ISPA: Exploiting Intra-SM Parallelism in GPUs via Fine-Grained Resource Management

Han Zhao, Weihao Cui, Quan Chen, and Minyi Guo, *Fellow, IEEE*

Abstract—Emerging GPUs have multiple Streaming Multiprocessors (SM), while each SM is comprised of CUDA Cores and Tensor Cores. While CUDA Cores do the general computation, Tensor Cores are designed to speed up matrix multiplication for deep learning applications. However, a GPU kernel often either uses CUDA Cores or Tensor Cores, leaving the other processing units idle. Although many prior research works have been proposed to co-locate kernels to improve GPU utilization, they cannot leverage the Intra-SM CUDA Core-Tensor Core Parallelism. Specifically, ISPA designs persistent and elastic block to solve the thread slot and shared memory contention between co-located kernels. ISPA also adopts the register allocation method to manage the register contention. These resource management methods are applicable for both white-box kernels and $cudnn$ kernels. Experimental results on an Nvidia 2080Ti GPU show that ISPA improves the system-wide throughput by 15.3% for white-box workloads, and 7.1% for $cudnn$-based workloads compared with prior co-location work.

Index Terms—Intra-SM parallelism, tensor core, GPU

1 Introduction

Numerous applications (for example, physical simulation [1], neuroscience [2], and deep learning [3]) are computationally intensive, and GPUs are commonly used to supply this computational capacity. Nvidia introduced Tensor Cores to accelerate matrix multiplication operations (GEMM operation) since the Volta architecture [4], [5], [6]. Tensor Cores were limited to usage with the GEMM operation. A GPU program can utilize the Tensor Cores by using the appropriate CUDA APIs [7] or *cudnn* library functions [8]. Without these proprietary APIs and *cudnn* kernels, applications that require matrix multiplication cannot take advantage of the Tensor Cores.

The hardware design of a streaming multiprocessor (SM) in today's modern GPUs is depicted in Fig. 1. In general, a GPU contains several SMs (an Nvidia RTX2080Ti GPU, for example, has 68 SMs), and kernels are scheduled to execute on the SMs. CUDA Cores and Tensor Cores are distinct units that share the SM's full memory stack. The general-purpose operation is performed by CUDA Cores, whereas matrix multiplication is accelerated by Tensor Cores.

In general, a GPU kernel is executed in warps (each warp contains 32 threads), and an SM can execute multiple warps concurrently [5]. When a warp's data and computational resources are ready, it starts to run. Thus, if two ready warps each use Tensor Cores and CUDA Cores, they can take use of two hardware's parallelism. However, as illustrated in the left SM of Fig. 1, the current GPU launches all the blocks of a kernel to the SM before the blocks of other kernels. While a single kernel either only uses CUDA Cores or primarily Tensor Cores, one computational resource is squandered. (Tensor Cores require only a little assistance from CUDA Cores, such as C matrix accumulation.)

This paper is aimed at the private datacenter scenario, where all applications' source codes are available, as many previous works [9], [10], [11], [12], [13], [14] have indicated. In private datacenters, multiple users concurrently submit various applications to the GPUs. For example, deep learning applications utilizing Tensor Cores and scientific programs utilizing CUDA Cores may coexist on the same GPU [9], [10], [15], [16]. In this case, by scheduling the kernel blocks as shown in the right SM of Fig. 1, CUDA Cores and Tensor Cores can be used concurrently, significantly increasing processing throughput. Therefore, we propose **ISPA**, which exploits the CUDA Core-Tensor Core parallelism by carefully scheduling the blocks in the kernels of co-located applications.

Apart from ISPA, there are prior works co-locating multiple GPU applications to optimize the throughput [9], [10]. Baymax [9] and Laius [10], for example, co-locate GPU workloads to improve the system throughput while maintaining low latency for high-priority applications. They either reorder GPU task invocations or adjust SM allocations between GPU kernels based on the Nvidia MPS [17] or CUDA stream [18]. However, both MPS and CUDA stream launch kernels sequentially. A kernel's block can be scheduled on an SM only if the resources in the SM have not been used up. Since GPU kernels generally have many blocks to hide the stall cycles due to data access, a kernel's resource occupation makes other kernels' blocks unable to be scheduled on the SM. Therefore, these interfaces cannot directly make use of the intra-SM parallelism due to their unawareness of resource contention.

- *The authors are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {zhaohan_miven, weihao, chen-quan}@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.*

Manuscript received 25 January 2022; revised 11 September 2022; accepted 2 October 2022. Date of publication 12 October 2022; date of current version 7 April 2023.
This work was supported by the National Natural Science Foundation of China under Grants 62022057, 61832006, and 61872240.
(Corresponding Authors: Quan Chen and Minyi Guo.)
Recommended for acceptance by A. R. Alameldeen.
Digital Object Identifier no. 10.1109/TC.2022.3214088

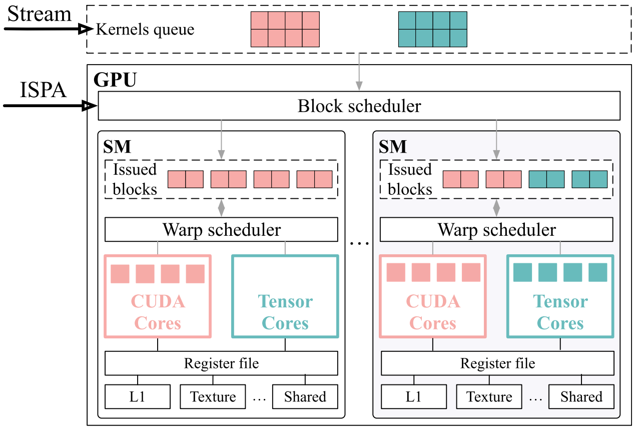


Fig. 1. Difference between ISPA and prior work.

There are four main challenges that must be resolved in ISPA, without modifying the GPU hardware. **Challenge-1:** the hardware driver provides the block scheduling algorithm, which leads to the thread slots contention on the SM. In this case, a mechanism is required to schedule the blocks of different kernels to an SM concurrently. **Challenge-2:** the size of shared memory in an SM is limited. A block cannot be launched when the current blocks already take all the shared memory space. A method is required to tune a block's shared memory usage to enable the intra-SM parallelism. **Challenge-3:** Many applications rely on *cudnn* library for high performance. *cudnn* kernels are black-box and have an extreme register usage. A method is needed to adjust the kernel's register usage. **Challenge-4:** A runtime scheduling strategy is required to carefully adjust the co-running kernels' block setup to maximize the throughput.

ISPA involves compilation and runtime schedule to tackle the four challenges. Specifically, ISPA adopts persistent block to solve GPU kernels' unnecessary thread slots occupation (**Challenge-1**). Moreover, ISPA discovers that the GEMM task's block size is adjustable. A smaller block size brings less shared memory usage. Based on this insight, ISPA proposes an elastic block technique to solve the shared memory contention (**Challenge-2**). Besides, ISPA discovers that DNN applications use the specific internal implementation though *cudnn* kernels have multiple implementations. ISPA locates the specific implementation through profiling the long-running applications. Besides, ISPA adjusts the kernel's register usage through compilation options (**Challenge-3**). Using these above three optimization methods, ISPA provides several versions for each kernel.

Lastly, ISPA uses an online-offline collaborative method to make scheduling decisions (**Challenge-4**). In the offline, ISPA searches the optimal configurations for mainstream kernel pairs and constructs their duration prediction models. When real-system applications arrive randomly, ISPA makes co-running decisions based on offline information and online queue status to maximize the system throughput. Note that, the main insight of ISPA is that the kernels could enjoy the intra-SM parallelism when the resource contention on the SM is solved. Therefore, ISPA utilizes these three resource management methods to exploit the intra-SM parallelism with CUDA stream, which is unaware of the resource contention. Besides, all kernel versions are generated automatically. The only effort for programmers is to check the correctness of elastic-block kernel versions.

The main contributions of ISPA are as follows:

- *Comprehensive analysis of the intra-SM CUDA Core-Tensor Core parallelism.* We identify the factors that impact the CUDA Core-Tensor Core parallelism. The analysis motivates the design of ISPA that maximizes the GPU throughput with co-location.
- *The design of fine-grained resource management techniques.* We could adjust a kernel's issued block number and shared memory usage by adopting the persistent and elastic block. We could also adjust the kernel's register usage using the compilation flag.
- *The in-depth analysis of the cudnn kernel's resource usage and their scheduling method.* We characterize the resource usage characteristics for *cudnn* kernels, and make the scheduling strategy.
- *The pure software implementation without hardware modification.* ISPA is applicable for current in-production GPUs to improve resource efficiency.

We evaluate ISPA on an Nvidia 2080Ti GPU. Our experimental results show that ISPA improves the system-wide throughput by 15.3% for white-box workloads and 7.1% for *cudnn*-based workloads compared with prior work.

2 Related Works

Co-locating applications in datacenters has been an active research area because it can improve the utilization. There are two main directions about the tasks co-location: throughput improvement and quality of service management.

There are prior works focus on improving the throughput of the GPU system. Some works improve the throughput by focusing on the scheduling mode, and other researches target the resource management. For example, SMK [15] enables block-level scheduling by adding the function of block preemption in the GPU. Maestro [19] is proposed to change the multitasking mode for better performance on GPUs dynamically. Besides, many works [20], [21], [22] focus on the SM management in multitasking GPUs. These approaches manage the SM allocation based on classification or prediction. Compared with ISPA, these works use simulators to validate their ideas' effectiveness, which is not supported in in-production GPUs. Besides, they do not consider the case of two computing units, which makes them fail to work.

Some researches focus on the kernel's block dimensions and thread-level parallelism to improve the throughput. For example, DYNCTA [23] allocates fewer blocks for applications suffering from memory bandwidth contention. However, GPU memory bandwidth has kept growing recent years while the maximum TLP remains the same. The memory bandwidth contention between blocks is not a critical problem nowadays. Pai et al. [24] propose elastic kernels to permit fine-grained resource usage. However, this method cannot be applied to the kernels using shared memory.

Quality of service management is also a popular research direction [9], [10], [25]. With the support of MPS scheduling, Baymax [9] predicts performance interference among co-located GPU applications for a temporally shared GPU. Laius [10] predicts the kernel duration and reorders the

TABLE 1
Specifications of an Nvidia RTX 2080Ti GPU

Resource	Value	Resource	Value
Number of SMs	68	Max Threads per SM	1024
Registers per SM	65536	Shared Memory per SM	64 KB

kernel on the spatial multitasking GPUs. TimeGraph [26] and GPUSync [27] use priority-based scheduling to guarantee the performance of real-time kernels. High-priority kernels are executed first if multiple kernels are launched to the same GPU. Since these works all rely on the MPS [17] which is kernel-level scheduling, they could not exploit the two hardware parallelism. All these works focus on high-priority applications' performance, which are inapplicable for throughput problems.

Besides these above researches, there are [28], [29], [30], [31], [32] also works for microbench's performance model development for NVIDIA GPUs, which are orthogonal to our work.

3 Motivation

In this section, we first present the background and the parallelism possibility of Tensor Core and CUDA Core. Second, we identify the constraints of existing scheduling interfaces on co-running tasks, which motivates our work.

3.1 Background and Experimental Setup

We use an Nvidia RTX 2080Ti GPU (Turing architecture) [5] as the experimental platform throughout this paper. Table 1 lists the detailed hardware specification of the experimental platform. In the conference version, we comprehensively validate the two computing units' parallelism by customizing two well-tuned GPU kernels. Briefly, we implement $Kernel_A$ to be a kernel that performs GEMM operation based on the Nvidia sample code [7] using Tensor Cores. We implement $Kernel_B$ to be a kernel that uses CUDA Cores. Both $Kernel_A$ and $Kernel_B$ have 68 blocks with 512 threads, and they have the same solo-run duration.

We use the metric $Makespan\ Reduction$ to measure the degree that the two kernels are processed in parallel. Equation (1) calculates the makespan when co-running two kernels. In this equation, T_1, T_2, and T_{colo} represent the solo-run time of the first kernel, the solo-run time of the second kernel, and the total makespan of completing the two kernels at co-location.

$$Makespan\ Reduction = \frac{T_1 + T_2 - T_{colo}}{T_1 + T_2} \tag{1}$$

When two $Kernel_A$ or two $Kernel_B$ co-run, the makespan reduction is 0. When $Kernel_A$ and $Kernel_B$ co-run, the makespan reduction is 45%. This is mainly because the two kernels' blocks run in parallel on the Tensor Cores and CUDA Cores. *There is potential intra-SM parallelism, if the co-running kernels use different processing units.* Note that, the makespan reduction does not reach 50%. This is because GEMM kernel has the computation part relying on CUDA Core, such as the C matrix accumulation.

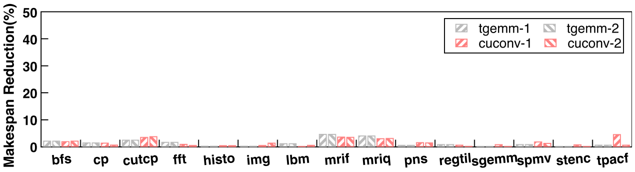


Fig. 2. Makespan reduction of kernel pairs.

3.2 Poor Utilization of the Intra-SM Parallelism

We then investigate whether real-system applications can benefit from the intra-SM parallelism. We refer to the kernel that uses CUDA Cores as *CD kernel*, and the kernel that uses Tensor Cores as *TC kernel* for easing of description. In this experiment, we choose an open-source GEMM kernel used in Nvidia cutlass [33], [34] (*tgemm*) and *cudnnConvolutionForward*() kernel (*cuconv*) as TC kernel. We use all fifteen scientific applications' kernels from Parboil benchmark [35] suite as CD kernel.

Besides, CD kernels' input parameters are all set as default. They have been extensively studied and have stable resource usage. For *cuconv* kernel, we choose the parameters of Resnet50's [36] first two convolution layers with a batch size of 32. Since convolution operation could be transformed to *im2col* operation [37] and GEMM operation, we configure the *tgemm* using the GEMM parameters corresponding to the above two convolutional layers. Fig. 2 shows the makespan reduction of co-running TC kernel and CD kernel using CUDA stream. As shown in the figure, all kernel pairs have no make reduction.

The real-system applications cannot utilize the intra-SM parallelism due to the intrinsic scheduling logic of CUDA stream. Only when all the blocks of a kernel are launched on the SM, and the SM's resources are not used up, another kernel's blocks could be scheduled on the SM. The resources include *thread slots*, *shared memory*, and *register*. Since CUDA stream is unaware of the resource contention, the kernels may execute sequentially due to the contention.

We then collect all kernels' issued block number per SM ("issued blk_num") and the maximum resident block number on SM ("max blk_num"). On this basis, we profile their resource usage on the SM. As shown in Table 2, we have three observations from this table.

First, all kernels launch a large number of blocks to the SM, which far exceeds the maximum resident block number. A launched kernel prevents the latter kernel's block from launching. This is identified as the thread slot contention. Second, 10 of 19 kernels require shared memory, and 3 of 4 TC kernel cases use all the shared memory. If there is no thread slot contention, co-located kernels also suffer serious shared memory contention. Third, *tgemm* kernel with different parameters have the same resource usage, while *cuconv* kernel does not. This is because *cudnn* is the packaged library. It has multiple internal implementations, and the calling logic is based on parameters. Since the internal implementations and calling logics are black-boxed, *cuconv*'s resource usage at runtime is unclear. Moreover, *cuconv* kernels use more than 69% registers with only 25% thread slots. There is severe register contention.

Based on the above analysis, we can conclude that kernel co-location is first limited by the sequential scheduling logic, bringing the thread slots contention. Second, GPU kernels contend for memory resources, such as shared

TABLE 2
Resource Usage of All Kernels

Kernel	max blk_num	issued blk_num	thread slot	shared mem	reg size
tgemm-1	1	3137	25%	**100%**	**50%**
tgemm-2	1	1568	25%	**100%**	**50%**
cuconv-1	**1**	**1568**	25%	64%	69.5%
cuconv-2	**2**	**1568**	25%	**100%**	79.2%
bfs	2	6	100%	39.34%	46.88%
cp	7	3	37.25%	0	41.02%
cutcp	8	258	100%	**25%**	**68.75%**
fft	8	15	100%	25%	**51.56%**
histo	1	3	100%	37.5%	40.63%
img	1	10	100%	**75%**	**60.94%**
lbm	8	30	100%	0	**93.75%**
mrif	4	15	100%	0	**54.69%**
mriq	4	120	100%	0	**53.13%**
pns	3	3	100%	9.38%	**73.83%**
regtil	8	15	100%	0	**90.63%**
sgemm	6	121	100%	4.69%	**91.41%**
spmv	8	16	100%	0	**76.56%**
stenc	8	15	100%	12.5%	**76.56%**
tpacf	3	3	100%	**56.25%**	**58.59%**

The bold values are used to prompt the readers about the three observations.

memory and registers. Third, *cudnn* kernels' resource usage is unclear, and their implementations are black-box. Therefore, the kernel co-location with CUDA stream suffers from low makespan reduction. We propose **ISPA** to exploit the intra-SM parallelism for higher system throughput.

4 Overview of ISPA

To solve the problems in taking advantage of Intra-SM parallelism, we design and implement ISPA. Fig. 3 shows the design overview of ISPA. ISPA targets the GPU scheduling optimization on the private cloud, and administrators have access to mainstream applications' implementations.

As shown in Fig. 3, ISPA is comprised of an *enhanced source-to-source compiler*, a *throughput-oriented strategy maker*, and an *online kernel scheduler*. The enhanced source-to-source compiler designs persistent and elastic block to manage the thread slot and shared memory usage, and also utilizes the compilation flags to manage the register allocation (**Challenge-1&2**). With the compiler's support, the throughput-oriented strategy maker searches the optimal co-running configurations and constructs the duration prediction models for mainstream kernel pairs. Besides the white-box TC kernels, the strategy maker also supports *cudnn* kernels (**Challenge-3**). Finally, based on these scheduling strategies, the kernel scheduler makes real-time scheduling decisions to maximize the GPU throughput (**Challenge-4**). In more detail, ISPA works as follows.

1) The enhanced source-to-source compiler provides three compilation optimization methods. First, the compiler could transform GPU kernels to persistent block mode to resolve thread slot contention. Second, the compiler could generate TC kernels' elastic block versions using smaller block sizes. Third, the compiler supports the register allocation using the *maxrregcount* compilation flags.

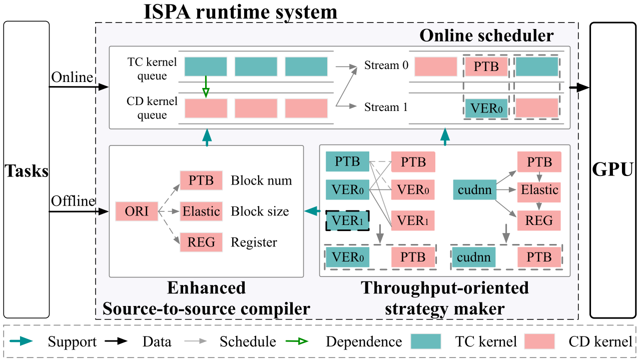


Fig. 3. Design overview of ISPA.

2) We collect all the TC kernels on the cloud and the mainstream CD kernels based on their historical usage. For the kernel pairs with white-box TC kernels, the throughput-oriented strategy maker searches the optimal co-running configurations from all the possible ones. The configuration includes the block size, persistent block number, and register usage. With the optimal co-running configuration, we further construct the duration prediction models. For the kernel pair with *cudnn* kernels, We first obtain these *cudnn* kernels' resource usage, and further adjust the co-located kernel's resource usage to get the maximum throughput.
3) When multiple GPU tasks arrive in real-time, the online kernel scheduler classifies the tasks' kernels into TC kernels and CD kernels. The online scheduler tracks the running kernels' status on the GPU and selects two co-running kernels from different tasks using different hardware.

Through the above scheduling method, ISPA improves the system-wide GPU throughput by exploiting the intra-SM parallelism. Note that, since GPU applications are often stable and long-running, administrators of private clouds generally have access to the mainstream applications' codes, and the offline overhead is acceptable (Section 7). Besides, ISPA could automatically use three resource management methods. All kernel versions are generated by our source-to-source compiler. While the GEMM task using Tensor Cores naturally supports the elastic block, legacy applications with CUDA Cores require programmers' directives. Therefore, the only effort for programmers is to double-check the correctness of elastic-block kernel versions.

5 Source-To-Source Compiler

5.1 Thread Slot Management

5.1.1 Persistent Block

As discussed in Section 3, GPU kernels often use a large number of blocks to hide the stall cycles due to data access, and the co-running kernels contend for the thread slots. To alleviate the slot contention, we adopt the persistent block technique (PTB) [38] to adjust a kernel's resident block number on an SM. The persistent block is abstracted as the block worker, which is permanently resident on the GPU until the kernel completes. Each persistent block is responsible for multiple original blocks' computation. The optimal persistent block number means minimum residency of blocks that

TABLE 3
Optimal PTB Block Number of Different Kernels

	tgemm	bfs	cp	cutcp	fft	histo	img	lbm
max	1	2	7	8	8	1	1	8
opt	1	1	4	8	2	1	1	1
	mrif	mriq	pns	regtil	sgem	spmv	stenc	tpa
max	4	4	3	8	6	8	8	3
opt	1	3	1	4	3	1	4	3

TABLE 4
Resource Usage for Different Kernel Versions

	fft	fft	sgemm	sgemm	tgemm	tgemm
Version	ORI	PTB	ORI	PTB	ORI	PTB
Issue unit (%)	14.9	14.7	29.1	28.8	9.54	9.51
Core (%)	15.3	15.1	41.5	40.9	33.1	32.9
L2 hit rate (%)	50.2	50.2	23.3	48.6	81.9	81.9
Dram (%)	5.0	4.9	74.4	47.1	24.6	24.7

has comparable performance to the maximum occupancy of SM.

For the Parboil benchmarks [35], Table 3 shows the optimal persistent block numbers ("opt") and the maximum resident block numbers ("max") of their main kernels. The optimal persistent block number is profiled using the algorithm in Section 6, and the maximum resident block number is profiled with CUDA interface. As observed, there is a gap between the optimal persistent block number and the maximum resident block number, not to mention the issued block number. *It is not always necessary to launch a large number of blocks for a kernel to achieve high performance.* Based on this observation, we adopt the persistent block technique to resolve the thread slot contention.

Table 4 shows the hardware resource usage of *fft*, *sgemm*, and *tgemm* with two kernel versions. Other kernels also have similar results. As observed, all kernels' two versions have the same core utilization and issue unit utilization. Since the issue unit utilization reflects the instruction execution speed, it directly determines the kernel's duration. Based on that, the PTB-based kernel version could achieve the same performance as the original kernel version. Also, two versions may have different L2 hit rates and DRAM utilization. This is because the PTB-based kernel has fewer active threads, which leads to fewer memory requests.

Some works [23] focus on optimizing kernel performance by tuning thread-level parallelism (TLP). They state that the maximum TLP leads to high amounts of idle time at the cores. The primary reason is high memory access latencies with limited memory bandwidth. This work is fully based on the GPU simulator. However, our experimental results indicate that tuning TLP with PTB for the same benchmark does not bring performance improvement. We have the same results on Nvidia 1080Ti, P100, V100, and 2080Ti. Therefore, our approach does not enjoy any performance improvement from PTB.

We also conduct application-level performance experiments after transforming all the kernels to the PTB version.

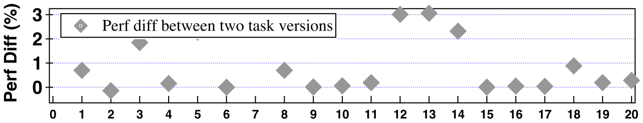


Fig. 4. Performance difference between PTB-based task and original task.

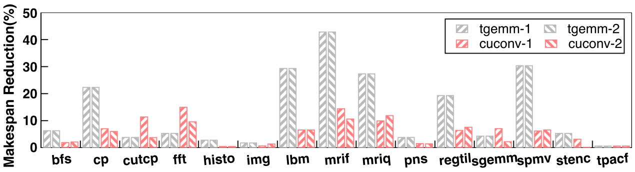


Fig. 5. Co-running two kernels in persistent block mode.

In Parboil, 11 of the 15 applications have multiple kernels. In addition to Parboil, we also choose five commonly used DNN models (Resnet50, ReNext, VGG16, Inception, Densenet) for this experiment. Fig. 4 shows the performance difference between the PTB-based and original tasks. Index 1 - 15 represents the tasks from Parboil, while index 16 - 20 represents five DNN models. As observed, the average performance difference of all tasks is 1.1%. This means that the PTB technique does not bring severe performance degradation. The main reason behind the result is PTB-based kernel has the same instruction issue efficiency.

5.1.2 Automatic Compilation

We use automatic source-to-source compilation to convert a kernel into the PTB version. The compilation process is divided into three steps. First, we identify the original kernel function. Second, we replace the "blockId" in the original function with new variables to ensure correctness. Lastly, we add related logic to loop the original blocks' computation.

5.1.3 Experimental Results

We also conduct the co-running experiments same as that in Section 3. While the *cudnn* kernels are black-box, we only apply the persistent block transformation to the code-available kernels. Fig. 5 shows the makespan reduction of the kernel pairs with the PTB technique.

As observed, the co-running between six CD kernels and *tgemm* has a makespan reduction of 28.8% on average. The co-running between eight CD kernels and *cuconv* reduces the makespan of 9.6% on average. Meantime, other kernel pairs have little makespan reduction. The improved makespan reduction comes from two reasons. First, after the kernels are converted to persistent block mode, they avoid unnecessary thread occupation. Second, the six CD kernels mainly contend for thread slots with TC kernels but not for memory resources.

5.2 Shared Memory Management

While there is no official shared memory multiplexing tool between kernels, we focus on the connection between shared memory size, block size, and performance. *We propose elastic block to solve the shared memory contention by adjusting the block size.*

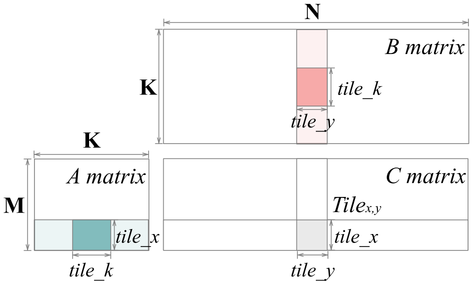


Fig. 6. Matrix mulitplication.

5.2.1 Elastic Block

TC Kernel. Tensor Cores can only perform GEMM task, which has been extensively studied. As shown in Fig. 6, a GEMM task generally divides the result matrix (C matrix) into multiple tiles, and each tile's computation corresponds to one block. At each moment, each block only loads partial A matrix and B matrix into the shared memory due to the space limitation. Each block then slides on the K dimension to complete all computations and obtain the correct result.

$$\begin{aligned} Shared\ mem = &(tile\_x \times tile\_k + tile\_y \times tile\_k) \times sizeof(half) \\ tile\_x \times tile\_k &\propto \quad block\_size \\ tile\_y \times tile\_k &\propto \quad block\_size \end{aligned} \tag{2}$$

Specifically, each block reads the A matrix of $tile\_k \times tile\_x$ and the B matrix of $tile\_k \times tile\_y$ to the shared memory. Equation (2) shows the shared memory usage used by a block. Since the block's threads load the data from the global memory to shared memory collaboratively, we can observe the linear relationship between block size and shared memory usage. According to Equation (2), when the block size reduces, the tile size becomes smaller, and the shared memory usage reduces. We refer to the kernel with a smaller block size as the kernel's elastic block version.

CD Kernel. Since Tensor Cores could only deal with the matrix multiplication, the TC kernels naturally support adjusting the block size. While CUDA Cores support various tasks, it is unknown for CD kernels. We then investigate whether the benchmarks of Parboil [35] have adjustable block sizes. Experimental results show that 13 of 15 benchmarks support block size adjustment. The rest two benchmarks with simple modification also support it.

These GPU kernels have adjustable block sizes because they all belong to one basic programming model, as shown the $Output_1$ in Fig. 7. While GPU programming divides a task into multiple subtasks, each block targets for a subtask. One GPU kernel may have multiple inputs and multiple outputs. As for each block, it performs the computation based on the sub-inputs to obtain the right sub-outputs. We could also calculate each block's shared memory usage, which is shown in Fig. 7. Assuming there are N sub-inputs, we could calculate the overall shared memory usage in Equation (3), similar to Equation (2). Therefore, the shared memory usage has a linear relationship with the block size.

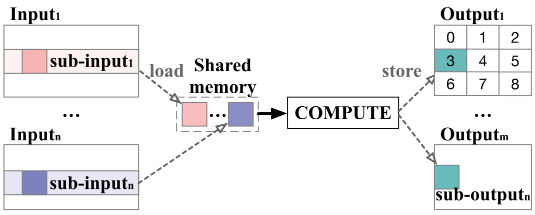


Fig. 7. The basic programming model.

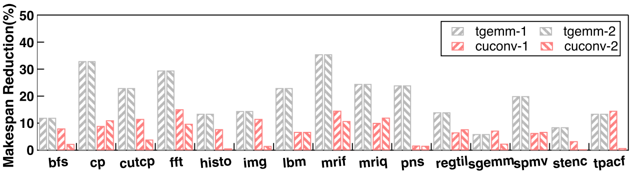


Fig. 8. Co-running two tasks in persistent and elastic block.

$$\begin{aligned} Shared\ mem &= \sum_{i=1}^{N} sub\_input_i \times sizeof(sub\_input_i) \\ sub\_input_i &\propto \quad block\_size \quad i \in [1, N] \end{aligned} \tag{3}$$

Note that, our elastic block technique is different from the *elastic kernel* proposed by Pai et al. [24]. They only focus on the kernels without shared memory usage. Their method is more like the PTB technique, which adjusts the TLP.

5.2.2 Automatic Compilation

We use the automatic source-to-source compilation to generate a kernel's elastic block versions. The compilation process is divided into four steps. First, we identify the macros or constant values that assign the shared memory usage. Second, we identify the variables related to the block dimensions. Third, we locate the variables related to the shared memory values and block dimensions using the abstract syntax tree. Fourth, we modify these variables to create the elastic block version, and give a kernel version with the same result as the original version under the default parameters. The only effort for programmers is to double check the correctness of these elastic-block kernel versions. Besides, since these kernel versions have different shared memory usage, we need to decide to call different kernel version at the runtime. This part is elaborate in Sections 6 and 6.3.

5.2.3 Experimental Results

We then verify the effectiveness of elastic block mechanism. As shown in Fig. 8, the co-running between the remaining nine CD kernels and *tgemm* gains a makespan reduction of 16.1%. The co-running between seven CD kernels and *cuconv* gains a makespan reduction of 10.1% on average. The improved makespan reduction comes from the fine-grained shared memory management. While the elastic block mode uses less shared memory, more parallelism is exploited. Besides, the *cuconv* kernel still mainly relies on GEMM for calculation [34], [39], though these kernels are black-box. If we can have access to their source code, we can also generate the corresponding elastic block version.

5.2.4 Discussion About Elastic Block

1) *Programming mode.* Besides the above mode, there are also other programming modes. For example, some threads in a block are not responsible for the output

computation, while common threads are. These threads go to the specific computation path based on the conditional expression. The expression is related to the thread index, which is generally hard-coded. In this case, adjusting the block size directly may incur the correctness problem of the condition. We further investigate the programming modes in mainstream benchmarks. We choose the kernels from Parboil [35], Rodinia [40], and CUDA official samples. 81.6% of GPU kernels in more than 120 kernels belong to the basic programming model. Since private cloud administrators have access to the applications' implementations, it is easy to check the possibility for elastic block.

2) *Performance degradation.* When the block size reduces, the shared memory usage reduces, which may bring more global memory accesses. More memory accesses may lead to performance degradation. Experimental results show that only bfs and $histo$ in Parboil have a 5.1% performance degradation on average. Besides, we collect the matrix multiplication parameter from five mainstream DNN networks. Experimental results show that 81.2% of input parameters have performance degradation under 7%.

To explain the above results, we take the $tgemm$ kernel as an example to make a qualitative analysis. Original $tgemm$ kernel only has one running block in each SM at each moment. The block is responsible for one result tile. As shown in Equation (4), the result tile's computation time could be divided into four parts: the loading time of A and B matrix, the computation time of the result tile, the store time of the result tile, and other auxiliary times. Since $tile\_x$ and $tile\_y$ are constant for the original kernel, we could derive Equation (5). One tile's computing time is proportional to K.

$$\begin{aligned} T_{origin} &= T_{load} + T_{compute} + T_{store} + T_{others} \\ T_{load} &= mem\_load(tile\_x * K + tile\_y * K) \\ T_{compute} &= compute(tile\_x * tile\_y * K) \\ T_{store} &= mem\_store(tile\_x * tile\_y) \qquad (4) \\ T\_origin &= a \times K + b \times K + c \qquad (5) \end{aligned}$$

When the block size is halved, two blocks perform computation on each SM at each moment. Each block is responsible for one sub-tile. Meanwhile, the halved block size brings halved $tile\_x$ and halved $tile\_y$. Therefore, the original tile is divided into four sub-tiles, and each block needs to complete the calculation of two sub-tiles. Since the sub-tile's computation division is the same as the original tile, we could get the two blocks' computation time and the kernel's duration from Equation (6).

$$\begin{aligned} T_{elastic} &= 2 \times T_{2*sub-tile} \\ &= 2 \times T_{load} + T_{compute} + T_{store} + T_{others} \\ &= 2 \times a \times K + b \times K + c \qquad (6) \end{aligned}$$

Based on Equations (5) and (6), we could derive the performance degradation, as shown in Equation (7).

$$Perf_{diff} = \frac{T_{elastic} - T_{origin}}{T_{origin}} = \frac{a \times K}{a \times K + b \times K + c} \qquad (7)$$

TABLE 5
Resources Limit of Different Kernels

kernel	bfs	cp	cutcp	fft	histo	img	lbm
PEB blk Resource	2.6 Shared	4.5 REG	3.6 REG	4.7 Shared	3.5 REG	4.0 REG	2.6 REG
kernel	mrif	mriq	pns	regtil	sgem	spmv	tpacf
PEB blk Resource	4.5 REG	4.7 REG	2.6 REG	2.7 REG	4.0 REG	3.2 REG	3.1 REG

Based on Equation (7), when K is relatively small, the performance degradation of elastic block is limited. We further investigate the input parameters of the GEMM kernel for all DNN models. 86.3% of K is less than 1152, while $tile\_x$ and $tile\_y$ are 128. Besides, the GEMM kernel uses many memory optimization methods, such as memory coalescing and continuous data loading (use float4 to load half data). These methods reduce the impact of extra memory accesses. Furthermore, while the block size is halved, the queuing of two blocks at the memory controller introduces an implicit pipeline. Since the original kernel only has a block, the memory access and computation could only be performed sequentially. The implicit pipeline also reduces the negative impact. Based on the above three reasons, the performance degradation of the elastic block version is limited. Likewise, the performance degradation of other kernels is also limited.

5.3 Register Management

As mentioned in Section 3, $cuconv$ kernel is prone to bring the register contention. We count the elastic block numbers that all kernels could launch to the SM while co-running with $cuconv$. Meanwhile, we also record the resource type that restricts the kernel from launching more blocks. As shown in Tables 5, 13 in 15 kernel pairs are limited by register contention, consistent with our analysis.

NVIDIA provides two register allocation methods to launch more blocks to the SM and increase the kernel's occupancy. First, an application could optionally provide additional information to the compiler in the form of launch bounds. Launch bounds specify $maxThreadsPerBlock$ and $minBlocksPerMultiprocessor$. If launch bounds are specified, the compiler derives the upper register limit L. Second, an application could optionally add a compiler flag $maxrregcount$ to hard limit the register number used by the kernel. It forces the compiler to rearrange the register usage.

Since the first method targets the solo-run kernel, it cannot set a hard upper limit for the register usage. Therefore, we choose the second allocation method. Note that when the compiler cannot stay below the imposed limit, it will simply spill the register to local memory. These local variables are stored in global DRAM memory, and they can be cached in L1 cache and L2 cache. Therefore, excessive register reduction could bring the memory system pressure. Therefore, we just adjust the register usage to launch one more block instead of the original resident block number.

We perform the experiments with $cuconv$ after the CD kernels are applied with register control. Fig. 9 shows the corresponding experimental results. While there is no register contention for the $tgemm$, the kernel pairs with $tgemm$ do not gain throughput improvement. Besides, four kernel

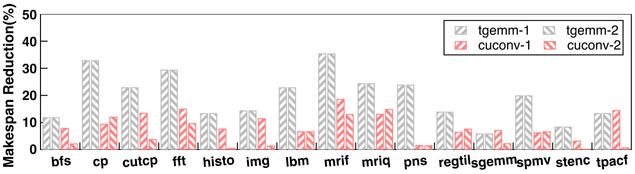


Fig. 9. Co-running two kernels with further register control.

pairs (cp, $cutcp$, $mrif$, $mriq$) with $cuconv$ have further reduce the makespan by 3.1% on average. This is because these four kernels launch one more block on the SM, which better utilizes the computing resources. Therefore, the kernel pairs exploit more parallelism.

6 Collaborative Scheduling

As mentioned in Section 5, we can use three methods to manage thread slot, shared memory, and register usage. However, how the kernel's elastic block version, persistent block number, and register usage are determined at runtime is still unknown. Besides, we are unaware of the $cudnn$ kernels' resource usage. Customizing the scheduling strategy for the $cudnn$ kernel is also a non-trivial problem.

Specifically, three problems should be solved for the runtime kernel scheduling. First, what are the resource usage characteristics of $cudnn$ kernels, and how to perceive their resource usage at runtime? Second, how to determine the kernel's resource usage at the runtime? Third, how to choose the co-running kernel pairs at the runtime?

ISPA uses an online-offline collaborative method to identify the scheduling that results in high system-wide throughput. While the cloud hosts long-running applications, we select mainstream kernels based on their usage, and generate all possible TC-CD kernel pairs. The offline strategy maker first perceives the resource usage of the $cudnn$ kernel used by a DNN service (Problem 1). Second, the strategy maker searches for the optimal persistent block number for all kernel versions. Based on the optimal persistent block number, the strategy maker locate the optimal configuration pair based on the co-running performance for each kernel pair (Problem 2). Third, the strategy maker constructs the duration prediction model for each kernel pair with optimal configurations. Finally, based on the configurations and duration models from the offline, the online scheduler performs online kernel scheduling(Problem 3).

6.1 Cudnn Kernel Profiling

As mentioned in Section 3, the $cudnn$ kernels have different resource usage when using different input parameters. Although the official document does not present any information about the kernel's resource usage, it indicates that the function calls of the $cudnn$ kernel are deterministic. Intuitively, we could record the resource usage of $cudnn$ kernels under all parameters of a DNN service, and customize the scheduling policies for them, respectively. However, such an approach brings severe offline profiling overhead.

We then comprehensively analyze the $cudnn$ kernel's resource usage to reduce the profiling overhead. In the full log from $nsight$, we find that each call has the internal function name it uses. The internal function selection is done by the $cudnn$ kernel. We statistic the internal implementations when the five DNN models in evaluation are configured with BS as 32 and 16. Experimental results show that there are seven internal implementations for these models. Fig. 10 presents the name of an example implementation and its definition rules. Table 6 shows the resource usage of these internal implementations.

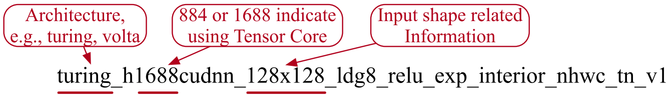


Fig. 10. The definition of internal implementation names.

We have several observations from the above table. First, we could get the internal implementation names through $nsight$, but we cannot control the launch of these kernels. Second, the internal implementation usage is determined by the $cudnn$ kernel, and the number of these implementations is limited. Third, the batch size of a DNN service also determines the $cudnn$ kernels' implementation. Fourth, existing $cudnn$ kernels exhibit similar resource usage characteristics. All kernels use limited thread slots and a large amount of shared memory. Half of the register usage is over 70%.

We maintain two tables to support the $cudnn$ kernel scheduling based on the above observations. The first table records the internal implementation used by a DNN service with different batch sizes. We inquire about the first table to perceive the $cudnn$ kernel's internal implementation at runtime. Besides, the second table record the resource usage of all the possible $cudnn$ kernel's internal implementations. We use these resource usage to tune the CD kernel's configuration for optimal throughput offline.

6.2 Locating the Optimal Persistent Block Setup

While persistent block is needed for resolving the thread slot contention, we need transform all the possible kernel's all the versions (original version or elastic version) to the persistent block mode. Therefore, we design a searching method based on dichotomy to locate the optimal persistent block number (blk_{opt}) for each kernel. The kernel's original performance is used as the baseline and the input for the search process. The searching range for blk_{opt} is between 1 and the maximum resident block number (blk_{max}).

Since the persistent block version's performance with blk_{max} is equal to the original kernel's performance, we skip its performance test and begin with $blk_{test} = (blk_{min} + blk_{max})/2$. If the performance is equal to the baseline, we set blk_{max} to blk_{test} and decrease blk_{test} to $(blk_{min} + blk_{max})/2$. If the performance is worse than the baseline, we set blk_{min} to blk_{test} and increase blk_{test} to $(blk_{min} + blk_{max})/2$. When all possible block numbers are searched, the optimal block number of the kernel is returned. While the max resident block of the SM is 16, each kernel requires up to 4 profiling steps to search for the optimal persistent block number.

TABLE 6
The Resource Usage of $Cudnn$ Kernels

CONV TYPE	T1	T2	T3	T4	T5	T6	T7
Register (%)	69.5	79.3	79.3	67.2	82.8	73.4	76.9
Shared memory (%)	64.0	100	64.0	64.0	100	76.8	76.8
Max DRAM BW (%)	32.5	64.1	42.8	70.3	50.2	41.9	32.2
FP32 utilization (%)	0	0.31	0	0.19	0	0	0

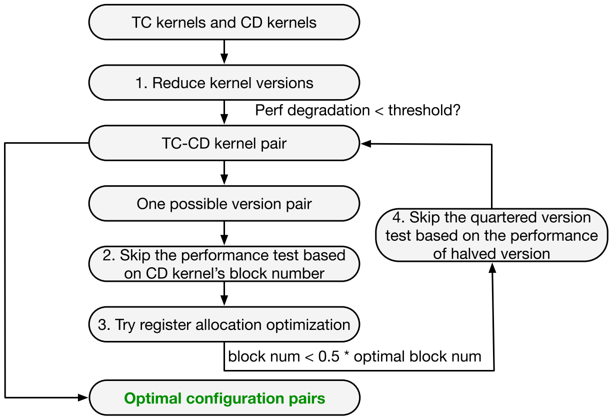

TC kernels and CD kernels
1. Reduce kernel versions
Perf degradation < threshold?
TC-CD kernel pair
One possible version pair
2. Skip the performance test based on CD kernel's block number
3. Try register allocation optimization
block num < 0.5 * optimal block num
4. Skip the quartered version test based on the performance of halved version
Optimal configuration pairs

Fig. 11. Identify the optimal configuration pairs.

6.3 Identifying Optimal Configuration Pairs

Then, we need to customize the scheduling strategy for all possible kernel pairs. For a kernel pair, we need to select each kernel's specific version (original version or elastic version) and its persistent block number. Since elastic block may bring performance degradation, we only create the halved and quartered block versions for each kernel. On this basis, each kernel has three kernel versions, which could be configured with multiple persistent block numbers. Besides, register optimization needs also be considered. A complete search process for the configurations has $O(N^4)$ complexity. Therefore, we design a configuration search method to optimize this search process, as shown in Fig. 11.

Our searching method is divided into four steps. First, we reduce the elastic block version of each kernel. We only focus on the kernels that have performance degradation within the specified threshold. In this paper, we set the threshold to 20%, because the average makespan reduction is about 25%. Second, for a version pair of a kernel pair, we determine the kernel's persistent block number based on resource usage. After the TC kernel's blocks reserve some resources, we calculate the possible CD kernel's block number based on resource slack. If the possible persistent block number is less than half of the optimal block number, we skip the co-running performance test. This is because a kernel's performance is almost halved when the possible persistent block number is half of the optimal one. Third, we consider the register allocation optimization method to enlarge the kernels' parallelism. As discussed in Section 5.3, we could only use the register allocation method to launch one more block on the SM. Fourth, if we do not gain the parallelism using the CD kernel's halved version while launching more than half the optimal block, we no longer perform the test with the quartered version. This is because the halved version already launches enough blocks.

The above searching method also supports *cudnn* kernels. The only difference is that *cudnn* kernels only have the original version. Moreover, each kernel pair requires four searches on average with the search method. Through the above configuration pair search, ISPA identifies the optimal co-running configuration pairs for mainstream TC-CD kernel pairs, and records their makespan reduction.

6.4 Duration Prediction Models

After we locate the optimal configurations for the kernel pairs, we need to consider the co-running decision for the runtime kernel scheduling. Equation (8) shows the throughput gain of a kernel pair. As observed, the kernels' co-running gain is determined by the solo-run durations of two kernels and the co-running duration of two kernels. Therefore, We need to construct the duration prediction models for these three durations for making runtime scheduling decisions to maximize the throughput.

$$Throughput\ gain = T_{seq} - T_{colo} = T_1 + T_2 - T_{colo} \tag{8}$$

For the kernel's solo-run duration prediction, previous works [9], [22] choose Linear Regression from various prediction models. We also choose LR as the duration prediction model due to its high precision. The inputs of LR models are the kernel's grid dimensions, and the output is the kernel's solo-run duration. For the kernel pair's co-running duration prediction, the models proposed by previous works could not provide accurate duration predictions. This is because these models work in cases where the kernel runs exclusively on the SM. Multiple kernels' blocks now could run in the SM simultaneously.

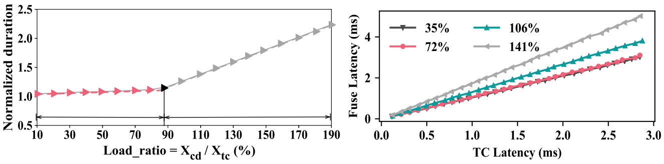
To construct a model for the kernel pair's duration prediction, we need to find a new model for the kernels' co-running duration. Previous works may rely on the hardware counters for duration prediction. However, since performance counters in real-world GPUs are not available at runtime, we could not predict the kernels' co-running duration with the hardware counters. Therefore, we try to study the co-running duration through extensive profiling.

Theoretically, the co-running duration could only be effected by two kernels' load. These two parts correspond to the original time of TC kernel and CD kernel, and we use T_{tc} and T_{cd} to represent them. To simplify the co-running duration modeling from two variables, we then define a metric $LoadRatio$ as $LoadRatio = T_{cd}/T_{tc}$. Based on that, our profiling experiments could be divided into two parts: changing load ratio with fixed TC kernel's load, and changing TC kernel's load with fixed load ratio.

For the first experiment, we fix the TC kernel's load, i.e., with static T_{tc}, and model the kernel pair's duration with CD kernel's different loads, i.e., a changing T_{cd}. Fig. 12a shows the kernel pair's duration of the *tgemm-fft*. The x-axis is the load ratio; and the y-axis is the kernel pair's co-running duration normalized to the T_{tc}. From the figure, the duration fits a two-stage linear regression model.

For the second experiment, we fix the load ratio, i.e., with static $LoadRatio$, and model the kernel pair's duration with TC kernel's different loads, i.e., a changing T_{tc}. Fig. 12b shows the duration curves with random load ratios. The x-axis is the TC kernel's load, and the y-axis is the kernel pair's duration. From the figure, the kernel pair's co-running duration has a linear relationship with the TC kernel's original duration while the load ratio is fixed.

While the kernel pairs with *cuconv* and other CD kernels also show similar profiling results, we have two observations. *First, the kernel pair's co-running 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 kernel pair's co-*



(a) fixed TC load and changing load ratio

(b) changing TC load and fixed load ratio

Fig. 12. The duration of co-running two kernels.

running duration has a linear relationship with the TC kernel's original time.

Therefore, we could predict the kernel pair's duration in three steps. 1) we predict the TC kernel and CD kernel's original time using LR models, which are T_{tc} and T_{cd}. 2) we compute the $LoadRatio$. 3) we predict the kernel pair's duration using the duration models in Fig. 12.

We randomly generate the workload to investigate the prediction accuracy of the duration models. Fig. 13 shows the prediction error of these single kernels prediction error. Index 1 - 15 represents the kernel from Parboil, while index 16 - 19 represents the $tgemm-1$, $tgemm-2$, $cuconv-1$, and $cuconv-2$. The predicted running time differs from the actual value by 2.1% on average and 6.1% at most. Therefore, Tacker is able to use linear regression to predict the kernels' solo-run durations. We also evaluate the kernel pairs' two-stage LR model's prediction accuracy. Fig. 14 shows the prediction accuracy. Index 1 - 15 represents the kernel pairs with $tgemm$, while index 16 - 30 represents the kernel pairs with $cuconv$. These LR models achieve an error rate of 3.7% on average and 8.5% at most.

6.5 Online Scheduling Decision

While the offline strategy maker locates the optimal co-running configurations and constructs the duration models, the online scheduler makes kernel co-running decisions at the runtime. The applications in the datacenter could be categorized into TC tasks and CD tasks. TC tasks contain TC kernels, and may contain CD kernels. CD tasks only contain the CD kernel. Since the kernels in a task have dependencies, we only focus on the co-running of TC kernel and CD kernel from different tasks in this paper.

The online scheduler maintains two kernel queues: the TC kernel queue and the CD kernel queue. When the tasks' kernels enqueue, the scheduler also saves these kernels' dependencies. Based on these kernel dependencies and the kernel queues status, the scheduler performs online scheduling as follows.

First, the online scheduler identifies the possible TC-CD kernel pairs at the moment based on the kernels' dependencies. Second, the scheduler predicts the durations of these kernel pairs. Based on these durations, the scheduler then calculates the throughput gain of these kernel pairs and chooses the kernel pair with the largest one for kernel scheduling. Third, if there is no possible TC-CD kernel pair for this moment, the kernels are scheduled to run with the persistent block mode in sequence.

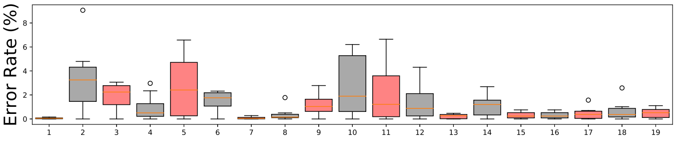


Fig. 13. The kernels' solo-run duration prediction errors.

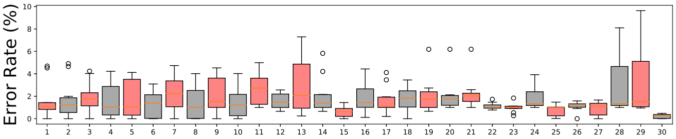


Fig. 14. The kernel pairs' co-running duration prediction errors.

$$\begin{cases} Thread_{TC} * blk\_num + Thread_{CD} * 1 \; < \; THREAD_{limit} \\ Shared_{TC} * blk\_num + Shared_{CD} * 1 \; < \; SHARED_{limit} \\ Reg_{TC} * blk\_num + Reg_{CD} * 1 \; < \; REG_{limit} \end{cases} \tag{9}$$

If some CD kernels have not been profiled with the offline customizer, the scheduler determines the co-running based on resource usage. If a possible TC-CD kernel pair satisfies the condition in Equation (9), one kernel could launch at least one block after the TC kernels' blocks launch. Therefore, the scheduler would schedule these two kernels to co-run for better throughput.

7 Evaluation

In this section, we first evaluate ISPA on the overall throughput, which enjoys the two hardware's parallelism. White-box TC kernels and black-box $cudnn$ kernels are both considered. Second, we evaluate the ideal makespan reduction of the kernel pairs. Third, we will evaluate the makespan reduction for multiple $cudnn$ internal kernels. Fourth, we will evaluate our system on the scheduling scenarios with various tasks. Finally, we discuss the performance difference between $cudnn$ kernels and $tgemm$ kernels, and the overhead of ISPA is discussed in detail.

7.1 Implementation

To evaluate ISPA method, we implement the source-to-source compiler and online kernel scheduler. We have described our source-to-source compilation methods in Section 5. The source-to-source compiler first converts all kernels to the PTB version and provides possible elastic block versions. After that, a dynamic-link library is created for online invocation. At runtime, the kernel scheduler maintains two kernel queues: TC kernel queue and CD kernel queue. Besides, the kernel scheduler maintains a state table for each running task. All kernels have three states in the table, which are waiting, ready and complete. While the kernel scheduler schedules a kernel on the SM, it sets the kernel's state as ready. The scheduler then sets the next kernel's state as ready. At each moment, the scheduler only considers the ready kernels for scheduling.

7.2 Experimental Setup

Benchmarks. We choose five commonly used DNN inference services [42] as the Tensor Core tasks, which are Resnet50, RexNext, VGG16, Inception, and Densenet. Since there are white-box TC kernels and black-box $cudnn$ kernels, we distinguish DNN inference service with $cudnn$ version and $tgemm$ version. For example, Resnet50-C is configured with $cudnn$ kernels, and Renset50-T uses $im2col + tgemm$ kernel to

TABLE 7
Evaluation Specifications

CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
GPU	NVIDIA RTX 2080Ti (68 SMs, 544 Tensor Cores)
OS	Ubuntu 16.04.5 LTS (kernel 4.15.0)
Inference system	Caffe 1.0 [41]
Software	GPU Driver Version: 450.51; CUDA Version: 10.0, CUDNN Version: 7.5

replace the *cudnn* convolution kernel. We use all the fifteen tasks from Parboil [35] as the CUDA Core tasks, which include various GPU tasks from different domains. TC tasks contain both TC kernels and CD kernels, while CD tasks only contain CD kernels. We do not choose other TC tasks from the benchmark suite because mainstream benchmarks do not have the tasks containing TC kernels. Besides, the batch sizes are set as 32.

Hardware and Software. The experiments are carried out on a server equipped with one Nvidia GPU RTX 2080Ti. The elaborate setups are summarized in Table 7. Note that ISPA does not rely on any hardware features of 2080Ti and is easy to serve on other GPUs that integrate Tensor Cores.

7.3 Overall Throughput

In this subsection, we evaluate ISPA's effectiveness in maximizing the throughput. We compare ISPA with CUDA stream. The throughput is the task number completed over a period of time.

Fig. 15 shows the system-wide throughput improvement with ISPA for *tgemm*-based task pairs compared with CUDA stream. As observed from this figure, ISPA improves the throughput in all the 5 * 15 = 75 co-location pairs. ISPA increases the throughput by 15.3% on average and up to 40.3% (*ResNext* and *lbm*). ISPA improves the throughput, because it solves the resource contention between co-running kernels. This allows ISPA to explore the parallelism of the two hardware. On the contrary, although CUDA stream is designed to co-running kernels, it could not utilize the intra-SM parallelism due to resource contention.

Fig. 16 shows the system-wide throughput improvement with ISPA for *cudnn*-based task pairs. As observed from this figure, ISPA improves the throughput in all the 5 * 13 pairwise co-location task pairs. We do not choose *pns* and *stenc* for experiments because these two kernels could not gain throughput improvement while co-running with *cudnn* kernels. ISPA increases the throughput by 7.1% on average and up to 15.6% (*ResNext* and *mrif*). ISPA improves the throughput because of the same reason as the *tgemm*-based task pairs. Since ISPA could solve the resource contention between kernels, it could exploit the parallelism between two hardwares.

Observed from Figs. 15 and 16, we find that the kernel pair with the same DNN service have different throughput improvements for different CD tasks. While VGG16-T has an makespan reduction of 38.4% with *mrif*, it only has 5.3% with *sgemm*. The VGG16-C also has similar results for *mrif* and *sgemm*. This difference comes from the CD task's different demands for the memory system, which includes the memory bandwidth, the bus bandwidth, etc. Therefore, compute-intensive CD tasks could enjoy a high co-running speepdup with *cudnn* kernels, while that of memory-intensive CD tasks are relatively low. Nonetheless, a large number of CD tasks could get throughput improvement, which demonstrates the effectiveness of ISPA.

We can also observe that the networks have throughput improvement differences. While ResNext-T increases the throughput by 17.2% on average, Densenet-T improves the throughput by 12.0%. This difference comes from the different network features due to the network design. Densenet introduces many small matrix multiplications compared to

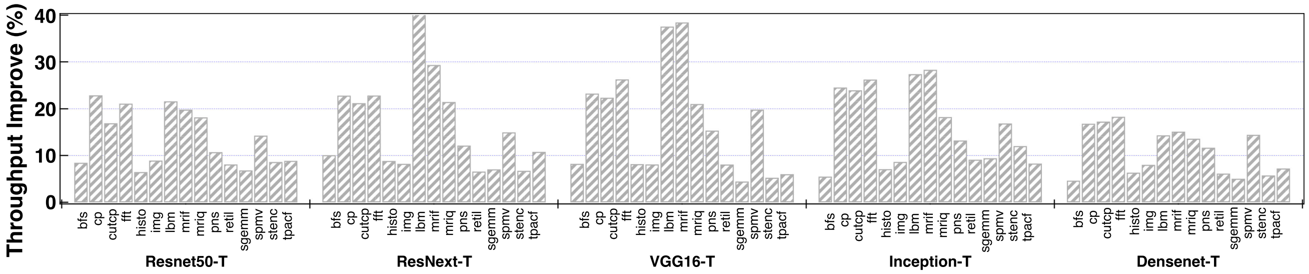


Fig. 15. The throughput improvement of ISPAnormalized to that of CUDA stream (*tgemm*-based services).

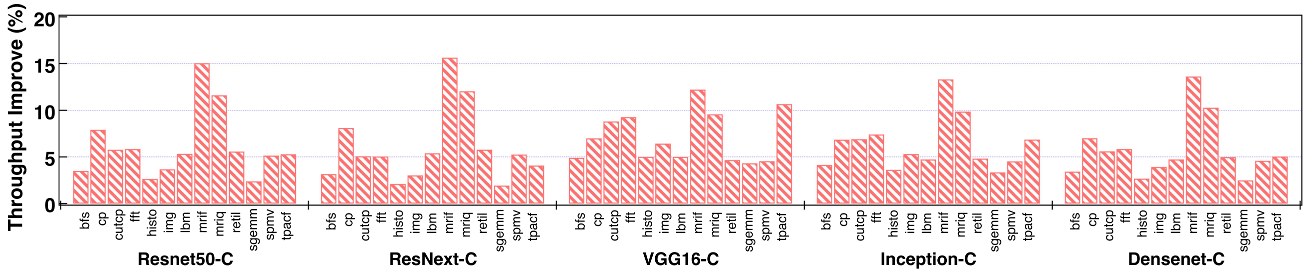


Fig. 16. The throughput improvement with ISPAnormalized to that of CUDA stream (*cuconv*-based services).

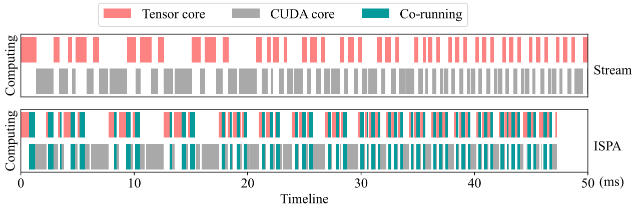


Fig. 17. The active timelines of Tensor and CUDA Cores.

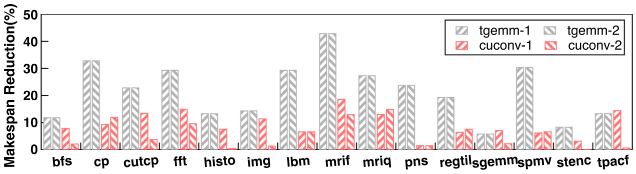


Fig. 18. The final makespan reduction at the kernel level.

other networks. The small matrix multiplication's short running time leads to the little potential for hardware parallelism. Besides, the small matrix multiplication may not occupy all the SM. In this case, CUDA stream could also enjoy the task parallelism, although it could not take advantage of two hardware's parallelism. Nevertheless, ISPA improves the GPU throughput on all the networks.

To better understand why ISPA performs better than CUDA stream, as an example, Fig. 17 shows the execution trace if TC task Resnet50 and CD task fft with ISPA and CUDA stream. In Fig. 17, the first row represents the Tensor Core active time, and the second row represents the CUDA Core active time. In these two rows, we use blue color to represent the co-running time. As shown in the figure, these color bars demonstrate that ISPA utilizes two hardware's parallelism and CUDA stream could not. We do not present the results of the kernel pair with $cudnn$ kernels due to page limitations, which have similar experimental results. Since ISPA could take advantage of these two hardware's parallelism, it improves the system-wide throughput.

We also collect the increased duration of each application. The experimental results show that the duration of each application increases by an average of 56.3%. The increased duration comes from two reasons. First, the makespan of co-running two kernels is longer than each kernel's solo-run duration. Second, when there is no co-running opportunity due to dependencies, the scheduler will try to launch the prerequisite kernel to create the co-running opportunity. This changes the kernel launch order.

7.4 Final Kernel-Level Makespan Reduction

This section proves the final makespan reduction of ISPA after the offline strategy customization. As shown in figure 18, all the kernel pairs with $tgemm$ have a makespan reduction of 21.3% on average and at least 9.1%. Likewise, all the kernel pairs (except pns and $stenc$) with $cuconv$ have a makespan reduction of 10.1% and at least 6.4%. These improved makespan come from the intra-SM parallelism.

The co-running experiments continue from Section 5 to Section 7, where there are three optimization techniques and one offline profiling method. Specifically, the persistent block first solves the thread slot contention. Six CD kernels (cp, lbm, $mrif$, $mriq$, $regtil$, $spmv$) have the best makespan reduction when they co-run with the original $tgemm$ kernel in persistent block mode. Eight CD kernels (cp, $cutcp$, fft, $mrif$, $mriq$, $regtil$, $sgemm$, $spmv$) gain the makespan reduction while co-running with the $cuconv$ kernel.

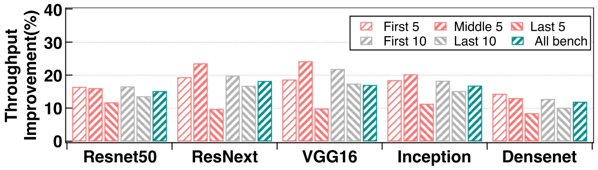


Fig. 19. The throughput of ISPAwith multiple tasks.

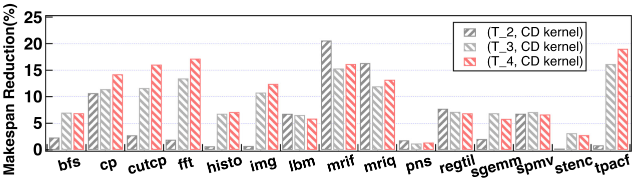


Fig. 20. The final makespan reduction of T_2 to T_4.

Second, the elastic block solves the shared memory contention, which comes from the memory system sharing. Six CD kernels (bfs, $cutcp$, fft, $sgemm$, $stenc$, $tpacf$) achieve the best performance while co-running with the halved block version of $tgemm$ kernel. Three CD kernels ($histo$, img, pns) gain the largest makespan reduction while they are also elastic block versions. Besides, five CD kernels (bfs, $histo$, img, lbm, $tpacf$) have improved makespan reduction while co-running with $cuconv$ kernel after they are transformed to elastic block version. Since elastic block could provide the kernel with fine-grained shared memory usage, two co-running kernels could co-exist on the SM.

Third, the register allocation method could support launching one more block on the SM, which increases the parallelism potential. Due to the fine-grained register allocation, four CD kernels (cp, $cutcp$, $mrif$, $mriq$) have further improved makespan reduction while co-running with $cuconv$ kernel. This indicates that register allocation could exploit the two computing units' parallelism.

Therefore, three methods could solve the three resource contention on the SM for the co-running kernels. Offline strategy customizer searches for the best co-running configurations, which bring the best makespan reduction.

7.5 Cudnn Kernels

As described in Section 6.1, five DNN models only use seven $cudnn$ internal implementations when they are configured with BS as 32 and 16. The first and second $cudnn$ internal implementation is comprehensively discussed in the above subsection. We also apply the three resource management techniques and one offline profiling method on the remaining five $cudnn$ internal implementations. The relative makespan reduction are shown in Figs. 20 and 21.

As observed, all the kernel pairs gain the improved makespan reduction, though the internal implementations have different resource usage. All kernel pairs have a makespan reduction of 9.2% and max speedup of 25.5%. From these two figures, two internal implementations (T_2 and T_5) have relatively low makespan reduction. This is because they require all the shared memory, CD kernels having shared memory demand have no makespan

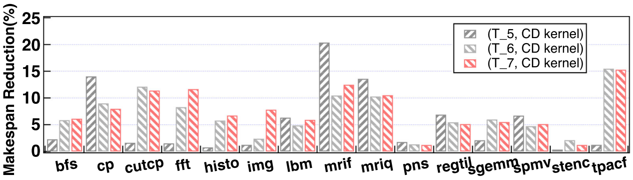


Fig. 21. The final makespan reduction of T_5 to T_7.

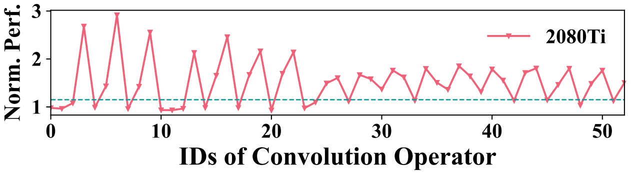


Fig. 22. Normalized performance of $im2col + tgemm$ method over $cuddnConv$ kernel.

reduction with them. Besides, two internal implementations (T_3 and T_4) have higher memory resource slack than the other two implementations (T_6 and T_7). That brings the higher speedup of them than the other two implementations.

7.6 Beyond Pair-Wise Co-Locations

To evaluate the robustness of ISPA in more complex co-locations scenarios, we pick the subsets of CD tasks and co-locate them all with the five TC tasks. The CD task sets include three five-task subsets, two ten-task subsets, and one task set with all the CD tasks. While the kernels are sorted by name, we randomly select the first five tasks, middle five tasks, last five tasks, first ten tasks, and last ten tasks to form the task set. Fig. 19 shows the system-wide throughput improvement with ISPA in these scenarios.

ISPA improves the system throughput by 16.0% on average and 26.2% at most, similar to previous results. This is because although there are more tasks for co-running, their co-running still relies on two hardware's parallel usage. The scheduler is only responsible for choosing the co-running candidate, and has no impact on the co-running configurations. For these tasks, they only perceive their execution, even when they are in parallel with another type of task.

7.7 Cuconv Versus Im2col+Tgemm

In the evaluation, we conduct the experiments with DNN services based on two different convolution methods. The first one is the $cuconvolutionForward()$, the second is the $im2col$ kernel and $tgemm$ kernel [33], [34]. These two methods may have performance differences. Fig. 22 shows the normalized performance of $im2col + tgemm$ implementation over $cuconvolutionForward()$ implementation in Resnet50 with BS 32. As shown, the performance gap between the two implementations is less than 14% for 43.3% of the convolution kernels. By only transforming the kernels with the low performance gap, the entire application has less than 2% performance loss after the transformation. We also choose the $tgemm$ as a performance setup for two reasons. First, we prove our resource management methods' effectiveness through white-box TC kernel and white-box CD kernel co-location. Second, there are many DNN models that rely on the GEMM operation directly, which also motivates our choice.

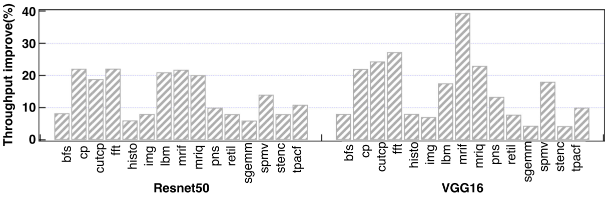


Fig. 23. Throughput improvement of ISPA (caffe 2.0).

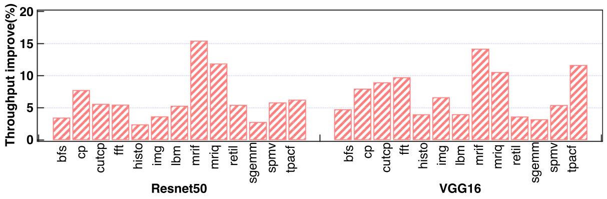


Fig. 24. Throughput improvement of ISPA (pytorch).

7.8 Inference System

The above experiments are based on Caffe 1.0, which seems a bit old. We also conduct experiments on other inference systems. We choose Caffe 2.0 and Pytorch to implement our ISPA system. Figs. 23 and 24 show the experimental results of Caffe 2.0 and Pytorch. Results of two DNN models are shown due to the tight space.

Since Pytorch heavily relies on cudnn kernels to improve performance, we configure Caffe 2.0 with the $tgemm$-based convolution method. As observed from Fig. 23, ISPA improves the throughput by 14.7% on average (up to 39.4%). By comparing Figs. 23 and 15, ISPA improves the throughput of all co-locations in a similar way. This is because the optimization of Caffe 2.0 compared to Caffe 1.0 mainly focuses on four aspects. These aspects include distributed training, mobile deployment, quantized computation, and vast-scale applications. However, these four aspects do not affect kernel scheduling and optimization. Therefore, Caffe 2.0 and Caffe 1.0 have similar results.

As shown in Fig. 24, ISPA increases the throughput by 6.9% on average (up to 15.4%). By comparing Figs. 24 and 16, ISPA also improves the throughput of all co-locations in a similar way. This is because Pytorch heavily relies on cudnn kernels to accelerate the model inference. cudnn kernels could serve mainstream operators in DNN models. DNN models may have similar experimental results on different inference systems. For example, Resnet50 of BS 32 needs about 13 ms to complete the computation under Caffe and Pytorch, which is the same result published by TensorFlow [43]. Therefore, ISPA can improve system throughput in other inference systems.

7.9 Overhead

Our overhead comes from the offline strategy customizer, which includes the $cudnn$ kernel's profiling, white-box kernels' optimal persistent block number determination, and the optimal configuration pair for kernel pairs.

As for the $cudnn$ profiling, we only need to profile each DNN service once to perceive their internal implementation, which generally takes several seconds. If there are new internal implementations, we need to update the table that stores the *cudnn* internal implementations. We then create the new TC-CD kernel pairs with the new *cudnn* kernel. As for the white-box kernels' optimal persistent block number, ISPAcould locate the optimal persistent block number in four profiling steps, which took 50ms on average.

As for the kernel pair's optimal configuration pair, we reduce the profiling overhead through four optimization steps. Therefore, each kernel pair need an average of four profiling steps, which takes 400 ms on average. Besides, since there are limited TC kernels for nowadays tasks, the search overhead is mainly decided by the CD kernel number. Current deep learning frameworks generally have tens of operators, indicating that ISPA's overhead is acceptable.

8 Conclusion

This papers bridges the research gap of improving the utilization of Tensor Core enabled GPU by proposing ISPA. A GPU kernel often either uses Tensor Cores or CUDA Cores, leaving another processing unit idle. ISPA proposes persistent and elastic block to solve thread slot and shared memory contention, and also adopts register allocation methods. Based on these methods, ISPA uses the compilation stage and the runtime schedule to co-locate TC kernels and CD kernels to exploit the intra-SM parallelism. Our experiments show that the throughput of ISPA outperforms existing co-location based solutions by 15.3% for white-box workloads and 7.1% for CUDNN-based workloads.

References

[1] M. J. Harris, "Fast fluid dynamics simulation on the GPU," *SIGGRAPH Courses*, vol. 220, no. 10.1145, pp. 1 198 555–1 198 790, 2005.

[2] J. B. Pezoa, D. Fasoli, and O. Faugeras, "Three applications of GPU computing in neuroscience," *Comput. Sci. Eng.*, vol. 14, no. 3, pp. 40–47, May/Jun. 2012.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] Nvidia volta GPU architecture, 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[5] Nvidia turing GPU architecture, 2018. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[6] Nvidia ampere GPU architecture, 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[7] Nvidia, Tensor core example code, 2017. [Online]. Available: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

[8] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014.

[9] 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.

[10] W. Zhang et al., "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proc. ACM Int. Conf. Supercomput.*, 2019, pp. 58–68.

[11] T. Patel and D. Tiwari, "Clite: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 193–206.

[12] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 167–179.

[13] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2015, pp. 450–462.

[14] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2013, pp. 607–618.

[15] 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 *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 358–369.

[16] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *Proc. ACM Int. Conf. Supercomput.*, 2015, pp. 119–130.

[17] Nvidia, Cuda MPS, 2022. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[18] Cuda stream, 2015. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency

[19] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking GPUs," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 527–540.

[20] W. Zhao, Q. Chen, and M. Guo, "KSM: Online application-level performance slowdown prediction for spatial multitasking GPGPU," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 187–191, Jul./Dec. 2018.

[21] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective sm partitioning in multitasking GPUs," in *Proc. Int. Conf. SuperComput.*, 2018, pp. 65–75.

[22] X. Zhao, M. Jahre, and L. Eeckhout, "HSM: A hybrid slowdown model for multitasking GPUs," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. operating Syst.*, 2020, pp. 1371–1385.

[23] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. IEEE 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 157–166.

[24] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proc. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 407–418.

[25] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-aware dynamic resource allocation for spatial-multitasking GPUs," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2014, pp. 726–731.

[26] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Annu. Techn. Conf.*, 2011, pp. 17–30.

[27] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time GPU management," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 33–44.

[28] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.

[29] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 564–576.

[30] W. Cui et al., "Ebird: Elastic batch for improving responsiveness and throughput of deep learning services," in *Proc. IEEE 37th Int. Conf. Comput. Des.*, 2019, pp. 497–505.

[31] 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 Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1307–1321, Jun. 2021.

[32] H. Zhao et al., "CODA: Improving resource utilization by slimming and co-locating DNN and CPU jobs," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 853–863.

[33] Tensor core example code. [Online]. Available: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/cudaTensorCoreGemm

[34] Nvidia, Nvidia cutlass. [Online]. Available: https://github.com/NVIDIA/cutlass

[35] J. A. Stratton et al., "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center Reliable High-Perform. Comput.*, vol. 127, 2012, Art. no. 27.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[37] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. IEEE 10th Int. Workshop Front. Handwriting Recognit.*, 2006.

[38] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proc. Conf. Innov. Parallel Comput.*, 2012, pp. 1–14.

[39] S. Chetlur et al., "cudnn: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[40] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[41] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.

[42] V. J. Reddi et al., "MLPerf inference benchmark," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2020, pp. 446–459.

[43] TensorFlow, NGC result: ResNet v1.5 for TensorFlow. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_tensorflow/performance

Han Zhao received the BSc degree from Shanghai Jiao Tong University, China. He is currently working toward the PhD degree with the field of computer science under supervision of Dr. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University. His research interests include high performance computing and resource management of accelerators in datacenters.

Weihao Cui received the BSc degree from Shanghai Jiao Tong University, China. He is currently working toward the PhD degree with the field of computer science under supervision of Dr. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University. His research interests include high performance computing and resource management of accelerators in datacenters.

Quan Chen received the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in June 2014. He is a tenure-track associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include High performance computing, task scheduling in various architectures, resource management in datacenter, runtime system and operating system.

Minyi Guo (Fellow, IEEE) received the PhD degree in computer science from the University of Tsukuba, Japan. He is currently Zhiyuan Chair professor and head with 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. He is now on the editorial board for *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing* and *Journal of Parallel and Distributed Computing*. He is a fellow of CCF.