Adaptive Kernel Fusion for Improving the GPU Utilization while Ensuring QoS

Han Zhao*, Junxiao Deng*, Weihao Cui, Quan Chen, Youtao Zhang, Deze Zeng, Minyi Guo

Abstract—The prosperity of machine learning applications has promoted the rapid development of GPU architecture. It continues to integrate more CUDA Cores, larger L2 cache and memory bandwidth within SM. Moreover, the GPU integrates Tensor Core dedicated to matrix multiplication. Although studies have shown that task co-location could effectively improve system throughput, existing works only focus on resource scheduling at the SM level and cannot improve resource utilization within the SM.
In this paper, we propose Aker, a static kernel fusion and scheduling approach to improve resource utilization inside the SM while ensuring the QoS (Quality-of-Service) of co-located tasks. Aker consists of a static kernel fuser, a duration predictor for fused kernels, an adaptive fused kernel selector, and an enhanced QoS-aware kernel manager. The kernel fuser enables the static and flexible fusion for a kernel pair. The kernel pair could be Tensor Core kernel and CUDA Core kernel, or computing-prefer CUDA Core kernel and memory-prefer CUDA Core kernel. After the kernel fuser provides multiple fused kernel versions for a kernel pair, the duration predictor precisely predicts the duration of the fused kernels and the adaptive fused kernel selector locates the optimal fused kernel version. 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 Aker improves the throughput of best-effort applications compared with state-of-the-art solutions by 50.1% on average, while ensuring the QoS of latency-critical tasks.

Index Terms—Kernel fusion, QoS, GPU scheduling

1 Introduction

GPUs have gained widespread acceptance as a flexible acceleration solution for many modern applications [1], [2]. With rapid technological advancements, GPUs are becoming increasingly powerful. It keeps integrating more CUDA Cores, larger L2 cache and memory bandwidth. Moreover, in response to the escalating demand for acceleration in artificial intelligence and machine learning (AI/ML) applications, recently released commercial GPUs, exemplified by Nvidia Volta [3] and subsequent architectures, have integrated Tensor Cores within streaming multiprocessors (SM) to accelerate general matrix multiplication (GEMM), which constitutes a fundamental operation in AI/ML applications.

Given the abundant computing resources in modern GPUs, studies have proposed the co-location of multiple applications onto the same GPU. This effectively improves resource utilization and reduces system energy consumption, particularly for computing servers in data centers. Based on QoS (Quality-of-Service) demands, we can classify data center applications into two categories: 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, necessitating the object detection algorithm to complete within 50ms [4], [5]. 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.

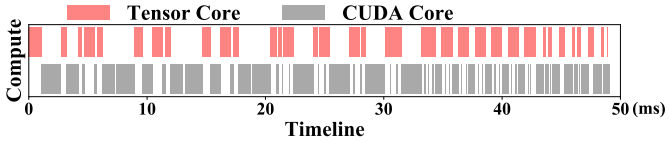


Fig. 1: The active timeline of Tensor Cores and CUDA Cores when Baymax is used to run *Resnet50* and *sgemm*.

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 [6] enables the BE applications to share the unutilized GPU cycles from LC applications. Preemptive methods, e.g., Rollover [7], 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 [8], [9]. This paper focuses on developing novel non-preemptive co-locating strategies, which are ready to deploy for existing commodity GPUs.

Since existing co-locating solutions only focus on the GPU-level time-sharing between LC and BE applications, they tend to produce suboptimal results on the GPUs with abundant computing and memory resources. Figure 1 first presents a suboptimal result example. In this experiment, we use Baymax [6] co-locates LC services (*Resnet50* [10]) and BE applications (*sgemm* from Parboil [11]) on an Nvidia RTX Ada6000 GPU. From the figure, we observe that, while the GPU can be identified as computation-busy, either Tensor

• *Han Zhao and Junxiao Deng contribute to this paper equally. All the authors except Youtao Zhang and Deze Zeng are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Youtao Zhang is with the Computer Science Department, University of Pittsburgh. Deze Deng is with the School of Computer Science, China University of Geosciences.*

Cores or CUDA Cores are idle at any given time.

Meanwhile, even for the GPU kernels only using CUDA Core, they may prefer computing resources or memory resources. The experimental results in Section 3.1 show that computing-prefer kernels exhibit 80.5% computing core utilization and 3.15% memory bandwidth utilization. Conversely, memory-prefer kernels exhibit 85.6% memory bandwidth utilization and 26.1% computing core utilization. When Baymax co-locates two BE applications only using CUDA Core, either computing resources or memory resources are in a low utilization state at any given time.

The above two problems are referred to as the false high utilization problem. The main reason behind this is that existing solutions ignore the fine-grained resource usage of the GPU kernel. By scheduling a single kernel at any given time, they lack the ability to utilize the abundant computing and memory resources. While commodity GPUs place their warp scheduling in the black box, we test various scheduling policies and have one observation. If different warps in a thread block of a kernel perform different computations, it allows for parallel utilization of Tensor Cores and CUDA Cores, computing and memory resources. This occurs because multiple warps within a thread block remain active simultaneously. By fusing the Tensor Core kernel and the CUDA Core kernel, or fusing computing-prefer kernel and memory-prefer from different applications, we could improve the resource utilization within the GPU.

In this paper, we propose **Aker**, a kernel fusion and scheduling approach for resolving the false high utilization problem. In order to ensure the required QoS of LC applications when fusing kernels, Aker is comprised of a *static kernel fuser*, a *duration predictor for fused kernels*, an *adaptive fused kernel selector* and an *enhanced QoS-aware kernel manager*. Aker introduces no extra security vulnerability compared with Nvidia MPS [12]. In both Aker and MPS, the original programs launch the kernels, and a server process manages the actual execution. Our contributions are as follows.

- We propose a static kernel fusion method to improve resource utilization without online generation overhead. This method uses the persistent thread block to deal with dynamic inputs, thus avoiding online fusion overhead.
- We propose accurate prediction models for fused kernels to ensure the QoS of LC applications. As a fused kernel runs longer than original LC kernel, we adopt a model-driven predictor to predict the fused kernel's duration.
- We propose an adaptive kernel selection method to search the optimal fused kernel version with maximum throughput gain adaptively. This method is proposed based on the theoretical analysis.
- We propose an online kernel management method to execute fused kernels. It enables kernel fusion with partial computation from original kernels. Based on that, it determines to invoke the original kernel or the fused kernel to maximize the throughput based on QoS targets.

We evaluate the proposed approach on real hardware (Nvidia RTX Ada6000 and V100 GPUs). Our experimental results show that Aker not only ensures the required QoS but also improves the throughput of the BE applications by 50.1% compared with Baymax on average (up to 91.6%).

2 Related Works

In recent years, several schemes have been developed to improve GPU throughput [13], [14]. To achieve better GPU scheduling, Wang *et al.* proposed SMK to exploit block preemption for block-level scheduling [13]. Based on block-level scheduling, SMK improves the system throughput by dividing the resources carefully. Wang *et al.* proposed to scale memory resources to manage memory bandwidth [15] so that an application-aware memory scheduler may be developed [16]. Punyala *et al.* [17] proposed to perform application classification and analyze the per-class interference and slow-down. Then they could find the best matching between classes to maximize the throughput. These methods focus on the SM-level resource allocation and try to minimize the interference based on related metrics.

It is important for ensure QoS (quality of service) in GPU scheduling [18]. Baymax [6] and Prophet [8] exploited MPS scheduling to predict performance interference among co-located GPU applications for a temporally shared GPU. TimeGraph [19] and GPUSync [20] adopted priority-based scheduling to guarantee the performance of real-time kernels. Sedighi *et al.* [21] proposed to optimize the SM allocation between component applications, which could improve the system throughput. Since these works rely on MPS [12] scheduling at the kernel level, they cannot exploit the parallelism from two types of computing cores. Wang *et al.* [7] proposed to employ fine-grained sharing of SM-internal resources to improve QoS. When one of co-located kernels has high priority, it would be scheduled first. These works could only perform the kernel scheduling at the SM level, which could not alleviate the false high utilization problem.

HSM [22] and GDP [23] predicted the slowdown of co-located GPU applications. Compared with Aker, many existing schemes [22], [23] 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. There are also researches [24]–[26] 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. Besides the above researches, there are some researches [27] targets the cluster-level analysis and optimization. They are orthogonal to Aker.

3 Motivation

In this section, we first elaborate on the false high utilization problem. We then discuss the potential to improve the resource utilization within the GPU, and summarize the challenges in exploiting the opportunity.

3.1 The False High Utilization Problem

3.1.1 Tensor Core and CUDA Core

To elaborate on the false high utilization problem, we first conduct an experiment to study the computing core utilization when co-locating the kernels of an LC service and a BE application on a modern GPU. We choose a non-preemptive co-locating strategy Baymax [6] to exploit the idle GPU

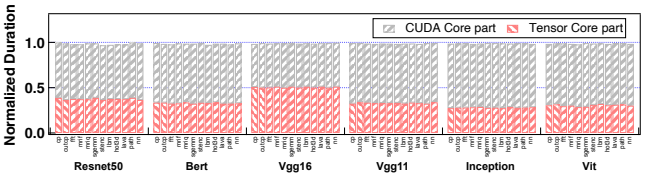


Fig. 2: The active time of the kernels with Baymax.

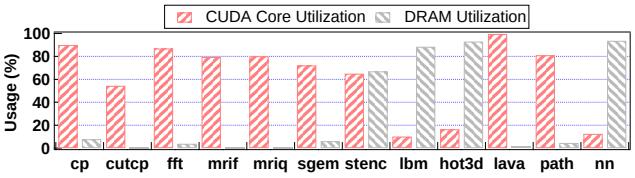


Fig. 3: The kernels' computing core and memory utilization.

cycles from LC services for BE kernels. We use five DNN models ($Resnet50$, $Bert$, $Vgg16$, $Vgg11$, $Inception3$, and Vit) as the LC services, and eight tasks (cp, $cutcp$, fft, $mrif$, $mriq$, $sgemm$, $stencil$, lbm) from Parboil [11] and four tasks ($hot3d$, $lava$, $path$, nn) from Rodinia [28] as the BE applications in the experiment. Each kernel's duration is collected to compute the duration of all the Tensor Core kernels and CUDA Core kernels.

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.

3.1.2 Computing and memory resources

Even for the GPU kernels only using CUDA Core, they may also have different preferences between computing and memory resources. We collect the CUDA Core utilization and DRAM utilization for the eight tasks from Parboil. Figure 3 shows the resource utilization of eight applications. By comparing the utilization of computing and memory resources, we could classify the applications into computing-prefer kernels (cp, $cutcp$, fft, $mrif$, $mriq$, $sgemm$, $lava$, $path$) and memory-prefer kernels ($stencil$, lbm, $hot3d$, nn). As shown from the figure, computing-prefer kernels exhibit 80.5% computing core utilization and 3.15% memory bandwidth utilization. Memory-prefer kernels exhibit 85.6% memory bandwidth utilization and 26.1% computing core utilization. When co-locating two applications using the same computing cores, either the computing core or memory bandwidth is in a low utilization state at any given time.

From the above experiments, we conclude that current strategies generate sequential and interleaving execution of co-located LC service and BE application, which leaves either computing and memory resources at low utilization state. This is referred to as the **false high utilization** problem in this paper. Our study shows that this problem exists widely when co-running LC services and BE applications.

TABLE 1: The normalized duration of the five 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
Bench-D	K_c	K_m	1.05
Bench-E	K_m	K_m	2

3.2 Potential Parallelism Opportunity

We next study the potential to improve the resource utilization within the GPU. We construct several micro-kernels, in which a kernel block has warps for different computations.

For example, we implement a micro-benchmark "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 [29], [30]. K_c has the same grid dimension and block dimension as K_t. 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 is for K_c. We also implement two more benchmarks, "Bench-B" and "Bench-C", as shown in Table 1. These two benchmarks run two K_t and two K_c.

Meanwhile, we implement a micro-benchmark "Bench-D" that fuses a CUDA Core kernel K_c and a CUDA Core kernel K_m into one kernel. K_c and K_m also have the same duration. K_c is a computing-prefer kernel as it performs pure computation with negligible memory operations. K_m is a memory-prefer kernel, in which each thread computes the average number between 10 variables. K_m does not use the shared memory, and each memory access reaches the DRAM. We also implement one more benchmark "Bench-E", which runs two K_m.

Table 1 also shows the processing time of different micro-benchmarks. The time is normalized to the duration of K_t. From the table, the normalized duration 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. **Exploiting the Tensor Cores and CUDA Cores in parallel can effectively improve the overall system throughput.**

As shown from Table 1, the normalized processing time of Bench-D is only 1.05, while that time of either Bench-C or Bench-E is 2. In this experiment, although K_c and K_m all use CUDA Core, they have different preferences over computing and memory resources. This further indicates that the improved execution time of "Bench-D" comes mainly from the parallel usage of computing and memory resources from different kernels. **Kernel fusion between computing-prefer kernel and memory-prefer kernel could also effectively improve the overall system throughput.**

3.3 Challenges in Utilizing Kernel Fusion

However, directly fusing two kernels does not always bring throughput improvement. For example, we choose K_t as the Tensor Core kernel and kernels from Parboil [11] as the CUDA Core kernel. Figure 4 shows the processing time

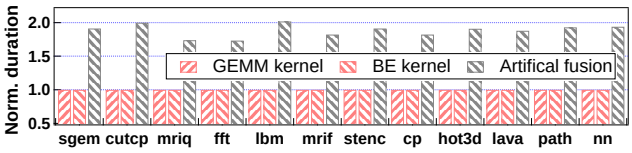


Fig. 4: The duration of fused kernels with artificial fusion.

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 direct kernel fusion brings no throughput improvement. Moreover, the kernel fusion between two CUDA Core kernels from Parboil shows the same experimental results.

Direct kernel fusion's inefficiency comes from the contention for SM resources. Since the fused kernel launches fewer blocks on an SM, both components are slowed down. Even with the flexible resource allocation for two component kernels, there may also exist many possible fusion versions for one kernel pair. How to search for the optimal one is also an open question. Besides, kernel fusion is very likely to introduce a longer return time. Thus, inappropriate fusion may result in QoS violations.

To summarize, there exist four challenges in utilizing the kernel fusion to improve the resource utilization.

- **The kernel fusion has to adapt to dynamic inputs and diverse kernels.** While online fusion methods bring high overhead, a static fusion method needs to adapt to dynamic inputs at runtime.
- **The kernel fusion has to search for the optimal fused kernel version.** Since there exist multiple kernel versions with limited SM resources, it is hard to locate the optimal fused kernel version.
- **The kernel fusion has to quickly and precisely predict the performance of the fused kernel.** It is challenging to make an accurate prediction, as different warps run different computations.
- **The kernel fusion demands QoS-aware online kernel management.** When multiple kernels from LC services and BE applications are available, Aker should identify the fusion decision that maximizes the throughput while ensuring the QoS of LC services.

4 The Aker Design

In this section, we present the Aker 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 5, Aker is a kernel fusion and scheduling approach that consists of a *static kernel fuser*, a *duration predictor for fused kernels*, an *adaptive fused kernel selector* and an *enhanced QoS-aware kernel manager*. The kernel fuser supports the static and flexible fusion for a kernel pair. The kernel pair could be Tensor Core kernel and CUDA Core kernel, or computing-prefer kernel and memory-prefer kernel. The duration predictor exploits a two-stage LR (linear regression) model to predict the duration of fused kernels. When the static kernel fuser could provide multiple fused kernel versions for a kernel pair, the adaptive fused kernel selector searches for the optimal fused kernel. Finally,

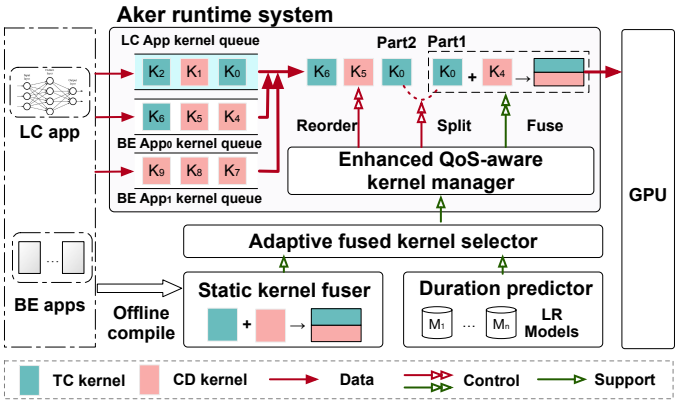


Fig. 5: The design overview of Aker.

the enhanced QoS-aware kernel manager determines the appropriate kernels (original kernel or fused kernel with partial workload fusion) to invoke at runtime.

To efficiently fuse two kernels, Aker 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 perceive the grid dimension online. Since the to-be-fused kernels use a different amount of resources (e.g., thread slots, registers, and shared memory), another challenge is how to enable flexible resource usage between two kernels. (Section 5).

As the fused kernel tends to finish in a longer time (compared to original runs), we need to predict their duration to ensure the QoS of LC services. The challenge here is that the widely-used linear regression for predicting a kernel's latency [8] 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 6).

When the static kernel fuser provides multiple fused kernel versions due to limited resources, Aker needs to locate the optimal fused kernel with maximum throughput gain. Since different kernel versions could show better performance under different inputs, it is hard to determine a fused kernel for all inputs. We resolve this problem by assisting the optimal kernel fusion using kernel split (Section 7).

We maintain a kernel queue for each BE application. The kernels within this kernel queue have temporal dependency. When making the scheduling decision, Aker picks the first kernel in LC kernel queue and checks whether there is a kernel in BE kernel queues can be fused with the picked LC kernel. 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 are executed first. After all LC kernels complete the computation, the kernel fusion with two BE kernels is also considered (Section 8).

Aker can be used to manage long-running LC services in private data centers where all the workloads are known, and Aker has access to the applications' codes. This is similar to those in prior works [6]–[8]. 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, Aker prepares fused kernels for its kernels. The threshold is adjustable.

5 Static Kernel Fusion

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

5.1 Classifying kernels

Before kernel fusion, we need to classify the kernels into different categories. TC kernels and CD kernels could be classified based on the hardware usage. Furthermore, we classify CD kernels into computing-prefer kernels, memory-prefer kernels, and neutral kernels. A kernel with memory bandwidth utilization exceeding 50% is classified as a memory-prefer kernel. When a kernel has computing core utilization greater than 50% and memory utilization less than 50%, it is classified as a computing-prefer kernel. If both two resource utilization of a kernel are below 50%, it is regarded as a neutral kernel. This is because memory-prefer kernels may also have high computing core utilization. In this paper, Aker first considers the kernel fusion of TC kernel and CD kernel, computing-prefer kernel and memory-prefer kernel. Secondly, Aker considers the kernel fusion of computing-prefer kernel and neutral kernel, memory-prefer kernel and neutral kernel.

Some kernels may have different features depending on the input. Specifically, when the input to a kernel is small, it has a relatively smaller thread block number. Consequently, it fails to leverage all the parallelism available on the GPU and might be classified as either a neutral kernel or a computing-prefer kernel. On the other hand, when the kernel has a larger input, it has a larger thread block number. This enables it to utilize all the parallelism on the GPU and might then be classified as a memory-prefer kernel. Faced with the above problem, we only categorize kernels based on their resource usage when the GPU parallelism is fully utilized. This is because 96.1% of GPU kernels make use of all the GPU parallelism in the benchmarks.

5.2 Direct Kernel Fusion

The direct fusion strategy is to fuse the thread blocks of two different kernels into a new block. Figure 6 shows an example process of fusing a CUDA Core kernel (CD kernel for short) and a Tensor Core kernel (TC kernel for short).

In the figure, TC kernel has 2 blocks, each block has 2 warps, and thread id ranges from 0 to 63. 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 is active

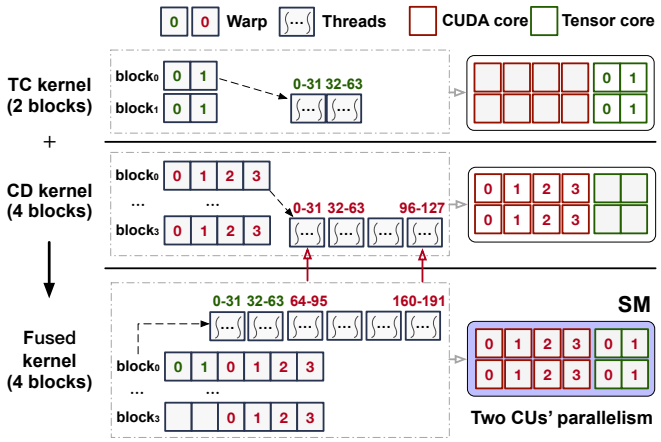


Fig. 6: An example process of direct kernel fusion.

while two warps are idle for the last two blocks. The direct fusion for two CD kernels is the same process.

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

5.3 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 [31], [32]. 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.

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.*

5.4 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 example, assuming there are two CD kernels K_1 and K_2 for fusion. To achieve original performance, K_1 needs 2 persistent block per SM, and each block uses 16KB shared memory; K_2 needs 1 persistent block per SM, and each block uses 32KB shared memory. When K_1 and K_2 are fused, a new block uses 48KB shared memory. In this case, only a block could be issued on one SM when an SM only has 64KB shared memory, and the K_1's performance drops seriously.

Faced with the above problem, we could enhance the fusion with flexible fusion ratio. Figure 7 shows one fusion example for fusing two kernels using a 2:1 ratio. The new block contains two block of K_1 and one block of K_2. After

```
dim3 mix_grid, mix_block;
mix_grid.x = SM_NUM;
mix_block.x = thread_num_cd1 * 2 + thread_num_cd2;

__global__ void new_fused_kernel(...) {
  if ( threadIdx.x < thread_num_cd1 * 1) {
      thread_step = thread_num_cd1 * 0;
      ptb_CD_kernel1_block0(...);
  } else if ( threadIdx.x < thread_num_cd1 * 2) {
      thread_step = thread_num_cd1 * 1;
      thread_id = threadIdx.x - thread_step;
      ptb_CD_kernel1__block1(params, thread_id);
  } else if (threadIdx.x <
     thread_num_cd1 * 2 + thread_num_cd2 ) {
      thread_step = thread_num_cd1 * 2;
      thread_id = threadIdx.x - thread_step;
      ptb_CD_kernel2_block0(params, thread_id);
  }
}
```

CD kernel1 Blocks

CD kernel2 Blocks

Fig. 7: A fused kernel's construct example.

supporting the flexible kernel fusion, the resource utilization on the SM could be maximized.

However, there is still possible that the resources of one SM could not host all the persistent blocks required by two component kernels. Since the resources on the SM have a hard limit, we have to make tradeoffs between two component kernels. Assuming that K_1 and K_2 all need 4 persistent blocks, the SM could support the possible 4:2, 3:3, 2:4 fusion ratios, but not 4:4 ratio. Without any prior knowledge, it is difficult to determine the optimal fusion ratio of a kernel pair. Static kernel fuser first generates all possible fused kernel versions, the fused kernel selector further locates the optimal one (Section 7).

Note that, we have also attempted to fuse three kernels into a single fused kernel. Experimental results reveal that this not only fails to bring about performance improvement but even leads to performance degradation. The reason for this is that the SM resources hardly host all the persistent blocks required by two component kernels. If a third kernel continues to be fused, it will cause severe slowdowns of the two existing kernels. Hence, in this paper, we only focus on kernel fusion with two kernels.

6 Modeling Fused Kernels

In this section, we propose a duration prediction approach that could accurately predict the fused kernel's duration.

6.1 Analyzing The Duration of Fused Kernel

To construct a model for predicting the duration of fused kernels, we study the fused kernel's duration through extensive profiling. Assuming that two kernels K_1 and K_2 are fused into kernel $Fuse_1$. Since the block setup of $Fuse_1$ is static, its duration could only be affected by the computation workloads of two component kernels. These two parts correspond to the original computation time of K_1 and K_2, and we use X_{k1} and X_{k2} 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 K_1's original time, and changing K_1's original time with fixed load ratio.

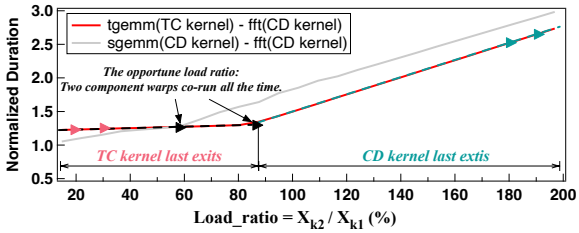


Fig. 8: The fused kernel's duration with changing load ratios, when the component kernel K_1 has fixed workload.

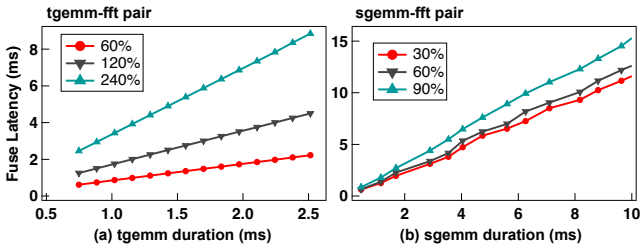


Fig. 9: The fused kernel's duration with fixed load ratios, when the component kernel K_1 has changing original time.

$$Load\_ratio = X_{k2} \ / \ X_{k1} \tag{1}$$

For the first experiment, we fix the K_1's workload, i.e, with static X_{k1}, and model the fused kernel's duration with different workloads of the K_2, i,e, a changing X_{k2}. Figure 8 shows the fused kernel's duration of the *tgemm-fft* and *sgemm-fft* pairs. *tgemm* and *sgemm* are K_1, and *fft* is K_2. *tgemm* uses Tensor Core. *sgemm* and *fft* use CUDA Core. In the figure, the x-axis is the load ratio, and the y-axis is the duration. 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 K_2 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 duration curve before the inflection point, the solo-run kernel in the fused kernel becomes the K_1. 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 K_1, i.e., a changing X_{k1}. We choose several load ratios randomly to show the experimental results better. Figure 9 shows the duration curves for *tgemm-fft* and *sgemm-fft* pair with different load ratios. Each curve in these two figures corresponds to one fixed load ratio. The x-axis is the K_1'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 K_1's original duration while the load ratio is fixed.

Based on the above analysis, we have two observations. *First, the fused kernel's duration shows a two-stage linear regression model, if the K_1's original duration is fixed. Second, when*

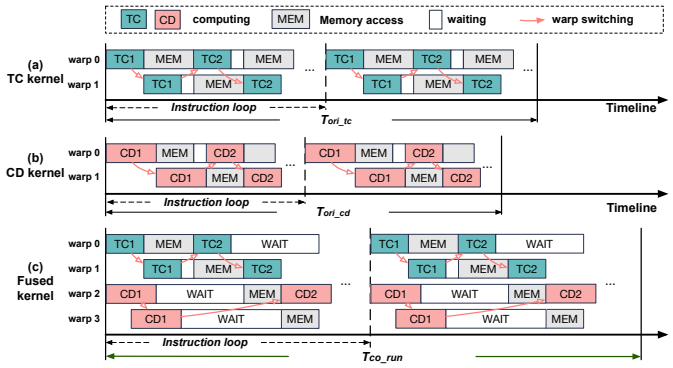


Fig. 10: The warp execution timelines of different kernels.

the load ratio is fixed, the fused kernel's duration has a linear relationship with the K_1's original time.

In addition to the above two observation, we also have another observation from Figure 8. As shown from the figure, the normalized fused kernel's duration is always longer than the LC kernel yet shorter than the serial execution. This is attributable to two reasons. Firstly, the two kernels contend for limited SM resources, which slows down the execution of the kernels. Secondly, since two kernels prefer different resources, the parallel execution brings about the throughput improvement. Since the longer duration may bring the QoS violations, we have already taken this into account in our scheduling method (Section 8.3.1).

6.2 The Two-stage Linear Regression Model

We infer the two-stage linear regression model through warp scheduling. For modern GPUs [33], warps are switched on the SM to hide the computation gap and the switching strategy is deterministic [34], [35]. When multiple warps perform computation alternately, warp switching is triggered by memory access or synchronization.

Assuming that K_1 is TC kernel and K_2 is CD kernel, Figure 10 (a) and (b) show the warp execution timeline of PTB-based K_1 and K_2, 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-based (linear regression) model can precisely predict the duration of PTB-based kernels [8], [22], [23].

Though the block of a fused kernel contains two component warps, they are scheduled with the same strategy. As shown in Figure 10(c), TC warps and CD warps run at the same time as they could not utilize all the resources alone. Due to memory contention, the execution behaviors of two component 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.*

As discussed in Section 6.1, the execution of a fused kernel can be divided into: the co-run of two component kernels, and the solo-run of one component kernel. *While both stages could be predicted using LR, the fused kernel's duration have a linear relationship with one component kernel's original duration if the two kernels have static load ratio.* This corresponds to the second observation in Section 6.1.

Meanwhile, the load ratio also determines the duration of the fused kernel. When a fused kernel has opportune load ratio, two component warps always co-run, and finish at the same time. The opportune load ratio corresponds to the inflection point in Figure 8, and the execution process is shown in Figure 10(c). When a fused kernel has a smaller load ratio, the fused kernel has the additional solo-run stage of K_1. The execution process on the left side of Figure 8 has the case (c) as the first stage and the case (a) as the second stage. When a fused kernel has a larger load ratio, the fused kernel has the additional solo-run stage of K_2. The execution process on the right side of Figure 8 has the case (c) as the first stage and the case (b) as the second stage.

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. This corresponds to the first observation in Section 6.1. We give the formulaic proof for the above conclusions in the conference version [36]. Due to page limitations, we skip this part here.

6.3 Building Duration Models

Based on the above observations, we could predict a fused kernel's duration in three steps. (1) we predict the K_1 and K_2's original time using LR models, which are X_{k1} and X_{k2}. (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 8.

In the step (1), each kernel needs its own LR model. The input is the block number in non-PTB mode, and the output is the kernel's duration, as prior studies [8], [22], [23]. Like these works, we collect runtime input and corresponding performance data when these applications are executed independently. Based on these performance data, we train its duration prediction model for each kernel.

In the step (3), each fused kernel needs its own two-stage LR model. For each fused kernel, we collect its duration in four load ratios: 10%, 20%, 180%, 190%, and build the initial duration model. Furthermore, we use online co-running data to update the model parameters. Whenever the prediction error exceeds 10%, Aker updates the model using online data. Note that, we always set the GPU to the highest frequency to ensure that we obtain accurate and stable experimental results.

7 Searching the Optimal Fused Kernel

As stated in Section 5.4, there may exist multiple fused kernel versions for a kernel pair with different fusion ratios. The blue line and the red line in Figure 11 show the durations of two fused kernel versions of the same kernel pair. Each thread block of $Fuse_1$ contains two persistent blocks of $tgemm$ and two persistent blocks of fft. Each thread block of $Fuse_2$ contains three persistent blocks of $tgemm$ and one persistent block of fft. Since there are different thread blocks in the kernels, these two lines are not overlapped.

As shown from the figure, the kernel pair prefers different fused kernel versions under different load ratios. Under these circumstances, an intuitive idea is to choose the

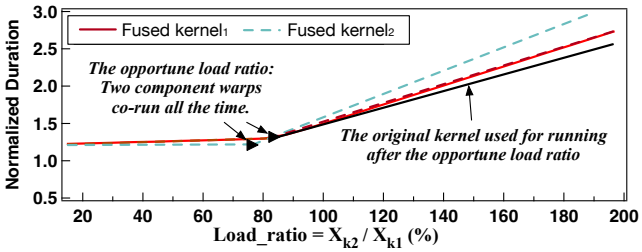


Fig. 11: The duration of different fused kernel version for the same kernel pair (tgemm-fft) with changing load ratios.

best-performing kernel version based on the runtime load ratio. However, such a naive approach brings great runtime selection overhead and storage overhead.

Faced with this problem, we further obtain two observations through extensive experiments. First, the makespan reduction of the fused kernel version reaches its maximum at the inflection point. (The makespan reduction is calculated using two component kernels' serial execution time and the fused kernel's duration.) This is because two component kernels always co-run with the opportune load ratio. Under other load ratios, the fused kernel always needs the solo-run stage after the co-run stage. The solo-run stage could not bring makespan reduction.

Secondly, since the component kernel may not be able to launch enough persistent blocks, the solo-run stage of the fused kernel suffers severe performance degradation. Specifically, $tgemm$ requires 3 persistent thread blocks to attain a similar performance of the original kernel, and fft necessitates 3 persistent thread blocks to achieve a similar original performance. However, the limited resources on the SM only support 2 $tgemm$ blocks and 2 fft blocks for Fuse1. The solo-run stage of the fused kernel could only utilize 2 $tgemm$ persistent thread blocks or 2 fft persistent thread blocks to perform the execution. Therefore, the solo-run stage of the fused kernel suffers from performance slowdown compared to the original kernel.

Under these circumstances, if a fused kernel always exits the computation with the opportune load ratio and uses the original kernel to execute the remaining workload, then this kernel pair can achieve better makespan reduction. The red line and the black line in Figure 11 show the execution time of fused kernel-only and opportune fused kernel plus original kernel respectively. As shown in the figure, the fused kernel exiting with the opportune load ratio has better makespan reduction.

Based on the above two observations, we can locate the optimal kernel fusion version. **Assuming that a fused kernel always exits the computation with the opportune load ratio and the original kernel is used for the remaining workload, the optimal fused kernel version is the one with maximum makespan reduction at its opportune load ratio.**

We further provide proof of the above conclusion. Suppose there are two kernels K_1 and K_2, and two fused kernel versions $Fuse_1$ and $Fuse_2$. The execution time of $Fuse_1$ under the opportune load ratio of $1:X$ is Y. The execution time of $Fuse_2$ is N under the opportune load ratio of $1:M$. The range of X and M is [0,1]. When the runtime load ratio of K_1 and K_2 is $1:R$, the duration of $Fuse_1$ is $Y+(R-X)$,

```
__global__ void ptb_CD_kernel(…, int start_block_id,
                              int end_block_id) {
    for (int block_pos = blockIdx.x + start_block_id;
         block_pos <= end_block_id;
         block_pos += issued_block_num) {

         int i = block_pos;
         ...
    }
}
```

Fig. 12: The PTB implementation enabling kernel split.

and the duration of $Fuse_2$ is $N+(R-M)$. At the same time, we can calculate that the makespan reduction of $Fuse_1$ at the opportune load ratio is $1+X-Y$, and the makespan reduction of $Fuse_2$ at the opportune load ratio is $1+M-N$.

$$\begin{aligned} Y+R-X &> N+R-M \\ 1+X-Y &< 1+M-N \end{aligned} \tag{2}$$

Equation 2 shows one possible comparison results. We find that these two inequalities are completely equivalent. This means that if a fused kernel achieves better makespan reduction at its opportune load ratio, it will achieve better makespan reduction at any load ratio.

Since we obtain the initial two-stage LR model of all fused kernel versions for a kernel pair in Section 6.3, we can further calculate their makespan reduction under their opportune load ratios. Furthermore, we can locate the optimal fused kernel version of this kernel pair. Experimental results in Section 9.6 show that Aker could locate all the kernel pairs benefiting from the kernel fusion, and search the fused kernel version with optimal makespan reduction.

8 Online Kernel Scheduling

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

8.1 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).

8.2 Kernel split implementation

As proved in Section 7, if a fused kernel always exits the computation with opportune load ratio and the original kernel is responsible for the remaining workload, the system throughput will be further improved. This execution process requires splitting one BE kernel's into two kernels: one for kernel fusion and one for original kernel solo-run.

As shown in the Figure 12, the PTB-based kernel could be added with two parameters $start\_block\_id$ and end_block_id to support kernel split. Assuming there is a BE kernel with 1024 blocks and Aker needs to split one BE kernel K_1 into two kernels K_{1-1} with 256 blocks and K_{1-2} with 768 blocks, Aker just needs to launch the kernel K

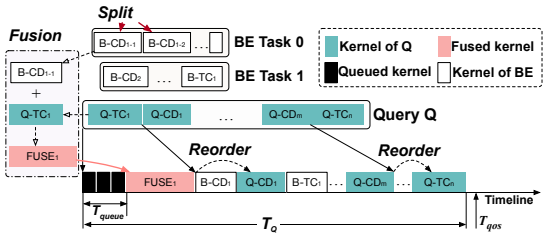


Fig. 13: The online scheduling an LC query Q with Aker.

two times. As for the first kernel launch, $start\_block\_id$ is 0 and end_block_id is 255. As for the second kernel launch, $start\_block\_id$ is 256 and end_block_id is 1023.

8.3 Scheduling Policy

Aker uses both kernel fusion and kernel reorder to maximize the system throughput. The kernel fusion could be assisted by kernel split to maximize the throughput improvement. 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 3 is satisfied.

$$T_Q = T_{queue} + T_{lc} + T_{fuse} + T_{be} \leq T_{qos} \tag{3}$$

The runtime kernel scheduler of Aker decides to perform kernel reorder or fusion for each LC kernel and BE kernel based on Equation 3 as follows.

8.3.1 Calculating QoS Headroom

As discussed above, T_{queue} is known and cannot be reduced when the query Q is launched. Aker 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, Aker 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}. The opportune load ratio requires split CD kernel into two kernels with duration T_{cd-1} and T_{cd-2}. The CD kernel with duration T_{cd-1} is used for kernel fusion and the CD kernel with duration T_{cd-2} is put back to the kernel queue.

Aker then predicts the duration of the kernel fused from the two kernels with opportune load ratio (denoted as T_{k_fuse}). If Equation 4 is satisfied, Aker actually fuses the two kernels and launches the fused kernel. Equation 4 states that the two kernels' fusion could improve the resource utilization inside the SM, and the fused kernel's duration is within the QoS headroom.

$$T_{k\_fuse} - T_{tc} < T_{hr} \tag{4}$$

More specifically, the kernel fusion spends $T_{k\_fuse} - T_{tc}$ to complete the CD kernel, which originally takes T_{cd-1}. After the kernel launch, Aker updates T_{hr} to be $T_{hr} - (T_{k\_fuse} - T_{tc})$.

If all the ready BE kernels may not be fused with the current kernel of Q, Aker 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, Aker fuses the kernels with the highest throughput gain. The throughput gain can be calculated to be $T_{gain} = T_{cd-1} - (T_{k\_fuse} - T_{tc})$. In this equation, $T_{k\_fuse} - T_{tc}$ is the time for Aker to finish the CD kernel, which has original time T_{cd-1}. Aker fuse the kernel of Q with the BE kernel with the largest T_{gain} to maximize the system throughput.

Furthermore, if all the kernels from LC service is launched, Aker also considers the kernel fusion from two BE applications. As long as the duration of the fused kernel is within the QoS headroom, the fused kernel is launched to maximize the system throughput. Also, the fused kernel of two BE kernels also utilize the kernel split to improve the system throughput using opportune load ratio.

8.3.2 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 5 calculates Q's QoS headroom when it is issued. If the T_{hr} of the new query is close to 0, Aker directly launches all the kernels to the GPU.

$$T_{hr} = T_{qos} - T_{queue} - T_{ori\_solo} - \sum_{i=1}^{n} T_{lc\_i} \tag{5}$$

9 Evaluation

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

9.1 Implementation of Aker

To evaluate Aker method, we implement the kernel fuser and the runtime kernel manager. The kernel fuser first transforms all original kernels to PTB mode in a source-to-source way. Second, the kernel fuser generates all possible fused kernel versions for a kernel pair following Section 5.4. Then, for each fused kernel version, the adaptive selector locates the opportune load ratio using only four profiling

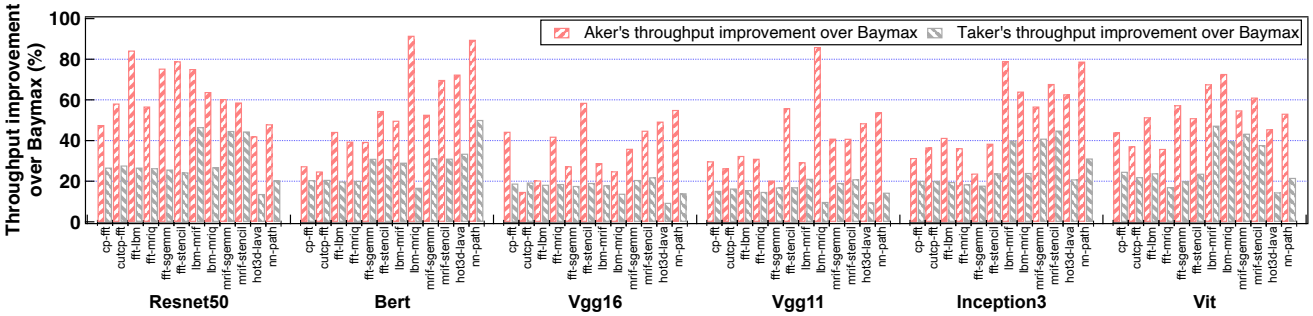


Fig. 14: The throughput improvement of BE applications at co-location with Aker and Tacker.

TABLE 2: Experimental specifications.

CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
GPU	Nvidia RTX Ada6000
Software	CUDA Version: 12.6, CUDNN Version: 9.3
LC Services	Resnet50 (batch size: 64), Bert (256), VGG16 (64) VGG11 (64), Inception3 (64), Vit(48)
BE Apps [11], [28]	cp, cutcp, fft, mrif, mriq, sgemm, stencil, lbm, hot3d, lava, path, nn

points. Based on that, the selector could find the optimal fused kernel version for the kernel pair. Lastly, a dynamic-link library is created for online invocation.

We implement the kernel manager from the scratch. 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. When a user request arrives, we record the current timestamp. Given that a DNN model comprises hundreds of kernels, synchronizing the timestamp upon the launch of each kernel would incur substantial overhead. Hence, we synchronize the timestamp every 10ms. In the interval between two synchronizations, we merely schedule the kernel based on the kernel duration prediction. This enables us to enhance hardware utilization and avoid QoS violations resulting from duration prediction errors.

To implement Aker method in the python-based frameworks like TensorFlow, the fused kernels are compiled into customized operators through *custom-op* [37]. At runtime, Tensorflow invokes the customized or original operators.

9.2 Experiment Setup

Table 2 shows the detailed experimental setup. We use six commonly used DNN models, $Resnet50$, $Bert$, $Vgg16$, $Vgg11$, $Inception3$, and Vit as LC applications; use twelve applications from Parboil [11] and Rodinia [28] as BE applications. The LC applications are generated by the DNN compiler Rammer [32]. The BE applications are categorized into computing-prefer (cp, $cutcp$, fft, $mrif$, $mriq$, $sgemm$, $lava$, $path$) and memory-prefer ($stencil$, lbm, $hot3d$, nn). We use 50ms to be the QoS target, and LC queries arrive in Poisson distribution [38]. The batch sizes of the LC services are the maximum available batch sizes under the QoS target. **All the benchmarks in Parboil use CUDA Cores, and LC applications use both Tensor Cores and CUDA Cores.**

The experiments are mainly carried out on a server equipped with an Nvidia RTX Ada6000 GPU. Aker does not rely on any particular features of Ada6000 and is easy to be set up on other GPUs that integrate Tensor Cores. We also evaluate Aker on an Nvidia V100 GPU in Section 9.9.

9.3 Improving Throughput

In this subsection, we compare Aker with Baymax [6] and Tacker [36]. Baymax improves GPU utilization while guaranteeing the QoS by reordering kernels. Tacker also guarantees the QoS and improves GPU utilization by kernel reorder and kernel fusion. Equation 6 calculates the throughput improvement [6], [39] of Aker and Tacker compared with Baymax. In the equation, T_{Baymax}, T_{Tacker}, and T_{Aker} represent the processing time of BE applications using Baymax, Tacker, and Aker. The throughput improvements only include the results from BE applications as ensuring QoS is sufficient for LC services [6], [39].

$$\begin{aligned} Throughput\ improvement_{Tacker} &= \frac{T_{Tacker} - T_{Baymax}}{T_{Baymax}} \\ Throughput\ improvement_{Aker} &= \frac{T_{Aker} - T_{Baymax}}{T_{Baymax}} \end{aligned} \tag{6}$$

Figure 14 presents the throughput improvement of Aker and Tacker compared with Baymax. From the figure, Aker achieves an average of 50.1% (and up to 91.6%) improvement over Baymax. Tacker achieves an average of 24.3% (and up to 47.4%) improvement over Baymax. Tacker and Aker improve the throughput for all 72 (=6×12) co-location sets. This is because Tacker and Aker exploit both adaptive kernel fusion and kernel reorder, which help to explore the intra-SM parallelism and the idle GPU time. As a comparison, Baymax only utilizes the idle cycles with kernel reorder.

Meanwhile, Aker achieves an average of 25.7% (and up to 74.9%) improvement over Tacker. This is because Aker further optimizes the kernel fusion. It enables the maximum makespan reduction when using kernel fusion. Besides, Aker could exploit the kernel fusion between two CD kernels, which also brings throughput improvement.

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. Aker also has a similar execution trajectory, which also helps Aker to enjoy the intra-SM parallelism using kernel fusion.

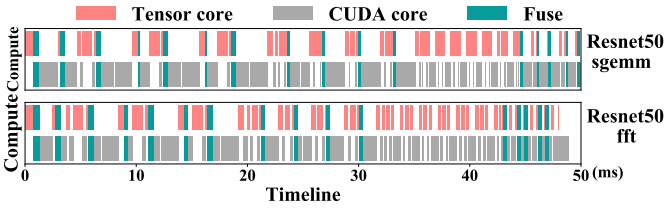


Fig. 15: The active timelines of the two types of cores.

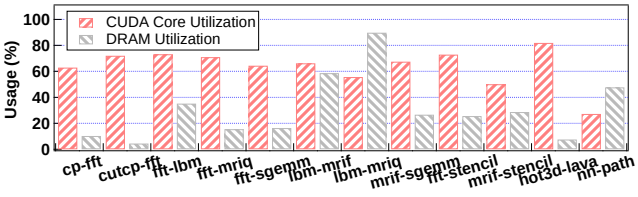


Fig. 16: The computing core and memory utilization of fused kernels from two CD kernels.

In addition, Aker also supports kernel fusion between computing-prefer kernel and memory-prefer kernel, even though two kernels all use CUDA Core. Figure 16 shows the hardware utilization after kernel fusion. Comparing Figure 3 and Figure 16, it is obvious that most fused kernels achieve the high computing core utilization and high memory utilization. Experimental results show that all fused kernels achieve an average of 63.8% computing core utilization and an average of 30.5% memory utilization. Therefore, the fused kernels improve the overall throughput.

9.4 Guaranteeing QoS

Figure 17 presents the 99%-ile of the LC applications under Aker and Tacker in the 72 co-location sets. As shown in the figure, Aker and Tacker ensure the QoS for LC applications under all the co-locations. This is because Aker and Tacker determine whether to perform kernel fusion based on the queries' QoS headroom in the runtime. If there is a possible QoS violation, Aker and Tacker launch the kernels of the LC application directly.

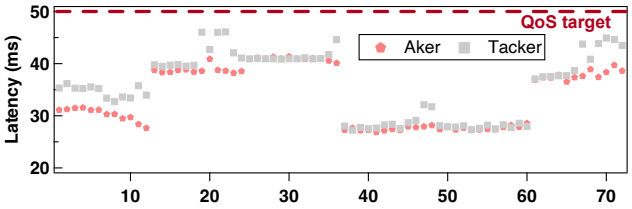


Fig. 17: The 99%-ile latencies of the LC services in all the 50 co-location cases with Aker and Tacker.

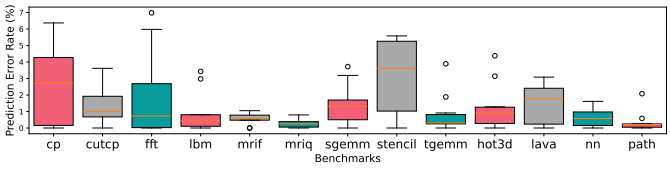


Fig. 18: The duration prediction errors of the PTB kernels.

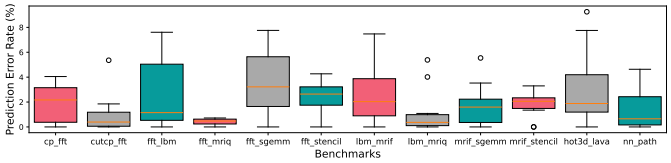


Fig. 19: The duration prediction errors of the fused kernels.

Moreover, LC applications in all the co-locations have different 99%-ile latency, because Aker and Tacker consider the kernel fusion based on the runtime load. When there is no opportunity for kernel fusion, Tacker and Aker complete the execution of the LC application as soon as possible and use the headroom before the next query to execute the BE application. In addition, Aker has smaller 99%-ile latency in all co-locations compared with Tacker. This is because Aker could support the opportune kernel fusion and Tacker has to suffer from the solo-run stage of the fused kernel. Nonetheless, Aker and Tacker all could effectively use the QoS headroom in all the co-locations to run the BE kernels, the 99%-ile latencies of the LC applications are satisfied within the QoS target.

9.5 Accuracy of The Duration Predictor

In this subsection, we evaluate the duration prediction accuracy for fused kernels. As presented in Section 6.1, Aker 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 18 shows the prediction error of these single kernels prediction error. The predicted running time differs from the actual value by at most 7.3%, and the average prediction error is less than 2.8%. Therefore, Aker 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. The experimental results about the fused kernel between TC kernel and CD kernel have been shown in the conference version [36]. In addition, Aker further supports the kernel fusion between two CD kernels. Figure 19 presents the prediction accuracy for these fused kernels. Since Aker enables the opportune kernel fusion using kernel split, Figure 19 shows the prediction accuracy for the opportune kernel fusion. As show from the figure, these models achieve an error rate lower than 8.9%.

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

9.6 Different format

While our main experiments are conducted using FP16 format for DNN models, we add one experiment using INT8 format, we investigate the throughput improvement of Aker and Tacker using INT8 format in this subsection.

Figure 20 shows the corresponding experimental results. As shown, Aker achieves an average of 51.3% (and up to 77.4%) improvement over Baymax. Tacker achieves an

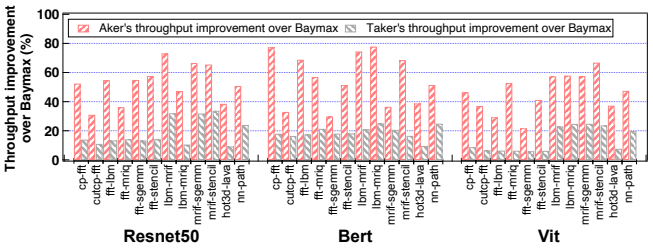


Fig. 20: The throughput improvement of Aker and Tacker under the INT8 format.

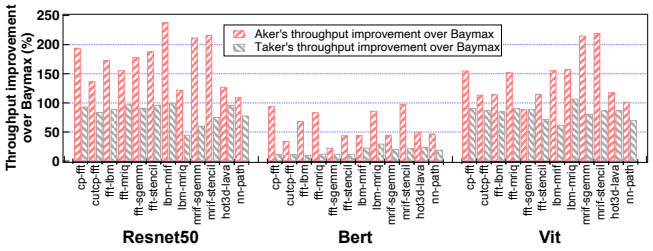


Fig. 21: The throughput improvement of Aker and Tacker under the 30ms QoS target.

average of 17.0% (and up to 33.7%) improvement over Baymax. Aker and Tacker still attain throughput improvement, because they do not rely on the specific format. The throughput improvement is rooted in the parallelism of Tensor Core and CUDA Core and the parallel usage of computing core and memory bandwidth. Therefore, as long as there is system throughput using kernel fusion, Aker and Tacker could improve the system throughput.

9.7 Contrained QoS headroom

In this subsection, we investigate the throughput improvement of Aker and Tacker under a more constrained system. The QoS targets are set as 30ms. While the QoS headroom in Sec 9.3 is 16.1ms on average, the QoS headroom for $Resnet50$ and Vit in this experiment is about 4ms and the headroom for $Bert$ is 11.2ms.

Figure 21 shows the throughput improvement of Aker and Tacker compared with Baymax under the 30ms QoS target. Aker achieves an average of 124.9% (and up to 245.7%) improvement over Baymax. Tacker achieves an average of 62.1% (and up to 99.8%) improvement over Baymax. Aker and Tacker achieve greater throughput improvements, because Baymax has less throughput gain under strict QoS conditions. Meanwhile, Aker's throughput improvement compared with Tacker is reduced. This is because the limited QoS headroom prevents Aker benefiting from the kernel fusion between two BE applications.

9.8 Adapting to Other GPU Generations

Besides RTX Ada6000, Figure 22 shows the throughput improvement of BE applications with Aker and Tacker on a V100 GPU [3]. As observed, Aker increases the throughput of BE applications by 45.7% on average (up to 83.8%), Tacker increases the throughput of BE applications by 24.9% on average (up to 44.3%). This demonstrates that Aker and Tacker could be easily adapted to other GPUs.

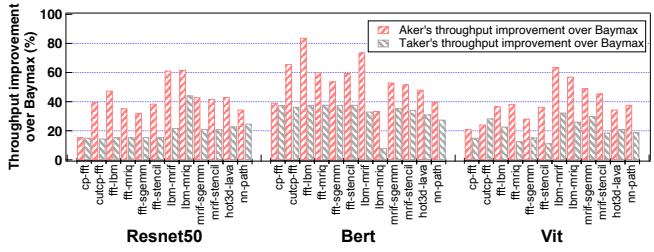


Fig. 22: Throughput improvement on an Nvidia V100.

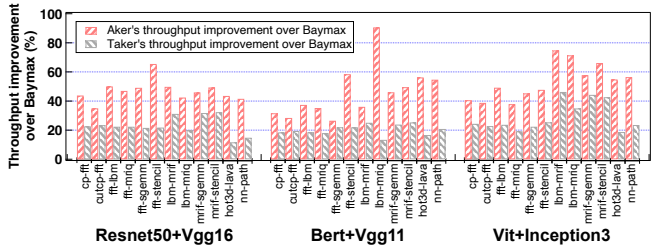


Fig. 23: Throughput improvement co-locating 2 LC tasks.

By comparing Figure 22 and Figure 14, Aker improves the throughput of BE applications more on RTX Ada6000 than on V100. This is because all DNN models have shorter completion time, which brings more idle cycles for Baymax. As the baseline value becomes larger, the performance improvement decreases slightly. In addition, we only need to update the prediction models to deploy Aker on other GPUs, as kernels show different performance on different GPUs. No other update is required.

9.9 Co-location of two LC tasks

In this subsection, we investigate the throughput improvement of Aker and Tacker while co-locating 2 LC tasks and 2 BE tasks. Specifically, we halve the load of all DNN models in Sec 9.3. Figure 23 shows the corresponding experimental results. As shown, Aker achieves an average of 48.7% (and up to 90.6%) improvement over Baymax. Tacker achieves an average of 24.1% (and up to 46.2%) improvement over Baymax. Aker and Tacker still attain throughput improvement, because they are designed to deal with multiple LC tasks. Since they are aware of the QoS target of all LC tasks, they could also utilize kernel fusion to improve the system throughput while guaranteeing the QoS.

9.10 Overhead

Aker brings slight offline overhead and online overhead. As for the online scheduling, Aker 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. When making the scheduling decision, Aker picks the first kernel in the LC kernel queue and checks whether there is a kernel in BE kernel queues that can be fused with the picked LC kernel. Therefore, Aker considers 50 kernel pairs for fusion. This operation takes 1.2 milliseconds. In the same case, we also measure the overhead of the static scheduling by forcing Aker 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.

Aker's offline overhead comes from the kernel fusion process, the optimal kernel search process and the model training process. For a BE application 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. Meantime, the DNN models contain 206 kernels on average. In total, there are 21 types of kernels. While preparing the fused kernels, we generate the 157 fused kernels and save 22 fused kernels that exhibit throughput enhancement. We implement the above process in 840 lines of code.

10 Conclusion

Aker uses kernel fusion to maximize the throughput of BE applications while ensuring the required QoS of LC services. It is comprised of a static kernel fuser, a duration predictor for fused kernels, an adaptive fused kernel selector and an enhanced QoS-aware kernel manager. The kernel fuser enables the static and flexible fusion for a kernel pair. The kernel pair could be Tensor Core kernel and CUDA Core kernel, or computing-prefer CUDA Core kernel and memory-prefer CUDA Core kernel. After preparing multiple fused kernel versions for a kernel pair, the duration predictor precisely predicts the duration of the fused kernels and the adaptive fused kernel selector locates the optimal fused kernel version. At runtime, the kernel manager determines whether to perform the kernel fusion. Aker improves the throughput of BE applications by 50.1% on average (up to 91.6%), while ensuring the required QoS target.

Acknowledgment

This work is partially sponsored by the National Key Research and Development Program of China (2022YFB4501400) and National Natural Science Foundation of China (62302302, 62232011, 62022057, 61832006). Quan Chen is the corresponding author.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] "Cuda mps," https://docs.nvidia.com/deploy/mps/index.html.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] S. R. Punyala, T. Marinakis, A. Komaee, and I. Anagnostopoulos, "Throughput optimization and resource allocation on gpus under multi-application execution," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 73–78.

[18] 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.

[19] 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.

[20] 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.

[21] H. Sedighi, D. Gehberger, and R. Glitho, "Workload-aware dynamic gpu resource management in component-based applications," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2022, pp. 213–220.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] B. Hanindhito and L. K. John, "Accelerating ml workloads using gpu tensor cores: The good, the bad, and the ugly," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 178–189.

[27] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[28] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

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

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 800–813.

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

[38] 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.

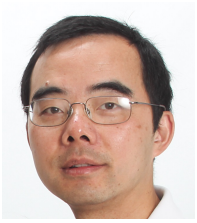
[39] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

Han Zhao is an assistant professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include high performance computing and resource management of accelerators in datacenters. He got his Ph.D. degree at June 2022 from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.

Junxiao Deng received his B.Sc. degree from Huazhong University of Science and Technology, China. He is currently an Ph.D. student in the field of computer science under supervision of Prof. Minyi Guo in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. His research interest is resource management of accelerators in datacenters.

Weihao Cui is an assistant professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include machine learning system and runtime system in datacenters. He got his Ph.D. degree at June 2023 from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.

Quan Chen is a professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include High performance computing, Task Scheduling in various architectures, Resource management in Datacenter. He got his Ph.D. degree at June 2014 from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.



Youtao Zhang is a Professor of Computer Science, University of Pittsburgh, Pittsburgh, USA. He received the Ph.D. degree in computer science from the University of Arizona in 2002. His current research interests include computer architecture and memory systems, and hardware-assisted AI/ML. Prof. Zhang was the recipient of the U.S. National Science Foundation Career Award in 2005. He is a member of ACM/IEEE.

Deze Zeng Deze Zeng received his Ph.D. and M.S. degrees in computer science from University of Aizu, Aizu-Wakamatsu, Japan, in 2013 and 2009, respectively. He is currently a professor in School of Computer Science, China University of Geosciences, Wuhan, China. His current research interests include: network function virtualization, cloud computing, software-defined networking, data center networking.

Minyi Guo received the Ph.D. degree in computer science from the University of Tsukuba, Japan. He is currently Zhiyuan Chair professor and head of the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, pervasive computing, big data and cloud computing. Prof. Guo is an IEEE Fellow and CCF Fellow.