Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks

Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University Shanghai, China
{zhaohan_miven,weihao}@sjtu.edu.cn, {chen-quan,zhao-jieru,leng-jw,guo-my}@cs.sjtu.edu.cn

2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00054

***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. We therefore propose Plasticine to exploit the intra-SM parallelism for maximizing the GPU throughput. Plasticine involves compilation and runtime schedule to achieve the above purpose. Experimental results on an Nvidia 2080Ti GPU show that Plasticine improves the system-wide throughput by 15.3% compared with prior co-location work.**

***Index Terms*—Tensor Core, Scheduling system, Throughput**

I. Introduction

Many emerging applications (e.g., physical simulation [1], neuroscience [2], deep learning [3]) are compute demanding, and GPUs are widely adopted to provide such computational power. For deep learning applications that heavily rely on matrix multiplication operations, Nvidia introduces Tensor Cores to speed up matrix multiplication since Volta architecture [4]–[6], and Tensor Cores could only perform the matrix multiplication task. A GPU program can utilize the Tensor Cores by invoking the corresponding APIs provided in CUDA 9.0 or later [7]. Without the specific APIs, legacy deep learning applications and other applications that do not use matrix multiplication cannot utilize the Tensor Cores.

Figure 1 shows the hardware design of a streaming multiprocessor (SM) in the current GPUs. In general, a GPU has multiple SMs (For instance, an Nvidia V100 GPU has 80 SMs), and the kernels are scheduled to run on the SMs. As shown in this figure, CUDA Cores and Tensor Cores are independent processing units, while they share the entire memory stack in the SM. CUDA Cores are used to do the general-purpose operation, and Tensor Cores are used to speed up matrix multiplication.

In general, a GPU kernel is executed in the granularity of warps (a warp has 32 threads), and an SM can run multiple warps simultaneously [5]. When the data and the computing resources of a warp are ready, it starts to run. Therefore, if two ready warps use Tensor Cores and CUDA Cores, respectively, they could utilize two hardware's parallelism. However, as shown in the left SM of Figure 1, the current GPU launches all the blocks of a kernel (a block has multiple warps) to the SM before the blocks of other kernels. While a single kernel either only uses CUDA Cores or mainly uses Tensor Cores, one computing resource is wasted. (Tensor Cores need insignificant help from CUDA Cores, such as addressing.)

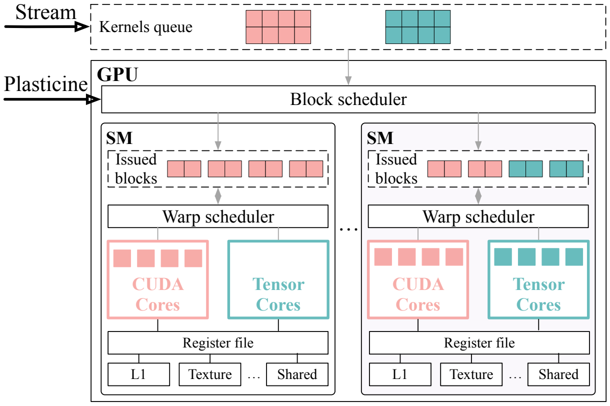

Fig. 1: Difference between Plasticine and prior work.

This paper targets the private datacenter scenario, where the source codes of all the GPU applications are available, and these applications can be profiled offline as many prior works [8]–[13]. In private datacenters, multiple users submit various applications to the GPUs concurrently. For instance, deep learning applications that use Tensor Cores and scientific applications that use CUDA Cores may run on the same GPU [8], [9], [14], [15]. In this case, if we schedule the blocks of the kernels as shown in the right SM of Figure 1, CUDA Cores and Tensor Cores can be used simultaneously, and the throughput can be greatly improved. Therefore, we propose **Plasticine** to exploit the intra-SM CUDA Core-Tensor Core parallelism, by carefully scheduling the blocks in the kernels of the co-located applications.

Besides Plasticine, there are prior research works on co-locating multiple GPU applications to improve GPU resource utilization [8], [9]. For instance, Baymax [8] and Laius [9] co-locate GPU applications to improve the utilization while ensuring the low latency of high-priority applications. They either re-order the GPU task invocations or adjust the SM allocations between the GPU kernels, based on the NVIDIA Multiple Process Service (MPS) [16] or CUDA stream [17]. As shown in the left SM of Figure 1, both MPS and CUDA

Quan Chen and Minyi Guo are the corresponding authors.

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00054

stream techniques operate at the kernel level. Only when a kernel has not used up the resources in an SM, another kernel's thread block can be scheduled on the SM. Therefore, the kernels generally run sequentially on an SM. They are not able to exploit the intra-SM parallelism between CUDA Cores and Tensor Cores.

There are three main challenges that have to be resolved in Plasticine, without modifying the GPU hardware. **Challenge-1:** the block scheduling algorithm is provided by the hardware driver, 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 number of registers and the size of shared memory in an SM are limited (Figure 1). A block cannot be launched when the current blocks already take all the registers or shared memory space, even if there are free thread slots. A method is required to tune a block's memory resource usage to enable the intra-SM parallelism. **Challenge-3:** A runtime scheduling strategy is required to carefully adjust the co-running kernels' block setup to maximize the system-wide throughput.

Plasticine involves compilation and runtime schedule to tackle the three challenges. Specifically, Plasticine adopts persistent block to solve GPU kernels' unnecessary thread slots occupation. Persistent block enables multiple kernels' block residing on the SM, which achieves similar block-level scheduling using the kernel-level interface (**Challenge-1**). Moreover, Plasticine discovers that the GEMM task's block size is adjustable. A smaller block size brings less shared memory and registers usage. Based on this insight, Plasticine proposes an elastic block technique to solve the kernels' memory resource contention (**Challenge-2**). Using the persistent and elastic block, Plasticine provides several versions for each GPU kernel. Finally, Plasticine uses an online-offline collaborative method to make scheduling decisions (**Challenge-3**). In the offline, Plasticine searches the optimal configurations for mainstream kernel pairs and records their overlap rates. When real-system applications arrive randomly, Plasticine makes co-running decisions based on offline information and online queue status to maximize the system throughput.

The main contributions of Plasticine 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 Plasticine that maximizes the system-wide GPU throughput with co-location.
- **The design of the persistent and elastic block.** Adopting this technique, we can adjust the number of blocks from each kernel and the resource usage of each thread block at kernel co-location.
- **The pure software implementation without hardware modification.** Plasticine is applicable for current in production GPUs to improve resource efficiency.

We evaluate Plasticine on an Nvidia 2080Ti GPU. Our experimental results show that Plasticine improves the system-wide throughput by 15.3% on average, and up to 40.3% compared with prior work.

II. 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 [18]. Some works improve the throughput by focusing on the scheduling mode, and other researches target the resource management. For example, SMK [14] 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 previous [20], [21] research works rely on heuristics to manage memory bandwidth to do careful task scheduling. In addition to the above research, many jobs [22]–[27] work on the SM management in multitasking GPUs. These approaches infer the performance impact of SM allocation based on related metrics. KSM [28] and Themis [29] predict the slowdown of co-located applications on spatial multitasking accelerators. Compared with Plasticine, these works use simulators to validates 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 without perceiving new hardware features.

Quality of service management is also a popular research direction [30], [31]. With the support of MPS scheduling, Baymax [8] predicts performance interference among co-located GPU applications for a temporally shared GPU. Laius [9] predicts the kernel duration and reorders the kernel on the spatial multitasking GPUs. TimeGraph [32] and GPUSync [33] 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 [16] scheduling which is kernel-level scheduling, they could not exploit the two hardware parallelism. Wang et al. [34] use fine-grained sharing of SM-internal resources to improve QoS, while FGPU [35] adopts careful memory isolation using page color to ensure the performance of applications. All these work mainly focus on ensuring high-priority applications' performance, which is not applicable for pure throughput problems.

Besides these above researches, there are [36]–[40] also works for microbench's performance model development for NVIDIA GPUs. These works are orthogonal to our work.

III. Motivation

In this section, we first validate two processing units' parallelism. Then, we identify the constraints of existing scheduling interfaces on co-running tasks, which motivates our work.

A. Background and Experimental Setup

With the development of deep learning, the complexity and the size of neural networks keep growing. Nvidia integrates Tensor Cores to deliver better performance for matrix multiplication in DL applications after Volta architecture. Each Tensor Core performs up to 64 floating points fused multiply-add

TABLE I: 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

TABLE II: Experimental results of CUDA stream.

$Task_A$ – Tensor Core kernel $Task_B$ – CUDA Core kernel	T_1	T_2	T_{colo}	Overlap Rate
$Task_A$ + $Task_A$	1	1	2	0
$Task_B$ + $Task_B$	1	1	2	0
$Task_A$ + $Task_B$	**1**	**1**	**1.1**	**45%**

(FMA) operations per clock using FP16 inputs. For instance, a V100 GPU (Volta architecture) integrates 640 Tensor Cores in 80 SMs, and an RTX 2080Ti GPU (Turing architecture) [5] integrates 544 Tensor Cores in 68 SMs. We use an Nvidia RTX 2080Ti GPU as the experimental platform throughout this paper. In an SM, the theoretical processing ability of Tensor Cores is 4X that of CUDA Cores. It is not efficient to totally waste the computational ability of CUDA Cores, even if a kernel only uses Tensor Cores. Table I lists the detailed hardware specification of the experimental platform.

B. Potential Intra-SM Parallelism

Current parallel interfaces (CUDA stream [17] and MPS [16]) launch all the blocks of a kernel before the blocks of another kernel. They cannot schedule two blocks that use different processing units to an SM at the same time. To this end, we implement two well-tuned GPU kernels ($Task_A$ and $Task_B$) to validate whether the Intra-SM parallelism can be achieved in real-system GPU.

We implement $Task_A$ to be a kernel that performs GEMM operation based on the Nvidia sample code [7] using Tensor Cores. We implement $Task_B$ to be a kernel that uses CUDA Cores. It performs pure computation using registers and does not perform any memory operations. This eliminates the impact of the memory access contention on experimental results. Both $Task_A$ and $Task_B$ have 68 blocks with 512 threads in each block, and they have the same solo-run processing time. In this case, each block is assigned to an SM, if the task runs alone on the GPU (The GPU has 68 SMs).

We then run $Task_A$ and $Task_B$ in different co-location pairs, and collect the makespan of completing the two tasks. We use the metric *Overlap Rate*, to measure the degree that the two tasks are processed in parallel. Equation 1 calculates the overlap rate of $Task_A$ and $Task_B$. In this equation, T_1, T_2, and T_{colo} represent the solo-run time of the first task, the solo-run time of the second task, and the total makespan of completing the two tasks at co-location. The overlap rate ranges from 0 to 50%.

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

Table II shows the normalized makespan when we run $Task_A$ and $Task_B$ in different co-location pairs using CUDA stream. As shown in the table, when two $Task_A$ or two $Task_B$ co-run, the overlap rate is 0. This is because they use the same processing units (Tensor Cores or CUDA Cores). Since the 512 threads of a block in a task already occupy most of the computing resources it needed, another block queues up.

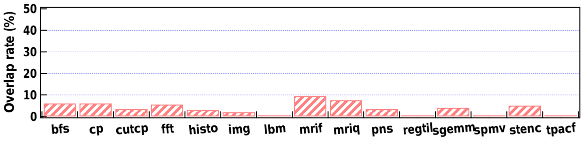

Fig. 2: Overlap rates of co-running CD tasks with the TC task.

TABLE III: Resource usage of tasks.

Task	max resident block	issued block	thread slot	shared mem	reg size
GEMM	1	94	25%	**100%**	**50%**
bfs	2	6	100%	39.34%	46.88%
cp	7	3	37.25%	0	41.02%
cutcp	8	258	100%	**50%**	**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%**

On the contrary, when $Task_A$ and $Task_B$ co-run, the overlap rate is 45%. This is mainly because the blocks of $Task_A$ and $Task_B$ 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.**

C. Poor Utilization of the Intra-SM Parallelism

We then investigate whether real-system applications can benefit from the Intra-SM parallelism. In this experiment, we co-locate an Nvidia official GEMM kernel (used in Nvidia cutlass [41], [42]) that uses Tensor Cores, with scientific applications that use CUDA Cores from Parboil benchmark suite [43]. The two applications are also co-located using the CUDA stream interface. We refer to the kernel that uses CUDA Cores as *CD task*, and the kernel that uses Tensor Cores as the *TC task* for easing of description.

Figure 2 shows the overlap rates of the co-located applications. As shown in the figure, the overlap rates in 4 out of the 15 co-location pairs are close to 0, and the rest pairs also have a low rate. The real-system applications are not able to efficiently utilize the intra-SM parallelism due to the kernel-level scheduling of CUDA stream. Only when all the blocks of a task are launched on the SM, and the SM's resources are not used up, another task's blocks could be scheduled on the SM. The resources include *thread slots*, *registers*, *shared memory*, etc (Table I). If any blocks of a task are queuing on the SM or any of the above resources are used up, another task's blocks could not be scheduled on the SM.

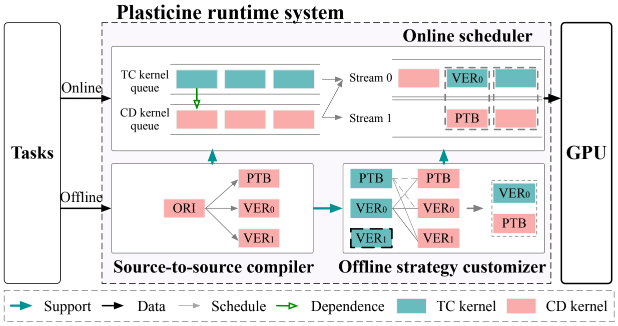

Fig. 3: Design overview of Plasticine.

We first collect all tasks' issued block number per SM and the maximum resident block number on SM. On this basis, we profile their resource usage on the SM. As shown in Table III, all tasks launch a large number of blocks to the SM, which far exceeds the maximum resident block number. This prevents the co-running task's block from launching on the SM. Besides, 14 of 16 tasks occupy 100% thread slots, 10 of 16 tasks require shared memory, and 13 of 16 tasks use more than half registers. It is easy to infer that there are severe contentions in thread slots, shared memory, and registers.

Based on the above analysis, we can conclude that GPU tasks at co-location are first limited by the kernel-level scheduling interface, which brings the thread slots contention. Secondly, GPU tasks still contend for memory resources, such as shared memory and registers. These contentions lead to the three challenges elaborated in Section I, and **Plasticine** has to solve them to exploit the intra-SM parallelism.

IV. Overview of Plasticine

Figure 3 shows the design overview of Plasticine that is comprised of a *source-to-source compiler*, an *offline strategy customizer*, and an *online kernel scheduler*. The source-to-source compiler enables approximate block-level scheduling from the compilation layer, with the current kernel-level scheduling interface. It transforms GPU kernels to persistent and elastic block versions for solving the resource contention (**Challenge-1&2**). Based on the transformed kernels, the strategy customizer searches the optimal co-running configurations for the kernel pairs. Based on these scheduling strategies, the kernel scheduler makes real-time scheduling decisions to maximize the GPU throughput (**Challenge-3**). In more detail, Plasticine works in the following steps.

1) The source-to-source compiler transforms all the kernels to persistent block mode automatically for resolving thread slot contention. Moreover, in order to alleviate the memory resource contention, the compiler generates several elastic block versions for TC kernels based on the persistent block version. The transformation does not seriously hurt the performance (will be discussed later), but alleviates the resource contention.

2) For the potential pairs of TC kernels and CD kernels, the offline strategy customizer searches the optimal co-running configurations, including the block sizes and persistent block setups for two kernels. Specifically, the strategy customizer filters the possible configuration pairs, and selects the one with the best overlap rate.

TABLE IV: Resources usage of different kernels.

kernel	gemm	bfs	cp	cutcp	fft	histo	img	lbm
max blk_num	1	2	7	8	8	1	1	8
opt blk_num	1	1	4	8	2	1	1	1
kernel	**mrif**	**mriq**	**pns**	**regtil**	**sgem**	**spmv**	**stenc**	**tpacf**
max blk_num	4	4	3	8	6	8	8	3
opt blk_num	1	2	1	1	2	1	4	3

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, Plasticine improves the system-wide GPU throughput by exploiting the intra-SM CUDA Core-Tensor Core parallelism. Note that, since GPU applications are relatively stable and long-running in a private datacenter, the overhead of offline customizer is acceptable.

V. Persistent and Elastic Block

We design *persistent and elastic block* to run the kernels. Specifically, persistent block solves thread slots contention with kernel-level scheduling, and elastic block solves memory resources contention.

A. Resolving Thread Slot Contention

As discussed in Section III, 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 [44] that is capable of adjusting a kernel's resident block number on an SM. The persistent block is abstracted as the block worker, which is always resident on the GPU until the kernel completes. Each persistent block is responsible for multiple original blocks' computation. The optimal persistent block number is the smallest block number that achieves the same performance as the original kernel.

For the Parboil benchmarks [43], Table IV shows the optimal persistent block numbers ("opt blk_num") and the maximum resident block numbers ("max blk_num") of their main kernels. The optimal persistent block number is profiled using the algorithm in Section VI, and the maximum resident block number is profiled with interface *cudaOccupancyMaxActiveBlocksPerMultiprocessor()*. 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 resident blocks for a kernel to achieve high performance.**

Figure 4 shows the overlap rates of the benchmarks with the GEMM task, after they are transformed to persistent block mode. As observed, the overlap rates of six benchmarks (*cp*, *lbm*, *mrif*, *mriq*, *regtil*, *spmv*) increase significantly, while other benchmarks' overlap rates are still low.

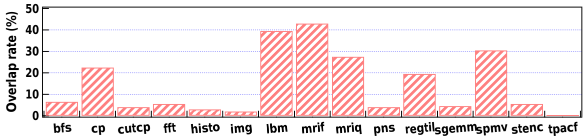

Fig. 4: Co-running two tasks in persistent block mode.

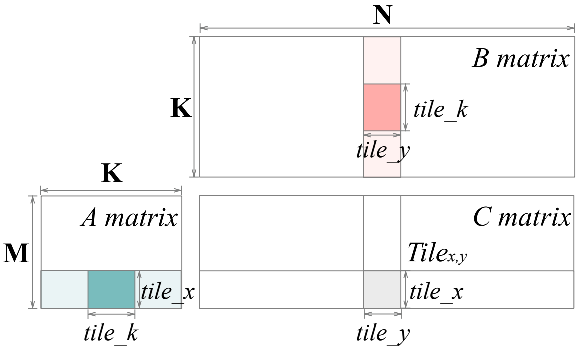

Fig. 5: Matrix mulitplication.

The improved overlap rate comes from two reasons. First, after the kernels are converted to persistent block mode, they avoid unnecessary thread occupation. Secondly, the kernels of the six benchmarks only contend for thread slots with the GEMM kernel but not for memory resources. The shared memory and registers required by them can be met even if they co-run with GEMM task. Whenever they could be scheduled on SM with GEMM tasks, the overlap rates are high.

B. Alleviating Memory System Contention

While persistent block could resolve the contention on thread slots, kernels still cannot leverage the intra-SM parallelism if they contend for *shared memory* or *registers* seriously (e.g., *fft* with the GEMM kernel in Figure 4). While there is no official shared memory multiplexing tools for co-running tasks, we focus on the connection between shared memory size, block size, and performance. **We propose elastic block to solve the memory system contention by elastically decreasing the block size.**

1) Elastic block of TC task: Tensor Cores can only perform GEMM task, which has been extensively studied. As shown in Figure 5, a GEMM task generally divides the result matrix into multiple tiles, and each tile's computation corresponds to one block. For each block, shared memory is used to store the two input matrices. Since the block's threads load the data from the global memory to shared memory collaboratively, Equation 2 shows the shared memory usage of the GEMM in Figure 5.

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

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 smaller block size as the kernel's elastic block version. It is crucial to ensure that using a smaller block size would not seriously hurt the performance. We therefore create the elastic block version of GEMM task by halving the block size.

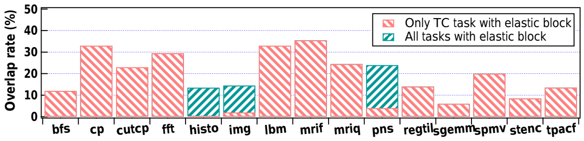

Fig. 6: Co-running two tasks in persistent and elastic block.

We collect the two kernel's performance with GEMM inputs of the convolutional layers in popular DNNs. Experimental results show that the performance gap is within 5%. While the original kernel uses 64KB shared memory, the elastic block version only uses 36KB shared memory. In this case, we can sacrifice tiny performance to reduce the shared memory usage, thus improve intra-SM parallelism. **Elastic block reduces shared memory and register usage by using a smaller tile. They also reduce the intermediate results on the SM, which brings more global memory access and possible performance degradation.**

We then verify the effectiveness of elastic block mechanism using the elastic block version TC task. As shown in Figure 6, the overlap rates of six benchmarks increase, benefitted from the shared memory released by the TC task. At the same time, three benchmarks still have low overlap rates.

2) Elastic block of CD task: Since Tensor Cores could only deal with the matrix multiplication, the TC task naturally supports adjusting the block size. While CUDA Cores support various tasks, it is unknown whether CD tasks can support the elastic block. We investigate whether the benchmarks of Parboil [43] 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 the block size adjustment.

These GPU tasks have adjustable block sizes because they all belong to one programming model. While GPU programming divides a large task into multiple subtasks, each block is responsible for a subtask. The subtask size has a relationship with the block size, and these subtasks use shared memory to accelerate memory access. Based on these two features, the smaller block size means the smaller subtask size, which leads to a reduction of the shared memory and register usage.

Based on the elastic block of CD tasks, we perform the co-running experiments with the elastic block version TC task. As shown in Figure 6, the last three benchmarks (with blue bars) with low overlap rates exploit the intra-SM parallelism.

It is not safe to directly modify the block size of CD tasks because there may be correctness issues. Therefore, the elastic block usage of CD tasks requires programmers to provide directives on whether their tasks support block size adjustment. In order to better support elastic block, we also put forward a

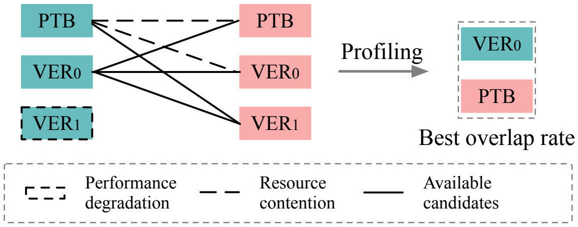

Fig. 7: Identify the optimal configuration pairs.

few suggestions to support elastic block for CD tasks.

- Use standard macro definitions. $block\_size$ and $tile\_size$ are computed from the same macro.
- Calculate the value related to $tile\_size$ in the code using the macro instead of constant value usage.

VI. Online-Offline Collaborative Scheduling

The persistent and elastic block enables the possibility to exploit the intra-SM parallelism. However, in the real-system scenario, customizing the kernel scheduling strategy from the co-running applications is still unsolved. Plasticine uses an online-offline collaborative method to identify the scheduling that results in high system-wide throughput. Specifically, the offline strategy customizes the scheduling strategy for mainstream TC-CD kernel pairs. The online scheduler makes scheduling decisions based on the strategies.

As mentioned in Section V-B, the source-to-source compiler divides the block size by the power of 2 to create multiple elastic block versions, and these versions have performance differences. While the cloud hosts long-running applications, we select mainstream kernels based on their usage, and generate all possible TC-CD kernel pairs. For each kernel pair, we need to collect the co-running performance under all configuration pairs to find the optimal one. In the above process, we first need to find the optimal persistent block number for all kernels with different block sizes. Secondly, we need to optimize the search process, because the complete search process has $O(N^2)$ complexity.

A. Locating the optimal persistent block setup

As mentioned in Section V-A, the optimal persistent block number (blk_{opt}) is the minimum block number with the same performance as the original kernel. We design a searching method based on dichotomy to locate the optimal persistent block number for each kernel. The kernel's original performance is used as the baseline for the search process. The searching range for blk_{opt} is between 1 and the maximum resident block number. 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.

B. Identifying optimal configuration pairs

After all the kernels have been converted to persistent block mode, co-running tasks may still suffer from memory resources contention. The elastic block version of GPU kernels will mitigate memory resource contention, which may introduce performance degradation. For a TC-CD kernel pair, we need to search for the optimal one among all configuration pairs. A complete search requires $O(N^2)$ time complexity, which is unacceptable. We reduce the search overhead from two aspects, as shown in Figure 7.

First, we reduce the elastic block version of each kernel. Since the elastic block version may bring performance degradation, 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 overlap rate is about 20%. When the kernel's performance drops by 20%, the kernels' co-running is hard to improve the throughput. When we filter the kernels using this threshold, 14 kernels in 16 tested kernels only have one elastic block version.

Second, we make corresponding judgments based on resource utilization. If the resource usage of two original persistent block kernels does not exceed the resource upper limit of the SM, we skip the search of other configuration pairs. Besides, for a possible configuration pair, if the resource contention prevents one kernel's block from launching, we skip the performance test of this configuration pair.

Based on the above two optimizations, each kernel pair requires 2 searches on average. Through the above configuration pair search, Plasticine identifies the optimal co-running configuration pairs for mainstream TC-CD kernel pairs, and record their overlap rates.

C. Online scheduling decision

While the offline strategy customizer locates the optimal co-running configurations for mainstream kernel pairs, the online scheduler makes kernel scheduling decisions for real-time applications. These applications 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 between 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 selects the kernel pair with the largest overlap rate based on the profiling information. These two kernels are configured with the optimal co-running configurations. 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.

$$\begin{cases} Thread_{TC} * blk\_num + Thread_{CD} * 1 < THREAD_{limit} \\ Shared_{TC} * blk\_num + Shared_{CD} * 1 < SHARED_{limit} \\ Reg_{TC} * blk\_num + Reg_{CD} * 1 < REG_{limit} \end{cases} \tag{3}$$

If some CD kernels have not been profiled with the offline customizer, the scheduler determines the co-running based on

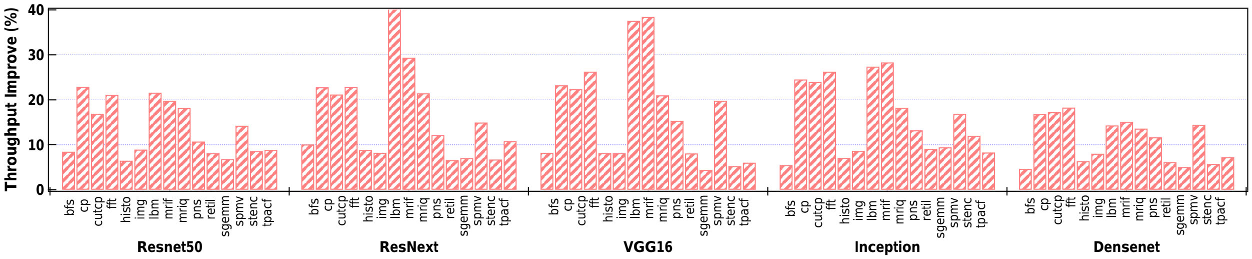

Fig. 8: The throughput improvement with Plasticine (normalized to their throughputs with CUDA stream).

TABLE V: 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 [45]
Software	GPU Driver Version: 450.51; CUDA Version: 10.0, CUDNN Version: 7.5

resource usage. If a possible TC-CD kernel pair satisfies the condition in Equation 3, one kernel could launch at least one block after the TC kernels' blocks launch. Therefore, the scheduler would schedule these two kernels to co-running for better throughput.

VII. Evaluation

In this section, we first evaluate Plasticine on the overall throughput, which enjoys the two hardware's parallelism. Second, we evaluate the ideal overlap at the kernel level. Then, we will evaluate our system on the scheduling scenarios with various tasks. Finally, the overhead of Plasticine is discussed in detail.

A. Experimental setup

Benchmarks. We choose five commonly used DNN models [46] as the Tensor Core tasks, which are Resnet50, ResNext, VGG16, Inception, and Densenet. We use all the fifteen tasks from Parboil [43] as the CUDA Core tasks, which include various GPU tasks from different domains. All tasks come in a Poisson distribution [46]. Besides, we set all the batch sizes for the DNN models as 32 except VGG16 with 16.

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 I. Note that Plasticine does not rely on any hardware features of 2080Ti and is easy to serve on other GPUs that integrate Tensor Cores.

B. Overall throughput

In this subsection, we evaluate Plasticine's effectiveness in maximizing the GPU throughput. We compare Plasticine with the CUDA stream, in which we issue the tasks without any modifications.

Figure 8 shows the system-wide throughput improvement with Plasticine compared with the CUDA stream. As observed from this figure, Plasticine improves the throughput in all the 5 * 15 = 75 pairwise co-location pairs. Plasticine increases the throughput by 15.3% on average and up to 40.3% (*ResNext* and *lbm*). Plasticine improves the throughput, because it solves the resource contention between co-running kernels. This allows Plasticine to explore the parallelism of the two hardware. On the contrary, although the CUDA stream is designed to co-running kernels, it could not utilize the two computing units' parallelism due to resource contention.

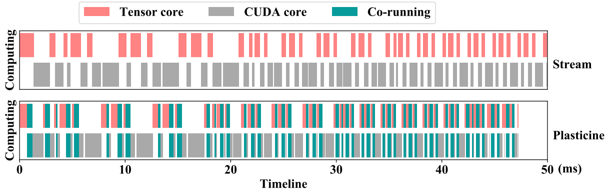

Fig. 9: The active timelines of Tensor Cores and CUDA Cores.

We can also observe that the networks have throughput improvement differences. While *ResNext* increases the throughput by 17.2% on average, *Densenet* 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 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, the CUDA stream could also enjoy the task parallelism, although it could not take advantage of two hardware's parallelism. Nevertheless, Plasticine improves the GPU throughput on all the networks.

To better understand why Plasticine performs better than CUDA stream, as an example, Figure 9 shows the execution trace if TC task *Resnet50* and CD task *fft* with Plasticine and CUDA stream. In Figure 9, 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 Plasticine utilizes two hardware's parallelism and CUDA stream could not. Since Plasticine could take advantage of these two hardware's parallelism, it improves the system-wide throughput.

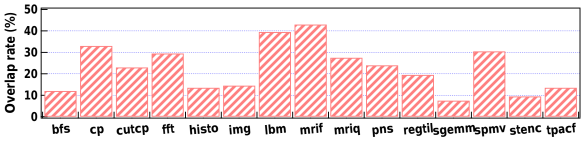

Fig. 10: The ideal overlap rate at the kernel level.

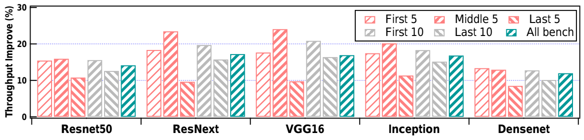

Fig. 11: The throughput of Plasticine with multiple tasks.

C. Kernel-level ideal overlap

This section proves the ideal overlap rate of Plasticine from the kernel level, which benefits from enabling two computing units' parallel usage. As shown in Figure 10, all the kernel pairs have an overlap rate of 21.3% on average and at least 9.1%, which comes from the two hardware's parallel usage.

This overlap experiment continues from Section V-B to Section VII, where there are two optimization techniques and one offline profiling method. Specifically, the persistent block first solves the thread slot contention introduced by the CUDA stream's kernel-level scheduling. Six kernels (*cp*, *lbm*, *mrif*, *mriq*, *regtil*, *spmv*) achieve the best performance while running with the original TC task in persistent block mode.

Secondly, the elastic block solves the memory resource contention, which comes from two hardware sharing the memory stack. The other six kernels (*bfs*, *cutcp*, *fft*, *sgemm*, *stenc*, *tpacf*) have the best overlap rate while running with the elastic block version of TC tasks. For the left three kernels (*histo*, *img*, *pns*), they also need to be adjusted to the elastic block version to get the best throughput. With the elastic block, these kernel pairs' resource requirements could also be satisfied.

Therefore, the persistent and elastic block solves the resource contention of co-running kernels. The offline strategy customizer searches for the optimal configurations of all kernel pairs, which brings the best overlap rate.

D. Beyond pair-wise co-locations

To evaluate the robustness of Plasticine in dealing with 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. Figure 11 shows the system-wide throughput improvement with Plasticine in these scenarios.

Plasticine improves the system throughput by 15.0% on average and 24.1% 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.

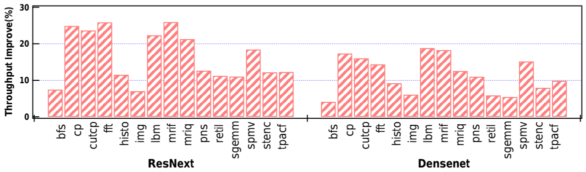

Fig. 12: The throughput for models with smaller batch sizes.

E. Models with smaller batch size

The above experiments all use the default batch size of each network, which is mostly 32. In this subsection, we also conduct experiments on the smaller batch size. Due to page limitations, we only show the results of ResNext and Densenet with batch size 16. As shown in Figure 12, Plasticine also achieves throughput improvements in all task pairs. ResNext increases the throughput by 16.5%, and Densenet increases the throughput by 11.5%. Although there is slight performance degradation, Plasticine still has considerable performance improvement with the smaller batch size.

F. Overhead

Our overhead comes from the offline strategy customizer. The customizer first filters the elastic kernels within the performance threshold. While 14 in 16 tasks only have one elastic kernel, this process generally needs two or three profile steps. Second, the customizer searches the optimal configuration pairs for possible kernel pairs. Suppose each kernel pair needs three profile steps, and each step needs 100 ms, the profiling overhead for one kernel pair is 300 ms. Since there are limited TC kernels for current tasks, the search overhead is mainly decided by the CD task number. Therefore, Plasticine's overhead is acceptable.

VIII. Conclusion

In this paper, we bridge the research gap of improving the utilization of Tensor Core enabled GPU by proposing Plasticine. A GPU kernel often either uses Tensor Cores or CUDA Cores, leaving another processing unit idle. Plasticine proposes persistent and elastic block to solve resources contention, and 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 Plasticine outperforms existing co-location based solutions by 15.3% on average, and up to 40.3%.

Acknowledgment

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61632017, 61872240, 62072297) and the Program of Shanghai Academic/Technology Research Leader (18XD1401800).

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," *Computing in Science & Engineering*.

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

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

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

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

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

[8] 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.

[9] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *ICS*.

[10] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *HPCA*. IEEE, 2020, pp. 193–206.

[11] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *HPCA*. IEEE, 2020, pp. 167–179.

[12] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ISCA*, 2015.

[13] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ISCA*. ACM, 2013, pp. 607–618.

[14] 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 *HPCA*. IEEE, 2016, pp. 358–369.

[15] 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 *ICS*, 2015, pp. 119—-130.

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

[17] "Cuda stream," https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency.

[18] F. Liu, W. Zhao, Z. Wang, T. Yang, and L. Jiang, "Im3a: Boosting deep neural network efficiency via in-memory addressing-assisted acceleration," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021.

[19] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking gpus," in *ASPLOS*, 2017.

[20] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multi-programming in gpus via effective bandwidth management," in *HPCA*. IEEE, 2018, pp. 247–258.

[21] 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 *MEMSYS*, 2015.

[22] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *HPCA*. IEEE, 2012, pp. 1–12.

[23] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of gpu resources for both performance and fairness," in *ICCD*. IEEE, 2014, pp. 440–447.

[24] Q. Hu, J. Shu, J. Fan, and Y. Lu, "Run-time performance estimation and fairness-oriented scheduling policy for concurrent gpgpu applications," in *ICPP*. IEEE, 2016, pp. 57–66.

[25] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *ASP-DAC*. IEEE, 2014, pp. 726–731.

[26] H. Lee and M. A. Al Faruque, "Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform," in *DATE*. IEEE, 2014, pp. 1–6.

[27] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. J. Reddi, "Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 44–57.

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

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

[30] Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo, "Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 399–408.

[31] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo, "Qos-aware and resource efficient microservice deployment in cloud-edge continuum," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 932–941.

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

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

[34] Z. Wang, J. Yang, R. Melhem, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *ISCA*, 2017.

[35] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional gpus: Software-based compute and memory bandwidth reservation for gpus," in *RTAS*, 2019.

[36] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *HPCA*. IEEE, 2011, pp. 382–393.

[37] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *HPCA*. IEEE, 2015, pp. 564–576.

[38] 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 *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019, pp. 497–505.

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

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

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

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

[43] 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.

[44] K. Gupta, J. A. Stuart, and J. D. Owens, *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[45] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[46] 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 *ISCA*, 2020.