.

The overhead of kernel fusion is determined by the fuse operation itself and the frequency of the fusion. While the fuse operation is intrinsic, we reduce the frequency of performing the fusion in two methods. First, if the aggregated per-block resource usage of two kernels exceeds the resource's capacity in an SM, Tacker does not perform kernel fusion. Second, if a CD kernel could not benefit from kernel fusion with one TC kernel, this CD kernel would not be considered for fusion. This is because all the TC kernels are GEMM-related, and they have similar resource usage.

IX. Conclusion

Tacker uses kernel fusion to maximize the throughput of BE applications while ensuring the required QoS of LC services. It is comprised of a Tensor-CUDA core kernel fuser, a duration predictor, and a runtime kernel manager. The kernel fuser enables the adaptive fusion of kernels that use the Tensor cores and CUDA cores. The duration predictor precisely predicts the duration of the fused kernels. The kernel manager determines whether to perform the kernel fusion. Tacker improves the throughput of BE applications by 18.6% on average (up to 41.1%), while ensuring the required QoS target.

Acknowledgment

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61632017, 61872240).

References

[1] "Cuda mps," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[2] "Cuda stream," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[3] "Nvidia ampere gpu architecture whitepaper," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[4] "Nvidia cutlass," https://github.com/NVIDIA/cutlass.

[5] "Nvidia issue efficiency," https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm.

[6] "Nvidia nsight compute," https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.

[7] "Nvidia parallel thread execution isa," https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[8] "Nvidia turing gpu architecture whitepaper," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[9] "Nvidia volta gpu architecture whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[10] "Nvidia warp instruction statistics," https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/instructionstatistics.htm.

[11] "tensor core example code," https://github.com/NVIDIA/cuda-samples/tree/master/Samples/cudaTensorCoreGemm.

[12] "tensor core example code," https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[13] "Tensorflow create customized ops," https://www.tensorflow.org/guide/create_op.

[14] "Tesla," https://en.wikipedia.org/wiki/Tesla_(microarchitecture).

[15] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of gpu resources for both performance and fairness," in *IEEE 32nd International Conference on Computer Design (ICCD 2014)*, pp. 440–447.

[16] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *19th Asia and South Pacific Design Automation Conference (ASP-DAC 2014)*, pp. 726–731.

[17] M. Awatramani, X. Zhu, J. Zambreno, and D. Rover, "Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications," in *International Conference on Parallel Architecture and Compilation (PACT 2015)*, pp. 1–12.

[18] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*, pp. 17–32.

[19] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[20] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[21] W. Cui, Q. Chen, H. Zhao, M. Wei, X. Tang, and M. Guo, "E 2 bird: Enhanced elastic batch for improving responsiveness and throughput of deep learning services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1307–1321, 2020.

[22] W. Cui, M. Wei, Q. Chen, X. Tang, J. Leng, L. Li, and M. Guo, "Ebird: Elastic batch for improving responsiveness and throughput of deep learning services," in *IEEE 37th International Conference on Computer Design (ICCD 2019)*, pp. 497–505.

[23] W. Cui, H. Zhao, Q. Chen, N. Zheng, J. Leng, J. Zhao, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2021)*, pp. 1–15.

[24] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *IEEE 34th Real-Time Systems Symposium (RTSS 2013)*, pp. 33–44.

[25] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462.

[26] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*, pp. 1–14.

[27] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, pp. 27–40.

[28] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pp. 620–629.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 2016)*, pp. 770–778.

[30] Q. Hu, J. Shu, J. Fan, and Y. Lu, "Run-time performance estimation and fairness-oriented scheduling policy for concurrent gpgpu applications," in *45th International Conference on Parallel Processing (ICPP 2016)*, pp. 57–66.

[31] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 2017)*, pp. 4700–4708.

[32] M. Jahre and L. Eeckhout, "Gdp: Using dataflow properties to accurately estimate interference-free performance at runtime," in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pp. 296–309.

[33] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional gpus: Software-based compute and memory bandwidth reservation for gpus," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019)*, pp. 29–41.

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia (MM 2014)*, pp. 675–678.

[35] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 223–234.

[36] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings USENIX ATC (ATC 2011)*, pp. 17–30.

[37] H. Lee and M. A. Al Faruque, "Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform," in *Design, Automation & Test in Europe Conference & Exhibition (DATE 2014)*, pp. 1–6.

[38] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA 2014)*, pp. 260–271.

[39] F. Liu, W. Zhao, Z. He, Y. Wang, Z. Wang, C. Dai, X. Liang, and L. Jiang, "Improving neural network efficiency via post-training quantization with adaptive floating-point," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV 2021)*.

[40] F. Liu, W. Zhao, Y. Zhao, Z. Wang, T. Yang, Z. He, N. Jing, X. Liang, and L. Jiang, "Sme: Reram-based sparse-multiplication-engine to squeeze-out bit sparsity of neural network," in *Proceedings of the 39th IEEE International Conference on Computer Design (ICCD 2021)*.

[41] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, pp. 450–462.

[42] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pp. 881–897.

[43] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2011)*, pp. 248–259.

[44] J. J. K. Park, Y. Park, and S. 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)*, pp. 527–540.

[45] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA 2020)*, pp. 446–459.

[46] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: a gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pp. 322–337.

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[48] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 39–58.

[49] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[50] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2015)*, pp. 62–75.

[51] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 2016)*, pp. 2818–2826.

[52] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multi-programming in gpus via effective bandwidth management," in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pp. 247–258.

[53] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA 2017)*, pp. 269–281.

[54] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2016)*, pp. 358–369.

[55] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "Cap: co-scheduling based on asymptotic profiling in cpu+ gpu hybrid systems," in *proceedings of the 2013 international workshop on programming models and applications for multicores and Manycores*, 2013, pp. 107–114.

[56] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA 2015)*, pp. 564–576.

[57] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 2017)*, pp. 1492–1500.

[58] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

[59] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proceedings of the ACM International Conference on Supercomputing (ICS 2019)*, pp. 58–68.

[60] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *IEEE 17th international symposium on high performance computer architecture (ISCA 2011)*, pp. 382–393.

[61] H. Zhao, W. Cui, Q. Chen, J. Leng, K. Yu, D. Zeng, C. Li, and M. Guo, "Coda: Improving resource utilization by slimming and co-locating dnn and cpu jobs," in *IEEE 40th International Conference on Distributed Computing Systems (ICDCS 2020)*, pp. 853–863.

[62] H. Zhao, W. Cui, Q. Chen, J. Zhao, J. Leng, and M. Guo, "Exploiting intra-sm parallelism in gpus via persistent and elastic blocks. international conference on computer design," in *IEEE 39th International Conference on Computer Design (ICCD 2021)*.

[63] W. Zhao, Q. Chen, and M. Guo, "Ksm: Online application-level performance slowdown prediction for spatial multitasking gpgpu," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 187–191, 2018.

[64] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*, pp. 653–663.

[65] X. Zhao, M. Jahre, and L. Eeckhout, "Hsm: A hybrid slowdown model for multitasking gpus," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*, pp. 1371–1385.